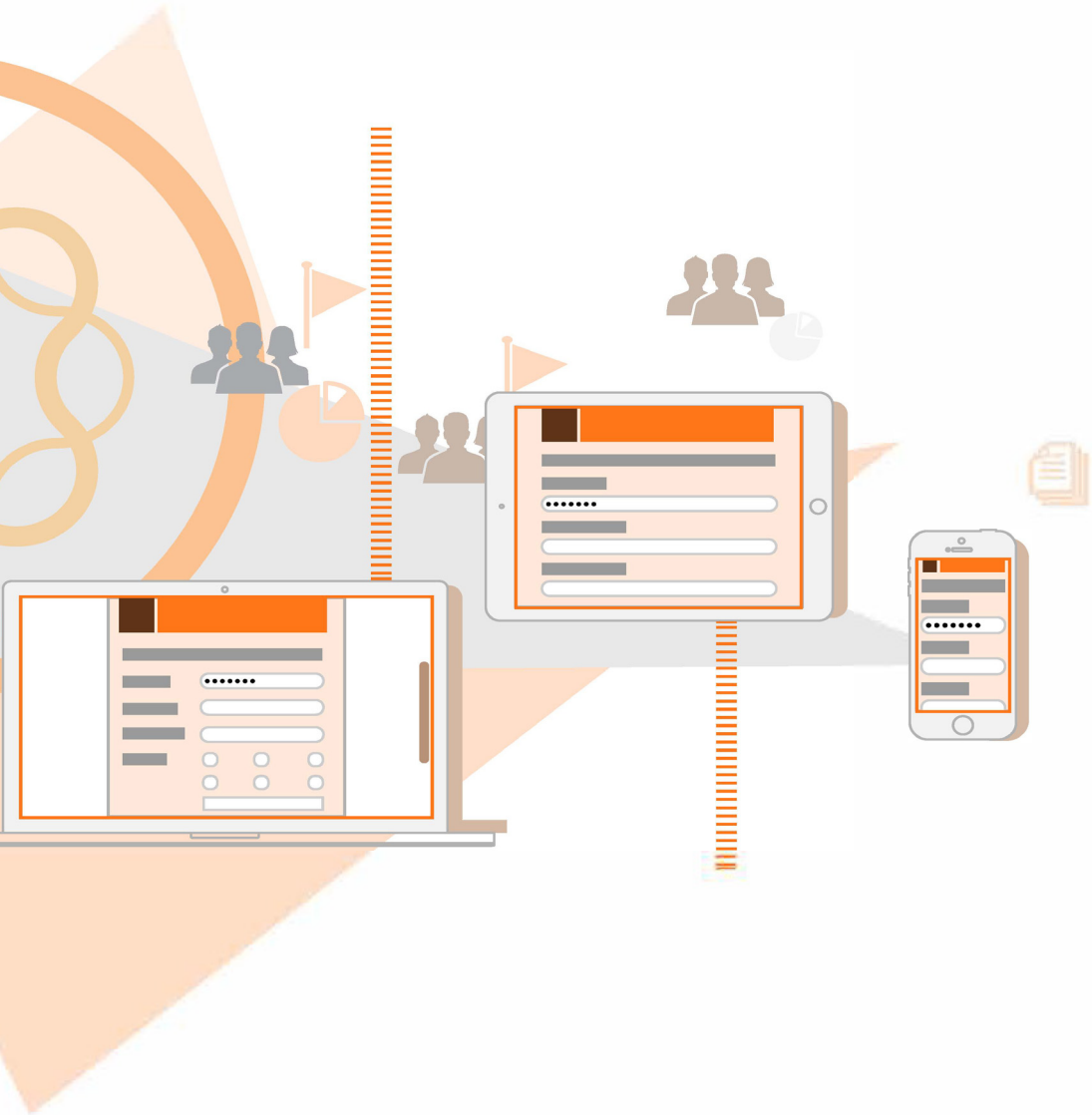
Programming with AEM Forms



AEM 6.3 Forms

Legal notices

Contents

Chapter 1: About This Help

Programming with AEM Forms is intended for developers who want to build components and client applications that programmatically interact with services in AEM Forms.

This Help does not provide service background information or service considerations that you need to know when programmatically invoking AEM Forms services. Before programmatically working with AEM Forms, it is recommended that you are familiar with AEM Forms.

Note: *Only documented APIs are supported by Adobe. Usage of any undocumented APIs are not supported.*

Use this Help to learn about the following aspects of AEM Forms SDK:

- How to invoke AEM Forms services (including processes built with *Programming with* Workbench). (See Using Workbench)
- How to develop client applications that access AEM Forms APIs by using ActionScript™ or Java™, or that use exposed WSDLs on native SOAP stacks.
- How to develop custom service providers for various services such as the Document Security service.
- How to develop components within a Java integrated development environment (IDE), such as Eclipse, that you can deploy to AEM Forms.

  This Help contains step-by-step information about using the APIs to develop components and client applications, and provides complete code examples called *Quick Starts* that you can use to get up and running immediately. (See "Introducing Java API(SOAP) Quick Start" on page 2.)

Chapter 2: Java API(SOAP) Quick Start (Code Examples)

Introducing Java API(SOAP) Quick Start

Adobe AEM Forms API Quick Start can help you accelerate your efforts to develop programs that interact with AEM Forms services. *Quick Start*s are complete programs that you can copy and paste into your own projects and use as a starting point. You can run a Quick Start to see how it behaves and modify it for your own needs.

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Java strongly-typed API Quick Start provides a listing of JAR files that are required to execute the Java application. Most Java Quick Starts are console application that run within `main`. However, the Forms Java strongly-typed API Quick Start is implemented as Java servlet that run within a web application.

The JAR file listing is located in a comment section located at the beginning of the Quick Start. For example, the following comment is located in an Output quick start and is a typical JAR file listing found in each Java Quick Start.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe--client.jar
    * 3. adobe-usermanager-client.jar
    *
    *  These JAR files are located in the following path:
    * <install directory>/Adobe/Adobe_Experience_Manager_forms/SDK/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/Adobe/Adobe_Experience_Manager_forms/SDK/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
    *
    * If you want to invoke a remote AEM Forms instance and there is a
    * firewall between the client application and AEM Forms, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/Adobe/Adobe_Experience_Manager_forms/SDK/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms library files" in Programming
    * with AEM Forms
    */
```

Multiple Services Quick Start

Most Quick Starts located in *Programming with AEM Forms* invoke a specific service in order to perform an operation. However, some Quick Starts invoke multiple AEM Forms services in order to perform a given workflow. The following list provides Java quick starts that invoke more than one AEM Forms service:

"Quick Start (SOAP mode): Passing a document located in the Repository to the Output service using the Java API" on page 245 (invokes the Repository and Output service)

"Quick Start (SOAP mode): Creating a PDF document based on fragments using the Java API" on page 255 (invokes the Assembler and Output service)

"Quick Start (SOAP mode): Creating PDF Documents with submitted XML data using the Java API" on page 185 (invokes the Forms, Output, and Document Management service)

"Quick Start (SOAP mode): Passing documents to the Forms Service using the Java API" on page 204 (invokes the Forms and Document Management service)

Quick Start (SOAP mode): Digitally signing a XFA-based Form using the Java API (invokes the Forms and Signature service)

"Quick Start (SOAP mode): Managing roles and permissions using the Java API" on page 426 (invokes the DirectoryManager and the AuthorizationManager service)

"Quick Start (SOAP mode): Passing documents to the Output Service using the Java API" on page 252 (invoke the Output and Document Management service)

Note: *Quick Start located in Programming with AEM Forms are based on AEM Forms being deployed on JBoss® Application Server and the Microsoft® Windows® operating system. However, if you are using another operating system, such as UNIX®, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Note: *Most web service Quick Starts are written in C# and uses the .NET framework. However, you can create client application logic that is able to invoke AEM Forms services in any development environment that supports SOAP standards. (See "Invoking AEM Forms using Web Services" on page 514.)*

Application Manager Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Application Manager service.

"Quick Start (SOAP mode): Deploying Applications using the Java API" on page 4

"Quick Start (SOAP mode): Removing an application using the Java API" on page 6

Note: *The application manager APIs support only AEM Forms LCA files. It does not support LCA files of LiveCycle ES2 and ES4.*

AEM Forms operations can be performed using the AEM Forms strongly typed API and the connection mode should be set to SOAP.

Note: *Java API(SOAP) Quick Start located in Programming with AEM forms are based on the Forms if you are using another operating system, such as Unix, replace windows specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Deploying Applications using the Java API

The following Java code example imports an application based on an existing LCA file named *EncryptDocument.lca*. (See "Deploying applications" on page 1105.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    * 19. adobe-workflow-client-sdk.jar
    * 20. adobe-applicationmanager-client-sdk.jar

    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
```

```
import java.io.FileInputStream;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.applicationmanager.application.ApplicationStatus;
import com.adobe.idp.applicationmanager.client.ApplicationManager;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class DeployApplication {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Get the AEM Forms application to deploy
            FileInputStream fileApp = new FileInputStream("C:\\Adobe\EncryptDocument.lca");
            Document lcApp = new Document(fileApp);

            //Create an ApplicationManager object
```

```
            ApplicationManager appManager = new ApplicationManager(myFactory);

            //Import the application into the production server
            ApplicationStatus appStatus = appManager.importApplicationArchive(lcApp);
            int status = appStatus.getStatusCode();

            //Determine if the application was successfully deployed
            if (status==1)
                    System.out.println("The application was successfully deployed");
            else
                    System.out.println("The application was not successfully deployed. The
status is "+status);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Removing an application using the Java API

The following Java code example removes an application named *EncryptDocument*. (See "Removing Applications" on page 1108.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    * 19. adobe-workflow-client-sdk.jar
    * 20. adobe-applicationmanager-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */


import java.util.*;

import com.adobe.idp.applicationmanager.application.Application;
import com.adobe.idp.applicationmanager.application.ApplicationId;

import com.adobe.idp.applicationmanager.client.ApplicationManager;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RemoveApplication {

    public static void main(String[] args) {


        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ComponentRegistryClient object
            ApplicationManager appManager = new ApplicationManager(myFactory);

            //Get all the deployed applications
            List allApps = appManager.getApplications();
```

```
            //Iterate through the applications
            Iterator iter= allApps.iterator();
            while (iter.hasNext()) {

             //Cast each element to an Application object
             Application myApplication  = (Application)iter.next();
             ApplicationId appID = myApplication.getApplicationId();
             String appName = appID.getApplicationName();

             System.out.println("The name of the AEM Forms application is "+
appID.getApplicationName());
             //Determine the name of the application
             if (appName.compareTo("EncryptDocument")==0)
             {
                //Remove the application
                appManager.removeApplication(appID);
                System.out.println("The  "+ appID.getApplicationName() +" application was
removed.");
             }
        }
        }
        catch(Exception e)
            {
            e.printStackTrace();
            }
    }
}
```

Application Manager Client Java API Quick Start(SOAP)

The following Java API Quick Start(SOAP) are available for the Application Manager Client.

"Quick Start (SOAP mode): Creating Application Version using the Java API" on page 9

"Quick Start (SOAP mode): Exporting Applications using the Java API" on page 10

"Quick Start (SOAP mode): Importing Applications using the Java API" on page 12

"Quick Start (SOAP mode): Getting a Application using the Java API" on page 14

"Quick Start (SOAP mode): Getting the applications using the Java API" on page 16

"Quick Start (SOAP mode): Getting status of applications using Java API" on page 18

"Quick Start (SOAP mode):Previewing the LiveCycle ES2 and later application archive using the Java API" on page 20

"Quick Start (SOAP mode):Deleting the Application archive using the Java API" on page 21

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick start located in Programming with AEM Forms are based on the Forms Server being deployed on JBoss and the Windows operating system. However, if you are using another operating system, such as Unix, replace windows-specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See Setting connection properties.)*

Quick Start (SOAP mode): Creating Application Version using the Java API

The following Java code example creates an application using the JAVA API.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. adobe-repository-client.jar
 * 5. activation.jar (required for SOAP mode)
 * 6. axis.jar (required for SOAP mode)
 * 7. commons-codec-1.3.jar (required for SOAP mode)
 * 8. commons-collections-3.1.jar  (required for SOAP mode)
 * 9. commons-discovery.jar (required for SOAP mode)
 * 10. commons-logging.jar (required for SOAP mode)
 * 11. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 12. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 13. jaxrpc.jar (required for SOAP mode)
 * 14. log4j.jar (required for SOAP mode)
 * 15. mail.jar (required for SOAP mode)
 * 16. saaj.jar (required for SOAP mode)
 * 17. wsdl4j.jar (required for SOAP mode)
 * 18. xalan.jar (required for SOAP mode)
 * 19. xbean.jar (required for SOAP mode)
 * 20. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
 *
 */
package com.adobe.idp.dsc.applicationmanager;
import java.util.Properties;
import java.util.Random;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.repository.bindings.ResourceRepositoryDelegate;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
public class CreateApplicationVersion_SOAP {
    private static String applicationFolder = "Applications";
    private static String defaultAppVersion = "1.0";
    public static void main(String[] args) {
        // Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT",
                "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
                ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE",
                ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);
```

```
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");
        // Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory
                .createInstance(connectionProps);
        // Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(
                myFactory);
        // Create ResourceRepositoryDelegate object
        ResourceRepositoryDelegate repositoryClient = new ResourceRepositoryClient(
                myFactory);
        final Random num = new Random();
        String appName = "App" + num.nextInt();
        String newAppName = null;
        try {
            // Create application with default application version
            newAppName = appClient.createApplication(appName);
            if (repositoryClient.resourceExists("/" + applicationFolder + "/"
                    + appName.toString() + "/" + defaultAppVersion)) {
                System.out.println("Application with name: " + appName + "/"
                        + defaultAppVersion + " is created succesfully!");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            // Create version of the new application
            appName = appClient.createApplicationVersion(newAppName,
                    defaultAppVersion, "2.0", "version increment");
            if (repositoryClient.resourceExists("/" + applicationFolder + "/"
                    + appName.toString() + "/" + "2.0")) {
                System.out.println("Application version 2.0 created : "
                        + appName + "/" + "2.0");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Exporting Applications using the Java API

The following Java code example exports an application using the JAVA API.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
 *
 */
package com.adobe.idp.dsc.applicationmanager;
import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;
public class ExportLCA_SOAP {
    public static void main(String[] args) {
        // Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT",
                "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
                ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE",
                ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");
        // Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory
```

```
            .createInstance(connectionProps);
        // Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(
                myFactory);
        Document doc = null;
        try {
            final FileInputStream fileApp = new FileInputStream(
                    "C:\\ImportSampleApp2.lca");
            doc = new Document(fileApp);
        } catch (Exception e) {
            e.printStackTrace();
        }
        String resourceID = null;
        try {
            // Import the application into the LC server
            resourceID = appClient.importApplication(doc);
            System.out.println("Import application with resource ID:"
                    + resourceID + " is completed successfully!");
        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }
        final String sampleAppName = "ExportSampleApp2";
        try {
            final List<String> eReqList = new ArrayList<String>();
            eReqList.add(resourceID);
            // Export the application imported above
            doc = appClient.export(eReqList, "lcaDescription");
            // Save into local LCA file
            final String archiveName = "C:/" + sampleAppName + "-" + "1.0"
                    + ".lca";
            final File fTemp = new File(archiveName);
            doc.copyToFile(fTemp);
            System.out.println("Export application completed with name: "
                    + archiveName);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Importing Applications using the Java API

The following Java code example imports an application using the JAVA API.

Note: *The Java API importApplication() replaces existing applications of the same name with newer application. To update an exsiting application, use API importApplication() in place of API updateApplication().*

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
 *
 */
package com.adobe.idp.dsc.applicationmanager;
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;
public class ImportLCA_SOAP {
    public static void main(String[] args) {
        // Set connection properties required to invoke AEM FOrms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT",
                "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
                ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE",
                ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");
        // Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory
                .createInstance(connectionProps);
        // Create ApplicationManagerClient object
```

```
        ApplicationManagerClient appClient = new ApplicationManagerClient(
                myFactory);
        Document doc = null;
        try {
            final FileInputStream fileApp = new FileInputStream(
                    "C:\\ImportSampleApp2.lca");
            doc = new Document(fileApp);
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            // Import the application into the LC server
            final String resourceID = appClient.importApplication(doc);
            System.out.println("Import application with resource ID:"
                    + resourceID + " is completed successfully!");
        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Getting a Application using the Java API

The following Java code example gets an application using the Java API.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
 *
 */
```

```
package com.adobe.idp.dsc.applicationmanager;

import java.io.FileInputStream;
import java.util.Properties;

import com.adobe.idp.Document;
import com.adobe.idp.applicationmanager.application.Application;
import com.adobe.idp.applicationmanager.application.ApplicationId;
import com.adobe.idp.applicationmanager.application.impl.ApplicationIdImpl;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;

public class GetApplication_SOAP {

    public static void main(String[] args) {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

        //Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        //Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(myFactory);

        Document doc = null;
        try {

            final FileInputStream fileApp = new FileInputStream("C:\\appraisal.lca");
            doc = new Document(fileApp);

        } catch (Exception e) {
            e.printStackTrace();
        }
        final String sampleAppName = "Samples - Performance Appraisal";
        try {

            //Import the application into the LC server
            final String resourceID = appClient.importApplication(doc);
            System.out.println("Import application with resource ID: " + resourceID + " is
completed successfully!");
            //Deploy the application
            boolean result = appClient.deployApplication(sampleAppName);
            if (result) {
                System.out.println("Imported application is deployed.");
```

```
            }
        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }

        //Initialize the ApplicationId instance
        ApplicationId appId = new ApplicationIdImpl();
        appId.setApplicationName(sampleAppName);

        try {
                //Get the application by application id
                Application app = appClient.getApplication(appId);
                System.out.println("Get application with name: " +
app.getApplicationId().getApplicationName());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Getting the applications using the Java API

The following Java code example gets the applications using the Java API.

Note: *Getting AEM Forms Application API, getApplications(), returns only deployed applications.*

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
```

```
 *
 */ package com.adobe.idp.dsc.applicationmanager;

import java.io.FileInputStream;
import java.util.List;
import java.util.Properties;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;

public class GetApplications_SOAP {

    public static void main(String[] args) {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

        //Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        //Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(myFactory);

        Document doc = null;
        try {

            final FileInputStream fileApp = new FileInputStream("C:\\appraisal.lca");
            doc = new Document(fileApp);

        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        try {

            //Import the application into the LC server
            final String resourceID = appClient.importApplication(doc);
            System.out.println("Import application with resource ID: " + resourceID + " is
completed successfully!");

        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }

        try {
                //Get applications from LC server
                List appList = appClient.getApplications();
                System.out.println("Get applications from LC Server: " + appList.size());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Getting status of applications using Java API

The following Java code example gets status of an application using the Java API.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
 *
 */
```

```
package com.adobe.idp.dsc.applicationmanager;

import java.io.FileInputStream;
import java.util.Properties;

import com.adobe.idp.Document;
import com.adobe.idp.applicationmanager.application.ApplicationId;
import com.adobe.idp.applicationmanager.application.ApplicationStatus;
import com.adobe.idp.applicationmanager.application.impl.ApplicationIdImpl;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;

public class GetApplicationStatus_SOAP {

    public static void main(String[] args) {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

        //Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        //Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(myFactory);

        Document doc = null;
        try {

            final FileInputStream fileApp = new FileInputStream("C:\\appraisal.lca");
            doc = new Document(fileApp);

        } catch (Exception e) {
            e.printStackTrace();
        }
        final String sampleAppName = "Samples - Performance Appraisal";
        try {

            //Import the application into the LC server
            final String resourceID = appClient.importApplication(doc);
            System.out.println("Import application with resource ID: " + resourceID + " is
completed successfully!");
            //Deploy the application
            boolean result = appClient.deployApplication(sampleAppName);
            if (result) {
                System.out.println("Imported application is deployed.");
```

```
            }
        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }

        //Initialize the ApplicationId instance
        ApplicationId appId = new ApplicationIdImpl();
        appId.setApplicationName(sampleAppName);

        try {
                //Get the application by application id
                ApplicationStatus status = appClient.getApplicationStatus(appId);
                System.out.println("Get application status with code: " +
status.getStatusCode());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode):Previewing the LiveCycle ES2 and later application archive using the Java API

The following Java code example is for prevewing AEM Forms and later application archive using the Java API.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
*/
```

```
package com.adobe.idp.dsc.applicationmanager;
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;
public class PreviewLCA_SOAP {
    public static void main(String[] args) {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");
        //Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        //Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(myFactory);

        Document doc = null;
        try {

            final FileInputStream fileApp = new FileInputStream("C:\\appraisal.lca");
            doc = new Document(fileApp);

        } catch (Exception e) {
            e.printStackTrace();
        }

        try {

            //Preview the LCA
            final Document newDoc = appClient.previewLCA(doc);

        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode):Deleting the Application archive using the Java API

The following Java code example is for deleting an application archive.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 * 3. adobe-application-remote-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/sdk/client-libs/common
 *
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/sdk/client-libs/thirdparty
 *
 *
 */
package com.adobe.idp.dsc.applicationmanager;

import java.io.FileInputStream;
import java.util.Properties;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClient;
import com.adobe.livecycle.applicationmanager.client.ApplicationManagerClientException;

public class DeleteApplication_SOAP {

    public static void main(String[] args) {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
        connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL",
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
        connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
        connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

        //Create ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
```

```
        //Create ApplicationManagerClient object
        ApplicationManagerClient appClient = new ApplicationManagerClient(myFactory);

        Document doc = null;
        try {

            final FileInputStream fileApp = new FileInputStream("C:\\appraisal.lca");
            doc = new Document(fileApp);

        } catch (Exception e) {
            e.printStackTrace();
        }

        try {

            //Import the application into the LC server
            final String resourceID = appClient.importApplication(doc);
            System.out.println("Import application with resource ID: " + resourceID + " is
completed successfully!");

        } catch (ApplicationManagerClientException e) {
            e.printStackTrace();
        }
        final String sampleAppName = "Samples - Performance Appraisal";
        try {
                //Delete the application imported above
                appClient.deleteApplication(sampleAppName, "1.0");
                System.out.println("Delete application completed with name: " + sampleAppName);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Assembler Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Assembler service

"Quick Start (SOAP mode): Assembling a PDF document using the Java API" on page 24

"Quick Start (SOAP mode): Disassembling a PDF document using the Java API" on page 27

"Quick Start (SOAP mode): Assembling an encrypted PDF document using the Java API" on page 29

"Quick Start (SOAP mode): Assembling a PDF document with bates numbering using the Java API" on page 32

"Quick Start (SOAP mode): Assembling a non-interactive PDF document using the Java API" on page 35

"Quick Start (SOAP mode): Determining whether a document is PDF/A compliant using the Java API" on page 37

"Quick Start (SOAP mode): Validating DDX documents using the Java API" on page 40

"Quick Start (SOAP mode): Assembling PDF documents with bookmarks using the Java API" on page 43

"Quick Start (SOAP mode): Dynamically creating a DDX document using the Java API" on page 46

"Quick Start (SOAP mode): Assembling PDF Portfolios using the Java API" on page 51

"Quick Start (SOAP mode): Assembling multiple XDP fragments using the Java API" on page 54

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Start located in Programming with AEM Forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Assembling a PDF document using the Java API

The following Java code example merges two PDF source documents named *map.pdf* and *directions.pdf* into a single PDF document. The name of the single PDF document is *AssemblerResultPDF.pdf*. The name of the DDX document is *shell.xml*. (See "Programmatically Assembling PDF Documents" on page 925.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
```

```
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document used in this quick start:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *   <PDF result="out.pdf">
    *       <PDF source="map.pdf" />
    *       <PDF source="directions.pdf" />
    *   </PDF>
    *</DDX>
    */
import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class InvokeAssemblerSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\shell.xml");
```

```
            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create a Map object to store PDF source documents
            Map inputs = new HashMap();
            FileInputStream mySourceMap = new FileInputStream("C:\\map.pdf");
            FileInputStream mySourceOptions = new FileInputStream("C:\\directions.pdf");

            //Create a Document object based on the map.pdf source file
            Document myPDFMapSource = new Document(mySourceMap);

            //Create a Document object based on the directions.pdf source file
            Document myPDFOptionsSource = new Document(mySourceOptions);

            //Place two entries into the Map object
            inputs.put("map.pdf",myPDFMapSource);
            inputs.put("directions.pdf",myPDFOptionsSource);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;

            //Iterate through the map object to retrieve the result PDF document
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
                Map.Entry e = (Map.Entry)i.next();

                //Get the key name as specified in the
                //DDX document
                String keyName = (String)e.getKey();
                if (keyName.equalsIgnoreCase("out.pdf"))
                {
                    Object o = e.getValue();
                    outDoc = (Document)o;

                    //Save the result PDF file
                    File myOutFile = new File("C:\\AssemblerResultPDF.pdf");
                    outDoc.copyToFile(myOutFile);
                }
            }
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Disassembling a PDF document using the Java API

The following Java code example disassembles a PDF document named *AssemblerResultPDF.pdf*. Notice that the name of the DDX document is *shell_disassemble.xml*. Each disassembled PDF document is named *ResultPDF[Number].pdf*. That is, the first disassembled PDF document is named *ResultPDF1.pdf*. For information about the *shell_disassemble.xml* DDX document used in this code example, see "Programmatically Disassembling PDF Documents" on page 932.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document used in this quick start:
    * <?xml version="1.0" encoding="UTF-8"?>
```

```
    *<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *  <PDFsFromBookmarks prefix="stmt">
    *   <PDF source="AssemblerResultPDF.pdf"/>
    *</PDFsFromBookmarks>
    *</DDX>
    */
import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class DisassemblePDFSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\shell_disassemble.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create a Map object to store PDF source documents
            Map inputs = new HashMap();
            FileInputStream mySourceMap = new FileInputStream("C:\\AssemblerResultPDF.pdf");

            //Create a Document object based on the map.pdf source file
            Document myPDFSource = new Document(mySourceMap);

            //Place two entries into the Map object
            inputs.put("AssemblerResultPDF.pdf",myPDFSource);
```

```
            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to the Assembler service
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF documents from the Map object
            Document outDoc = null;
            int index = 0;

            //Iterate through the map object to retrieve the result PDF document
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
                Map.Entry e = (Map.Entry)i.next();
                Object o = e.getValue();

                //Cast the Object to a Document
                //and save to a file
                outDoc = (Document)o;
                File myOutFile = new File("C:\\ResultPDF"+index +".pdf");
                outDoc.copyToFile(myOutFile);
                index++;
            }
            if (index > 0)
                System.out.println("The PDF document was disassembled into "+index+" PDF
documents.");
            else
                System.out.println("The PDF document was not disassembled.");

        }catch (Exception e) {
            System.out.println("Error OCCURRED: "+e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Assembling an encrypted PDF document using the Java API

The following Java code example assembles a password-encrypted PDF document. The unsecured PDF document is named *Loan.pdf*. Notice that the name of the DDX document is *shell_Encrypt.xml*. The encrypted PDF document is named *AssemblerEncryptedPDF.pdf*. (See "Assembling Encrypted PDF Documents" on page 937.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document used in this quick start:
    *<?xml version="1.0" encoding="UTF-8"?>
    *  <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *   <PDF result="EncryptLoan.pdf" encryption="userProtect">
    *       <PDF source="inDoc" />
    *   </PDF>
    *   <PasswordEncryptionProfile name="userProtect" compatibilityLevel="Acrobat7">
    *       <OpenPassword>AdobeOpen</OpenPassword>
    *   </PasswordEncryptionProfile>
    * </DDX>
```

```
    */

import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AssembleEncryptedDocumentSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            /*
             * Create a FileInputStream object based on an existing DDX file
             * This DDX document contains instructions to encrypt the PDF document
             */
            FileInputStream myDDXFile = new FileInputStream("C:\\shell_Encrypt.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Reference an unsecured PDF document
            FileInputStream mySourceLoan = new FileInputStream("C:\\Loan.pdf");
```

```
            //Create a Document object based on the Loan.pdf source file
            Document myPDFLoanSource = new Document(mySourceLoan);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service
            Document jobResult =
assemblerClient.invokeOneDocument(myDDX,myPDFLoanSource,assemblerSpec);

            //Create the output file
            File myOutFile = new File("C:\\AssemblerEncryptedPDF.pdf");
            jobResult.copyToFile(myOutFile);

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Assembling a PDF document with bates numbering using the Java API

The following Java code example assembles a PDF document with unique page identifiers (bates numbering). Notice that the name of the DDX document is *shell_Bates.xml*. The PDF document that is returned from the Assembler service is saved as a PDF file named *AssemblerResultBatesPDF.pdf*. (See "Assembling Documents Using Bates Numbering" on page 947.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
```

```
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document used in this quick start:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *   <PDF result="out.pdf">
    *   <Header>
    *       <Center>
    *           <StyledText>
    *               <p font-size="20pt"><BatesNumber/></p>
    *           </StyledText>
    *       </Center>
    *   </Header>
    *      <PDF source="map.pdf" />
    *     <PDF source="directions.pdf" />
    *     </PDF>
    * </DDX>
    */

import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class AssembleBatesNumberDocumentSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
```

```
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\shell_Bates.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create a Map object to store PDF source documents
            Map<String, Object> inputs = new HashMap<String, Object>();
            FileInputStream mySourceMap = new FileInputStream("C:\\Adobe\map.pdf");
            FileInputStream mySourceOptions = new FileInputStream("C:\\Adobe\directions.pdf");

            //Create a Document object based on the map.pdf source file
            Document myPDFMapSource = new Document(mySourceMap);

            //Create a Document object based on the directions.pdf source file
            Document myPDFOptionsSource = new Document(mySourceOptions);

            //Place two entries into the Map object
            inputs.put("map.pdf",myPDFMapSource);
            inputs.put("directions.pdf",myPDFOptionsSource);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Set the initial number to 100
            assemblerSpec.setFirstBatesNumber(100);

            //Submit the job to Assembler service
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;

            //Iterate through the map object to retrieve the result PDF document
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
                Map.Entry e = (Map.Entry)i.next();
```

```
                //Get the key name as specified in the
                //DDX document
                String keyName = (String)e.getKey();
                if (keyName.equalsIgnoreCase("out.pdf"))
                {
                    //Cast the Object to a Document
                    Object o = e.getValue();
                    outDoc = (Document)o;

                    //Create the output file
                    File myOutFile = new File("C:\\AssemblerResultBatesPDF.pdf");
                    outDoc.copyToFile(myOutFile);
                }
            }
            System.out.println("The PDF that contains Bates numbering was assembled.");
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Assembling a non-interactive PDF document using the Java API

The following Java code example assembles a non-interactive PDF document. The interactive PDF document that is passed to the Assembler service is named *Loan.pdf*. Notice that the name of the DDX document is *shell_XFA.xml*. The non-interactive PDF document is saved as a PDF file named *AssembleNonInteractivePDF.pdf*. (See "Assembling Non-Interactive PDF Documents" on page 942.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
```

```
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document used in this quick start:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    * <PDF result="out.pdf">
    *   <PDF source="inDoc"/>
    *   <NoXFA/>
    * </PDF>
    * </DDX>
    */
import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AssembleNonInteractiveSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
```

```
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            /*
             * Create a FileInputStream object based on an existing DDX file
             * This DDX document contains instructions to create
             * a non-interactive PDF document
             */
            FileInputStream myDDXFile = new FileInputStream("C:\\shell_XFA.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Reference an interactive PDF document
            FileInputStream mySourceLoan = new FileInputStream("C:\\Adobe\Loan.pdf");

            //Create a Document object based on the Loan.pdf source file
            Document myPDFLoanSource = new Document(mySourceLoan);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service and get back a
            //non-interactive PDF document
            Document outDoc =
assemblerClient.invokeOneDocument(myDDX,myPDFLoanSource,assemblerSpec);

            //Save the non-interactive PDF document
            File myOutFile = new File("C:\\AssembleNonInteractivePDF.pdf");
            outDoc.copyToFile(myOutFile);

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Determining whether a document is PDF/A compliant using the Java API

The following Java code example determines whether the input PDF document is PDF/A compliant. The input PDF document that is passed to the Assembler service is named *Loan.pdf*. The name of the DDX document is shell_PDFA.xml. The XML document that is returned from the Assembler service and specifies whether the input PDF document is PDF/A compliant is saved as an XML file named result.xml. For information about the *shell_PDFA.xml* DDX document used in this code example, see "Determining Whether Documents Are PDF/A- Compliant" on page 952.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * This quick start validates the following DDX document:
    *<?xml version="1.0" encoding="UTF-8"?>
    *<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *   <DocumentInformation source="Loan.pdf" result="Loan_result.xml">
    *   <PDFAValidation compliance="PDF/A-1b" resultLevel="Detailed"
ignoreUnusedResources="true" allowCertificationSignatures="true" />
    *   </DocumentInformation>
    *</DDX>
    */
```

```
import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AssembleDeterminePDFASOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\shell_PDFA.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create a Map object to store PDF source documents
            Map inputs = new HashMap();
            FileInputStream mySourceMap = new FileInputStream("C:\\Adobe\Loan.pdf");

            //Create a Document object based on the map.pdf source file
            Document myPDFMapSource = new Document(mySourceMap);

            //Place two entries into the Map object
            inputs.put("Loan.pdf",myPDFMapSource);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service
```

```
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;

            //Iterate through the map object to retrieve the result XML
            //document that specifies if the input document is
            //PDF/A compliant
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
                Map.Entry e = (Map.Entry)i.next();

                //Get the key name as specified in the
                //DDX document
                String keyName = (String)e.getKey();
                if (keyName.equalsIgnoreCase("Loan_result.xml"))
                {
                    //Get the element value
                    Object o = e.getValue();

                    //Cast the Object to a Document
                    outDoc = (Document)o;

                    //Save the XML file
                    File myMXLFile = new File("C:\\Adobe\result.xml");
                    outDoc.copyToFile(myMXLFile);
                    }
                }

            System.out.println("The results are written to result.xml.");
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Validating DDX documents using the Java API

The following Java code example validates a DDX document based on a file named *bookmarkDDX.xml*. (See "Validating DDX Documents" on page 965.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * This quick start validates the following DDX document:
    *&<?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *   <PDF result="out.pdf">
    *       <PDF source="map.pdf" />
    *       <PDF source="directions.pdf" />
    *   </PDF>
    * </DDX>
    */
import com.adobe.livecycle.assembler.client.*;
```

```
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ValidateDDXSOAP
{
    public static void main(String[] args) {

        boolean isValid = false;
           Document outLog = null;

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\bookmarkDDX.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setValidateOnly(true);
            assemblerSpec.setLogLevel("FINE");
            assemblerSpec.setFailOnError(false);

            //Validate the DDX document
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,null,assemblerSpec);
            outLog = jobResult.getJobLog();
            isValid = true;

        }catch (Exception e) {
```

```
            if (e instanceof OperationException) {
                   OperationException oe = (OperationException) e;
                   outLog = oe.getJobLog();
                   File myOutFile = new File("C:\\test.xml");
                   outLog.copyToFile(myOutFile);
               }
            e.printStackTrace();
         } finally {
              if (outLog != null) {
                  File myOutFile = new File("C:\\test.xml");
                  outLog.copyToFile(myOutFile);
              }
          if (isValid) {
                // do something
          } else {
               // do something else
            }
          }
    }
}
```

Quick Start (SOAP mode): Assembling PDF documents with bookmarks using the Java API

The following Java code example assembles a PDF document that contains bookmarks. The name of the DDX document is *bookmarkDDX.xml*. The name of the bookmark XML document that describes the bookmarks to add to the PDF document is bookmarks.xml. The result PDF document is saved as a PDF file named AssemblerResultBookmarks.pdf. (See "Assembling PDF Documents with Bookmarks" on page 958.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
```

```
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * * This quick start uses the following DDX document:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *  <PDF result="FinalDoc.pdf">
    *     <PDF source="Loan.pdf">
    *        <Bookmarks source="doc2" />
    *     </PDF>
    *  </PDF>
    * </DDX>
    *
    * This quick start also uses the following bookmarks XML
    * to assemble a PDF document containing bookmarks:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <Bookmarks xmlns="http://ns.adobe.com/pdf/bookmarks" version="1.0">
    *  <Bookmark>
    *     <Action>
    *        <Launch NewWindow="true">
    *           <File Name="C:\Adobe\LoanDetails.pdf" />
    *        </Launch>
    *     </Action>
    *       <Title>Open the Loan document</Title>
    *  </Bookmark>
    * <Bookmark>
    *     <Action>
    *        <Launch>
    *           <Win Name="C:\WINDOWS\notepad.exe" />
    *        </Launch>
    *     </Action>
    *   <Title>Launch NotePad</Title>
    *  </Bookmark>
    * </Bookmarks>
    *
    */
    import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
```

```
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AssembleBookmarksSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\bookmarkDDX.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create a Map object to store an input PDF document and a Bookmark
            //XML document
            Map inputs = new HashMap();
            FileInputStream mySourceMap = new FileInputStream("C:\\Loan.pdf");
            FileInputStream bookmarkInfo = new FileInputStream("C:\\bookmarks.xml");

            //Create a Document object based on the Loan.pdf source file
            Document myPDFMapSource = new Document(mySourceMap);

            //Create a Document object based on the bookmarks.xml file
            Document myBookmarkXML= new Document(bookmarkInfo);

            //Place two entries into the Map object
            inputs.put("Loan.pdf",myPDFMapSource);
            inputs.put("doc2",myBookmarkXML);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);
```

```
            //Submit the job to Assembler service
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;

            //Iterate through the map object to retrieve the result PDF document
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
                Map.Entry e = (Map.Entry)i.next();

                //Get the key name as specified in the
                //DDX document
                String keyName = (String)e.getKey();
                if (keyName.equalsIgnoreCase("FinalDoc.pdf"))
                {
                    Object o = e.getValue();
                    outDoc = (Document)o;

                    //Save the result PDF file
                    File myOutFile = new
File("C:\\Adobe\Assembler\Output\AssemblerResultBookmarks.pdf");
                    outDoc.copyToFile(myOutFile);
                }
            }
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Dynamically creating a DDX document using the Java API

The following Java code example dynamally creates a DDX document that disassembles a PDF document. A new PDF document is created for each level 1 bookmark in the input PDF document. This code example contains two user-defined methods:

- `createDDX`: Creates an `org.w3c.dom.Document` object that represents the DDX document that is sent to the Assembler service. This user-defined method returns the `org.w3c.dom.Document` object.
- `convertDDX`: Converts an `org.w3c.dom.Document` object to a `com.adobe.idp.Document` object. This method accepts an `org.w3c.dom.Document` object as an input parameter and returns a `com.adobe.idp.Document` object.

  Both of these methods are invoked in this quick start. (See "Dynamically Creating DDX Documents" on page 969.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document created in this quick start:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    * <PDF result="out.pdf">
    *   <PDF source="inDoc"/>
    *   <NoXFA/>
    * </PDF>
    * </DDX>
    */
import com.adobe.livecycle.assembler.client.*;
```

```
import java.util.*;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Element;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AssemblePDFWithDynamicDDXSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCl
ientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Dynamically create a DDX document
            org.w3c.dom.Document myDDX= createDDX() ;

            //Covert the DDX document to a com.adobe.idp.Document instance
            com.adobe.idp.Document ddx = convertDDX(myDDX);

            //Create a Map object to store PDF source documents
            Map inputs = new HashMap();
            FileInputStream mySourceMap = new FileInputStream("C:\\AssemblerResultPDF.pdf");
```

```
            //Create a Document object based on the map.pdf source file
            Document myPDFSource = new Document(mySourceMap);

            //Place the entry into the Map object
            inputs.put("AssemblerResultPDF.pdf",myPDFSource);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service and use the dynamically created DDX document
            AssemblerResult jobResult = assemblerClient.invokeDDX(ddx,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;
            int index = 1;

            //Iterate through the map object to retrieve the result PDF documents
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
                Map.Entry e = (Map.Entry)i.next();
                Object o = e.getValue();

                //Cast the Object to a Document
                //and save to a file
                outDoc = (Document)o;
                File myOutFile = new File("C:\\ResultPDF"+index +".pdf");
                outDoc.copyToFile(myOutFile);
                index++;
            }

        }catch (Exception e) {
            e.printStackTrace();
        }
    }

     //Creates a DDX document using an org.w3c.dom.Document object
     private static org.w3c.dom.Document createDDX()
     {
        org.w3c.dom.Document document = null;

        try
        {
            //Create DocumentBuilderFactory and DocumentBuilder objects
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();

            //Create a new Document object
            document = builder.newDocument();

            //Create the root element and append it to the XML DOM
            Element root = (Element)document.createElement("DDX");
            root.setAttribute("xmlns","http://ns.adobe.com/DDX/1.0/");
            document.appendChild(root);

            //Create the PDFsFromBookmarks element
```

```
            Element PDFsFromBookmarks =
(Element)document.createElement("PDFsFromBookmarks");
            PDFsFromBookmarks.setAttribute("prefix","stmt");
            root.appendChild(PDFsFromBookmarks);

            //Create the PDF element
            Element PDF = (Element)document.createElement("PDF");
            PDF.setAttribute("source","AssemblerResultPDF.pdf");
            PDFsFromBookmarks.appendChild(PDF);
      }
     catch (Exception e) {
               System.out.println("The following exception occurred: "+e.getMessage());
             }
    return document;
     }


     //Converts an org.w3c.dom.Document object to a
     //com.adobe.idp.Document object
     private static Document convertDDX(org.w3c.dom.Document myDOM)
     {
        byte[] mybytes = null;

        try
        {
        //Create a Java Transformer object
        TransformerFactory transFact = TransformerFactory.newInstance();
        Transformer transForm = transFact.newTransformer();

        //Create a Java ByteArrayOutputStream object
        ByteArrayOutputStream myOutStream = new ByteArrayOutputStream();

        //Create a Java Source object
        javax.xml.transform.dom.DOMSource myInput = new DOMSource(myDOM);

        //Create a Java Result object
        javax.xml.transform.stream.StreamResult myOutput = new StreamResult(myOutStream);

        //Populate the Java ByteArrayOutputStream object
        transForm.transform(myInput,myOutput);
```

```
            // Get the size of the ByteArrayOutputStream buffer
             int myByteSize = myOutStream.size();

            //Allocate myByteSize to the byte array
             mybytes = new byte[myByteSize];

            //Copy the content to the byte array
             mybytes = myOutStream.toByteArray();
            }
            catch (Exception e) {
              System.out.println("The following exception occurred: "+e.getMessage());
          }

        //Create a com.adobe.idp.Document object and copy the
        //contents of the byte array
        Document myDocument = new Document(mybytes);
        return myDocument;
        }

    }
```

Quick Start (SOAP mode): Assembling PDF Portfolios using the Java API

The following Java code example creates a PDF portfolio. The PDF portfolio is saved as a PDF file named *AssemblerResultPortfolio.pdf*. (See "Assembling PDF Portfolios" on page 976.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
```

```
    *
    * This is the DDX file used to create a PDF portfolio:
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *   <PDF result="portfolio1.pdf">
    *       <Portfolio>
    *           <Navigator source="myNavigator">
    *               <Resource name="navigator/image.xxx" source="myImage.png"/>
    *           </Navigator>
    *       </Portfolio>
    *       <PackageFiles source="dog1"  >
    *           <FieldData name="X">72</FieldData>
    *           <FieldData name="Y">72</FieldData>
    *           <File filename="saint_bernard.jpg" mimetype="image/jpeg"/>
    *       </PackageFiles>
    *       <PackageFiles source="dog2"  >
    *           <FieldData name="X">120</FieldData>
    *           <FieldData name="Y">216</FieldData>
    *           <File filename="greyhound.pdf"/>
    *       </PackageFiles>
    *   </PDF>
    * </DDX>
    */
import com.adobe.livecycle.assembler.client.*;

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class CreatePDFPortfolioSOAP {
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);
```

```
            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new
FileInputStream("C:\\Adobe\portfolioAssembly.xml");
            FileInputStream myNavFile = new FileInputStream("C:\\Adobe\AdobeOnImage.nav");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);
            Document myNav = new Document(myNavFile);

            //Create a Map object to store PDF source documents
            Map<String,Object> input = new HashMap<String, Object>();
            FileInputStream mySourceNavImage = new FileInputStream("C:\\Adobe\myImage.png");
            FileInputStream mySourceDog1 = new FileInputStream("C:\\Adobe\saint_bernard.jpg");
            FileInputStream mySourceDog2 = new FileInputStream("C:\\Adobe\greyhound.pdf");

            //Create a Document object based on the myImage.png source file
            Document myPDFNavImageSource = new Document(mySourceNavImage);

            //Create a Document object based on the MyFirstFile.pdf source file
            Document myPDFDog1Source = new Document(mySourceDog1);

            //Create a Document object based on the MySecondFile.txt source file
            Document myPDFDog2Source = new Document(mySourceDog2);

            //Place two entries into the Map object
            input.put("myNavigator", myNav);
            input.put("myImage.png",myPDFNavImageSource);
            input.put("dog1",myPDFDog1Source);
            input.put("dog2",myPDFDog2Source);

            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,input,assemblerSpec);
            Map<String,Document> allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;

            //Iterate through the map object to retrieve the result PDF document
            for (Iterator<Map.Entry<String,Document>> i = allDocs.entrySet().iterator();
i.hasNext();) {
                // Retrieve the Map object?s value
                Map.Entry<String,Document> e = (Map.Entry<String,Document>)i.next();

                //Get the key name as specified in the
```

```
                //DDX document
                String keyName = (String)e.getKey();
                if (keyName.equalsIgnoreCase("portfolio1.pdf"))
                {
                    Object o = e.getValue();
                    outDoc = (Document)o;

                    //Save the result PDF file
                    File myOutFile = new File("C:\\Adobe\AssemblerResultPortfolio.pdf");
                    outDoc.copyToFile(myOutFile);
                }
            }

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Assembling multiple XDP fragments using the Java API

The following Java code example assembles XDP fragments that are based on the following XDP files: *tuc018_template_flowed.xdp*, *tuc018_contact.xdp*, and *tuc018_patient.xdp*. The assembled XDP document that contains all fragments is saved as a XDP file named *AssemblerResultXDP.xdp*. (See "Assembling Multiple XDP Fragments" on page 983.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-assembler-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
```

```
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * The following XML represents the DDX document used in this quick start:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *    <XDP result="tuc018result.xdp">
    *       <XDP source="tuc018_template_flowed.xdp">
    *      <XDPContent insertionPoint="ddx_fragment" source="tuc018_contact.xdp"
fragment="subPatientContact" required="false"/>
    *          <XDPContent insertionPoint="ddx_fragment" source="tuc018_patient.xdp"
fragment="subPatientPhysical" required="false"/>
    *          <XDPContent insertionPoint="ddx_fragment" source="tuc018_patient.xdp"
fragment="subPatientHealth" required="false"/>
    *       </XDP>
    *    </XDP>
    * </DDX>
    */
import com.adobe.livecycle.assembler.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AssembleFragmentsSOAP
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
```

```
connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an AssemblerServiceClient object
            AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

            //Create a FileInputStream object based on an existing DDX file
            FileInputStream myDDXFile = new FileInputStream("C:\\Adobe\fragmentDDX.xml");

            //Create a Document object based on the DDX file
            Document myDDX = new Document(myDDXFile);

            //Create a Map object to store the input XDP files
            Map inputs = new HashMap();
            FileInputStream inSource = new
FileInputStream("C:\\Adobe\tuc018_template_flowed.xdp");
            FileInputStream inFragment1 = new FileInputStream("C:\\Adobe\tuc018_contact.xdp");
            FileInputStream inFragment2 = new FileInputStream("C:\\Adobe\tuc018_patient.xdp");

            //Create a Document object
            Document myMapSource = new Document(inSource);

            //Create a Document object
            Document inFragment1Doc = new Document(inFragment1);

            //Create a Document object
            Document inFragment2Doc = new Document(inFragment2);

            //Place all of the XDP files into the MAP
            inputs.put("tuc018_template_flowed.xdp",myMapSource);
            inputs.put("tuc018_contact.xdp",inFragment1Doc);
            inputs.put("tuc018_patient.xdp",inFragment2Doc);


            //Create an AssemblerOptionsSpec object
            AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
            assemblerSpec.setFailOnError(false);

            //Submit the job to Assembler service
            AssemblerResult jobResult = assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
            java.util.Map allDocs = jobResult.getDocuments();

            //Retrieve the result PDF document from the Map object
            Document outDoc = null;

            //Iterate through the map object to retrieve the result XDP document
            for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                // Retrieve the Map object's value
```

```
                Map.Entry e = (Map.Entry)i.next();

                //Get the key name as specified in the
                //DDX document
                String keyName = (String)e.getKey();
                if (keyName.equalsIgnoreCase("tuc018result.xdp"))
                {
                    Object o = e.getValue();
                    outDoc = (Document)o;

                    //Save the result XDP file
                    File myOutFile = new File("C:\\AssemblerResultXDP.xdp");
                    outDoc.copyToFile(myOutFile);
                }
            }
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Redacting a PDF document using the Java API

The following code example redacts a PDF document using `PDFUtility`.

Note: *`PDFUtility` can redact only those PDFs which are marked for redaction using Acrobat.*

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-pdfutility-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. adobe-utilities.jar
 * 5. jboss-client.jar (use a different JAR file if AEM Forms is not deployed
 * on JBoss)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * The adobe-utilities.jar file is located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * The jboss-client.jar file is located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * If you want to invoke a remote AEM Forms instance and there is a
 * firewall between the client application and AEM Forms, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include additional JAR files located in the following
 * path
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * For information about the SOAP
 * mode and the additional JAR files that need to be included,
 * see "Setting connection properties" in Programming
 * with AEM Forms
```

```
 *
 * For complete details about the location of the AEM Forms JAR files,
 * see "Including AEM Forms library files" in Programming
 * with AEM Forms
 */

import java.util.*;
import com.adobe.livecycle.pdfutility.client.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RedactPDF
{
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a PDF Utility client
            PDFUtilityServiceClient pdfUt = new PDFUtilityServiceClient(myFactory);
```

```
            // Specify a PDF document to Redact
         FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\\RedactMarked.pdf");
            Document inDoc = new Document(fileInputStream);
            RedactionOptionSpec spec = new RedactionOptionSpec();

            // Convert the PDF document to redact
            RedactionResult redRes = pdfUt.redact(inDoc,spec);

            Document redactPDF = redRes.getDocument();

            //Save the returned Document object as an XDP file
            File redactedFile = new File("C:\\Adobe\\Redacted.pdf");
            redactPDF.copyToFile(redactedFile);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Backup and Restore Service API Quick Starts

Java API Quick Start(SOAP) are available for the Backup and Restore Service API.

"Quick Start (SOAP mode): Entering backup mode using the Java API" on page 59

"Quick Start (SOAP mode): Leaving backup mode using the Java API" on page 62

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

> ***Note:*** *Quick Starts located in Programming with AEM Forms are based on the Forms operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.*

Quick Start (SOAP mode): Entering backup mode using the Java API

The following Java code example enters into backup mode with a unique label for two hours. After the backup time expires or if backup mode is explicitly exited, the forms server returns to purging files from the Global Document Storage. (See "Entering Backup Mode on the forms server" on page 762.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-backup-restore-client-sdk.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;

import com.adobe.idp.backup.dsc.client.BackupServiceClient;
import com.adobe.idp.backup.dsc.service.BackupModeEntryResult;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class BackupRestoreEnter
```

```
{
    public static void main(String[] args)
    {
        try
        {
            // Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,"administra
tor");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a BackupService client object
            BackupServiceClient backup = new BackupServiceClient(myFactory);

            // Specify a generic label, 120 minutes to perform the backup,
            // and not to provide continous backup mode coverage (used for snapshot backups)
            String backUpLabel = new String("Snapshot2008July01");
            int minsInBackupMode = 120;
            boolean continousCoverage = false;


            // Enter backup mode on the forms server server
            BackupModeEntryResult backupResult =
backup.enterBackupMode(backUpLabel,minsInBackupMode, continousCoverage);

            // Get information from entering backup mode on the the forms server server.
            if (backupResult != null)
            {
                System.out.println("Start time is: " + backupResult.getStartTime());
                System.out.println("Backup Current ID is: " + backupResult.getId());
```

```
                System.out.println("Backup Previous ID is: " +
backupResult.getPreviousReservationId());
                System.out.println("Backup Label is: " + backupResult.getLabel());
                System.out.println("Backup Time to complete is: " +
backupResult.getReservationTimeout());
            }
            else
            {
                System.out.println("Could not enter backup mode.");
            }

        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return;
    }
}
```

Quick Start (SOAP mode): Leaving backup mode using the Java API

The following Java code example explicitly causes a Forms Server to leave backup mode and return to purging files from the Global Document Storage. (See "Leaving Backup Mode on the forms server" on page 765.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-backup-restore-client-sdk.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;

import com.adobe.idp.backup.dsc.client.BackupServiceClient;
import com.adobe.idp.backup.dsc.service.BackupModeResult;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class BackupRestoreLeave
{

    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[host]");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,"administra
tor");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a BackupService object
            BackupServiceClient backup = new BackupServiceClient(myFactory);

            // Leave backup mode on the forms server
```

```
            BackupModeResult leaveBackupResult = backup.leaveBackupMode();

            //Get result information from leaving backup mode
            if (leaveBackupResult != null)
            {
                System.out.println("Backup Mode ID is : " + leaveBackupResult.getId());
            }
            else
            {
                System.out.println("Forms server is not in backup mode.");
            }

        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return;
    }
}
```

Barcoded Forms Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Barcoded Forms service:

"Quick Start (SOAP mode): Decoding barcoded form data using the Java API" on page 64

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM Forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Decoding barcoded form data using the Java API

The following Java code decodes form data that is located in a PDF form that is saved as Loan.pdf. The decoded data is saved as an XML file named extractedData.xml. This code example converts a `org.w3c.dom.Document` object into a `com.adobe.idp.Document` object. (See "Decoding Barcoded Form Data" on page 746.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-barcodedforms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.*;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import com.adobe.livecycle.barcodedforms.CharSet;
import com.adobe.livecycle.barcodedforms.Delimiter ;
import com.adobe.livecycle.barcodedforms.XMLFormat ;
```

```
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.barcodedforms.client.*;

public class DecodeFormDataSOAP {

    public static void main(String[] args) {

    try
        {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        BarcodedFormsServiceClient barClient = new BarcodedFormsServiceClient(myFactory);

        //Specify a PDF document to convert to a XDP file
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanBarForms.pdf");
        Document inDoc = new Document (fileInputStream);

        java.lang.Boolean myFalse = new java.lang.Boolean(false);
        java.lang.Boolean myTrue = new java.lang.Boolean(true);

        //Decode barcoded form data
        org.w3c.dom.Document decodeXML = barClient.decode(
            inDoc,
            myTrue,
            myFalse,
            myFalse,
            myFalse,
            myFalse,
            myFalse,
            myFalse,
            myFalse,
            CharSet.UTF_8);

        //Convert the decoded data to XDP data
        List extractedData = barClient.extractToXML(
            decodeXML,
            Delimiter.Carriage_Return,
            Delimiter.Tab,
            XMLFormat.XDP);

        //Create an Iterator object and iterate through
        //the List object
```

```
    Iterator iter = extractedData.iterator();
    int i = 0 ;

    while (iter.hasNext()) {

        //Get the org.w3c.dom.Document object in each element
        org.w3c.dom.Document myDom = (org.w3c.dom.Document)iter.next();

        //Convert the org.w3c.dom.Document object to a
        //com.adobe.idp.Document object
        com.adobe.idp.Document myDocument = convertDOM(decodeXML);

        //Save the XML data to extractedData.xml
        File myFile = new File("C:\\Adobe\extractedData"+i+".xml");

        myDocument.copyToFile(myFile);
        i++;
        }
    }
catch(Exception e)
    {
        e.printStackTrace();
    }
}


//This user-defined method converts an org.w3c.dom.Document to a
//com.adobe.idp.Document object
public static com.adobe.idp.Document convertDOM(org.w3c.dom.Document doc)
    {

        byte[] mybytes = null ;
    com.adobe.idp.Document myDocument = null;
    try
        {

        //Create a Java Transformer object
    TransformerFactory transFact = TransformerFactory.newInstance();
    Transformer transForm = transFact.newTransformer();

    //Create a Java ByteArrayOutputStream object
    ByteArrayOutputStream myOutStream = new ByteArrayOutputStream();

    //Create a Java Source object
    Source myInput = new DOMSource(doc);

    //Create a Java Result object
    Result myOutput = new StreamResult(myOutStream);

    //Populate the Java ByteArrayOutputStream object
    transForm.transform(myInput,myOutput);

    //Get the size of the ByteArrayOutputStream buffer
```

```
        int myByteSize = myOutStream.size();

        //Allocate myByteSize to the byte array
        mybytes = new byte[myByteSize];

        //Copy the content to the byte array
        mybytes = myOutStream.toByteArray();
        com.adobe.idp.Document myDoc = new com.adobe.idp.Document(mybytes);

        myDocument = myDoc ;
        }

        catch(Exception ee)
        {
            ee.printStackTrace();
        }

    return myDocument;
      }
}
```

Note: *When using both an* `org.w3c.dom.Document` *object and a* `com.adobe.idp.Document` *object in the same application logic, it is good practice to fully qualify both objects.*

Components and Services Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for components and services.

"Quick Start (SOAP mode): Deploying a component using the Java API" on page 68

"Quick Start (SOAP mode): Setting the execution context of a service using the Java API" on page 70

"Quick Start (SOAP mode): Disabling service security using the Java API" on page 72

"Quick Start (SOAP mode): Starting a service using the Java API" on page 75

"Quick Start (SOAP mode): Modifying a services configuration values using the Java API" on page 76

"Quick Start (SOAP mode): Removing components using the Java API" on page 78

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *You cannot programmatically manipulate components and services by using web services.*

Note: *Quick starts located in Programming with AEM forms are based on the Forms server being deployed on JBoss and the Windows operating system. However, if you are using another operating system, such as Unix, replace windows specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Deploying a component using the Java API

The following Java example deploys a component that is based on a JAR file named *adobe-emailSample-dsc.jar*.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
        * 5. axis.jar (required for SOAP mode)
        * 6. commons-codec-1.3.jar (required for SOAP mode)
        * 7. commons-collections-3.2.jar  (required for SOAP mode)
        * 8. commons-discovery.jar (required for SOAP mode)
        * 9. commons-logging.jar (required for SOAP mode)
        * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
        * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
        * 12. jaxrpc.jar (required for SOAP mode)
        * 13. log4j.jar (required for SOAP mode)
        * 14. mail.jar (required for SOAP mode)
        * 15. saaj.jar (required for SOAP mode)
        * 16. wsdl4j.jar (required for SOAP mode)
        * 17. xalan.jar (required for SOAP mode)
        * 18. xbean.jar (required for SOAP mode)
        * 19. xercesImpl.jar (required for SOAP mode)
*
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.component.client.*;
import com.adobe.idp.dsc.registry.infomodel.Component;

public class DeployComponents {
```

```
    public static void main(String[] args) {

        try{

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ComponentRegistryClient object
            ComponentRegistryClientcomponentReg = new ComponentRegistryClient(myFactory);

            //Reference a JAR file that represents the component to deploy
        //  FileInputStream componentFile = new FileInputStream("C:\\Adobe\adobe-emailSample-
dsc.jar");

            FileInputStream componentFile = new FileInputStream("C:\\A22\Bank.jar");
            Document component = new Document(componentFile);

            //Install the component
            Component myComponent = componentReg.install(component);
            componentReg.start(myComponent);

            System.out.println("The component has been deployed");
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Setting the execution context of a service using the Java API

The following Java code example sets the Run-As Invoker execution context to an example service named *EncryptDocument*.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
        * 5. axis.jar (required for SOAP mode)
        * 6. commons-codec-1.3.jar (required for SOAP mode)
        * 7. commons-collections-3.2.jar  (required for SOAP mode)
        * 8. commons-discovery.jar (required for SOAP mode)
        * 9. commons-logging.jar (required for SOAP mode)
        * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
        * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
        * 12. jaxrpc.jar (required for SOAP mode)
        * 13. log4j.jar (required for SOAP mode)
        * 14. mail.jar (required for SOAP mode)
        * 15. saaj.jar (required for SOAP mode)
        * 16. wsdl4j.jar (required for SOAP mode)
        * 17. xalan.jar (required for SOAP mode)
        * 18. xbean.jar (required for SOAP mode)
        * 19. xercesImpl.jar (required for SOAP mode)
    *  The JBoss files must be kept in the jboss\bin\client folder. You can copy the client
folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.infomodel.ServiceConfiguration;
import com.adobe.idp.dsc.registry.service.ModifyServiceConfigurationInfo;
import com.adobe.idp.dsc.registry.service.client.ServiceRegistryClient;

/*
    * This Java quick start sets the Run-As Invoker to a service named EncryptDocument
    */
public class SetRunAsConfiguration {

     public static void main(String[] args) {
```

```
    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"tblue");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceRegistryClient object
        ServiceClientFactory _factory = ServiceClientFactory.createInstance(connectionProps);
        ServiceRegistryClient _src = new ServiceRegistryClient(_factory);

        //Reference the EncryptDocument service
        ServiceConfiguration _config  = _src.getHeadActiveConfiguration("EncryptDocument");

        //Set the RUN_AS_INVOKER execution context
        ModifyServiceConfigurationInfo _configModifyInfo = new
ModifyServiceConfigurationInfo();
        _configModifyInfo.setServiceId(_config.getServiceId());
        _configModifyInfo.setMajorVersion(_config.getMajorVersion());
        _configModifyInfo.setMinorVersion(_config.getMinorVersion());
        _configModifyInfo.setRunAsConfiguration(ServiceConfiguration.RUN_AS_INVOKER);
        _config = _src.modifyConfiguration(_configModifyInfo);
    }catch (Exception e) {
         e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Disabling service security using the Java API

The following Java code example disables security from the example EncryptDocument service and the services that are invoked from within this service (the Set Value and Encryption services).

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
        * 5. axis.jar (required for SOAP mode)
        * 6. commons-codec-1.3.jar (required for SOAP mode)
        * 7. commons-collections-3.2.jar  (required for SOAP mode)
        * 8. commons-discovery.jar (required for SOAP mode)
        * 9. commons-logging.jar (required for SOAP mode)
        * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
        * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
        * 12. jaxrpc.jar (required for SOAP mode)
        * 13. log4j.jar (required for SOAP mode)
        * 14. mail.jar (required for SOAP mode)
        * 15. saaj.jar (required for SOAP mode)
        * 16. wsdl4j.jar (required for SOAP mode)
        * 17. xalan.jar (required for SOAP mode)
        * 18. xbean.jar (required for SOAP mode)
        * 19. xercesImpl.jar (required for SOAP mode)
*
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.infomodel.ServiceConfiguration;
import com.adobe.idp.dsc.registry.service.ModifyServiceInfo;
import com.adobe.idp.dsc.registry.service.client.ServiceRegistryClient;

/*
    * This Java quick start disables security from the EncryptDocument process
```

```
    * and each service that is located in this process
    */
public class DisableSecurity{

     public static void main(String[] args) {

    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");


        //Create a ServiceRegistryClient object
        ServiceClientFactory _factory = ServiceClientFactory.createInstance(connectionProps);
        ServiceRegistryClient _src = new ServiceRegistryClient(_factory);

        //Reference the EncryptDocument process and each service that is
        //invoked from within the EncryptDocument process
        ServiceConfiguration encryptDocumentService  =
_src.getHeadActiveConfiguration("EncryptDocument");
        ServiceConfiguration setValueService  = _src.getHeadActiveConfiguration("SetValue");
        ServiceConfiguration encryptionService  =
_src.getHeadActiveConfiguration("EncryptionService");

        //Create a ModifyServiceInfo object
        ModifyServiceInfo si = new ModifyServiceInfo();

        //Disable security from the EncryptDocument service
        si.setId(encryptDocumentService.getServiceId());
        si.setSecurityEnabled(false);
        _src.modifyService(si);

        //Disable security from the SetValue service
        si.setId(setValueService.getServiceId());
        si.setSecurityEnabled(false);
        _src.modifyService(si);

        //Disable security from the EncryptionService
        si.setId(encryptionService.getServiceId());
        si.setSecurityEnabled(false);
        _src.modifyService(si);

    }catch (Exception e) {
         e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Starting a service using the Java API

The following Java code example starts a service named *SendEmailService*.

```
package com.adobe.sample.servicemanager;

/**
    * This Java Quick Start uses the following JAR files:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. adobe-workflow-client-sdk.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if AEM Forms is not deployed on Jboss)
    * 6. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 7. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.registry.infomodel.ServiceConfiguration;
import com.adobe.idp.dsc.registry.service.client.ServiceRegistryClient;

public class StartService {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties ConnectionProps = new Properties();
            ConnectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
            ConnectionProps.setProperty("DSC_TRANSPORT_PROTOCOL","SOAP");
            ConnectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
            ConnectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
            ConnectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(ConnectionProps);

            //Create a ServiceRegistryClient object
            ServiceRegistryClient serviceReg = new ServiceRegistryClient(myFactory);

            //Reference the SendEmailService
            ServiceConfiguration myServiceConfig =
serviceReg.getHeadActiveConfiguration("SendEmailService");

            //Start the SendEmailService
            serviceReg.start(myServiceConfig);
            }

                catch(Exception e)
                {
                    e.printStackTrace();
                }
    }
}
```

Quick Start (SOAP mode): Modifying a services configuration values using the Java API

The following Java example modifies configuration values that belong to SendEmail Service. For information about creating the sample email component, see Creating Your First Component.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
        * 5. axis.jar (required for SOAP mode)
        * 6. commons-codec-1.3.jar (required for SOAP mode)
        * 7. commons-collections-3.2.jar  (required for SOAP mode)
        * 8. commons-discovery.jar (required for SOAP mode)
        * 9. commons-logging.jar (required for SOAP mode)
        * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
        * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
        * 12. jaxrpc.jar (required for SOAP mode)
        * 13. log4j.jar (required for SOAP mode)
        * 14. mail.jar (required for SOAP mode)
        * 15. saaj.jar (required for SOAP mode)
        * 16. wsdl4j.jar (required for SOAP mode)
        * 17. xalan.jar (required for SOAP mode)
        * 18. xbean.jar (required for SOAP mode)
        * 19. xercesImpl.jar (required for SOAP mode)
*
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.registry.infomodel.ConfigParameter;
import com.adobe.idp.dsc.registry.infomodel.ServiceConfiguration;
import com.adobe.idp.dsc.registry.service.ModifyServiceConfigurationInfo;
```

```
import com.adobe.idp.dsc.registry.service.client.ServiceRegistryClient;


public class ModifyService {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties ConnectionProps = new Properties();
            ConnectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT", "http://[server]:[port]");
            ConnectionProps.setProperty("DSC_TRANSPORT_PROTOCOL","SOAP");
            ConnectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
            ConnectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
            ConnectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(ConnectionProps);

            //Create a ServiceRegistryClient object
            ServiceRegistryClient serviceReg = new ServiceRegistryClient(myFactory);

            //Reference the SendEmailService
            ServiceConfiguration myServiceConfig =
serviceReg.getHeadServiceConfiguration("SendEmailService");

            //Create a ModifyServiceConfigurationInfo object
                ModifyServiceConfigurationInfo modService = new
ModifyServiceConfigurationInfo();

            //Set configuration values required by the SendEmailService
            String serviceId = myServiceConfig.getServiceId();
            modService.setServiceId(serviceId);
            modService.setMajorVersion(1);
            modService.setConfigParameterAsText("smtpHost","mySMTPSERVER");
            modService.setConfigParameterAsText("smtpUser","smyUserName");
            modService.setConfigParameterAsText("smtpPassword","myPassword");
```

```
            //Modify the service's configuration values
            serviceReg.modifyConfiguration(modService);

            //Conform the new configuration values
            ServiceConfiguration serviceConfig =
serviceReg.getServiceConfiguration("SendEmailService",1,0);
            ConfigParameter cp = serviceConfig.getConfigParameter("smtpUser");
            String configValue = cp.getTextValue();
            System.out.println(configValue);
            }

            catch(Exception e)
            {
                e.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Removing components using the Java API

The following Java code example removes a component by using the Java API.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. adobe-workflow-client-sdk.jar
    * 8. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 9. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
```

```
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.component.client.*;
import com.adobe.idp.dsc.registry.infomodel.Component;

public class RemoveComponent {

    public static void main(String[] args) {

        try{
             //Set connection properties required to invoke AEM Forms
              Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
              connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ComponentRegistryClient object
            ComponentRegistryClientcomponentReg = new ComponentRegistryClient(myFactory);

            //Retrieve the Id of the component to remove from the service container
            Component myComponent =
componentReg.getComponent("com.adobe.livecycle.sample.email.emailSampleComponent", "1.0");

            //Determine if the component is in a running state
            if (myComponent.getState()== Component.RUNNING)
            {
                //Stop the component
```

```
                Component stoppedComponent = componentReg.stop(myComponent);

                //Uninstall the component
                componentReg.uninstall(stoppedComponent);
            }
            else
                componentReg.uninstall(myComponent);

            System.out.println("The component was removed.");
            }

            catch(Exception e)
            {
                e.printStackTrace();
            }
    }
}
```

Convert PDF Service Java API Quick Start(SOAP)

The following Quick Starts are available for the Convert PDF service API.

"Quick Start (SOAP mode): Converting a PDF document to PostScript using the Java API" on page 80

"Quick Start (SOAP mode): Converting a PDF document to JPEG files using the Java API" on page 83

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Start located in Programming with AEM forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Converting a PDF document to PostScript using the Java API

The following code example converts a PDF document called *Loan.pdf* to a PostScript document called *Loan.ps*. (See "Converting PDF Documents to PostScript" on page 771.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-convertpdf-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7.  commons-collections-3.1.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.convertpdfservice.client.ConvertPdfServiceClient;
import com.adobe.livecycle.convertpdfservice.client.ToPSOptionsSpec;
import com.adobe.livecycle.convertpdfservice.client.enumeration.Color;
import com.adobe.livecycle.convertpdfservice.client.enumeration.LineWeight;
import com.adobe.livecycle.convertpdfservice.client.enumeration.PSLevel;
import com.adobe.livecycle.convertpdfservice.client.enumeration.PageSize;
import com.adobe.livecycle.convertpdfservice.client.enumeration.Color;
```

```
public class JavaAPIConvertPDFtoPSSOAP
{
    public static void main(String[] args)
    {
    try
        {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a ConvertPdfServiceClient object
        ConvertPdfServiceClient convertPDFClient= new ConvertPdfServiceClient(myFactory);

        //Get a PDF file document to convert to a PS document
        //and populate a com.adobe.idp.Document object
        String inputFileName = "C:\\Adobe\Loan.pdf";
        FileInputStream fileInputStream = new FileInputStream(inputFileName);
        Document inDoc = new Document(fileInputStream);

        //Create a ToPSOptionsSpec object that defines run-time options
        ToPSOptionsSpec psSpec = new ToPSOptionsSpec();
        psSpec.setPsLevel(PSLevel.LEVEL_3);
        psSpec.setShrinkToFit(true);
        psSpec.setPageSize(PageSize.A4);
        psSpec.setRotateAndCenter(true);
        psSpec.setColor(Color.compositeGray);
        psSpec.setLineWeight(LineWeight.point25);

        //Convert the PDF document to a PostScript file
        Document createdDocument =convertPDFClient.toPS2(
            inDoc,
            psSpec
            );

        //Save the PostScript file
        createdDocument.copyToFile(new File("C:\\Adobe\Loan.ps"));
        }
    catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Converting a PDF document to JPEG files using the Java API

The following Java code example converts a PDF document called *Loan.pdf* to a set of JPEG files and stores them in the C:\Adobe directory. Each file is named *tempFile[index].jpg*, where the first image file is named *tempFile0.jpg*. (See "Converting PDF Documents to Image Formats" on page 775.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-convertpdf-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7.  commons-collections-3.1.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
```

```
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.convertpdfservice.client.ConvertPdfServiceClient;
import com.adobe.livecycle.convertpdfservice.client.ToImageOptionsSpec;
import com.adobe.livecycle.convertpdfservice.client.enumeration.CMYKPolicy;
import com.adobe.livecycle.convertpdfservice.client.enumeration.ColorCompression;
import com.adobe.livecycle.convertpdfservice.client.enumeration.ColorSpace;
import com.adobe.livecycle.convertpdfservice.client.enumeration.GrayScaleCompression;
import com.adobe.livecycle.convertpdfservice.client.enumeration.GrayScalePolicy;
import com.adobe.livecycle.convertpdfservice.client.enumeration.ImageConvertFormat;
import com.adobe.livecycle.convertpdfservice.client.enumeration.Interlace;
import com.adobe.livecycle.convertpdfservice.client.enumeration.JPEGFormat;
import com.adobe.livecycle.convertpdfservice.client.enumeration.MonochromeCompression;
import com.adobe.livecycle.convertpdfservice.client.enumeration.PNGFilter;
import com.adobe.livecycle.convertpdfservice.client.enumeration.RGBPolicy;

public class JavaAPIConvertPDFtoImageSOAP {

    public static void main(String[] args)
    {
    try
    {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create the ConvertPDF service client
        ConvertPdfServiceClient serviceClient = new ConvertPdfServiceClient(myFactory);

        //Get a PDF file document to convert to a JPEG document and populate a
com.adobe.idp.Document object
        String inputFileName = "C:\\Adobe\Loan.pdf";
        FileInputStream fileInputStream = new FileInputStream(inputFileName);
        Document inDoc = new Document(fileInputStream);

        // Set up the runtime options for the new JPEG file to be created
        ToImageOptionsSpec spec = new ToImageOptionsSpec();
        spec.setImageConvertFormat(ImageConvertFormat.JPEG);
        spec.setGrayScaleCompression(GrayScaleCompression.Low);
        spec.setColorCompression(ColorCompression.Low);
        spec.setFormat(JPEGFormat.BaselineOptimized);
        spec.setRgbPolicy(RGBPolicy.Off);
        spec.setCmykPolicy(CMYKPolicy.Off);
        spec.setColorSpace(ColorSpace.RGB);
```

```
        spec.setResolution("72");
        spec.setMonochrome(MonochromeCompression.None);
        spec.setFilter(PNGFilter.Sub);
        spec.setInterlace(Interlace.Adam7);
        spec.setTileSize(180);
        spec.setGrayScalePolicy(GrayScalePolicy.Off);

        //Perform the conversion and get the containing the newly created JPEG files
        List allImages = serviceClient.toImage2(
            inDoc,
            spec
        );

        //Create an Iterator object and iterate through
        //the List object to get all images
        Iterator iter = allImages.iterator();
        int i = 0 ;
        while (iter.hasNext()) {
            Document file = (Document)iter.next();
            file.copyToFile(new File("C:\\Adobe\tempFile"+i+".jpg"));
            i++;
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        }
    }
}
```

Credential Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Credential service.

"Quick Start (SOAP mode): Importing credentials using the Java API" on page 85

"Quick Start (SOAP mode): Deleting credentials using the Java API" on page 87

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick starts located in Programming with AEM forms are based on the FormsServer being deployed on JBoss and the Windows operating system. However, if you are using another operating system, such as Unix, replace Windows-specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Note: *You cannot perform Credential service operations using web services.*

Quick Start (SOAP mode): Importing credentials using the Java API

The following code example imports a credential based on a file named *cred.p12*. The alias value used to import the credential is `Secure`. (See "Importing Credentials by using the Trust Manager API" on page 1060.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-truststore-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.truststore.client.CredentialServiceClient;

public class ImportCredentialSoap {

    public static void main(String[] args) {
```

```
            try {

         //Set connection properties required to invoke AEM Forms using SOAP mode
             Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
             connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
             connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
             connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

             //Create a ServiceClientFactory object
             ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

             //Create a CredentialServiceClient instance
             CredentialServiceClient certClient = new CredentialServiceClient(myFactory);

             //Reference a credential based on a P12 file
             FileInputStream myCred = new FileInputStream("C:\\Adobe\cred.p12");
             Document credential = new Document(myCred);

             //Create a string array to store usage values
             String[] usage = new String[1];
             usage[0] = "truststore.usage.type.sign";

             //Import the credential
             certClient.importCredential("secure",credential,"password",usage);
             System.out.println("Credential was uploaded");

            }catch (Exception e) {
                e.printStackTrace();
            }

            }
}
```

Quick Start (SOAP mode): Deleting credentials using the Java API

The following code example deletes a credential based on an alias value *secure*. (See "Deleting Credentials by using the Trust Manager API" on page 1063.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-truststore-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.truststore.client.CredentialServiceClient;

public class DeleteCertificateSoap {


    public static void main(String[] args)
```

```
    {
    try {

        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a CredentialServiceClient instance
        CredentialServiceClient certClient = new CredentialServiceClient(myFactory);

        //Delete the certificate
        certClient.deleteCredential("secure");
        System.out.println("Credential was deleted");

       }catch (Exception e) {
           e.printStackTrace();
       }

       }

}
```

Distiller Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Distiller® service:

"Quick Start (SOAP mode): Converting a PostScript file to a PDF document using the Java API" on page 90

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Converting a PostScript file to a PDF document using the Java API

The following code example converts a PostScript file called *Loan.ps* to a PDF file called *Loan.pdf*. (See "Converting PostScript to PDF documents" on page 767.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-distiller-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
```

```
import com.adobe.livecycle.generatepdf.client.CreatePDFResult;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.distiller.client.DistillerServiceClient;

public class JavaAPICreatePDFSoap {

    public static void main(String[] args)
    {
        try
        {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        // Create a ServiceClientFactory instance
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        DistillerServiceClient disClient = new DistillerServiceClient(factory );

        // Get a PS file document to convert to a PDF document and populate a
com.adobe.idp.Document object
        String inputFileName = "C:\\Adobe\Loan.ps";
        FileInputStream fileInputStream = new FileInputStream(inputFileName);
        Document inDoc = new Document(fileInputStream);

        //Set run-time options
        String adobePDFSettings = "Standard";
        String securitySettings = "No Security";

        //Convert a PS  file into a PDF file
        CreatePDFResult result = new CreatePDFResult();
```

```
        result = disClient.createPDF(
                inDoc,
                inputFileName,
                    adobePDFSettings,
                securitySettings,
                null,
                null
            );

        //Get the newly created document
        Document createdDocument = result.getCreatedDocument();

        //Save the PDF file
        createdDocument.copyToFile(new File("C:\\Adobe\Loan.pdf"));
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

DocConverter Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the DocConverter service.

"Quick Start (SOAP mode): Determining PDF/A compliancy using the Java API" on page 94

"Quick Start (SOAP mode): Converting a document to a PDF/A document using the Java API" on page 92

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Converting a document to a PDF/A document using the Java API

The following Java code example converts a PDF document named *Loan.pdf* to a PDF/A document that is saved as a PDF file named *LoanArchive.pdf*. (See "Converting Documents to PDF/A Documents" on page 990.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-docconverter-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.docconverter.client.DocConverterServiceClient;
import com.adobe.livecycle.docconverter.client.PDFAConversionOptionSpec;
import com.adobe.livecycle.docconverter.client.PDFAConversionResult;
```

```
public class CreatePDFADocumentSOAP {

    public static void main(String[] args) {
    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a DocConverterServiceClient object
        DocConverterServiceClient docConverter = new DocConverterServiceClient(myFactory);

        //Reference a PDF document to convert to a PDF/A document
        FileInputStream myPDF = new FileInputStream("C:\\Adobe\Loan.pdf");
        Document inDoc = new Document(myPDF);

        //Create a PDFAConversionOptionSpec object and set
        //tracking information
        PDFAConversionOptionSpec spec = new PDFAConversionOptionSpec();
        spec.setLogLevel("FINE");

        //Convert the PDF document to a PDF/A document
        PDFAConversionResult result =  docConverter.toPDFA(inDoc,spec);

        //Save the PDF/A file
        Document pdfADoc= result.getPDFADocument();
        File pdfAFile = new File("C:\\Adobe\LoanArchive.pdf");
        pdfADoc.copyToFile(pdfAFile);
      }catch (Exception e) {
        e.printStackTrace();
    }
     }
}
```

Quick Start (SOAP mode): Determining PDF/A compliancy using the Java API

The following Java code example determines whether the input PDF document is PDF/A-compliant. The input PDF document that is passed to the DocConverter service is named *LoanArchive.pdf*. Validation results are written to an XML file named *ValidationResults.xml*. (See "Programmatically Determining PDF/A Compliancy" on page 994.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-docconverter-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.docconverter.client.DocConverterServiceClient;
import com.adobe.livecycle.docconverter.client.PDFAValidationOptionSpec;
import com.adobe.livecycle.docconverter.client.PDFAValidationResult;
```

```
public class IsDocumentPDFASOAP {

    public static void main(String[] args) {
    try{
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a DocConverterServiceClient object
        DocConverterServiceClient docConverter = new DocConverterServiceClient(myFactory);

        //Reference a PDF document used to determine PDF/A compliancy
        FileInputStream myPDF = new FileInputStream("C:\\Adobe\LoanArchive.pdf");
        Document inDoc = new Document(myPDF);

        //Create a PDFAValidationOptionSpec object and set
        //run-time values
        PDFAValidationOptionSpec spec = new PDFAValidationOptionSpec();
        spec.setCompliance(PDFAValidationOptionSpec.Compliance.PDFA_1B);
        spec.setResultLevel(PDFAValidationOptionSpec.ResultLevel.DETAILED);
        spec.setLogLevel("FINE");
        spec.setIgnoreUnusedResource(true);

        //Determine if the PDF document is PDF/A compliant
        PDFAValidationResult result =  docConverter.isPDFA(inDoc,spec);

        //Get the results of the operation
        Boolean isPDFA = result.getIsPDFA();

        //Get XML data that contains validaction results
        Document validationResults =  result.getValidationLog();
        File file= new File("C:\\Adobe\ValidationResults.xml");
        validationResults .copyToFile(file);

    }catch (Exception e) {
        e.printStackTrace();
    }
     }
}
```

Document Management Service (Deprecated) Java API Quick Start(SOAP)

The following Quick Starts are available for the Document Management service (Deprecated).

Note: *Effective August 5th 2011, Adobe is migrating Content Services ES customers to the Adobe Digital Enterprise Platform Experience Services. The product roadmap for customers that use Content Services is to move to the new ADEP Experience Services - Core, which includes a native Content Repository built on the modern, modular CRX architecture, acquired during the Adobe acquisition of Day Software.*

"Quick Start (SOAP mode): Create Content Services spaces using the Java API (Deprecated)" on page 97

"Quick Start (SOAP mode): Delete Content Services content using the Java API (Deprecated)" on page 99

"Quick Start (SOAP mode): Add content to Content Services using the Java API (Deprecated)" on page 101

"Quick Start (SOAP mode): Retrieve content from Content Services using the Java API (Deprecated)" on page 104

"Quick Start (SOAP mode): Move Content Services content using the Java API (Deprecated)" on page 106

"Quick Start (SOAP mode): List Content Services content using the Java API (Deprecated)" on page 108

"Quick Start (SOAP mode): Search Content Services content using the Java API (Deprecated)" on page 110

"Quick Start (SOAP mode): Setting Content Services Permissions using the Java API (Deprecated)" on page 113

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick starts located in Programming with AEM forms are based on the Forms Server being deployed on JBoss and the Windows operating system. However, if you are using another operating system, such as UNIX, replace windows-specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Create Content Services spaces using the Java API (Deprecated)

The following Java code example creates a new space named *Test Directory* located in Company Home. The identification value of the new space is written to the console. (See Creating Content Services Spaces.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;

public class CreateNewSpaceSoap {

    public static void main(String[] args) {
```

```
        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the name of the store and node
            String storeName ="SpacesStore";
            String nodeName = "/Company Home/Test Directory" ;

            //Create a new space
            String spaceId = docManager.createSpace(storeName,nodeName);
            System.out.println("The identifier value of the new space is " +spaceId);
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Delete Content Services content using the Java API (Deprecated)

The following Java code example deletes a space named /Company Home/Test Directory. (See Creating Content Services Spaces.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;

public class DeleteContentSoap {

    public static void main(String[] args) {
```

```
        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the name of the store and node
            String storeName ="SpacesStore";
            String nodeName = "/Company Home/Test Directory" ;

            //Delete the content from /Company Home/Test Directory
            Boolean ans = docManager.deleteContent(storeName, nodeName);

            if (ans == true)
                System.out.println("The content was successfully deleted");
            else
                System.out.println("The content was not deleted");
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Add content to Content Services using the Java API (Deprecated)

The following Java code example adds a PDF file named *MortgageForm.pdf* to a folder named /Company Home/Test Directory. The creator and description attributes are set. The identification value of the new content is written to the console. (See Adding Content to Content Services.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.util.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.CRCResult;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;
import com.adobe.livecycle.contentservices.client.impl.UpdateVersionType;

public class AddContentSoap {
```

```
    public static void main(String[] args) {

        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the store and node name
            String storeName ="SpacesStore";
            String nodeName = "/Company Home/Test Directory" ;

            //Retrieve the document to store in /Company Home/Test Directory
            Document content =  new Document(new File("C:\\Adobe\MortgageForm.pdf"), false);

            //Create a MAP instance to store attributes
            Map<String,Object> inputs = new HashMap<String,Object>();

            //Specify attributes that belong to the new content
            String creator = "{http://www.alfresco.org/model/content/1.0}creator";
            String description = "{http://www.alfresco.org/model/content/1.0}description";

            inputs.put(creator,"Tony Blue");
            inputs.put(description,"A mortgage application form");

            //Store MortgageForm.pdf in /Company Home/Test Directory
            CRCResult result = docManager.storeContent(storeName,
                     nodeName,
```

```
                    "MortgageForm.pdf",
                    "{http://www.alfresco.org/model/content/1.0}content",
                    content,
                    "UTF-8",
                    UpdateVersionType.INCREMENT_MAJOR_VERSION,
                    null,
                    inputs);

            //Get the identifier value of the new content
            String id = result.getNodeUuid();
            System.out.println("The identifier value of the new content is "+id);
    }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieve content from Content Services using the Java API (Deprecated)

The following Java code example retrieves a PDF file named *MortgageForm.pdf* from /Company Home. The PDF file is saved to the local file system and is named *UpdatedMortgageForm.pdf*. (See Retrieving Content from Content Services.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.CRCResult;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;

public class RetrieveContentSoap {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the name of the store and the content to retrieve
```

```
                String storeName = "SpacesStore";
                String nodeName  = "/Company Home/MortgageForm.pdf";

                //Retrieve /Company Home/MortgageForm.pdf
                CRCResult content = docManager.retrieveContent(
                     storeName,
                     nodeName,
                     "");

                //Write the PDF file to the local file system
                File myFile = new File("C:\\Adobe\UpdatedMortgageForm.pdf");
                Document doc =content.getDocument();
                doc.copyToFile(myFile);
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Move Content Services content using the Java API (Deprecated)

The following Java code example moves a PDF file named *MortgageForm.pdf* from /Company Home/Test Directory to /Company Home. The identification value of the moved content is written to the console. (See Moving Content Services Content.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
```

```
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;

public class MoveContentSoap {

    public static void main(String[] args) {

        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);


            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);
```

```
            //Specify the name of the store and the content to move
               String storeName = "SpacesStore";
               String nodeName = "/Company Home/Test Directory/MortgageForm.pdf";
               String newSpace = "/Company Home";

               //Move the content from /Company Home/Test Directory
               //to /Company Home and display the identifier value of the
               //moved content
               String contentID = docManager.moveContent(storeName, nodeName, newSpace);
               System.out.println("The identifier value of the moved content is "+contentID);
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): List Content Services content using the Java API (Deprecated)

The following Java code example lists content that is located in /Company Home. Each node type and node name is displayed. (See Listing Content Services Content.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.CRCResult;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;

public class ListingContentSoap {

    public static void main(String[] args) {

        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the name of the store and the space
               String storeName = "SpacesStore";
```

```
            String nodeName  = "/Company Home";

            //List the contents of /Company Home
            List<CRCResult> allImages = docManager.getSpaceContents(
                  storeName,
                  nodeName,
                  false);

        //Create an Iterator object and iterate through
        //the List object
        Iterator iter = allImages.iterator();
        int i = 0 ;
        while (iter.hasNext()) {

            //Get the node content type and name
            CRCResult sinContent = (CRCResult)iter.next();
            String nodeType =  sinContent.getNodeType();
            String name = sinContent.getNodeName();
            System.out.println("The node type is "+nodeType  +". The node name is "+name);
        }
    }

    catch(Exception e)
    {
        e.printStackTrace();
    }
  }
}
```

Quick Start (SOAP mode): Search Content Services content using the Java API (Deprecated)

The following Java code searches /Company Home for a document that contains the text MortgageForm. The sub folders are also searched. (See Searching Content Services Content.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.ResultSet;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;
import com.adobe.livecycle.contentservices.client.impl.QueryImpl;
import com.adobe.livecycle.contentservices.client.impl.StatementImpl;

public class SearchSpaceSoap {
```

```
    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the name of the store and node
            String path ="/Company Home";
             String storeName = "SpacesStore";

            //Create a Query expression
            QueryImpl qImpl = new QueryImpl();
            String myName = "{http://www.alfresco.org/model/content/1.0}name";
            StatementImpl statement = new StatementImpl(myName, StatementImpl.OPERATOR_CONTAINS,
"MortgageForm" );
            qImpl.addStatement(statement);

            //Perform the search for a document that contains the text MortgageForm
            ResultSet rs = docManager.searchRepository(storeName, path, true, qImpl, 200);
            long resultSize = rs.getResultSize();

            //Determine if the document is located in Content space
            if (resultSize > 0)
            {
                System.out.println("MortgageForm is located in the Repository");
            }
        }
    catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Setting Content Services Permissions using the Java API (Deprecated)

The following Java code example sets a permission for a user named tony blue. The domain that is specified is the default domain. The Consumer permission is specified and the node is `/Company Home/Test Directory`. (See Setting Content Services Permissions.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;
```

```
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;
import com.adobe.livecycle.contentservices.client.impl.ContentAccessPermission;

public class SetPermissionsSoap {

    public static void main(String[] args) {

        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the store and node name
            String storeName ="SpacesStore";
            String nodeName = "/Company Home/Test Directory/";

             //Create a new permission
            ContentAccessPermission permission = new ContentAccessPermission();
            permission.setAuthority("tblue/DefaultDom");
            permission.setIsAllowed(false);
            permission.setPermission("Consumer");
```

```
            //Create a collection to hold the values
            List<ContentAccessPermission> permissionList = new
ArrayList<ContentAccessPermission>();
            permissionList.add(0,permission);

            //Set the permission
            docManager.writePermissions(storeName,
                     nodeName,
                     permissionList,
                    false);
    }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP Mode): Creating Associations using the Java API (Deprecated)

The following Java code creates an association an XML data file and a PDF form. This type of association is named LinkedBy.The PDF document must have the aspect linkable applied to it. (See Creating Content Services Associations.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-contentservices-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;

public class CreateAssociationsSoap {

    public static void main(String[] args) {

        try{

            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the input values
            String storeName ="SpacesStore";
            String associationType = "{http://www.adobe.com/lc/datacapture/1.0}linkedBy";
```

```
            String aspect = "{http://www.adobe.com/lc/datacapture/1.0}linkable";
            String parentPath= "/Company Home/MortgageForm.pdf";
            String childPath= "/Company Home/Loan.xml";

            //Set the linkable aspect to MortgageForm.pdf
            List<String> aspectList = new ArrayList();
            aspectList.add(aspect);

            //Create an attribute map
            Map<String,Object> inputs = new HashMap<String,Object>();

            //Specify attributes that belong to the new content
            String creator = "{http://www.alfresco.org/model/content/1.0}creator";
            String description = "{http://www.alfresco.org/model/content/1.0}description";

            inputs.put(creator,"Tony Blue");
            inputs.put(description,"Link the PDF document to loan data");

            //Set the aspects
            docManager.setContentAttributes(storeName,parentPath,aspectList,inputs);

            //Create an association between MortgageForm.pdf and Loan.xml
            docManager.createAssociation(storeName,
                    associationType,
                    parentPath,
                    childPath);
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
```

Encryption Service Java API Quick Start(SOAP)

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Encrypting a PDF document using the Java API

The following Java code example encrypts a PDF document named *Loan.pdf* with a password value of `OpenPassword`. The master password is `PermissionPassword`. The secured PDF document is saved as a PDF file named *EncryptLoan.pdf*. (See "Encrypting PDF Documents with a Password" on page 806.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
```

```
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.encryption.client.*;

public class PasswordEncryptPDFSoap{

    public static void main(String[] args) {

    try{
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an EncryptionServiceClient object
        EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);

        //Specify the PDF document to encrypt with a password
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
        Document inDoc = new Document (fileInputStream);

        //Create a PasswordEncryptionOptionSpec object that stores encryption run-time values
        PasswordEncryptionOptionSpec passSpec = new PasswordEncryptionOptionSpec();

        //Specify the PDF document resource to encrypt
        passSpec.setEncryptOption(PasswordEncryptionOption.ALL);

        //Specify the permission associated with the password
        //These permissions enable data to be extracted from a password
        //protected PDF form
        List<PasswordEncryptionPermission> encrypPermissions = new
ArrayList<PasswordEncryptionPermission>();
        encrypPermissions.add(PasswordEncryptionPermission.PASSWORD_EDIT_ADD);
        encrypPermissions.add(PasswordEncryptionPermission.PASSWORD_EDIT_MODIFY);
        passSpec.setPermissionsRequested(encrypPermissions);
```

```
        //Specify the Acrobat version
        passSpec.setCompatability(PasswordEncryptionCompatability.ACRO_7);

        //Specify the password values
        passSpec.setDocumentOpenPassword("OpenPassword");
        passSpec.setPermissionPassword("PermissionPassword");

        //Encrypt the PDF document
        Document encryptDoc = encryptClient.encryptPDFUsingPassword(inDoc,passSpec);

        //Save the password-encrypted PDF document
        File outFile = new File("C:\\Adobe\EncryptLoan.pdf");
        encryptDoc.copyToFile (outFile);

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Removing password-based encryption using the Java API

The following Java code example removes password-based encryption from a PDF document named *EncryptLoan.pdf*. The master password value used to remove password-based encryption is *PermissionPassword*. The unsecured PDF document is saved as a PDF file named *noEncryptionLoan.pdf*. (See "Removing Password Encryption" on page 821.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.encryption.client.*;

public class RemovePasswordFromPDFSOAP {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
```

```
            //Create an EncryptionServiceClient object
            EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);

            //Get the encrypted PDF from which to remove password-based encryption
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\EncryptLoan.pdf");
            Document inDoc = new Document (fileInputStream);

            //Remove password-based encryption from the PDF document
            Document encryptDoc =
encryptClient.removePDFPasswordSecurity(inDoc,"PermissionPassword");

            //Save the unsecured PDF document
            File outFile = new File("C:\\Adobe\noEncryptionLoan.pdf");
            encryptDoc.copyToFile (outFile);

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Encrypting a PDF document with a certificate using the Java API

The following Java code example encrypts a PDF document named *Loan.pdf* with a certificate named *Encryption.cer*. The encrypted PDF document is saved as a PDF file named *EncryptLoanCert.pdf*. (See "Encrypting PDF Documents with Certificates" on page 811.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.encryption.client.*;

public class PKIEncryptPDFSoap {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an EncryptionServiceClient object
            EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);
```

```
            //Specify the PDF document to encrypt with a certificate
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document (fileInputStream);

            //Set the List that stores PKI information
            List pkiIdentities = new ArrayList();

            //Set the Permission List
            List permList = new ArrayList();
            permList.add(CertificateEncryptionPermissions.PKI_ALL_PERM) ;

            //Create a Recipient object to store certificate information
            Recipient recipient = new Recipient();

            //Specify the private key that is used to encrypt the document
            FileInputStream fileInputStreamCert = new
FileInputStream("C:\\Adobe\Encryption.cer");
            Document privateKey = new Document (fileInputStreamCert);
            recipient.setX509Cert(privateKey);

            //Create an EncryptionIdentity object
            CertificateEncryptionIdentity encryptionId = new CertificateEncryptionIdentity();
            encryptionId.setPerms(permList);
            encryptionId.setRecipient(recipient);

            //Add the EncryptionIdentity to the list
            pkiIdentities.add(encryptionId);

            //Set encryption run-time options
            CertificateEncryptionOptionSpec certOptionsSpec = new
CertificateEncryptionOptionSpec();
            certOptionsSpec.setOption(CertificateEncryptionOption.ALL);
            certOptionsSpec.setCompat(CertificateEncryptionCompatibility.ACRO_7);

            //Encrypt the PDF document with a certificate
            Document encryptDoc =
encryptClient.encryptPDFUsingCertificates(inDoc,pkiIdentities, certOptionsSpec);

            //Save the encrypted PDF document
            File outFile = new File("C:\\Adobe\EncryptLoanCert.pdf");
            encryptDoc.copyToFile (outFile);

            }catch (Exception e) {
                e.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Removing certificate-based encryption using the Java API

The following Java code example removes certificate-based encryption from a PDF document named *EncryptLoanCert.pdf*. The alias of the public key that is used to remove encryption is `Encryption`. The unsecured PDF document is saved as a PDF file named *noEncryptionLoan.pdf*. (See "Removing Certificate Based Encryption" on page 817.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
```

```
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.encryption.client.*;

public class RemovePKIFromPDFSOAP {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an EncryptionServiceClient object
            EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);

            //Get the encrypted PDF document
            FileInputStream fileInputStream = new
FileInputStream("C:\\Adobe\EncryptLoanCert.pdf");
            Document inDoc = new Document (fileInputStream);

            //Remove certificate-based encryption from the PDF document
            Document encryptDoc = encryptClient.removePDFCertificateSecurity(inDoc,
"Encryption");

            //Save the unsecured PDF document
            File outFile = new File("C:\\Adobe\noEncryptionLoan.pdf");
            encryptDoc.copyToFile (outFile);

        }catch (Exception e) {
                e.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Unlocking an encrypted PDF document using the Java API

The following Java code example unlocks a password-encrypted PDF document named *EncryptLoan.pdf*. (See "Unlocking Encrypted PDF Documents" on page 824.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
```

```
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.encryption.client.*;

public class UnlockPDFSOAP {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an EncryptionServiceClient object
            EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);

            //Get the password-encrypted PDF document to unlock
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\EncryptLoan.pdf");
            Document inDoc = new Document (fileInputStream);

            //Specify the password to open the password-encrypted PDF document
            String openPassword = "OpenPassword" ;

            //Unlock the password-encrypted PDF document
            Document unlockedDoc = encryptClient.unlockPDFUsingPassword(inDoc,openPassword);

        }catch (Exception e) {
                System.out.println("The following error occurred during this operation "
+e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Determining encryption type using the Java API

The following Java code example determines the type of encryption that is protecting a PDF document named *EncryptLoan.pdf*. (See "Determining Encryption Type" on page 828.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.encryption.client.*;

public class GetEncryptionTypeSOAP {

    public static void main(String[] args) {
```

```
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a EncryptionServiceClient object
            EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);

            //Get the PDF document
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\EncryptLoan.pdf");
            Document inDoc = new Document (fileInputStream);

            //Determine the type of encryption of the PDF document
            EncryptionTypeResult encryptTypeResult = encryptClient.getPDFEncryption(inDoc);

            if (encryptTypeResult.getEncryptionType() == EncryptionType.PASSWORD)
                System.out.println("The PDF document is protected with password-based
encryption");
            else if (encryptTypeResult.getEncryptionType() == EncryptionType.POLICY_SERVER)
                System.out.println("The PDF document is protected with policy");
            else if (encryptTypeResult.getEncryptionType() == EncryptionType.CERTIFICATE)
                System.out.println("The PDF document is protected with certificate-based
encryption");
            else if (encryptTypeResult.getEncryptionType() == EncryptionType.OTHER)
                System.out.println("The PDF document is protected with another type of
encryption");
            else if (encryptTypeResult.getEncryptionType() == EncryptionType.NONE)
                System.out.println("The PDF document is not protected.");
        }catch (Exception e) {
                e.printStackTrace();
        }
    }
}
```

Endpoint Registry Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Endpoint Registry.

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick start located in Programming with AEM forms are based on the Forms if you are using another operating system, such as Unix, replace windows specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Note: *You cannot work with endpoints by using a web service.*

QuickStart: Adding an EJB endpoint using the Java API

The following Java code example adds an EJB endpoint to a service named *MyApplication/EncryptDocument*. (See "Adding EJB Endpoints" on page 1112.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
```

```
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

public class AddEJBEndPoint {

    public static void main(String[] args) {

    try{
        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();

        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create an EndpointRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);
```

```
        //Create an SOAP endpoint for the MyApplication/EncryptDocument process
        CreateEndpointInfo e = new CreateEndpointInfo();
        e.setConnectorId("EJB");
        e.setDescription("EJB endpoint for the MyApplication/EncryptDocument proces");
        e.setName("MyApplication/EncryptDocument");
        e.setServiceId("MyApplication/EncryptDocument");
        e.setOperationName("*");
        Endpoint endPoint = endPointClient.createEndpoint(e);

        //Enable the SOAP Endpoint
        endPointClient.enable(endPoint);

    }catch (Exception e) {
         e.printStackTrace();
        }

    }
}
```

QuickStart: Adding a SOAP endpoint using the Java API

The following Java code example adds a SOAP endpoint to a service named *MyApplication/EncryptDocument*. (See "Adding SOAP Endpoints" on page 1114.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *   your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *   These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
```

```
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;


public class AddSoapEndPoint {

    public static void main(String[] args) {

    try{
        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create an EndpointRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);
```

```
        //Create a SOAP Endpoint for the MortgageLoan - Prebuilt process
        CreateEndpointInfo e = new CreateEndpointInfo();
        e.setConnectorId("SOAP");
        e.setDescription("SOAP endpoint for the MyApplication/EncryptDocument proces");
        e.setName("MyApplication/EncryptDocument");
        e.setServiceId("MyApplication/EncryptDocument");
        e.setOperationName("*");
        Endpoint endPoint = endPointClient.createEndpoint(e);

        //Enable the SOAP Endpoint
        endPointClient.enable(endPoint);


    }catch (Exception e) {
         e.printStackTrace();
        }

    }
}
```

QuickStart: Adding a Watched Folder endpoint using the Java API

The following Java code example adds a Watched Folder endpoint to a service named *MyApplication/EncryptDocument*. (See "Adding Watched Folder Endpoints" on page 1116.)

Note: *You must include the WatchedFolderEndpointConfigConstants.java file in your project to compile and run the following quick start. (See "Watched folder configuration values constant file" on page 1122.)*

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *   your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *   These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

public class AddWatchFolderEndPoint {

    public static void main(String[] args) {

    try{
        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create an EndpointRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);

        //Create a Watched Folder endpoint for the MyApplication/EncryptDocument process
        CreateEndpointInfo e = new CreateEndpointInfo();
        e.setConnectorId("WatchedFolder");
        e.setDescription("WatchedFolder endpoint for the EncryptDocument process");
        e.setName("MyApplication/EncryptDocument");
        e.setServiceId("MyApplication/EncryptDocument");
        e.setOperationName("invoke");
```

```
        //Set configuration values for a Watched Folder EndPoint

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_URL,"C:
\\EncryptFolder");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_PROPERTY_ASYNCHRONOU
S,"true");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_PURGE_DURATION,"-
1");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_REPEAT_INTERVAL,"5")
;

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_REPEAT_COUNT,"-1");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_THROTTLE,"false");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_USERNAMER,"SuperAdmi
n");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_DOMAINNAME,"DefaultD
om");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_BATCH_S
IZE,"2");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_WAIT_TI
ME,"0");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_EXCLUDE_FILE_PATTERN
,".txt");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_INCLUDE_FILE_PATTERN
,"*");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_RESULT_
FOLDER_NAME,"result/%Y/%M/%D/");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_PRESERV
E_FOLDER_NAME,"preserve/%Y/%M/%D/");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_FAILURE
_FOLDER_NAME,"failure/%Y/%M/%D/");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_PRESERV
E_ON_FAILURE,"true");

e.setConfigParameterAsText(WatchedFolderEndpointConfigConstants.PROPERTY_FILEPROVIDER_OVERWRI
TE_DUPLICATE_FILENAME,"false");

        //Define input parameter values
        e.setInputParameterMapping("inDoc",
                "com.adobe.idp.Document",
                "variable",
                "*.pdf");
```

```
        //Define the output parameter values
        e.setOutputParameterMapping("outDoc",
                "com.adobe.idp.Document",
                "%F.pdf");

        //Create the Watched Folder Endpoint
        Endpoint endPoint = endPointClient.createEndpoint(e);

        //Enable the Endpoint
        endPointClient.enable(endPoint);


    }catch (Exception e) {
         e.printStackTrace();
        }

    }
}
```

QuickStart: Adding an Email endpoint using the Java API

The following Java code example adds an Email endpoint to a service named *MyApplication/EncryptDocumen*t. (See "Adding Email Endpoints" on page 1122.)

Note: *You must include the EmailEndpointConfigConstants.java file in your project to compile and run the following quick start. (See "Email configuration values constant file" on page 1129.)*

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *   your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *   These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

public class AddEmailEndPoint {

    public static void main(String[] args) {

    try{
        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create an EndpointRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);

        //Create a new Email endpoint for the MyApplication/EncryptDocument process
        CreateEndpointInfo e = new CreateEndpointInfo();
        e.setConnectorId("Email");
        e.setDescription("Email endpoint for the MyApplication/EncryptDocument proces");
        e.setName("MyApplication/EncryptDocument");
        e.setServiceId("MyApplication/EncryptDocument");
        e.setOperationName("invoke");
```

```
        //Set Configuration values for the Email endPoint

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_CRON_EXPRESSIO
N,"");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_REPREAT_COUNT,
"-1");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_REPREAT_INTERV
AL,"10");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_START_DELAY,"0
");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_BATCH_SIZE,"2"
);

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_USERNAME,"Supe
rAdmin");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_DOMAINNAME,"De
faultDom");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_DOMAINPATTERN,
"*");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_FILEPATTERN,"*
");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_RECIPIENT_SUCC
ESSFUL_JOB,"sender");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_RECIPIENT_FAIL
ED_JOB,"sender");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_INBOX_HOST,"sj
-lost");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_INBOX_PORT,"0"
);

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_PROTOCOL,"pop3
");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_INBOX_TIMEOUT,
"60");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_INBOX_USER,"sc
ott");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_INBOX_PASSWORD
,"password");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_INBOX_SSL,"fal
se");
```

```
e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_SMTP_HOST,"sj-
lost");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_SMTP_PORT,"25"
);

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_SMTP_USER,"sco
tt");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_SMTP_PASSWORD,
"password");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_CHARSET,"passw
ord");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_SMTP_SSL,"fals
e");

e.setConfigParameterAsText(EmailEndpointConfigConstants.PROPERTY_EMAILPROVIDER_FAILED_FOLDER,
"failedJobFolder");

        //Define input parameter values
        e.setInputParameterMapping("InDoc",
                "com.adobe.idp.Document",
                "variable",
                "*.pdf");

        //Define the output parameter values
        e.setOutputParameterMapping("SecuredDoc",
                "com.adobe.idp.Document",
                "%F.pdf");

        //Create the Email Endpoint
        Endpoint endPoint = endPointClient.createEndpoint(e);

        //Enable the Email Endpoint
        endPointClient.enable(endPoint);

    }catch (Exception e) {
         e.printStackTrace();
        }

    }
}
```

QuickStart: Adding a Remoting endpoint using the Java API

The following Java code example adds a Remoting endpoint to a service named *MyApplication/EncryptDocument*. (See "Adding Remoting Endpoints" on page 1129.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

/**
    * This Java Quick Start adds a Remoting endpoint to a service named
```

```
MyApplication/EncryptDocument
    */
public class AddRemotingEndPoint {

    public static void main(String[] args) {

    try{
        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();

        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create a ConnectorRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);

        //Create an Remoting Endpoint for the MyApplication/EncryptDocument process
        CreateEndpointInfo e = new CreateEndpointInfo();
        e.setConnectorId("Remoting");
        e.setDescription("Remoting endpoint for the MyApplication/EncryptDocument proces");
        e.setName("EncryptDocumentRemoting");
        e.setServiceId("MyApplication/EncryptDocument");
        e.setOperationName("*");

        //Create the EndPoint
        Endpoint endPoint = endPointClient.createEndpoint(e);

        //Enable the Endpoint
        endPointClient.enable(endPoint);

    }catch (Exception e) {
         e.printStackTrace();
        }

    }
}
```

QuickStart: Adding a TaskManager endpoint using the Java API

The following Java code example adds a TaskManager endpoint to a service named *MyApplication/EncryptDocument*. Notice that the name of the category is *EncryptProcess*. (See "Adding TaskManager Endpoints" on page 1132.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointCategoryInfo;
import com.adobe.idp.dsc.registry.endpoint.CreateEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;
import com.adobe.idp.dsc.registry.infomodel.EndpointCategory;

public class AddTaskManagerEndPoint {
```

```
    public static void main(String[] args) {

    try{
        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();

        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create a ConnectorRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);

        //Create the category associated with this TaskManager endpoint
        CreateEndpointCategoryInfo catInfo = new CreateEndpointCategoryInfo("EncryptProcess",
"Enables this process to be invoked from within Workspace");
        EndpointCategory cat = endPointClient.createEndpointCategory(catInfo);

        //Set TaskManager endpoint attributes
        CreateEndpointInfo e = new CreateEndpointInfo();
        e.setConnectorId("TaskManagerConnector");
        e.setDescription("TaskManagerConnector endpoint for the MyApplication/EncryptDocument
process");
        e.setName("MyApplication/EncryptDocument");
        e.setServiceId("MyApplication2/EncryptDocument");
        e.setCategoryId(cat.getId());
        e.setOperationName("invoke");

        //Create the TaskManagerConnector endpoint
        Endpoint endPoint = endPointClient.createEndpoint(e);

        //Enable the endpoint
        endPointClient.enable(endPoint);

    }catch (Exception e) {
         e.printStackTrace();
        }

    }
}
```

QuickStart: Modifying an endpoint using the Java API

The following Java code example modifies a Watched Folder endpoint.The endpoint is for the *MyApplication/EncryptDocument* process. The watched folder is changed to `C:\NewWatchedFolder`. (See "Modifying Endpoints" on page 1135.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
```

```
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.filter.PagingFilter;
import com.adobe.idp.dsc.registry.endpoint.ModifyEndpointInfo;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

public class ModifyEndPoint {

    public static void main(String[] args) {

    try{
        Endpoint _endpoint = null;

        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create an EndpointRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);

        //Retrieve all endpoints
        List allEndpoints = endPointClient.getEndpoints((PagingFilter)null);

        //Iterate through the returned list of endpoints
        Iterator iter = allEndpoints.iterator();
        int i =0;
        while (iter.hasNext()) {
            _endpoint = (Endpoint) iter.next();

            //Look for an endpoint that belongs to the
            //EncryptDocument service
            String serviceID = _endpoint.getServiceId();

            if (serviceID.matches("MyApplication/EncryptDocument"))
            {
                //Get the WatchedFolder endpoint
                String connId = _endpoint.getConnectorId();
                if (connId.matches("WatchedFolder"))
                {

                    //Create a ModifyEndpointInfo object
                    ModifyEndpointInfo endpointInfo =new ModifyEndpointInfo();

                    //Modify configuration values
                    endpointInfo.setId(_endpoint.getId());
```

```
                    endpointInfo.setConfigParameterAsText("url", "C:\\NewWatchedFolder");
                    endpointInfo.setConfigParameterAsText("asynchronous","true");
                    endpointInfo.setConfigParameterAsText("repeatInterval","5");
                    endpointInfo.setConfigParameterAsText("repeatCount","-1");
                    endpointInfo.setConfigParameterAsText("throttleOn","false");
                    endpointInfo.setConfigParameterAsText("userName","SuperAdmin");
                    endpointInfo.setConfigParameterAsText("domainName","DefaultDom");
                    endpointInfo.setConfigParameterAsText("batchSize","2");
                    endpointInfo.setConfigParameterAsText("waitTime","0");
                    endpointInfo.setConfigParameterAsText("excludeFilePattern",".txt");
                    endpointInfo.setConfigParameterAsText("includeFilePattern","*");

endpointInfo.setConfigParameterAsText("resultFolderName","result/%Y/%M/%D/");

endpointInfo.setConfigParameterAsText("preserveFolderName","preserve/%Y/%M/%D/");

endpointInfo.setConfigParameterAsText("failureFolderName","failure/%Y/%M/%D/");
                    endpointInfo.setConfigParameterAsText("preserveOnFailure","true");

endpointInfo.setConfigParameterAsText("overwriteDuplicateFilename","false");

                    //Define input parameter values
                    endpointInfo.setInputParameterMapping("inDoc",
                            "com.adobe.idp.Document",
                            "variable",
                            "*.pdf");

                    //Define the output parameter values
                    endpointInfo.setOutputParameterMapping("outDoc",
                            "com.adobe.idp.Document",
                            "%F.pdf");

                    //Modify the endpoint for this service
                    endPointClient.modifyEndpoint(endpointInfo);
                    System.out.println("The EJB endpoint for the EncryptDocument service was
modified");
                }
            i++;
            }
        }
    }catch (Exception e) {
         e.printStackTrace();
        }
    }
}
```

QuickStart: Removing an endpoint using the Java API

The following Java code removes an EJB endpoint from a service named *MyApplication/EncryptDocument*. (See "Removing Endpoints" on page 1137.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.filter.PagingFilter;
import com.adobe.idp.dsc.registry.endpoint.client.EndpointRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

/**
```

```
    * This Java Quick Start removes an EJB endpoint from a service named
MyApplication/EncryptDocument
    */
public class RemoveEndPoints {

    public static void main(String[] args) {

    try{
        Endpoint _endpoint = null;

        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create an EndpointRegistryClient object
        EndpointRegistryClient endPointClient = new EndpointRegistryClient(myFactory);

        //Get all endpoints
        List allEndpoints = endPointClient.getEndpoints((PagingFilter)null);

        //Iterate through the returned list of endpoints
        Iterator iter = allEndpoints.iterator();
        int i =0;
        while (iter.hasNext()) {
            _endpoint = (Endpoint) iter.next();

            //Look for an endpoint that belongs to the
            //EncryptDocument service
            String serviceID = _endpoint.getServiceId();

            if (serviceID.matches("MyApplication/EncryptDocument"))
```

```
            {
                //Get the EJB endpoint that belongs to
                //this service
                String connId = _endpoint.getConnectorId();
                if (connId.matches("EJB"))
                {
                    //Remove the EJB endpoint for this service
                    endPointClient.remove(_endpoint);
                    System.out.println("The EJB endpoint for the EncryptDocument service was
removed");
                }
            i++;
            }
        }
    }catch (Exception e) {
         e.printStackTrace();
        }
    }
}
```

QuickStart: Retrieving endpoint connector information using the Java API

The following Java code retrieves information about a Watched Folder endpoint. Information about each configuration value is retrieved and displayed. This code list specifies whether each configuration value is required or optional. In addition, the name and value for each configuration value is displayed. (See "Retrieving Endpoint Connector Information" on page 1139.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.Properties;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.registry.connector.client.ConnectorRegistryClient;
import com.adobe.idp.dsc.registry.infomodel.ConfigParameter;
import com.adobe.idp.dsc.registry.infomodel.Endpoint;

public class RetrieveConnectorInfo {

    public static void main(String[] args) {

    try{
        Endpoint _endpoint = null;

        //Set connection propertiesrequired to invoke AEM Forms
        Properties ConnectionProps = new Properties();
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(ConnectionProps);

        //Create a ConnectorRegistry Client object
        ConnectorRegistryClient conClient = new ConnectorRegistryClient(myFactory);

        //Specify WatchedFolder as the connector type
        Endpoint endpoint = conClient.getEndpointDefinition("WatchedFolder");

        //Get all the configuration values associated with this connector type
        ConfigParameter[] allConfigParams = endpoint.getConfigParameters();
```

```
        int len = allConfigParams.length;

        //Get the value of the individual configuration parameter values
        //and which ones are required and which ones are optional
        for (int i=0; i<len; i++)
        {
            //Get an individual ConfigParameter object
            ConfigParameter cp = (ConfigParameter)allConfigParams[i];

            //Determine if this configuration value is required
            if (cp.isRequired() == true)
                System.out.println("This required configuration value name is "+cp.getName() +
". Its value is "+cp.getTextValue());
            else
                System.out.println("This optional configuration value name is "+cp.getName() +
". Its value is "+cp.getTextValue());
        }
    }catch (Exception e) {
         e.printStackTrace();
        }
    }
}
```

Forms Service API Quick Starts

The following Quick Starts are available for the Forms service:

Application logic that uses the Forms service API is implemented as Java servlets. AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick starts located in Programming with v are based on the forms server being you are using another operating system, such as Unix, replace windows specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

The Adobe Developer web site contains the following article that discusses how to create a ASP.NET application that invokes the Forms service and renders forms. See Creating form rendering ASP.NET applications.

Quick Start (SOAP mode): Rendering an interactive PDF form using the Java API

The following code example renders an interactive PDF form named *Loan.xdp* to a client web browser. A file is attached to the form. Notice that the form design is part of an application and is referenced by using the content root URI value `repository:///`. (See "Rendering Interactive PDF Forms" on page 582.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms.
    */
import java.io.FileInputStream;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
```

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderPDFForm extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Set the parameter values for the renderPDFForm method
            String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";

            byte[]cData = "".getBytes();
            Document oInputData = new Document(cData);

            //Set run-time options using a PDFFormRenderSpec instance
            PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
            pdfFormRenderSpec.setCacheEnabled(new Boolean(true));
            pdfFormRenderSpec.setAcrobatVersion(AcrobatVersion.Acrobat_9);

            //Specify URI values that are required to render a form
            URLSpec uriValues = new URLSpec();
```

```
            uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
            uriValues.setContentRootURI("repository:///");
            uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");

            //Specify file attachments to attach to the form
            FileInputStream fileAttachment = new FileInputStream("C:\\rideau1.jpg");
            Document attachment1 = new Document(fileAttachment);
            String fileName = "rideau1.jpg";
            Map fileAttachments = new HashMap();
            fileAttachments.put(fileName, attachment1);

            //Invoke the renderPDFForm method and write the
            //results to a client web browser
            FormsResult formOut = formsClient.renderPDFForm(
                        formName,               //formQuery
                        oInputData, //inDataDoc
                        pdfFormRenderSpec,      //PDFFormRenderSpec
                        uriValues,      //urlSpec
                        fileAttachments//attachments
                        );

            //Create a Document object that stores form data
            Document myData = formOut.getOutputContent();

            //Get the content type of the response and
            //set the HttpServletResponse objects content type
            String contentType = myData.getContentType();
            resp.setContentType(contentType);

            //Create a ServletOutputStream object
            ServletOutputStream oOutput = resp.getOutputStream();

            //Create an InputStream object
            InputStream inputStream = myData.getInputStream();

            //Write the data stream to the web browser
            byte[] data = new byte[4096];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(data)) > 0)
            {
                oOutput.write(data, 0, bytesRead);
            }

            }catch (Exception e) {
                 e.printStackTrace();
              }
        }
}
```

Quick Start (SOAP mode): Rendering a form at the client using the Java API

The following code example renders a form named *Loan.xdp* at the client using the Forms service Java API. Notice that the form design is part of an application and is referenced by using the content root URI value `repository:///`. (See "Rendering Forms at the Client" on page 596.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms.
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderPDFFormClient extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
```

```
    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a FormsServiceClient object
        FormsServiceClient formsClient = new FormsServiceClient(myFactory);

        //Set parameter values required by the renderPDFForm method
        String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";
        byte[]cData = "".getBytes();
        Document oInputData = new Document(cData);

        //Set a run-time option required to render a form on the client
        PDFFormRenderSpec pdfRenderSpec = new PDFFormRenderSpec();
        pdfRenderSpec.setRenderAtClient(RenderAtClient.Yes);

        //Specify URI values required to render a form
        URLSpec uriValues = new URLSpec();
        uriValues.setApplicationWebRoot("http://[server]:[port]/FormsServiceClientApp");
        uriValues.setContentRootURI("repository:///");
        uriValues.setTargetURL("http://[server]:[port]/FormsServiceClientApp/HandleData");

        //Invoke the renderPDFForm method to render
        //an interactive PDF form on the client
        FormsResult formOut = formsClient.renderPDFForm(
                formName,
                oInputData,
                pdfRenderSpec,
                uriValues,
                null
            );

        //Create a Document object that stores form data
        Document myData = formOut.getOutputContent();

        //Get the content type of the response and
        //set the HttpServletResponse objects content type
        String contentType = myData.getContentType();
        resp.setContentType(contentType);
```

```
        //Create a ServletOutputStream object
        ServletOutputStream oOutput = resp.getOutputStream();

        //Create an InputStream object
        InputStream inputStream = myData.getInputStream();

        //Write the data stream to the web browser
        byte[] data = new byte[4096];
        int bytesRead = 0;
        while ((bytesRead = inputStream.read(data)) > 0)
        {
            oOutput.write(data, 0, bytesRead);
        }

        }catch (Exception e) {
             System.out.println("The following exception occurred: "+e.getMessage());
         }
    }
}
```

Quick Start (SOAP mode): Rendering a Guide (deprecated) using the Java API

The following code example renders a Guide (deprecated) named *TLALifeClaim.xdp* to a client web browser.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
```

```
    */
import java.io.IOException;
import java.io.InputStream;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderFormGuide extends HttpServlet implements Servlet {


    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    try{

        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        FormsServiceClient formsClient = new FormsServiceClient(myFactory);

        //Specify the parameters for the renderActivityGuide method
        String formName = "Applications/FormsApplication/1.0/FormsFolder/TLALifeClaim.xdp";
        byte[] cData = "".getBytes();
        Document oInputData = new Document(cData);

        //Cache the PDF form
        PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
        pdfFormRenderSpec.setCacheEnabled(new Boolean(true));

        //Set Form Guide run-time options
        ActivityGuideRenderSpec renderSpec = new ActivityGuideRenderSpec();
        renderSpec.setGuidePDF(false);
```

```
        //Specify URI values that are required to render a form
        //design located in the AEM Forms repository
        URLSpec uriValues = new URLSpec();
        uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
        uriValues.setContentRootURI("repository:///");
        uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");


        //Invoke the renderFormGuide method
        FormsResult formOut = formsClient.renderFormGuide(
                formName,            //formQuery
                oInputData,          //inDataDoc
                pdfFormRenderSpec,    //pdfFormRenderSpec
                renderSpec,           //activityGuideRenderSpec
                uriValues     //urlSpec
                );

        //Create a Document object that stores form data
        Document myData = formOut.getOutputContent();

        //Get the content type of the response
        String contentType = myData.getContentType();
        resp.setContentType(contentType);

        //Create a ServletOutputStream object
        ServletOutputStream oOutput = resp.getOutputStream();

        //Create an InputStream object
        InputStream inputStream = myData.getInputStream();

        //Write the data stream to the web browser
        byte[] data = new byte[4096];
        int bytesRead = 0;
        while ((bytesRead = inputStream.read(data)) > 0)
        {
            oOutput.write(data, 0, bytesRead);
        }

    }catch (Exception e) {
             System.out.println("The following exception occured: "+e.getMessage());
              }
     }
}
```

Quick Start (SOAP mode): Rendering a form based on fragments using the Java API

The following code example renders a form that is based on fragments. The name of the form design is *PurchaseOrderDynamic.xdp* and it is located in the AEM Forms repository (the XDP file is stored in a folder named FormsFolder located in the repository). Also the fragments that the POFragment form references must also be located in the repository. (See "Rendering Forms Based on Fragments" on page 600.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.FileInputStream;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderFormFragments extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);

    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
```

```
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Set the parameter values for the renderPDFForm method
            String formName =
"Applications/FormsApplication/1.0/FormsFolder/PurchaseOrderDynamic.xdp";

            FileInputStream myFormData = new FileInputStream("C:\\Adobe\Purchase Order
US.xml");
            Document oInputData = new Document(myFormData);

            //Cache the PDF form
            PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
            pdfFormRenderSpec.setCacheEnabled(new Boolean(true));

            //Specify URI values that are required to render a form
            //design based on fragments
            URLSpec uriValues = new URLSpec();
            uriValues.setApplicationWebRoot("http://[server]:[port]/FormsServiceClientApp");
            uriValues.setContentRootURI("repository:///");
            uriValues.setTargetURL("http://[server]:[port]/FormsServiceClientApp/HandleData");

            //Invoke the renderPDFForm method and write the
            //results to a client web browser
            FormsResult formOut = formsClient.renderPDFForm(
                        formName,               //formQuery
                        oInputData, //inDataDoc
                        pdfFormRenderSpec,      //PDFFormRenderSpec
                        uriValues,      //urlSpec
                        null                //attachments
                        );

            //Create a Document object that stores form data
            Document myData = formOut.getOutputContent();
```

```
            //Get the content type of the response and
            //set the HttpServletResponse object's content type
            String contentType = myData.getContentType();
            resp.setContentType(contentType);

            //Create a ServletOutputStream object
            ServletOutputStream oOutput = resp.getOutputStream();

            //Create an InputStream object
            InputStream inputStream = myData.getInputStream();

            //Write the data stream to the web browser
            byte[] data = new byte[4096];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(data)) > 0)
            {
                oOutput.write(data, 0, bytesRead);
            }
        }catch (Exception e) {
             System.out.println("The following exception occurred: "+e.getMessage());
      }
    }
}
```

Quick Start (SOAP mode): Rendering a rights-enabled form using the Java API

The following code example renders a rights-enabled form to a client web browser. The usage rights set in this code example enable a user to add comments in the form and save form data. (See "Rendering Rights-Enabled Forms" on page 605.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
```

```
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class RenderUsageRightsForms extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a FormsServiceClient object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        FormsServiceClient formsClient = new FormsServiceClient(myFactory);

        //Set parameter values for the renderPDFFormWithUsageRights method
        String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";
        byte[]cData = "".getBytes();
        Document oInputData = new Document(cData);

        //Set run-time options
```

```
        PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
        pdfFormRenderSpec.setCacheEnabled(new Boolean(true));

        //Set usage-rights run-time options
        ReaderExtensionSpec reOptions = new ReaderExtensionSpec();
        reOptions.setReCredentialAlias("RE2");
        reOptions.setReCommenting(true);
        reOptions.setReFillIn(true);

        //Specify URI values required to render the form
        URLSpec uriValues = new URLSpec();
        uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
        uriValues.setContentRootURI("repository:///");
        uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");

        //Render a rights-enabled PDF form
        FormsResult formOut = formsClient.renderPDFFormWithUsageRights(
            formName,            //formQuery
            oInputData,          //inDataDoc
            pdfFormRenderSpec,   //renderFormOptionsSpec
            reOptions,   //applicationWebRoot
            uriValues            //targetURL
            );

        //Create a Document object that stores form data
        Document myData = formOut.getOutputContent();

        //Get the content type of the response and
        //set the HttpServletResponse objects content type
        String contentType = myData.getContentType();
        resp.setContentType(contentType);

        //Create a ServletOutputStream object
        ServletOutputStream oOutput = resp.getOutputStream();

        //Create an InputStream object
        InputStream inputStream = myData.getInputStream();

        //Write the data stream to the web browser
        byte[] data = new byte[4096];
        int bytesRead = 0;
        while ((bytesRead = inputStream.read(data)) > 0)
        {
            oOutput.write(data, 0, bytesRead);
        }

    }catch (Exception e) {
         System.out.println("The following exception occurred: "+e.getMessage());
      }
    }
}
```

Quick Start (SOAP mode): Rendering an HTML form using the Java API

The following code example renders an HTML form using the Forms service Java API. A toolbar is added to the HTML form as well as two file attachments. In addition, the user agent value is obtained from the `HttpServletRequest` object. (See "Rendering Forms as HTML" on page 609.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;

import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import java.io.FileInputStream;


public class RenderHTMLForms extends HttpServlet implements Servlet {
```

```
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

            try{

                //Set connection properties required to invoke AEM Forms
                Properties connectionProps = new Properties();
                connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT",
"http://[server]:[port]");
                connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL","SOAP");
                connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
                connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
                connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

                //Create a FormsServiceClient object
                ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
                FormsServiceClient formsClient = new FormsServiceClient(myFactory);

                //Set parameter values for the (Deprecated) renderHTMLForm method
                String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";
                byte[]cData = "".getBytes();
                Document oInputData = new Document(cData);

                //Obtain the user agent value from the HttpServletRequest object
                String userAgent = req.getHeader("user-agent");

                //Create an HTMLRenderSpec object to store HTML run-time options
                HTMLRenderSpec htmlRS = new HTMLRenderSpec();
                htmlRS.setHTMLToolbar(HTMLToolbar.Vertical);

                //Specify the locale value
                htmlRS.setLocale("en_US");

                //Render the HTML form within full HTML tags
                htmlRS.setOutputType(OutputType.FullHTMLTags);

                //Set style information that controls the presentation of the HTML form
                htmlRS.setStyleGenerationLevel(StyleGenerationLevel.InlineAndInternalStyles);

                //Specify URI values that are required to render a form
                URLSpec uriValues = new URLSpec();
                uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
                uriValues.setContentRootURI("repository:///");

uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleSubmittedHTMLForm");

                //Specify file attachments
                FileInputStream myForm = new FileInputStream("C:\\Attach1.txt");
                Document attachment1 = new Document(myForm);
                FileInputStream myForm2 = new FileInputStream("C:\\Attach2.txt");
                Document attachment2 = new Document(myForm2);
```

```
                String fileName = "Attach1.txt";
                String fileName2 = "Attach2.txt";

                Map fileAttachments = new HashMap();
                fileAttachments.put(fileName, attachment1);
                fileAttachments.put(fileName2, attachment2);

                //Invoke the (Deprecated) renderHTMLForm method
                FormsResult formOut = formsClient.renderHTMLForm(
                    formName,               //formQuery
                    TransformTo.MSDHTML,//transformTo
                    oInputData, //inDataDoc
                    htmlRS,             //renderHTMLSpec
                    userAgent,      //User Agent
                    uriValues,      //urlSpec
                    fileAttachments//attachments
                    );

                //Create a Document object that stores form data
                Document myData = formOut.getOutputContent();

                //Get the content type of the response and
                //set the HttpServletResponse object's content type
                String contentType = myData.getContentType();
                resp.setContentType(contentType);

                //Create a ServletOutputStream object
                ServletOutputStream oOutput = resp.getOutputStream();

                //Create an InputStream object
                InputStream inputStream = myData.getInputStream();

                //Write the data stream to the web browser
                byte[] data = new byte[4096];
                int bytesRead = 0;
                while ((bytesRead = inputStream.read(data)) > 0)
                {
                    oOutput.write(data, 0, bytesRead);
                }

                }catch (Exception e) {
                     System.out.println("The following exception occurred: "+e.getMessage());
                     }
            }
}
```

Quick Start (SOAP mode): Rendering an HTML form that uses a CSS file using the Java API

The following code example renders an HTML form using the Forms service Client API. The name of the custom CSS file that is referenced is *custom.css*. (See "Rendering HTML Forms Using Custom CSS Files" on page 618.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;

import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

import java.io.FileInputStream;


public class RenderHTMLCSS extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }
```

```
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

            try{
                //Set connection properties required to invoke AEM Forms
                Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClientF
actoryProperties.DSC_SOAP_PROTOCOL);
                connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD, "password");

                //Create a FormsServiceClient object
                ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
                FormsServiceClient formsClient = new FormsServiceClient(myFactory);

                //Set parameter values for the (Deprecated) renderHTMLForm method
                String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";
                byte[]cData = "".getBytes();
                Document oInputData = new Document(cData);
                String userAgent = "" ;

                //Create an HTMLRenderSpec object to store HTML run-time options
                HTMLRenderSpec htmlRS = new HTMLRenderSpec();

                //Specify the locale value
                htmlRS.setLocale("en_US");

                //Specify a custom CSS file to use
                htmlRS.setCustomCSSURI("C:\\Adobe\custom.css");

                //Render the HTML form within full HTML tags
                htmlRS.setOutputType(OutputType.FullHTMLTags);

                //Specify URI values that are required to render a form
                URLSpec uriValues = new URLSpec();
                uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
                uriValues.setContentRootURI("repository:///");
                uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");

                //Specify file attachments
                FileInputStream myForm = new FileInputStream("C:\\Attach1.txt");
                Document attachment1 = new Document(myForm);
                FileInputStream myForm2 = new FileInputStream("C:\\Attach2.txt");
                Document attachment2 = new Document(myForm2);
                String fileName = "Attach1.txt";
                String fileName2 = "Attach2.txt";
```

```
                Map fileAttachments = new HashMap();
                fileAttachments.put(fileName, attachment1);
                fileAttachments.put(fileName2, attachment2);

                //Invoke the (Deprecated) renderHTMLForm method
                FormsResult formOut = formsClient.renderHTMLForm(
                    formName,               //formQuery
                    TransformTo.MSDHTML,//transformTo
                    oInputData, //inDataDoc
                    htmlRS,              //renderHTMLSpec
                    userAgent,       //User Agent
                    uriValues,       //urlSpec
                    fileAttachments//attachments
                    );

                //Create a Document object that stores form data
                Document myData = formOut.getOutputContent();

                //Get the content type of the response and
                //set the HttpServletResponse object's content type
                String contentType = myData.getContentType();
                resp.setContentType(contentType);

                //Create a ServletOutputStream object
                ServletOutputStream oOutput = resp.getOutputStream();

                //Create an InputStream object
                InputStream inputStream = myData.getInputStream();

                //Write the data stream to the web browser
                byte[] data = new byte[4096];
                int bytesRead = 0;
                while ((bytesRead = inputStream.read(data)) > 0)
                {
                    oOutput.write(data, 0, bytesRead);
                }

            }catch (Exception e) {
                     System.out.println("The following exception occurred: "+e.getMessage());
             }
    }
}
```

Quick Start (SOAP mode): Rendering an HTML Form with a custom toolbar using the Java API

The following code example renders an HTML form with a toolbar that is displayed in French. The location of the fscmenu.xml is C:\Adobe (this folder must be on the server hosting AEM Forms). Notice that the locale value is `fr_FR`. The section that discusses how to render an HTML form with a custom toolbar shows the syntax of the fscmenu.xml file used in this quick start. (See "Rendering HTML Forms with Custom Toolbars" on page 623.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;

import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import java.io.FileInputStream;


public class RenderCustomToolbar extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
```

```
        throws ServletException, IOException {

            try{
                //Set connection properties required to invoke AEM Forms
                Properties connectionProps = new Properties();
                connectionProps.setProperty("DSC_DEFAULT_SOAP_ENDPOINT",
"http://[server]:[port]");
                connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL","SOAP");
                connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
                connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
                connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

                //Create a FormsServiceClient object
                ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
                FormsServiceClient formsClient = new FormsServiceClient(myFactory);

                //Set parameter values for the renderHTMLForm method
                String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";
                byte[]cData = "".getBytes();
                Document oInputData = new Document(cData);
                String userAgent = "" ;

                //Create an HTMLRenderSpec object to store HTML run-time options
                HTMLRenderSpec htmlRS = new HTMLRenderSpec();
                htmlRS.setHTMLToolbar(HTMLToolbar.Vertical);

                //Specify the URI location of the
                // fscmenu.xml file that contains French
                htmlRS.setToolbarURI("C:\\Adobe");

                //Specify the locale value
                htmlRS.setLocale("fr_FR");

                //Render the HTML form within full HTML tags
                htmlRS.setOutputType(OutputType.FullHTMLTags);

                //Specify URI values that are required to render a form
                URLSpec uriValues = new URLSpec();
                uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
                uriValues.setContentRootURI("repository:///");
                uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");

                //Specify file attachments
                FileInputStream myForm = new FileInputStream("C:\\Attach1.txt");
                Document attachment1 = new Document(myForm);
                FileInputStream myForm2 = new FileInputStream("C:\\Attach2.txt");
                Document attachment2 = new Document(myForm2);
                String fileName = "Attach1.txt";
                String fileName2 = "Attach2.txt";

                Map fileAttachments = new HashMap();
                fileAttachments.put(fileName, attachment1);
                fileAttachments.put(fileName2, attachment2);

                //Invoke the renderHTMLForm method
                FormsResult formOut = formsClient.renderHTMLForm(
```

```
                    formName,               //formQuery
                    TransformTo.MSDHTML,//transformTo
                    oInputData, //inDataDoc
                    htmlRS,             //renderHTMLSpec
                    userAgent,      //User Agent
                    uriValues,      //urlSpec
                    fileAttachments//attachments
                    );

                //Create a Document object that stores form data
                Document myData = formOut.getOutputContent();

                //Get the content type of the response and
                //set the HttpServletResponse object's content type
                String contentType = myData.getContentType();
                resp.setContentType(contentType);

                //Create a ServletOutputStream object
                ServletOutputStream oOutput = resp.getOutputStream();

                //Create an InputStream object
                InputStream inputStream = myData.getInputStream();

                //Write the data stream to the web browser
                byte[] data = new byte[4096];
                int bytesRead = 0;
                while ((bytesRead = inputStream.read(data)) > 0)
                {
                    oOutput.write(data, 0, bytesRead);
                }

                }catch (Exception e) {
                     System.out.println("The following exception occurred: "+e.getMessage());
                     }
            }
}
```

Quick Start (SOAP mode): Handling PDF forms submitted as XML using the Java API

The following code example handles a form that is submitted as XML. The content type value passed to the `processFormSubmission` method is `CONTENT_TYPE=text/xml`. The values that correspond to the fields named `mortgageAmount`, `lastName`, and `firstName` are displayed. A user-defined method named `getNodeText` is used in this quick start. It accepts an `org.w3c.dom.Document` instance and a string value that specifies the node name. This method returns a string value that represents the value of the node. (See "Handling Submitted Forms" on page 630.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 9. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.DataInputStream;
import java.io.File;
import java.io.InputStream;
import java.io.PrintWriter;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

//Import DOM libraries
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import javax.xml.parsers.*;

public class HandleData extends HttpServlet implements Servlet {
```

```
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
                doPost(req,resp);
        }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        try{
            PrintWriter pp = resp.getWriter();

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Get Form data to pass to the processFormSubmission method
            Document formData = new Document(req.getInputStream());

            //Set run-time options
             RenderOptionsSpec processSpec = new RenderOptionsSpec();
             processSpec.setLocale("en_US");

            //Invoke the processFormSubmission method
            FormsResult formOut = formsClient.processFormSubmission(formData,
            "CONTENT_TYPE=text/xml",
            "",
            processSpec);

            //Get the processing state
            short processState = formOut.getAction();

            //Determine if the form data is ready to be processed
            //This code example checks only for submitted data (value is 0)
            if (processState == 0)
            {
              //Determine the content type of the data
              String myContentType = formOut.getContentType();
```

```
            System.out.println("THE CONTENT TYPS IS" +myContentType);

             if (myContentType.equals("application/vnd.adobe.xdp+xml")){

              //Get the form data
              Document formOutput = formOut.getOutputContent();
              InputStream formInputStream = new DataInputStream(formOutput.getInputStream());

              //Create DocumentBuilderFactory and DocumentBuilder objects
              DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
              DocumentBuilder builder = factory.newDocumentBuilder();
              org.w3c.dom.Document myDOM = builder.parse(formInputStream);

              //Call for each field in the form
              String Amount = getNodeText("mortgageAmount", myDOM);
              String myLastName =  getNodeText("lastName", myDOM);
              String myFirstName = getNodeText("firstName", myDOM);

              //Write the form data to the web browser
              pp.println("<p> The form data is :<br><br>" +
                      "<li> The mortgage amount is "+ Amount+"" +
                      "<li> Last name is "+ myLastName+"" +
                      "<li> First name is "+ myFirstName+"");


              }
            }
           }
        catch (Exception e) {
             e.printStackTrace();
          }
    }

    //This method returns the value of the specified node
    private String getNodeText(String nodeName, org.w3c.dom.Document myDOM)
    {
      //Get the XML node by name
      NodeList oList = myDOM.getElementsByTagName(nodeName);
      Node myNode = oList.item(0);
      NodeList oChildNodes = myNode.getChildNodes();

     String sText = "";
     for (int i = 0; i < oChildNodes.getLength(); i++)
     {
         Node oItem = oChildNodes.item(i);
        if (oItem.getNodeType() == Node.TEXT_NODE)
         {
           sText = sText.concat(oItem.getNodeValue());
         }
     }
    return sText;
    }
}
```

Note: *When using a* `com.adobe.idp.Document` *object and an* `org.w3c.dom.Document` *in the same application, fully qualify* `org.w3c.dom.Document`.

Quick Start (SOAP mode): Handling PDF forms submitted as PDF using the Java API

The following code example handles a form that is submitted as PDF data. The content type value passed to the `processFormSubmission` method is `CONTENT_TYPE=application/pdf`. The submitted form is saved as a PDF file named *tempPDF.pdf*. Also, because the form is submitted as PDF, file attachments can be retrieved. Any file attachments are saved as JPEG files. (See "Handling Submitted Forms" on page 630.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.DataInputStream;
import java.io.File;
import java.io.InputStream;
import java.io.PrintWriter;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
```

```
//Import DOM libraries
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import javax.xml.parsers.*;

public class HandleSubmittedPDFData extends HttpServlet implements Servlet {


    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
                doPost(req,resp);
        }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        try{
            PrintWriter pp = resp.getWriter();

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Get Form data to pass to the processFormSubmission method
            Document formData = new Document(req.getInputStream());

            //Set run-time options
             RenderOptionsSpec processSpec = new RenderOptionsSpec();
             processSpec.setLocale("en_US");

            //Invoke the processFormSubmission method
            FormsResult formOut = formsClient.processFormSubmission(formData,
            "CONTENT_TYPE=application/pdf",
            "",
            processSpec);

            //Determine if the form contains file attachments
            //It is assumed that file attachments are JPG files
```

```
            List fileAttachments = formOut.getAttachments();

            //Create an Iterator object and iterate through
            //the List object
            Iterator iter = fileAttachments.iterator();
            int i = 0 ;
            while (iter.hasNext()) {
                Document file = (Document)iter.next();
                file.copyToFile(new File("C:\\Adobe\tempFile"+i+".jpg"));
                i++;
            }

            //Get the processing state
            short processState = formOut.getAction();

            //Determine if the form data is ready to be processed
            //This code example checks only for submitted data (value is 0)
            if (processState == 0)
            {
              //Determine the content type of the data
              String myContentType = formOut.getContentType();

              if (myContentType.equals("application/pdf")){

                    //Get the form data
                    Document myPDFfile = formOut.getOutputContent();

                    //Create a PDF object
                    File myPDFFile = new File("C:\\Adobe\tempPDF.pdf");

                    //Populate the PDF file
                    myPDFfile.copyToFile(myPDFFile);
                    pp.println("<p> The PDF file is saved as C:\\Adobe\tempPDF.pdf") ;

                }
              }
        }
        catch (Exception e) {
             e.printStackTrace();
          }
    }

}
```

Quick Start (SOAP mode): Handling HTML forms submitted as XML using the Java API

The following code example handles an HTML form that is submitted as XML data. The content type value passed to the `processFormSubmission` method is `CONTENT_TYPE=application/x-www-form-urlencoded`.The values that correspond to the fields named `mortgageAmount`, `lastName`, and `firstName` are displayed. A user-defined method named `getNodeText` is used in this quick start. It accepts an `org.w3c.dom.Document` instance and a string value that specifies the node name. This method returns a string value that represents the value of the node. (See "Handling Submitted Forms" on page 630.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.DataInputStream;
import java.io.File;
import java.io.InputStream;
import java.io.PrintWriter;
import com.adobe.idp.Document;
```

```
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

//Import DOM libraries
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import javax.xml.parsers.*;

/*
    * This quick start handles data submitted as XML from a rendered HTML form
    */
public class HandleSubmittedHTMLForm extends HttpServlet implements Servlet {


    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
                doPost(req,resp);
        }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        try{
            PrintWriter pp = resp.getWriter();

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Get Form data to pass to the processFormSubmission method
            Document formData = new Document(req.getInputStream());

            //Set run-time options
             RenderOptionsSpec processSpec = new RenderOptionsSpec();
             processSpec.setLocale("en_US");

            //Invoke the processFormSubmission method
            FormsResult formOut = formsClient.processFormSubmission(formData,
            "CONTENT_TYPE=application/x-www-form-urlencoded",
```

```
            "",
            processSpec);

            //Get the processing state
            short processState = formOut.getAction();

            //Determine if the form data is ready to be processed
            //This code example checks only for submitted data (value is 0)
            if (processState == 0)
            {

                    //Get the form data
                    Document formOutput = formOut.getOutputContent();
                    InputStream formInputStream = new
DataInputStream(formOutput.getInputStream());

                    //Create DocumentBuilderFactory and DocumentBuilder objects
                    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
                    DocumentBuilder builder = factory.newDocumentBuilder();
                    org.w3c.dom.Document myDOM = builder.parse(formInputStream);

                    //Call for each field in the form
                    String Amount = getNodeText("mortgageAmount", myDOM);
                    String myLastName =  getNodeText("lastName", myDOM);
                    String myFirstName = getNodeText("firstName", myDOM);

                    //Write the form data to the web browser
                    pp.println("<p> The form data is :<br><br>" +
                            "<li> The mortgage amount is "+ Amount+"" +
                            "<li> Last name is "+ myLastName+"" +
                            "<li> First name is "+ myFirstName+"");
                     }
                }
            catch (Exception e) {
                 e.printStackTrace();
              }
        }

        //This method returns the value of the specified node
```

```
    private String getNodeText(String nodeName, org.w3c.dom.Document myDOM)
    {
      //Get the XML node by name
      NodeList oList = myDOM.getElementsByTagName(nodeName);
      Node myNode = oList.item(0);
      NodeList oChildNodes = myNode.getChildNodes();

     String sText = "";
     for (int i = 0; i < oChildNodes.getLength(); i++)
     {
         Node oItem = oChildNodes.item(i);
        if (oItem.getNodeType() == Node.TEXT_NODE)
         {
           sText = sText.concat(oItem.getNodeValue());
         }
     }
    return sText;
    }
}
```

Quick Start (SOAP mode): Creating PDF Documents with submitted XML data using the Java API

The following Java code example handles form data that is submitted as XML. Form data is retrieved from the Form submission using the Forms API and sent to the Output service. The form data and a form design are used to create a non-interactive PDF document. The non-interactive PDF document is stored in a Content Services (deprecated) node named `/Company Home/Test Directory`. The name of the form is dynamically created. That is, the user's first and last name are used to name the PDF file. The resource identifier of the new content is written out to the client web browser. (See "Creating PDF Documents with Submitted XML Data" on page 639.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-output-client.jar
    * 21. adobe-contentservices-client.jar
    *
```

```
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.adobe.livecycle.contentservices.client.CRCResult;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;
import com.adobe.livecycle.contentservices.client.impl.UpdateVersionType;
import com.adobe.livecycle.formsservice.client.*;
import com.adobe.livecycle.output.client.OutputClient;
import com.adobe.livecycle.output.client.OutputResult;
import com.adobe.livecycle.output.client.PDFOutputOptionsSpec;
import com.adobe.livecycle.output.client.TransformationFormat;

import java.util.*;
import java.io.DataInputStream;
import java.io.File;
import java.io.InputStream;
import java.io.PrintWriter;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.InvocationRequest;
import com.adobe.idp.dsc.InvocationResponse;
import com.adobe.idp.dsc.clientsdk.ServiceClient;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

//Import DOM libraries
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import javax.xml.parsers.*;

public class HandleDataSendToOutput extends HttpServlet implements Servlet {


    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
                doPost(req,resp);
        }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        try{
            PrintWriter pp = resp.getWriter();
```

```
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Get Form data to pass to the processFormSubmission method
            Document formData = new Document(req.getInputStream());

            //Set run-time options
             RenderOptionsSpec processSpec = new RenderOptionsSpec();
             processSpec.setLocale("en_US");

            //Invoke the processFormSubmission method
            FormsResult formOut = formsClient.processFormSubmission(formData,
            "CONTENT_TYPE=text/xml",
            "",
            processSpec);

            //Get the processing state
            short processState = formOut.getAction();

            //Determine if the form data is ready to be processed
            //This code example checks only for submitted data (value is 0)
            if (processState == 0)
            {
              //Determine the content type of the data
              String myContentType = formOut.getContentType();

              if (myContentType.equals("application/vnd.adobe.xdp+xml")){

                //Get the form data
                Document formOutput = formOut.getOutputContent();
                InputStream formInputStream = new DataInputStream(formOutput.getInputStream());

                //Create DocumentBuilderFactory and DocumentBuilder objects
                DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
                DocumentBuilder builder = factory.newDocumentBuilder();
```

```
                org.w3c.dom.Document myDOM = builder.parse(formInputStream);

                //Call for each field in the form
                String Amount = getNodeText("mortgageAmount", myDOM);
                String myLastName =  getNodeText("lastName", myDOM);
                String myFirstName = getNodeText("firstName", myDOM);

                //Write the form data to the web browser
                pp.println("<p> The form data is :<br><br>" +
                        "<li> The mortgage amount is "+ Amount+"" +
                        "<li> Last name is "+ myLastName+"" +
                        "<li> First name is "+ myFirstName+"");


                //Create a non-interactive PDF document by invoking the Output service
                Document myPDFform = GeneratePDFDocument(myFactory, formOutput);

                //Create the name of the PDF file to store
                String pdfName = "Loan_"+myLastName+"_"+myFirstName+".pdf" ;
                String userName = myFirstName+" "+myLastName ;

                //Store the PDF form into Content Services (deprecated)
                String resourceID = StorePDFDocument(myFactory, myPDFform, pdfName,userName);
                pp.println("<p> The pdf document was store in :<br><br>" +
                        "<li> /Company home "+
                        "<li> The identifier value of the new resource is "+ resourceID+"");
                  }
            }
        }
        catch (Exception e) {
             e.printStackTrace();
          }
    }


    //Store the PDF document in /Company Home/Test Directory using the
    //AEM Forms Content Service API
    private String StorePDFDocument(ServiceClientFactory myFactory, com.adobe.idp.Document
pdfDoc, String formName, String userName)
    {
        try
        {
            //Create a DocumentManagementServiceClientImpl object
            DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

            //Specify the store and node name
            String storeName ="SpacesStore";
            String nodeName = "/Company Home/Test Directory";

            //Create a MAP instance to store attributes
            Map<String,Object> inputs = new HashMap<String,Object>();

            //Specify attributes that belong to the new content
            String creator = "{http://www.alfresco.org/model/content/1.0}creator";
            String description = "{http://www.alfresco.org/model/content/1.0}description";
```

```
            inputs.put(creator,userName);
            inputs.put(description,"A mortgage application form");

            //Store MortgageForm.pdf in /Company Home/Test Directory
            CRCResult result = docManager.storeContent(storeName,
                     nodeName,
                     formName,
                    "{http://www.alfresco.org/model/content/1.0}content",
                    pdfDoc,
                    "UTF-8",
                    UpdateVersionType.INCREMENT_MAJOR_VERSION,
                    null,
                    inputs);
            //Get the identifier value of the new resource
            String id = result.getNodeUuid();
            return id;
    }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
        return null ;
    }


    //This method returns the value of the specified node
    private com.adobe.idp.Document GeneratePDFDocument(ServiceClientFactory myFactory,
com.adobe.idp.Document formData)
    {
        try
        {
        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Set PDF run-time options
        com.adobe.livecycle.output.client.PDFOutputOptionsSpec outputOptions = new
PDFOutputOptionsSpec();
        outputOptions.setLocale("en_US");

        //Set rendering run-time options
        com.adobe.livecycle.output.client.RenderOptionsSpec pdfOptions = new
com.adobe.livecycle.output.client.RenderOptionsSpec();
        pdfOptions.setLinearizedPDF(true);

        //Create a PDF document
        OutputResult outputDocument = outClient.generatePDFOutput(
            TransformationFormat.PDF,
            "Loan.xdp",
            "C:\\Adobe",
            outputOptions,
            pdfOptions,
            formData
        );

        //Get the Generated PDF file
        Document ouputDoc = outputDocument.getGeneratedDoc();
        return ouputDoc ;
```

```
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
        return null;
    }

    //This method returns the value of the specified node
    private String getNodeText(String nodeName, org.w3c.dom.Document myDOM)
    {
      //Get the XML node by name
      NodeList oList = myDOM.getElementsByTagName(nodeName);
      Node myNode = oList.item(0);
      NodeList oChildNodes = myNode.getChildNodes();

     String sText = "";
     for (int i = 0; i < oChildNodes.getLength(); i++)
     {
         Node oItem = oChildNodes.item(i);
        if (oItem.getNodeType() == Node.TEXT_NODE)
         {
           sText = sText.concat(oItem.getNodeValue());
         }
     }
    return sText;
    }
}
```

Quick Start (SOAP mode): Prepopulating Forms with Flowable Layouts using the Java API

The following code example prepopulates a form with a dynamic data source. That is, the data source is created at run-time and is not contained within an XML file or created during design time. This code example contains three user-defined methods:

- `createDataSource`: Creates an `org.w3c.dom.Document` object that represents the data source that is used to prepopulate the form. This user-defined method returns the `org.w3c.dom.Document` object.
- `convertDataSource`: Converts an `org.w3c.dom.Document` object to a `com.adobe.idp.Document` object. This method accepts an `org.w3c.dom.Document` object as an input parameter and returns a `com.adobe.idp.Document` object.
- `renderPOForm`: Uses the Forms service Java API to render a dynamic purchase order form. The `com.adobe.idp.Document` object that was returned by the `convertDataSource` method is used to prepopulate the form.

  All of these methods are invoked from within the Java servlet's `doPost` method. (See "Prepopulating Forms with Flowable Layouts" on page 644.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;

import java.util.*;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

import org.w3c.dom.Element;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
```

```
public class RenderDynamicForm extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }


    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

            //Render a dynamic purchase order form
            //Create an org.w3c.dom.Document object
            org.w3c.dom.Document myDom = createDataSource();

            //Convert the org.w3c.dom.Document object
            //to a com.adobe.idp.Document object
            com.adobe.idp.Document formData = convertDataSource(myDom);

            //Render the dynamic form using data located within the
            //com.adobe.idp.Document object
            renderPOForm(resp,formData);
     }


     //Creates an org.w3c.dom.Document object
     private org.w3c.dom.Document createDataSource()
     {
        org.w3c.dom.Document document = null;

        try
        {
            //Create DocumentBuilderFactory and DocumentBuilder objects
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();

            //Create a new Document object
            document = builder.newDocument();

            //Create the root element and append it to the XML DOM
            Element root = (Element)document.createElement("transaction");
            document.appendChild(root);

            //Create the header element
            Element header = (Element)document.createElement("header");
            root.appendChild(header);

            //Create the txtPONum element and append it to the
            //header element
            Element txtPONum = (Element)document.createElement("txtPONum");
            txtPONum.appendChild(document.createTextNode("8745236985"));
            header.appendChild(txtPONum);

            //Create the dtmDate element and append it to the
            //header element
            Element dtmDate = (Element)document.createElement("dtmDate");
            dtmDate.appendChild(document.createTextNode("2007-02-08"));
```

```
            header.appendChild(dtmDate);

            //Create the orderedByAddress element and append
            //it to the header element
            Element orderedByAddress =(Element)document.createElement("orderedByAddress");
            orderedByAddress.appendChild(document.createTextNode("222, Any Blvd"));
            header.appendChild(orderedByAddress);

            //Create the txtOrderedByPhone element and append
            //it to the header element
            Element txtOrderedByPhone =
(Element)document.createElement("txtOrderedByPhone");
            txtOrderedByPhone.appendChild(document.createTextNode("(555) 555-2334"));
            header.appendChild(txtOrderedByPhone);

            //Create the txtOrderedByFax element and append
            //it to the header element
            Element txtOrderedByFax = (Element)document.createElement("txtOrderedByFax");
            txtOrderedByFax.appendChild(document.createTextNode("(555) 555-9334"));
            header.appendChild(txtOrderedByFax);

            //Create the txtOrderedByContactName element and append
            //it to the header element
            Element txtOrderedByContactName =
(Element)document.createElement("txtOrderedByContactName");
            txtOrderedByContactName.appendChild(document.createTextNode("Frank Jones"));
            header.appendChild(txtOrderedByContactName);

            //Create the deliverToAddress element and append
            //it to the header element
            Element deliverToAddress = (Element)document.createElement("deliverToAddress");
            deliverToAddress.appendChild(document.createTextNode("555, Any Blvd"));
            header.appendChild(deliverToAddress);

            //Create the txtDeliverToPhone element and append
            //it to the header element
            Element txtDeliverToPhone =
(Element)document.createElement("txtDeliverToPhone");
            txtDeliverToPhone.appendChild(document.createTextNode("(555) 555-9098"));
            header.appendChild(txtDeliverToPhone);

            //Create the txtDeliverToFax element and append
            //it to the header element
            Element txtDeliverToFax = (Element)document.createElement("txtDeliverToFax");
            txtDeliverToFax.appendChild(document.createTextNode("(555) 555-9000"));
            header.appendChild(txtDeliverToFax);

            //Create the txtDeliverToContactName element and
            //append it to the header element
            Element txtDeliverToContactName =
(Element)document.createElement("txtDeliverToContactName");
            txtDeliverToContactName.appendChild(document.createTextNode("Jerry Johnson"));
            header.appendChild(txtDeliverToContactName);

            //Create the detail element and append it to the root
            Element detail = (Element)document.createElement("detail");
            root.appendChild(detail);
```

```
            //Create the txtPartNum element and append it to the
            //detail element
            Element txtPartNum = (Element)document.createElement("txtPartNum");
            txtPartNum.appendChild(document.createTextNode("00010-100"));
            detail.appendChild(txtPartNum);

            //Create the txtDescription element and append it
            //to the detail element
            Element txtDescription = (Element)document.createElement("txtDescription");
            txtDescription.appendChild(document.createTextNode("Monitor"));
            detail.appendChild(txtDescription);

            //Create the numQty element and append it to
            //the detail element
            Element numQty = (Element)document.createElement("numQty");
            numQty.appendChild(document.createTextNode("1"));
            detail.appendChild(numQty);

            //Create the numUnitPrice element and append it
            //to the detail element
            Element numUnitPrice = (Element)document.createElement("numUnitPrice");
            numUnitPrice.appendChild(document.createTextNode("350.00"));
            detail.appendChild(numUnitPrice);

            //Create another detail element named detail2 and
            //append it to root
            Element detail2 = (Element)document.createElement("detail");
            root.appendChild(detail2);

            //Create the txtPartNum element and append it to the
            //detail2 element
            Element txtPartNum2 = (Element)document.createElement("txtPartNum");
            txtPartNum2.appendChild(document.createTextNode("00010-200"));
            detail2.appendChild(txtPartNum2);

            //Create the txtDescription element and append it
            //to the detail2 element
            Element txtDescription2 = (Element)document.createElement("txtDescription");
            txtDescription2.appendChild(document.createTextNode("Desk lamps"));
            detail2.appendChild(txtDescription2);

            //Create the numQty element and append it to the
            //detail2 element
            Element numQty2 = (Element)document.createElement("numQty");
            numQty2.appendChild(document.createTextNode("3"));
            detail2.appendChild(numQty2);

            //Create the NUMUNITPRICE element
            Element numUnitPrice2 = (Element)document.createElement("numUnitPrice");
            numUnitPrice2.appendChild(document.createTextNode("55.00"));
            detail2.appendChild(numUnitPrice2);
          }
     catch (Exception e) {
              System.out.println("The following exception occurred: "+e.getMessage());
           }
    return document;
```

```
    }


    //Converts an org.w3c.dom.Document object to a
    //com.adobe.idp.Document object
    private Document convertDataSource(org.w3c.dom.Document myDOM)
    {
       byte[] mybytes = null;

       try
       {
       //Create a Java Transformer object
       TransformerFactory transFact = TransformerFactory.newInstance();
       Transformer transForm = transFact.newTransformer();

       //Create a Java ByteArrayOutputStream object
       ByteArrayOutputStream myOutStream = new ByteArrayOutputStream();

       //Create a Java Source object
       javax.xml.transform.dom.DOMSource myInput = new DOMSource(myDOM);

       //Create a Java Result object
       javax.xml.transform.stream.StreamResult myOutput = new StreamResult(myOutStream);

       //Populate the Java ByteArrayOutputStream object
       transForm.transform(myInput,myOutput);

       // Get the size of the ByteArrayOutputStream buffer
        int myByteSize = myOutStream.size();

       //Allocate myByteSize to the byte array
        mybytes = new byte[myByteSize];

       //Copy the content to the byte array
        mybytes = myOutStream.toByteArray();
       }
       catch (Exception e) {
         System.out.println("The following exception occurred: "+e.getMessage());
     }

    //Create a com.adobe.idp.Document object and copy the
    //contents of the byte array
    Document myDocument = new Document(mybytes);
    return myDocument;
    }


    //Render the purchase order form using the specified
    //com.adobe.idp.Document object
    private void renderPOForm(HttpServletResponse resp, Document formData)
    {
       try{

           //Set connection properties required to invoke AEM Forms
           Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
```

```
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceC
lientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Set the parameter values for the renderPDFForm method
            String formName = "Applications/FormsApplication/1.0/FormsFolder/PO.xdp";

            //Cache the form
            PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
            pdfFormRenderSpec.setCacheEnabled(new Boolean(true));

            //Specify URI values that are required to render a form
            URLSpec uriValues = new URLSpec();
            uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
            uriValues.setContentRootURI("repository:///");
            uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");

            //Invoke the renderForm method
            FormsResult formOut = formsClient.renderPDFForm(
                formName,               //formQuery
                formData,       //inDataDoc
                pdfFormRenderSpec,      //PDFFormRenderSpec
                uriValues,      //urlSpec
                null                //attachments
                );

            //Create a ServletOutputStream object
```

```
            ServletOutputStream oOutput = resp.getOutputStream();

            //Create a Document object that stores form data
            Document myData = formOut.getOutputContent();

            //Create an InputStream object
            InputStream inputStream = myData.getInputStream();

            //Write the data stream to the web browser
            byte[] data = new byte[4096];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(data)) > 0)
            {
                oOutput.write(data, 0, bytesRead);
            }
            }catch (Exception e) {
                 System.out.println("The following exception occurred: "+e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Handling a form containing a calculation script using the Java API

The following code example processes a form that contains a calculation script and writes the results back to the client web browser. (See "Calculating Form Data" on page 656.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class CalculateData extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }


    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Get form data to pass to the processFormSubmission method
            Document formData = new Document(req.getInputStream());

            //Set run-time options
            RenderOptionsSpec processSpec = new RenderOptionsSpec();
```

```
            processSpec.setLocale("en_US");

            //Invoke the processFormSubmission method
            FormsResult formOut =
formsClient.processFormSubmission(formData,"CONTENT_TYPE=application/pdf&CONTENT_TYPE=applicati
on/vnd.adobe.xdp+xml","",processSpec);

            //Get the processing state
            short processState = formOut.getAction();

            //Determine if the form data is calculated
            if (processState == 1)
            {

                //Write the data back to to the client web browser
                ServletOutputStream oOutput = resp.getOutputStream();
                Document calData = formOut.getOutputContent();

                //Create an InputStream object
                InputStream inputStream = calData.getInputStream();

                //Write the data stream to the web browser
                byte[] data = new byte[4096];
                int bytesRead = 0;
                while ((bytesRead = inputStream.read(data)) > 0)
                {
                    oOutput.write(data, 0, bytesRead);
                }
             }
            }
        catch (Exception e) {
             System.out.println("The following exception occurred: "+e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Optimizing performance using the Java API

The following code example optimizes performance by setting the caching, standalone, and linearized options. A linearized file is optimized for delivery on the web. (See "Optimizing the Performance of the Forms Service" on page 666.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderFormsPerformance extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    try{
```

```
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a FormsServiceClient object
        FormsServiceClient formsClient = new FormsServiceClient(myFactory);

        //Set the parameter values for the renderForm method
        String formName = "Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";
        byte[]cData = "".getBytes();
        Document oInputData = new Document(cData);

        //Set performance run-time options
        PDFFormRenderSpec renderSpec = new PDFFormRenderSpec();
        renderSpec.setCacheEnabled(new Boolean(true));
        renderSpec.setLinearizedPDF(true);

        //Specify URI values that are required to render a form
        //design located in the AEM Forms Repository
        URLSpec uriValues = new URLSpec();
        uriValues.setApplicationWebRoot("http://[server]:[port]/FormsServiceClientApp");
        uriValues.setContentRootURI("repository:///");
        uriValues.setTargetURL("http://[server]:[port]/FormsServiceClientApp/HandleData");

        //Invoke the renderPDFForm method and write the
        //results to a client web browser
        FormsResult formOut = formsClient.renderPDFForm(
                    formName,       //formQuery
                    oInputData, //inDataDoc
                    renderSpec,     //PDFFormRenderSpec
                    uriValues,//urlSpec
                    null        //attachments
                    );

        //Create a ServletOutputStream object
        ServletOutputStream oOutput = resp.getOutputStream();
```

```
        //Create a Document object that stores form data
        Document myData = formOut.getOutputContent();

        //Create an InputStream object
        InputStream inputStream = myData.getInputStream();

        //Write the data stream to the web browser
        byte[] data = new byte[4096];
        int bytesRead = 0;
        while ((bytesRead = inputStream.read(data)) > 0)
        {
            oOutput.write(data, 0, bytesRead);
        }

        }catch (Exception e) {
             System.out.println("The following exception occurred: "+e.getMessage());
      }
    }
}
```

Quick Start (SOAP mode): Rendering by value using the Java API

The following Java quick start renders an interactive PDF form that is based on a form design named *Loan.xdp* by value. Notice that the form design is used to populate a `com.adobe.idp.Document` object named *inputXDP*. (See "Rendering Forms By Value" on page 661.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms.
    */
import java.io.FileInputStream;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderByValue extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Retrieve the form design
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xdp");
            Document inputXDP = new Document(fileInputStream);
```

```
            //Specify URI values that are required to render a form
            URLSpec uriValues = new URLSpec();
            uriValues.setApplicationWebRoot("http://[server]:[port]/FormsQS");
            uriValues.setTargetURL("http://[server]:[port]/FormsQS/HandleData");

            //Invoke the renderPDFForm method and pass the
            //form design by value
            FormsResult formOut = formsClient.renderPDFForm(
                        "",                 //formQuery
                        inputXDP,       //inDataDoc
                        new PDFFormRenderSpec(), //PDFFormRenderSpec
                        uriValues,      //urlSpec
                        null                //attachments
                        );

            //Create a Document object that stores form data
            Document myData = formOut.getOutputContent();

            //Get the content type of the response and
            //set the HttpServletResponse object?s content type
            String contentType = myData.getContentType();
            resp.setContentType(contentType);

            //Create a ServletOutputStream object
            ServletOutputStream oOutput = resp.getOutputStream();

            //Create an InputStream object
            InputStream inputStream = myData.getInputStream();

            //Write the data stream to the web browser
            byte[] data = new byte[4096];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(data)) > 0)
            {
                oOutput.write(data, 0, bytesRead);
            }

            }catch (Exception e) {
                 e.printStackTrace();
              }
        }
}
```

Quick Start (SOAP mode): Passing documents to the Forms Service using the Java API

The following Java quick start retrieves the file Loan.xdp from Content Services (deprecated). This XDP file is located in the space `/Company Home/Form Designs`. The XDP file is returned in a `com.adobe.idp.Document` instance. The `com.adobe.idp.Document` instance is passed to the Forms service. The interactive form is written to a client web browser. (See "Passing Documents to the Forms Service" on page 591.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-contentservices-client.jar
    * 3. adobe-livecycle-client.jar
    * 4. adobe-usermanager-client.jar
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms.
    */
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.adobe.livecycle.contentservices.client.CRCResult;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;
import com.adobe.livecycle.formsservice.client.*;

import java.util.*;
import java.io.InputStream;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderFormsFromContentServices extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
```

```
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Create an empty Document that represents form data
            byte[]cData = "".getBytes();
            Document oInputData = new Document(cData);

            //Get the form design from Content Services (deprecated)
            Document formDesign =  GetFormDesign(myFactory);

            //Cache the PDF form
            PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
            pdfFormRenderSpec.setCacheEnabled(new Boolean(true));

            //Invoke the renderPDFForm2 and pass to the
            //Document that contains the form design
            FormsResult formOut = formsClient.renderPDFForm2(
                    formDesign,
                    oInputData,
                    pdfFormRenderSpec,
                    null,
                    null
                    );

            //Create a Document object that stores form data
            Document myData = formOut.getOutputContent();

            //Get the content type of the response and
            //set the HttpServletResponse object?s content type
            String contentType = myData.getContentType();
            resp.setContentType(contentType);

            //Create a ServletOutputStream object
            ServletOutputStream oOutput = resp.getOutputStream();

            //Create an InputStream object
            InputStream inputStream = myData.getInputStream();

            //Write the data stream to the web browser
            byte[] data = new byte[4096];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(data)) > 0)
            {
                oOutput.write(data, 0, bytesRead);
            }

            }catch (Exception e) {
                 e.printStackTrace();
```

```
            }
        }

    //Retrieve the form design from Content Services (deprecated)
    private Document GetFormDesign(ServiceClientFactory myFactory)
    {
        try{

        //Create a DocumentManagementServiceClientImpl object
        DocumentManagementServiceClientImpldocManager = new
DocumentManagementServiceClientImpl(myFactory);

        //Specify the name of the store and the content to retrieve
           String storeName = "SpacesStore";
           String nodeName  = "/Company Home/Form Designs/Loan.xdp";

           //Retrieve /Company Home/Form Designs/Loan.xdp
           CRCResult content = docManager.retrieveContent(
                    storeName,
                    nodeName,
                    "");

           //Return the Document instance
            Document doc =content.getDocument();
            return  doc;
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
        return null;
    }

}
```

Form Data Integration Service Java API Quick Start(SOAP)

The following Quick Starts are available for the Form Data Integration service.

"Quick Start (SOAP mode): Importing form data using the Java API" on page 208

"Quick Start (SOAP mode): Exporting form data using the Java API" on page 210

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Start located in Programming with AEM forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Importing form data using the Java API

The following Java code example imports data into a PDF form. The data is located in an XML file named *Loan_data.xml* and the PDF form is saved as a PDF file named *ResultLoanForm.pdf*. (See "Importing Form Data" on page 737.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-formdataintegration-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
```

```
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.formdataintegration.client.*;

public class ImportDataSOAP {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

             //Create a ServiceClientFactory object
             ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormDataIntegrationClient object
            FormDataIntegrationClient dataClient = new FormDataIntegrationClient(myFactory);

            //Import XDP XML data into an XFA PDF document
            //Reference an XFA PDF form
            FileInputStream inputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inputPDF = new Document(inputStream);

            FileInputStream dataInput = new FileInputStream("C:\\Adobe\Loan_data.xml");
            Document inputDataFile = new Document(dataInput);

            //Import data into the form
            Document resultPDF = dataClient.importData(inputPDF,inputDataFile);

            //Save the PDF file
            File resultFile = new File("C:\\Adobe\ResultLoanForm.pdf");
            resultPDF.copyToFile(resultFile);

        }catch (Exception e) {
             e.printStackTrace();
        }
      }
    }
```

Quick Start (SOAP mode): Exporting form data using the Java API

The following Java code example exports data from a PDF form. The form data is saved as an XML file named *Loan_data.xml*. (See "Exporting Form Data" on page 742.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-formdataintegration-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
```

```
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.formdataintegration.client.*;

public class ExportDataSOAP {

    public static void main(String[] args) {

    try{
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

         //Create a ServiceClientFactory object
         ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

         //Create a FormDataIntegrationClient object
         FormDataIntegrationClient dataClient = new FormDataIntegrationClient(myFactory);

         //Reference a PDF form from which to export data
         FileInputStream fileInputStream2 = new FileInputStream("C:\\Adobe\LoanForm.pdf");
         Document inputPDF = new Document(fileInputStream2);

         //Export data from the form
         Document resultPDF = dataClient.exportData(inputPDF);

         //Save the exported form data as an XML file
         File resultFile = new File("C:\\Adobe\Loan_data.xml");
         resultPDF.copyToFile(resultFile);

    }catch (Exception e) {
              e.printStackTrace();
        }
    }
}
```

Generate PDF Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Generate PDF service.

"Quick Start (SOAP mode): Converting a Microsoft Word document to a PDF document using the Java API" on page 212

"Quick Start (SOAP mode): Converting HTML content to a PDF document using the Java API" on page 214

"Quick Start (SOAP mode): Converting a PDF document to an RTF file using the Java API (SOAP mode)" on page 216

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Start located in Programming with AEM Forms are based on the Forms Server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Converting a Microsoft Word document to a PDF document using the Java API

The following code example converts a Word file named *Loan.doc* to a PDF document named *Loan.pdf*. (See "Converting Word Documents to PDF Documents" on page 780.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-generatepdf-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
```

```
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.generatepdf.client.CreatePDFResult;
import com.adobe.livecycle.generatepdf.client.GeneratePdfServiceClient;

public class ConvertWordDocumentSOAP {

    public static void main(String[] args)
    {
        try{
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a GeneratePdfServiceClient object
        GeneratePdfServiceClient pdfGenClient = new GeneratePdfServiceClient(myFactory);

        //Get a Microsoft Word file document to convert to a PDF document
        String inputFileName = "C:\\Adobe\\Loan.doc";
        FileInputStream fileInputStream = new FileInputStream(inputFileName);
        Document inDoc = new Document(fileInputStream);

        //Set createPDF2 parameter values
        String adobePDFSettings = "Smallest_File_Size";
        String securitySettings = "No Security";
        String fileTypeSettings = "Filetype Settings";

        //Convert the Word document to a PDF document
```

```
        CreatePDFResult result = pdfGenClient.createPDF2(
            inDoc,
            inputFileName,
            fileTypeSettings,
            adobePDFSettings,
            securitySettings,
            null,
            null);

        //Get the newly created document
        Document createdDocument = result.getCreatedDocument();

        //Save the converted PDF document as a PDF file
        createdDocument.copyToFile(new File("C:\\Adobe\\Loan.pdf"));
    }
    catch (Exception e) {
        System.out.println("Error OCCURRED: " + e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Converting HTML content to a PDF document using the Java API

The following Java code example converts HTML content located at http://www.adobe.com to a PDF document named *AdobeHTML.pdf*. (See "Converting HTML Documents to PDF Documents" on page 784.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-generatepdf-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.generatepdf.client.GeneratePdfServiceClient;
import com.adobe.livecycle.generatepdf.client.HtmlToPdfResult;

public class ConvertHTMLSOAP {

    public static void main(String[] args)
    {
        try{
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a GeneratePdfServiceClient object
        GeneratePdfServiceClient pdfGenClient = new GeneratePdfServiceClient(myFactory);

        //Get an HTML document to convert to a PDF document a
        String inputFileName = "http://www.adobe.com";

        String securitySettings = "No Security";
        String fileTypeSettings = "Standard";
```

```
        //Convert HTML content to a PDF document
        HtmlToPdfResult result = pdfGenClient.htmlToPDF2(
                inputFileName,
                fileTypeSettings,
                securitySettings,
                null,
                null);

        //Get the newly created document
        Document createdDocument = result.getCreatedDocument();

        //Save the PDF document as a PDF file
        createdDocument.copyToFile(new File("C:\\AdobeHTML.pdf"));
    }
    catch (Exception e) {
        System.out.println("Error OCCURRED: " + e.getMessage());
    }
    }
}
```

Quick Start (SOAP mode): Converting a PDF document to an RTF file using the Java API (SOAP mode)

The following code example converts a PDF document named *Loan.pdf* to an RTF document named *Loan.rtf*. (See "Converting PDF Documents to Non-image Formats" on page 787.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-generatepdf-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.generatepdf.client.ConvertPDFFormatType;
import com.adobe.livecycle.generatepdf.client.ExportPDFResult;
import com.adobe.livecycle.generatepdf.client.GeneratePdfServiceClient;


public class GeneratePdf_ExportPDFSOAP {

    public static void main(String[] args)
    {
        try{
            //Set connection properties required to invoke AEM Forms using SOAP mode
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a GeneratePdfServiceClient object
            GeneratePdfServiceClient pdfGenClient = new GeneratePdfServiceClient(factory);

            //Get a PDF document to convert to an RTF document
```

```
            String inputFileName = "C:\\Adobe\\Loan.pdf.pdf";
            FileInputStream fileInputStream = new FileInputStream(inputFileName);
            Document inDoc = new Document(fileInputStream);

            //Convert a PDF document to a RTF document
            ExportPDFResult result = pdfGenClient.exportPDF2(
                inDoc,
                inputFileName,
                ConvertPDFFormatType.RTF,
                null);

        //Get the newly created RTF document
        Document createdDocument = result.getConvertedDocument();

        //Save the RTF file
        createdDocument.copyToFile(new File("C:\\Adobe\\Loan.pdf.rtf"));
        }
        catch (Exception e) {
            System.out.println("Error OCCURRED: " + e.getMessage());
        }
    }
}
```

Invocation API Quick Starts

The following Quick Starts are available for programmatically invoking AEM Forms services:

Description	Remoting API	Java API	Web service API
"Invoking Human-Centric Long-Lived Processes" on page 560	"Invoking a long-lived process using Remoting" on page 577	"Quick Start: Invoking a long-lived process using the Invocation API" on page 564	"Quick Start: Invoking a long-lived process using the web service API" on page 572
"Invoking a short-lived process using the Invocation API" on page 512	N/A	"Quick Start: Invoking a short-lived process using the Invocation API" on page 219	N/A
"Invoking AEM Forms using Base64 encoding" on page 525 (Java web service proxy)	N/A	N/A	"Quick Start: Invoking a service using Java proxy files and Base64 encoding" on page 223
"Invoking AEM Forms using Base64 encoding" on page 525 (.NET web service proxy)	N/A	N/A	"Quick Start: Invoking a service using base64 in a Microsoft .NET project" on page 222
"Invoking AEM Forms using MTOM" on page 529 (.NET web service example)	N/A	N/A	"Quick Start: Invoking a service using MTOM in a .NET project" on page 236
"Invoking AEM Forms using SwaRef" on page 531 (Java web service example)	N/A	N/A	"Quick Start: Invoking a service using SwaRef in a Java project" on page 238

Description	Remoting API	Java API	Web service API
"Invoking AEM Forms using BLOB data over HTTP" on page 533 (Java web service example)	N/A	N/A	"Quick Start: Invoking a service using BLOB data over HTTP in a .NET project" on page 234
"Invoking AEM Forms using BLOB data over HTTP" on page 533 (.NET web service example)	N/A	N/A	"Quick Start: Invoking a service using BLOB data over HTTP in a Java project" on page 232
"Invoking AEM Forms using DIME" on page 536 (Java web service example)	N/A	N/A	"Quick Start: Invoking a service using DIME in a Java project" on page 230
"Invoking AEM Forms using Remoting" on page 444	"Quick Start: Invoking a short-lived process by passing an unsecure document using (Deprecated for AEM forms) AEM Forms Remoting" on page 225	N/A	N/A
"Passing secure documents to invoke processes using Remoting" on page 456	"Quick Start: Invoking a short-lived process by passing a secure document using Remoting" on page 458	N/A	N/A
"Invoking custom component services using Remoting" on page 463	"Quick Start: Invoking the Customer custom service using Remoting" on page 466	N/A	N/A

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM forms are based on Forms server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start: Invoking a short-lived process using the Invocation API

The following Java code example invokes a short-lived process named `MyApplication/EncryptDocument`. Notice that this process is invoked synchronously. The input parameter for this process is named `inDoc`. The output parameter for this process is named `outDoc`. The password encrypted PDF document is saved as a PDF file named `EncryptLoan.pdf`. (See "Invoking a short-lived process using the Invocation API" on page 512.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-convertpdf-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 7. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.InvocationRequest;
import com.adobe.idp.dsc.InvocationResponse;
import com.adobe.idp.dsc.clientsdk.ServiceClient;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


    public class InvokeDocumentEncryptLooselyTypedAPI {
```

```
        public static void main(String[] args)
        {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ServiceClient object
            ServiceClient myServiceClient = factory.getServiceClient();

            //Create a Map object to store the parameter value
            Map params = new HashMap();

            InputStream inFile = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document(inFile);

            //Populate the Map object with a parameter value
            //required to invoke the MyApplication/EncryptDocument short-lived process
            //inDoc refers to the name of the input parameter for the process
            params.put("inDoc", inDoc);

            //Create an InvocationRequest object
            InvocationRequest request =  factory.createInvocationRequest(
```

```
                    "MyApplication/EncryptDocument",    //Specify the short-lived process name
                    "invoke",       //Specify the operation name
                    params,         //Specify input values
                    true);          //Create a synchronous request

            //Send the invocation request to the short-lived process and
            //get back an invocation response -- outDoc refers to the output parameter for the
            //MyApplication/EncryptDocument process
            InvocationResponse response = myServiceClient.invoke(request);
            Document encryptDoc = (Document) response.getOutputParameter("outDoc");

            //Save the encrypted PDF document returned by the process
            //Save the password-encrypted PDF document
            File outFile = new File("C:\\Adobe\EncryptLoan.pdf");
            encryptDoc.copyToFile (outFile);
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

Quick Start: Invoking a service using base64 in a Microsoft .NET project

The following C# code example invokes a process named `MyApplication/EncryptDocument` from a Microsoft .NET project using Base64 encoding. (See "Invoking AEM Forms using Base64 encoding" on page 525.)

An unsecured PDF document based on a PDF file named *Loan.pdf* is passed to the AEM Forms process. The process returns a password-encrypted PDF document that is saved as a PDF file named *EncryptedPDF.pdf*.

```
/*
    * Ensure that you create a .NET client assembly that uses
    * base64 encoding. This is required to populate a BLOB
    * object with data or retrieve data from a BLOB object.
    *
    * For information, see "Invoking AEM Forms using Base64 Encoding" in
    * Programming with AEM forms
    */
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;

namespace InvokeEncryptDocumentBase64
{

       class InvokeEncryptDocumentUsingBase64
       {

           const int BUFFER_SIZE = 4096;
           [STAThread]
           static void Main(string[] args)
           {

               try
               {
                   String pdfFile = "C:\\Adobe\Loan.pdf";
```

```
                String encryptedPDF = "C:\\Adobe\EncryptedPDF.pdf";


            //Create an MyApplication_EncryptDocumentService object and set authentication
values
                MyApplication2_EncryptDocumentService encryptClient = new
MyApplication2_EncryptDocumentService();
             encryptClient.Credentials = new System.Net.NetworkCredential("administrator",
"password");

                //Reference the PDF file to send to the EncryptDocument process
                FileStream fs = new FileStream(pdfFile, FileMode.Open);

                //Create a BLOB object
                BLOB inDoc = new BLOB();

                //Get the length of the file stream
                int len = (int)fs.Length;
                byte[] ByteArray = new byte[len];

                //Populate the byte array with the contents of the FileStream object
                fs.Read(ByteArray, 0, len);
                inDoc.binaryData = ByteArray;

                //Invoke the EncryptDocument process
                BLOB outDoc = encryptClient.invoke(inDoc);

                //Populate a byte array with BLOB data
                byte[] outByteArray = outDoc.binaryData;

                //Create a new file named UsageRightsLoan.pdf
                FileStream fs2 = new FileStream(encryptedPDF, FileMode.OpenOrCreate);

                //Create a BinaryWriter object
                BinaryWriter w = new BinaryWriter(fs2);
                w.Write(outByteArray);
                w.Close();
                fs2.Close();
             }
            catch (Exception ee)
            {
                Console.WriteLine(ee.Message);
            }
        }
    }
}
```

Quick Start: Invoking a service using Java proxy files and Base64 encoding

The following Java code example invokes a process named `MyApplication/EncryptDocument` using Java proxy files created using JAX-WS and Base64 encoding. (See "Invoking AEM Forms using Base64 encoding" on page 525.)

An unsecured PDF document based on a PDF file named *Loan.pdf* is passed to the AEM Forms process. The process returns a password-encrypted PDF document that is saved as a PDF file named *EncryptedDocument.pdf*.

```
/**
    * Ensure that you create Java proxy files that consume
    * the the AEM Forms service WSDL. You can use JAX-WS to create
    * the Java proxy files.
    *
    * This Java quick start uses Base64 to invoke a short-lived process named
    * EncryptDocument. For information, see
    * "Invoking AEM Forms using Base64" in Programming with AEM forms.
    */

import java.io.*;

import javax.xml.ws.BindingProvider;
import com.adobe.idp.services.*;

public class InvokeEncryptDocumentBase64 {
    public static void main(String[] args){

        try{
            //Create a MyApplicationEncryptDocument object
            MyApplicationEncryptDocumentService encClient = new
MyApplicationEncryptDocumentService();
            MyApplicationEncryptDocument encryptDocClient = encClient.getEncryptDocument();

            //Set connection values required to invoke AEM Forms
            String url =
"[server]:[port]/soap/services/MyApplication/EncryptDocument?blob=base64";
            String username = "administrator";
            String password = "password";
            ((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
            ((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.USERNAME_PROPERTY, username);
            ((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, password);

               // Get the input PDF document to send to the EncryptDocument process
               BLOB inDoc = new BLOB();

            // Get the input DDX document and input PDF sources
            File fileName = new File("C:\\Adobe\Loan.pdf");
            FileInputStream inFs = new FileInputStream(fileName);

            // Get the length of the file stream and create a byte array
            int inLen = (int)fileName.length();
            byte[] inByteArray = new byte[inLen];

               // Populate the byte array with the content of the file stream
            inFs.read(inByteArray, 0, inLen);

               // Populate the BLOB objects
            inDoc.setBinaryData(inByteArray);

               //invoke the short-lived process named MyApplication/EncryptDocument
            BLOB outDoc = encryptDocClient.invoke(inDoc);

            //Save the encrypted file as a PDF file
```

```
            byte[] encryptedDocument = outDoc.getBinaryData();

            //Create a File object
            File outFile = new File("C:\\Adobe\EncryptedDocument.pdf");

            //Create a FileOutputStream object.
            FileOutputStream myFileW = new FileOutputStream(outFile);

            //Call the FileOutputStream object's write method and pass the pdf data
            myFileW.write(encryptedDocument);

            //Close the FileOutputStream object
            myFileW.close();
              System.out.println("The short-lived process named MyApplication/EncryptDocument
was successfully invoked.");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

    }

}
```

Quick Start: Invoking a short-lived process by passing an unsecure document using (Deprecated for AEM forms) AEM Forms Remoting

The following Flex code example invokes a short-lived process named `MyApplication/EncryptDocument`. (See "Invoking AEM Forms using Remoting" on page 444.)

Note: *This quick start invokes a AEM Forms process and uploads an unsecure document. To execute this quick start, AEM Forms has to be configured to upload unsecure documents. For information about how to configure AEM Forms to accept unsecure documents, see "Configuring AEM Forms to accept secure and unsecure documents" on page 458.*

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
     creationComplete="initializeChannelSet();">
     <mx:Script>
       <![CDATA[

     import mx.rpc.AEM Forms.DocumentReference;
     import flash.net.FileReference;
     import flash.net.URLRequest;
     import flash.events.Event;
     import flash.events.DataEvent;
     import mx.messaging.ChannelSet;
     import mx.messaging.channels.AMFChannel;
     import mx.rpc.events.ResultEvent;
     import mx.collections.ArrayCollection;
     import mx.rpc.AsyncToken;

      // Classes used in file retrieval
     private var fileRef:FileReference = new FileReference();
     private var docRef:DocumentReference = new DocumentReference();
     private var parentResourcePath:String = "/";
     private var serverPort:String = "[server]:[port]";
     private var now1:Date;
     private  var cs:ChannelSet

     // Holds information returned from AEM Forms
     [Bindable]
     public var progressList:ArrayCollection = new ArrayCollection();

     // Set up channel set to invoke AEM Forms.
     // This must be done before calling any service or process, but only
     // once for the entire application.
      private function initializeChannelSet():void {
       cs = new ChannelSet();
       cs.addChannel(new AMFChannel("remoting-amf", "http://" + serverPort +
"/remoting/messagebroker/amf"));
       EncryptDocument.setCredentials("administrator", "password");
       EncryptDocument.channelSet = cs;
     }

     // Call this method to upload the file.
     // This creates a file picker and lets the user select a PDF file to pass to the
EncryptDocument process.
     private function uploadFile():void {
       fileRef.addEventListener(Event.SELECT, selectHandler);
       fileRef.browse();
     }

      private function selectHandler(event:Event):void
            {
            var authTokenService:RemoteObject = new
RemoteObject("LC.FileUploadAuthenticator");
            authTokenService.addEventListener("result", authTokenReceived);
            authTokenService.channelSet = cs;
            authTokenService.getFileUploadToken();
            }
```

```
    private function authTokenReceived(event:ResultEvent):void
            {
            var token:String = event.result as String;
            var request:URLRequest =
DocumentReference.constructRequestForUpload("http://[server]:[port]", token);

            try
            {
         fileRef.upload(request);
            }
            catch (error:Error)
            {
            trace("Unable to upload file.");
            }
            }


     // Called once the file is completely uploaded.
     private function completeHandler(event:DataEvent):void {
       now1 = new Date();

       // Set the docRefs url and referenceType parameters
       docRef.url = event.data as String;
       docRef.referenceType=DocumentReference.REF_TYPE_URL;
       executeInvokeProcess();
     }

     //This method invokes the EncryptDocument process
     public function executeInvokeProcess():void {

        //Create an Object to store the input value for the EncryptDocument process
        var params:Object = new Object();
        params["inDoc"]=docRef;

        // Invoke the EncryptDocument process
        var token:AsyncToken;
        token = EncryptDocument.invoke(params);
        token.name = name;
     }

     // This method handles a successful conversion invocation
     public function handleResult(event:ResultEvent):void
     {
          //Retrieve information returned from the service invocation
         var token:AsyncToken = event.token;
         var res:Object = event.result;
         var dr:DocumentReference = res["outDoc"] as DocumentReference;
         var now2:Date = new Date();

         // These fields map to columns in the DataGrid
         var progObject:Object = new Object();
         progObject.filename = token.name;
         progObject.timing = (now2.time - now1.time).toString();
         progObject.state = "Success";
         progObject.link = "<a href=" + dr.url + "> open </a>";
         progressList.addItem(progObject);
     }
```

```
     private function resultHandler(event:ResultEvent):void {
       // Do anything else here.
     }

       ]]>
     </mx:Script>
     <mx:RemoteObject id="EncryptDocument" destination="MyApplication/EncryptDocument"
result="resultHandler(event);">
          <mx:method name="invoke" result="handleResult(event)"/>
     </mx:RemoteObject>

     <!--//This consists of what is displayed on the webpage-->
     <mx:Panel id="lcPanel" title="EncryptDocument  (Deprecated for AEM forms) AEM Forms
Remoting Example"
          height="25%" width="25%" paddingTop="10" paddingLeft="10" paddingRight="10"
          paddingBottom="10">
        <mx:Label width="100%" color="blue"
               text="Select a PDF file to pass to the EncryptDocument process"/>
       <mx:DataGrid x="10" y="0" width="500" id="idProgress" editable="false"
          dataProvider="{progressList}" height="231" selectable="false" >
         <mx:columns>
           <mx:DataGridColumn headerText="Filename" width="200" dataField="filename"
editable="false"/>
          <mx:DataGridColumn headerText="State" width="75" dataField="state" editable="false"/>
           <mx:DataGridColumn headerText="Timing" width="75" dataField="timing"
editable="false"/>
           <mx:DataGridColumn headerText="Click to Open" dataField="link" editable="false" >
            <mx:itemRenderer>
               <mx:Component>
                  <mx:Text x="0" y="0" width="100%" htmlText="{data.link}"/>
               </mx:Component>
            </mx:itemRenderer>
           </mx:DataGridColumn>
         </mx:columns>
       </mx:DataGrid>
       <mx:Button label="Select File" click="uploadFile()" />
     </mx:Panel>
</mx:Application>
```

Quick Start: Invoking a service using DIME in a .NET project

The following C# code example invokes a process named `MyApplication/EncryptDocument` from a Microsoft .NET project using Dime. (See "Invoking AEM Forms using Base64 encoding" on page 525.)

An unsecured PDF document based on a PDF file named *map.pdf* is passed to the AEM Forms process using DIME. The process returns a password-encrypted PDF document that is saved as a PDF file named *mapEncrypt.pdf*.

```
/**
    *
    * Ensure that you create a .NET project that uses
    * Web Services Enhancements 2.0. This is required to send a
    * AEM Forms process an attachment using DIME.
    *
    * For information, see "Invoking AEM Forms using DIME" in Programming with AEM forms.
    */

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;
using Microsoft.Web.Services2.Dime;
using Microsoft.Web.Services2.Attachments;
using Microsoft.Web.Services2.Configuration;
using Microsoft.Web.Services2;

//The following statement represents a web reference to
//the forms server that contains the process that
//is invoked
using ConsoleApplication1.LC_Host;

namespace ConsoleApplication1
{

     class InvokeEncryptDocumentUsingDime
       {

           const int BUFFER_SIZE = 4096;
           [STAThread]
           static void Main(string[] args)
           {

           try
              {
                   String pdfFile = "C:\\Adobe\map.pdf";
                   String encryptedPDF = "C:\\Adobe\mapEncrypt.pdf";

                   //Create an EncryptDocumentServiceWse object and set authentication values
                   EncryptDocumentServiceWse encryptClient = new EncryptDocumentServiceWse();
                  encryptClient.Credentials = new System.Net.NetworkCredential("administrator",
"password");

                   // Create the DIME attachment representing a PDF document
                   DimeAttachment inputDocAttachment = new DimeAttachment(
                       System.Guid.NewGuid().ToString(),
                       "application/pdf",
                       TypeFormat.MediaType,
                       pdfFile);

                    //Create a BLOB object
                    BLOB inDoc = new BLOB();

                    //Set the DIME attachment ID
                    inDoc.attachmentID = inputDocAttachment.Id;
```

```
                    encryptClient.RequestSoapContext.Attachments.Add(inputDocAttachment);

                    //Invoke the EncryptDocument process
                    BLOB outDoc = encryptClient.invoke(inDoc);

                    //Get the returned attachment identifier value
                    String encryptedDocId = outDoc.attachmentID;
                    FileStream myStream = new FileStream(encryptedPDF, FileMode.Create,
FileAccess.Write);

                    //Iterate through the attachments
                foreach (Attachment attachment in encryptClient.ResponseSoapContext.Attachments)
                      {
                        if (attachment.Id.Equals(encryptedDocId))
                          {
                            //Create a byte array that contains the encrypted PDF document
                            System.IO.Stream mySteam2 = attachment.Stream;
                            byte[] myBytes = new byte[mySteam2.Length];
                            int size = (int)mySteam2.Length;
                            mySteam2.Read(myBytes, 0, size);

                            //Save the encrypted PDF document as a PDF file
                           FileStream fs2 = new FileStream(encryptedPDF, FileMode.OpenOrCreate);

                            //Create a BinaryWriter object
                            BinaryWriter w = new BinaryWriter(fs2);
                            w.Write(myBytes);
                            w.Close();
                            fs2.Close();
                           Console.Out.WriteLine("Saved converted document at:" + encryptedPDF);
                          }
                    }
                 }
                 catch (Exception ee)
                {
                    Console.WriteLine(ee.Message);
                 }
            }
        }
}
```

Quick Start: Invoking a service using DIME in a Java project

The following Java code example invokes a process named `MyApplication/EncryptDocument` using DIME. (See "Invoking AEM Forms using DIME" on page 536.)

An unsecured PDF document based on a PDF file named *Loan.pdf* is passed to the AEM Forms process using DIME. The process returns a password-encrypted PDF document that is saved as a PDF file named *EncryptLoan.pdf*.

```
/**
    * Ensure that you create Java Axis files that
    * are required to send a AEM Forms process
    * an attachment using DIME.
    *
    * For information, see "Invoking AEM Forms using DIME" in Programming with AEM forms.
    */
import com.adobe.idp.services.*;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.URL;
import javax.activation.DataHandler;
import javax.activation.FileDataSource;

import org.apache.axis.attachments.AttachmentPart;

public class InvokeDocumentEncryptDime {
    public static void main(String[] args) {

    try{

        //Create a MyApplicationEncryptDocumentServiceLocator object
        MyApplicationEncryptDocumentServiceLocator locate = new
MyApplicationEncryptDocumentServiceLocator ();

        //specify the service target URL and object type
        URL serviceURL = new
URL("http://[server]:[port]/soap/services/MyApplication/EncryptDocument?blob=dime");

        //Use the binding stub with the locator
        EncryptDocumentSoapBindingStub encryptionClientStub = new
EncryptDocumentSoapBindingStub(serviceURL,locate);
        encryptionClientStub.setUsername("administrator");
        encryptionClientStub.setPassword("password");

        //Get the DIME Attachments - which is the PDF document to encrypt
        java.io.File file = new java.io.File("C:\\Adobe\Loan.pdf");

        //Create a DataHandler object
        DataHandler buildFile = new DataHandler(new FileDataSource(file));

        //Use the DataHandler object to create an AttachmentPart object
        AttachmentPart part = new AttachmentPart(buildFile);
        //get the attachment ID
        String attachmentID = part.getContentId();

        //Add the attachment to the encryption service stub
        encryptionClientStub.addAttachment(part);

        //Inform ES where the attachment is stored by providing the attachment id
        BLOB inDoc = new BLOB();
        inDoc.setAttachmentID(attachmentID);

        BLOB outDoc = encryptionClientStub.invoke(inDoc);

        //Go through the returned attachments and get the encrypted PDF document
```

```
        byte[] resultByte = null;
        attachmentID = outDoc.getAttachmentID();

        //Find the proper attachment
        Object[] parts = encryptionClientStub.getAttachments();
        for (int i=0;i<parts.length;i++){
            AttachmentPart attPart = (AttachmentPart)  parts[i];
            if (attPart.getContentId().equals(attachmentID)) {
                //DataHandler
                buildFile = attPart.getDataHandler();
                InputStream stream = buildFile.getInputStream();

                byte[] pdfStream = new byte[stream.available()];
                stream.read(pdfStream);

                //Create a File object
                File outFile = new File("C:\\Adobe\EncryptLoan.pdf");

                //Create a FileOutputStream object.
                FileOutputStream myFileW = new FileOutputStream(outFile);

                //Call the FileOutputStream object?s write method and pass the pdf data
                myFileW.write(pdfStream);

                //Close the FileOutputStream object
                myFileW.close();

            }
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    }

}
```

Quick Start: Invoking a service using BLOB data over HTTP in a Java project

The following Java code example invokes a process named `MyApplication/EncryptDocument` using data over HTTP. (See "Invoking AEM Forms using BLOB data over HTTP" on page 533.)

An unsecured PDF document based on a PDF file named *Loan.pdf* is passed to the AEM Forms process using SOAP over HTTP. The PDF file is located at the following URL: `http://[server]:[port]/FormsQS`. The process returns a password-encrypted PDF document that is saved as a PDF file named *EncryptedDocument.pdf*.

```
/**
    * Ensure that you create Java proxy files that consume
    * the the AEM Forms service WSDL. You can use JAX-WS to create
    * the Java proxy files.
    *
    * This Java quick start uses BLOB over HTTP to invoke a short-lived process named
    * EncryptDocument. For information, see
    * "Invoking AEM Forms using BLOB over HTTP" in Programming with AEM forms.
    */
import java.io.*;
import java.net.URL;

import javax.xml.ws.BindingProvider;
import com.adobe.idp.services.*;

public class InvokeEncryptDocumentHTTP {
    public static void main(String[] args){

        try{
            //Create a MyApplicationEncryptDocument object
            MyApplicationEncryptDocumentService encClient = new
MyApplicationEncryptDocumentService();
            MyApplicationEncryptDocument encryptDocClient = encClient.getEncryptDocument();

            //Set connection values required to invoke AEM Forms using BLOB over HTTP
            String url =
"http://[server]:[port]/soap/services/MyApplication/EncryptDocument?blob=http";
            String username = "administrator";
            String password = "password";
            ((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
            ((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.USERNAME_PROPERTY, username);
            ((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, password);

            //Create a BLOB object and populate it by invoking the setRemoteURL method
            BLOB inDoc = new BLOB();
            inDoc.setRemoteURL("http://[server]:[port]/FormsQS/Loan.pdf");

               //invoke the short-lived process named MyApplication/EncryptDocument
            BLOB outDoc = encryptDocClient.invoke(inDoc);

            //Retrieve an InputStream from the returned BLOB instance
            URL myURL = new URL(outDoc.getRemoteURL());
            InputStream inputStream = myURL.openStream();

            //Create a new file containing the returned PDF document
            File f = new File("C:\\Adobe\EncryptedDocument.pdf");
            OutputStream out = new FileOutputStream(f);

            //Iterate through the buffer
```

```
            byte buf[] = new byte[1024];
            int len;
            while ((len = inputStream.read(buf)) > 0)
                out.write(buf, 0, len);
            out.close();
            inputStream.close();

             System.out.println("The short-lived process named EncryptDocument was
successfully invoked.");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

    }

}
```

Quick Start: Invoking a service using BLOB data over HTTP in a .NET project

The following C# code example invokes a process named `MyApplication/EncryptDocument` from a Microsoft .NET project using data over HTTP. (See "Invoking AEM Forms using BLOB data over HTTP" on page 533.)

An unsecured PDF document based on a PDF file named *Loan.pdf* is passed to the AEM Forms process using BLOB over HTTP. The process returns a password-encrypted PDF document that is saved as a PDF file named *EncryptedPDF.pdf*.

```
/*
    * Ensure that you create a .NET client assembly that uses
    * SOAP over HTTP. This is required to populate a BLOB
    * object's remote URL data memeber.
    *
    * For information, see "Invoking AEM Forms using BLOB data over HTTP" in
    * Programming with AEM forms
    */
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;
using System.Security.Policy;

namespace InvokeEncryptDocumentHTTP
{

       class InvokeEncryptDocumentUsingHTTP
       {

           const int BUFFER_SIZE = 4096;
           [STAThread]
           static void Main(string[] args)
           {
```

```
            try
            {
                String urlData = "http://[server]:[port]/FormsQS/Loan.pdf";

             //Create a MyApplication_EncryptDocumentService object and set authentication
values
                MyApplication_EncryptDocumentService encryptClient = new
MyApplication_EncryptDocumentService();
             encryptClient.Credentials = new System.Net.NetworkCredential("administrator",
"password");

                //Create a BLOB object
                BLOB inDoc = new BLOB();

                //Populate the BLOB object's remoteURL data member
                inDoc.remoteURL = urlData;

                //Invoke the EncryptDocument process
                BLOB outDoc = encryptClient.invoke(inDoc);

                //Create a UriBuilder object using the
                //BLOB object's remoteURL data member field
                UriBuilder uri = new UriBuilder(outDoc.remoteURL);

               //Convert the UriBuilder to a Stream object
                System.Net.WebRequest wr = System.Net.WebRequest.Create(uri.Uri);
                System.Net.WebResponse response = wr.GetResponse();
                System.IO.StreamReader sr = new
System.IO.StreamReader(response.GetResponseStream());
                Stream mySteam = sr.BaseStream;

                //Create a byte array
                byte[] myData = new byte[BUFFER_SIZE];

               //Populate the byte array
               PopulateArray(mySteam, myData);

               //Create a new file named UsageRightsLoan.pdf
               FileStream fs2 = new FileStream("C:\\Adobe\EncryptedPDF.pdf",
FileMode.OpenOrCreate);

               //Create a BinaryWriter object
               BinaryWriter w = new BinaryWriter(fs2);
               w.Write(myData);
               w.Close();
               fs2.Close();
            }
            catch (Exception ee)
            {
                Console.WriteLine(ee.Message);
```

```
            }
        }

        public static void PopulateArray(Stream stream, byte[] data)
        {
            int offset = 0;
            int remaining = data.Length;
            while (remaining > 0)
            {
                int read = stream.Read(data, offset, remaining);
                if (read <= 0)
                    throw new EndOfStreamException();
                remaining -= read;
                offset += read;
            }
        }

    }
}
```

Quick Start: Invoking a service using MTOM in a .NET project

The following C# code example invokes a process named `MyApplication/EncryptDocument` from a Microsoft .NET project using MTOM. (See "Invoking AEM Forms using MTOM" on page 529.)

An unsecured PDF document based on a PDF file named *loan.pdf* is passed to the AEM Forms process using MTOM. The process returns a password-encrypted PDF document that is saved as a PDF file named *EncryptedDocument.pdf*.

```
???/**
    * Ensure that you create a .NET project that uses
    * MS Visual Studio 2008 and version 3.5 of the .NET
    * framework. This is required to invoke a
    * AEM Forms service using MTOM.
    *
    * For information, see "Invoking AEM Forms using MTOM" in Programming with AEM forms
    */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using EncryptDocumentMTOM.ServiceReference1;
using System.IO;

//Invoke the EncryptDocument process using MTOM
namespace EncryptDocumentUsingMTOM
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                //Specify the name of the PDF file to encrypt
                String pdfFile = "C:\\Adobe\loan.pdf";
```

```
                    //Create an EncryptDocumentClient object
                    MyApplication_EncryptDocumentClient encryptProcess = new
MyApplication_EncryptDocumentClient();
                    encryptProcess.Endpoint.Address = new
System.ServiceModel.EndpointAddress("http://[server]:[port]/soap/services/MyApplication/Encr
yptDocument?blob=mtom");
                    BasicHttpBinding b = (BasicHttpBinding)encryptProcess.Endpoint.Binding;
                    b.MessageEncoding = WSMessageEncoding.Mtom;

                    //Enable BASIC HTTP authentication
                    encryptProcess.ClientCredentials.UserName.UserName = "administrator";
                    encryptProcess.ClientCredentials.UserName.Password = "password";
                  b.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
                    b.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
                    b.MaxReceivedMessageSize = 4000000;
                    b.MaxBufferSize = 4000000;
                    b.ReaderQuotas.MaxArrayLength = 4000000;

                    //Reference the PDF file to send to the EncryptDocument process
                    FileStream fs = new FileStream(pdfFile, FileMode.Open);

                    //Create a BLOB object
                    BLOB inDoc = new BLOB();

                    //Get the length of the file stream
                    int len = (int)fs.Length;
                    byte[] ByteArray = new byte[len];

                    //Populate the byte array with the contents of the FileStream object
                    fs.Read(ByteArray, 0, len);
                    inDoc.MTOM = ByteArray;

                    //Invoke the EncryptDocument short-lived process
                    BLOB outDoc = encryptProcess.invoke(inDoc);
                    byte[] encryptDoc = outDoc.MTOM;

                    //Create a new file containing the encrypted PDF document
                    string FILE_NAME = "C:\\Adobe\EncryptedDocument.pdf";
                    FileStream fs2 = new FileStream(FILE_NAME, FileMode.OpenOrCreate);
                    BinaryWriter w = new BinaryWriter(fs2);
                    w.Write(encryptDoc);
                    w.Close();
                    fs2.Close();
                }
                catch (Exception ee)
                {
                    Console.WriteLine(ee.Message);
                }
            }
        }
}
```

Note: *Many quick starts that show how to perform AEM Forms service operations include a MTOM code example.*

Quick Start: Invoking a service using SwaRef in a Java project

The following Java code example invokes a process named `MyApplication/EncryptDocument` from a Java project. This Java project using proxy classes that were created using JAX-WS and SwaRef as the encoding type. (See "Invoking AEM Forms using SwaRef" on page 531.)

An unsecured PDF document based on a PDF file named *Loan.pdf* is passed to the AEM Forms process using SwaRef. The encrypted PDF document is saved as a PDF file named *EncryptedDocument.pdf*.

```
/**
    * Ensure that you create Java proxy files that consume
    * the the AEM Forms service WSDL. You can use JAX-WS to create
    * the Java proxy files.
    *
    * This Java quick start uses SwaRef to invoke a short-lived process named
    * EncryptDocument. For information, see
    * "Invoking AEM Forms using SwaRef" in Programming with AEM forms.
    */

import javax.xml.ws.BindingProvider;
import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;
import java.io.*;

import com.adobe.idp.services.*;

public class InvokeEncryptDocumentSwaRef {

public static void main(String[] args) {

        try{

        //Specify connection values required to invoke the MyApplication/EncryptDocument process
        //using SwaRef
        String url =
"http://[server]:[port]/soap/services/MyApplication/EncryptDocument?blob=swaref";
        String username = "administrator";
        String password = "password";
        String pdfFile = "C:\\Adobe\Loan.pdf";

        //Create a MyApplicationEncryptDocument object
        MyApplicationEncryptDocumentService encClient = new
MyApplicationEncryptDocumentService();
        MyApplicationEncryptDocument encryptDocClient = encClient.getEncryptDocument();


((BindingProvider)encryptDocClient).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_P
ROPERTY, url);

((BindingProvider)encryptDocClient).getRequestContext().put(BindingProvider.USERNAME_PROPERTY,
username);

((BindingProvider)encryptDocClient).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY,
password);

        //Create a file object
```

```
        File pdf = new File(pdfFile);

        //Create a DataSource object
        DataSource myDS = new FileDataSource(pdf);

        //Create a DataHandler object
        DataHandler dataHandler = new DataHandler(myDS);

        //Create a BLOB object and populate it with the DataHandler
        BLOB inDoc = new BLOB();
        inDoc.setSwaRef(dataHandler);

        //Invoke the EncryptDocument process
        BLOB outDoc = encryptDocClient.invoke(inDoc);

        //Save the encrypted file as a PDF file
        DataHandler handler = outDoc.getSwaRef();

        //Create a new file containing the returned PDF document
        File f = new File("C:\\Adobe\EncryptedDocument.pdf");
        InputStream inputStream = handler.getInputStream();
        OutputStream out = new FileOutputStream(f);

        //Iterate through the buffer
        byte buf[] = new byte[1024];
        int len;
        while ((len = inputStream.read(buf)) > 0)
            out.write(buf, 0, len);
        out.close();
        inputStream.close();

         System.out.println("The short-lived process named MyApplication/EncryptDocument was
successfully invoked.");

        }catch (Exception e) {
             e.printStackTrace();
            }
        }
}
```

Note: *Many quick starts that show how to perform service operations include a SwaRef code example.*

Output Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Output service.

"Quick Start (SOAP mode): Creating a PDF document using the Java API" on page 240

"Quick Start (SOAP mode): Creating a PDF document based on an application XDP file using the Java API" on page 242

"Quick Start (SOAP mode): Creating a PDF/A document using the Java API" on page 250

"Quick Start (SOAP mode): Passing documents to the Output Service using the Java API" on page 252

"Quick Start (SOAP mode): Passing a document located in the Repository to the Output service using the Java API" on page 245

Quick Start (SOAP mode): Creating a PDF document based on fragments using the Java API

Quick Start (SOAP mode): Printing to a file using the Java API

"Quick Start (SOAP mode): Sending a print stream to a network printer using the Java API" on page 261

"Quick Start (SOAP mode): Creating multiple PDF files using the Java API" on page 264

"Quick Start (SOAP mode): Creating search rules using the Java API" on page 266

"Quick Start (SOAP mode): Transforming a PDF document using the Java API" on page 269

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

> ***Note:*** *Quick Starts located in Programming with AEM forms are based on the Forms Server operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Creating a PDF document using the Java API

The following Java code example creates a PDF document named *Loan.pdf*. This PDF document is based on a form design named *Loan.xdp* and an XML data file named *Loan.xml*. The *Loan.pdf* is written to the C:\Adobe folder located on the J2EE application server hosting AEM Forms, not the client computer. (See "Creating PDF Documents" on page 681.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
```

```
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class CreatePDFDocument {

    public static void main(String[] args) {

        try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Reference form data
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
        Document inXMData = new Document (fileInputStream);
```

```
        //Set PDF run-time options
        PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
        outputOptions.setFileURI("C:\\Adobe\Loan.pdf");

        //Set rendering run-time options
        RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
        pdfOptions.setLinearizedPDF(true);
        pdfOptions.setAcrobatVersion(AcrobatVersion.Acrobat_9);

        //Create a PDF document
        OutputResult outputDocument = outClient.generatePDFOutput(
            TransformationFormat.PDF,
            "Loan.xdp",
            "C:\\Adobe",
            outputOptions,
            pdfOptions,
            inXMData
        );

        //Retrieve the results of the operation
        Document metaData = outputDocument.getStatusDoc();
        File myFile = new File("C:\\Adobe\Output.xml");
        metaData.copyToFile(myFile);
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Creating a PDF document based on an application XDP file using the Java API

The following Java code example creates a PDF document named *Loan.pdf*. This PDF document is based on a form design named *Loan.xdp* and an XML data file named *Loan.xml*. The XDP file is deployed as part of an AEM Forms application named `Applications/FormsApplication`. Notice that the URI path is `repository:///Applications/FormsApplication/1.0/FormsFolder/`. The *Loan.pdf* is written to the C:\Adobe folder located on the J2EE application server hosting AEM Forms, not the client computer. (See "Creating PDF Documents" on page 681.)

Before running this quick start, ensure that you create an AEM Forms application named Applications/FormsApplication. Create a folder within the application named FormsFolder and place the XDP file in the folder. For more information, see .

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/Adobe/adobe_experience_manager_forms/SDK/client-libs/common
    *
    * <install directory>/Adobe/adobe_experience_manager_forms/SDK/client-libs/jboss
    *
    * <install directory>/Adobe/adobe_experience_manager_forms/jboss/bin/client
    *
    * If you want to invoke a remote AEM Forms instance and there is a
    * firewall between the client application and AEM Forms, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/Adobe/adobe_experience_manager_forms/SDK/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class CreatePDFDocumentFromLCApp {
```

```
    public static void main(String[] args) {

        try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Reference form data
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
        Document inXMData = new Document (fileInputStream);

        //Set PDF run-time options
        PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
        outputOptions.setFileURI("C:\\Adobe\Loan.pdf");

        //Set rendering run-time options
        RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
        pdfOptions.setLinearizedPDF(true);
        pdfOptions.setAcrobatVersion(AcrobatVersion.Acrobat_9);

        //Create a PDF document -- reference an XDP file named Loan.xdp that is deployed as
part of
        //a AEM Forms application named Applications/FormsApplication. The XDP file is located
        //in a folder named FormsFolder
        OutputResult outputDocument = outClient.generatePDFOutput(
            TransformationFormat.PDF,
```

```
            "Loan.xdp",
            "repository:///Applications/FormsApplication/1.0/FormsFolder/",
            outputOptions,
            pdfOptions,
            inXMData
        );

        //Retrieve the results of the operation
        Document metaData = outputDocument.getStatusDoc();
        File myFile = new File("C:\\Adobe\Output.xml");
        metaData.copyToFile(myFile);
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Passing a document located in the Repository to the Output service using the Java API

The following Java code retrieves an XDP file from the Repository and passes it to the Output service within `com.adobe.idp.Document` instance. The XDP file is deployed as part of a AEM Forms application named `Applications/FormsApplication`. Notice that the URI path is `repository:///Applications/FormsApplication/1.0/FormsFolder/`.

Note: *The Repository API is used to retrieve the XDP file from this location. (See "Reading Resources" on page 1043.)*

Also notice the content root value `repository:///Applications/FormsApplication/1.0/FormsFolder/` is passed to the `OutputClient` object's `generatePDFOutput2` method (the second parameter). This value is passed to the Output service to inform the Output service that form collateral, such as images, are stored in this location.

Note: *You can set the content root value in the same way when invoking the `generatePrintedOutput2` method.*

The *Loan.pdf* is written to the C:\Adobe folder located on the J2EE application server hosting AEM Forms. (See "Passing Documents located in the Repository to the Output Service" on page 700.)

Before running this quick start, ensure that you create a AEM Forms application named Applications/FormsApplication. Create a folder within the application named FormsFolder and place the XDP file in the folder.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-repository-client.jar
    * 3. adobe-livecycle-client.jar
    * 4. adobe-usermanager-client.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class CreatePDFFFromRepository {

    public static void main(String[] args) {

        try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
```

```
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Reference form data
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
        Document inXMData = new Document (fileInputStream);

        //Set PDF run-time options
        PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
        outputOptions.setFileURI("C:\\Adobe\Loan.pdf"); // this PDF form is saved on the server

        //Get the form design from the AEM Forms Repository
        Document formDesign =  GetFormDesign(myFactory);

        //Set rendering run-time options
        RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
        pdfOptions.setLinearizedPDF(true);
        pdfOptions.setAcrobatVersion(AcrobatVersion.Acrobat_9);

        //Create a non-interactive PDF document
        OutputResult outputDocument = outClient.generatePDFOutput2(
            TransformationFormat.PDF,
            "repository:///Applications/FormsApplication/1.0/FormsFolder/",
            formDesign,
            outputOptions,
            pdfOptions,
            inXMData
        );

        //Save the non-interactive PDF form as a PDF file on the client computer
        Document pdfForm = outputDocument.getGeneratedDoc();
        File myFile = new File("C:\\Adobe\Loan.pdf");
        pdfForm.copyToFile(myFile);
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }

    // Retrieve the form design from the following Repository path:
    // /Applications/FormsApplication/1.0/FormsFolder/Loan.xdp
    private static Document GetFormDesign(ServiceClientFactory myFactory)
    {
    try{

        // Create a ResourceRepositoryClient object using the service client factory
        ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);
```

```
        // Specify the path in the Repository to Loan.xdp
        String resourceUri =  "/Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";

        // Retrieve the XDP file
        Document doc = repositoryClient.readResourceContent(resourceUri);

           //Return the Document instance
           return  doc;
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
        return null;
    }
}
```

Quick Start (SOAP mode): Creating a PDF document using the Java API

The following Java code example creates a PDF document named *Loan.pdf*. This PDF document is based on a form design named *Loan.xdp* and an XML data file named *Loan.xml*. The *Loan.pdf* is written to the C:\Adobe folder located on the J2EE application server hosting AEM Forms, not the client computer. (See "Creating PDF Documents" on page 681.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
```

```
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class CreatePDFDocumentSOAP {

    public static void main(String[] args) {

    try{
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Reference form data
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
        Document inXMData = new Document (fileInputStream);

        //Set PDF run-time options
        PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
        outputOptions.setFileURI("C:\\Adobe\Loan.pdf");

        //Set rendering run-time options
        RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
```

```
        pdfOptions.setLinearizedPDF(true);
        pdfOptions.setAcrobatVersion(AcrobatVersion.Acrobat_9);

        //Create a PDF document
        OutputResult outputDocument = outClient.generatePDFOutput(
            TransformationFormat.PDF,
            "Loan.xdp",
            "C:\\Adobe",
            outputOptions,
            pdfOptions,
            inXMData
        );

        //Retrieve the results of the operation
        Document metaData = outputDocument.getStatusDoc();
        File myFile = new File("C:\\Adobe\Output.xml");
        metaData.copyToFile(myFile);
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Creating a PDF/A document using the Java API

The following Java code example creates a PDF/A document named *LoanArchive.pdf*. This PDF document is based on a form design named *Loan.xdp* and an XML data file named *Loan.xml*. The *LoanArchive.pdf* is written to the C:\Adobe folder located on the J2EE application server hosting AEM Forms, not the client computer. (See "Creating PDF/A Documents" on page 690.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
```

```
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class CreatePDFADocument {

    public static void main(String[] args) {

    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);
```

```
        //Reference an XML data source to merge with the form design
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
        Document inXMData = new Document (fileInputStream);

        //Set PDF run-time options
        PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
        outputOptions.setFileURI("C:\\Adobe\LoanArchive.pdf");

        //Set rendering run-time options
        RenderOptionsSpec pdfAOptions = new RenderOptionsSpec();
        pdfAOptions.setPDFAConformance(PDFAConformance.A);
        pdfAOptions.setPDFARevisionNumber(PDFARevisionNumber.Revision_1);


        //Create a PDF/A document
        OutputResult outputDocument = outClient.generatePDFOutput(
            TransformationFormat.PDFA,
            "Loan.xdp",
            "C:\\Adobe",
            outputOptions,
            pdfAOptions,
            inXMData
        );

        //Write the results of the operation to OutputLog.xml
        Document resultData = outputDocument.getStatusDoc();
        File myFile = new File("C:\\Adobe\OutputLog.xml");
        resultData.copyToFile(myFile);

        }catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Passing documents to the Output Service using the Java API

The following Java quick start retrieves the file *Loan.xdp* from Content Services. This XDP file is located in the `space /Company Home/Form Designs`. The XDP file is returned in a `com.adobe.idp.Document` instance. The `com.adobe.idp.Document` instance is passed to the Output service. The non-interactive form is saved as a PDF file named *Loan.pdf* on the client computer. Because the File URI option is set, the PDF file *Loan.pdf* is also saved on the J2EE application server hosting AEM Forms. (See "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-contentservices-client.jar
    * 3. adobe-livecycle-client.jar
    * 4. adobe-usermanager-client.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.contentservices.client.CRCResult;
import com.adobe.livecycle.contentservices.client.impl.DocumentManagementServiceClientImpl;
import com.adobe.livecycle.output.client.*;

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;


public class CreatePDFFFromContentServices {

    public static void main(String[] args) {

        try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
```

```
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Reference form data
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
        Document inXMData = new Document (fileInputStream);

        //Set PDF run-time options
        PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
        outputOptions.setFileURI("C:\\Adobe\Loan.pdf"); // this PDF form is saved on the server

        //Get the form design from Content Services
        Document formDesign =  GetFormDesign(myFactory);

        //Set rendering run-time options
        RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
        pdfOptions.setLinearizedPDF(true);
        pdfOptions.setAcrobatVersion(AcrobatVersion.Acrobat_9);

        //Create a non-interactive PDF document
        OutputResult outputDocument = outClient.generatePDFOutput2(
            TransformationFormat.PDF,
            "C:\\Adobe",
            formDesign,
            outputOptions,
            pdfOptions,
            inXMData
        );

        //Save the non-interactive PDF form as a PDF file on the client computer
        Document pdfForm = outputDocument.getGeneratedDoc();
        File myFile = new File("C:\\Adobe\Loan.pdf");
        pdfForm.copyToFile(myFile);
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }

    //Retrieve the form design from Content Services ES2
    private static Document GetFormDesign(ServiceClientFactory myFactory)
    {
        try{

        //Create a DocumentManagementServiceClientImpl object
        DocumentManagementServiceClientImpldocManager = new
```

```
DocumentManagementServiceClientImpl(myFactory);

        //Specify the name of the store and the content to retrieve
           String storeName = "SpacesStore";
           String nodeName  = "/Company Home/Form Designs/Loan.xdp";

           //Retrieve /Company Home/Form Designs/Loan.xdp
           CRCResult content = docManager.retrieveContent(
                    storeName,
                    nodeName,
                    "");

           //Return the Document instance
            Document doc =content.getDocument();
            return  doc;
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
        return null;
    }
}
```

Quick Start (SOAP mode): Creating a PDF document based on fragments using the Java API

The following Java code example creates a PDF document that is based on a form design assembled by the Assembler service. The Assembler service assembles fragments located in multiple XDP files into a single form design. Application logic that invokes the Assembler service is located in a user-defined method named `GetFormDesign`. The non-interactive form is saved as a PDF file named *Loan.pdf* on the client computer. (See "Creating PDF Documents Using Fragments" on page 703.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
```

```
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-assembler-client.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    *
    *  This is the DDX file is used to assemble multiple XDP documents:
    * <?xml version="1.0" encoding="UTF-8"?>
    * <DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    *    <XDP result="tuc018result.xdp">
    *       <XDP source="tuc018_template_flowed.xdp">
    *      <XDPContent insertionPoint="ddx_fragment" source="tuc018_contact.xdp"
fragment="subPatientContact" required="false"/>
    *          <XDPContent insertionPoint="ddx_fragment" source="tuc018_patient.xdp"
fragment="subPatientPhysical" required="false"/>
    *          <XDPContent insertionPoint="ddx_fragment" source="tuc018_patient.xdp"
fragment="subPatientHealth" required="false"/>
    *       </XDP>
    *    </XDP>
    * </DDX>
    */
import com.adobe.livecycle.assembler.client.AssemblerOptionSpec;
import com.adobe.livecycle.assembler.client.AssemblerResult;
import com.adobe.livecycle.assembler.client.AssemblerServiceClient;
import com.adobe.livecycle.output.client.*;

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
```

```
public class CreatePDFFromFragments {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an OutputClient object
            OutputClient outClient = new OutputClient(myFactory);

            //Reference form data
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
            Document inXMData = new Document (fileInputStream);

            //Set PDF run-time options
            PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
            outputOptions.setFileURI("C:\\Adobe\Loan.pdf"); // this PDF form is saved on the
server

            //Get the form design from Assembler service
            Document formDesign =  GetFormDesign(myFactory);

            //Set rendering run-time options
            RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
            pdfOptions.setLinearizedPDF(true);
            pdfOptions.setAcrobatVersion(AcrobatVersion.Acrobat_9);

            //Create a non-interactive PDF document
            OutputResult outputDocument = outClient.generatePDFOutput2(
                TransformationFormat.PDF,
                "C:\\Adobe",
                formDesign,
                outputOptions,
                pdfOptions,
                inXMData
            );

            //Save the non-interactive PDF form as a PDF file on the client computer
```

```
            Document pdfForm = outputDocument.getGeneratedDoc();
            File myFile = new File("C:\\Adobe\Loan.pdf");
            pdfForm.copyToFile(myFile);
            }
            catch (Exception ee)
            {
                ee.printStackTrace();
            }
        }

        //Retrieve the form design from Assembler service
        private static Document GetFormDesign(ServiceClientFactory myFactory)
        {
            try{

                //Create an AssemblerServiceClient object
                AssemblerServiceClient assemblerClient = new AssemblerServiceClient(myFactory);

                //Create a FileInputStream object based on an existing DDX file
                FileInputStream myDDXFile = new FileInputStream("C:\\Adobe\fragmentDDX.xml");

                //Create a Document object based on the DDX file
                Document myDDX = new Document(myDDXFile);

                //Create a Map object to store the input XDP files
                Map inputs = new HashMap();
                FileInputStream inSource = new
FileInputStream("C:\\Adobe\tuc018_template_flowed.xdp");
                FileInputStream inFragment1 = new
FileInputStream("C:\\Adobe\tuc018_contact.xdp");
                FileInputStream inFragment2 = new
FileInputStream("C:\\Adobe\tuc018_patient.xdp");

                //Create a Document object
                Document myMapSource = new Document(inSource);

                //Create a Document object
                Document inFragment1Doc = new Document(inFragment1);

                //Create a Document object
                Document inFragment2Doc = new Document(inFragment2);

                //Place all of the XDP files into the MAP
                inputs.put("tuc018_template_flowed.xdp",myMapSource);
                inputs.put("tuc018_contact.xdp",inFragment1Doc);
                inputs.put("tuc018_patient.xdp",inFragment2Doc);


                //Create an AssemblerOptionsSpec object
                AssemblerOptionSpec assemblerSpec = new AssemblerOptionSpec();
                assemblerSpec.setFailOnError(false);

                //Submit the job to Assembler service
                AssemblerResult jobResult =
assemblerClient.invokeDDX(myDDX,inputs,assemblerSpec);
                java.util.Map allDocs = jobResult.getDocuments();
```

```
                //Retrieve the result PDF document from the Map object
                Document outDoc = null;

                //Iterate through the map object to retrieve the result XDP document
                for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
                    // Retrieve the Map object?s value
                    Map.Entry e = (Map.Entry)i.next();

                    //Get the key name as specified in the
                    //DDX document
                    String keyName = (String)e.getKey();
                    if (keyName.equalsIgnoreCase("tuc018result.xdp"))
                    {
                        Object o = e.getValue();
                        outDoc = (Document)o;

                    }
                }

            return outDoc;
            }catch (Exception e) {
                e.printStackTrace();
            }
            return null;
        }
    }
```

Quick Start (SOAP mode): Printing to a file using the Java API

The following Java code example prints an output stream to a PostScript file named *MortgageForm.ps*. (See "Printing to Files" on page 708.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
```

```
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class PrintToFile {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
```

```
            //Create an OutputClient object
            OutputClient outClient = new OutputClient(myFactory);

            //Reference XML data that represents form data
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
            Document inputXML = new Document(fileInputStream);

            //Set print run-time options required to print to a file
            PrintedOutputOptionsSpec printOptions = new PrintedOutputOptionsSpec();
            printOptions.setFileURI("C:\\Adobe\MortgageForm.ps");

            //Print the print stream to a PostScript file
            OutputResult outputDocument = outClient.generatePrintedOutput(
                    PrintFormat.PostScript,
                    "Loan.xdp",
                    "C:\\Adobe",
                    null,
                    printOptions,
                    inputXML);

            //Write the results of the operation to OutputLog.xml
            Document resultData = outputDocument.getStatusDoc();
            File myFile = new File("C:\\Adobe\OutputLog.xml");
            resultData.copyToFile(myFile);
            System.out.println("AEM Forms printed to MortgageForm.ps");
        }
    catch (Exception ee)
            {
            ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Sending a print stream to a network printer using the Java API

The following Java code example sends a PostScript print stream to a network printer named *Printer1**Printer*. Two copies are sent to the printer. (See "Sending Print Streams to Printers" on page 713.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.output.client.*;

public class SendToPrinter {
```

```
    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an OutputClient object
            OutputClient outClient = new OutputClient(myFactory);

            //Reference XML data that represents form data
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
            Document inputXML = new Document(fileInputStream);

            //Set print run-time options required to print to a file
            PrintedOutputOptionsSpec printOptions = new PrintedOutputOptionsSpec();

            //Set the number of copies to print
            printOptions.setCopies(2);

            //Turn on the Staple option
            printOptions.setStaple(Staple.on);

            //Create a PostScript output stream based on the form design named Loan.xdp and
            //the data located in the XML file
            OutputResult outputDocument = outClient.generatePrintedOutput(
                    PrintFormat.PostScript,
                    "Loan.xdp",
                    "C:\\Adobe",
                    "C:\\Adobe",
                    printOptions,
                    inputXML);

            //Get a Document object that stores the PostScript print stream
```

```
            Document psPrintStream = outputDocument.getGeneratedDoc();

            //Specify the print server and the printer name
            String printServer = "\\\ottprint";
            String printerName = "\\\ottprint\Balsom";

            //Send the PostScript print stream to the printer
            outClient.sendToPrinter(
                    psPrintStream,
                    PrinterProtocol.SharedPrinter,
                    printServer,
                    printerName);
            }
        catch (Exception ee)
            {
            ee.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Creating multiple PDF files using the Java API

The following Java code creates multiple PDF files for each data record that is located in an XML data file named *Loan_data_batch.xml*. The files are written to the C:\Adobe directory. The PDF files are written to the C:\Adobe folder located on the J2EE application server hosting AEM Forms, not the client computer. (See "Creating Multiple Output Files" on page 718.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.output.client.*;

public class CreateBatchFiles {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an OutputClient object
            OutputClient outClient = new OutputClient(myFactory);

            //Reference form data that contains multiple records
            FileInputStream fileInputStream = new
```

```
FileInputStream("C:\\Adobe\Loan_data_batch.xml");
            Document inXMData = new Document (fileInputStream);

            //Set run-time options to generate many PDF files
            PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
            outputOptions.setFileURI("C:\\Adobe\Loan.pdf");
            outputOptions.setGenerateManyFiles(true);
            outputOptions.setRecordName("LoanRecord");


            //Set rendering run-time options
            RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
            pdfOptions.setCacheEnabled(new Boolean(true));


            //Create multiple PDF files
            OutputResult outputDocument = outClient.generatePDFOutput(
                TransformationFormat.PDF,
                "Loan.xdp",
                "C:\\Adobe",
                outputOptions,
                pdfOptions,
                inXMData
                );

            //Retrieve the results of the operation
            Document metaData = outputDocument.getStatusDoc();
            File myFile = new File("C:\\Adobe\Output.xml");
            metaData.copyToFile(myFile);
            }
            catch (Exception ee)
            {
                ee.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Creating search rules using the Java API

The following Java code example creates two text patterns that the Output service searches for. The first text pattern is Mortgage. If found, the Output service uses the form design named *Mortgage.xdp*. The second text pattern is Automobile. If found, the Output service uses the form design named *AutomobileLoan.xdp*. If neither text pattern is located, the Output service uses the default form design named *Loan.xdp.* (See "Creating Search Rules" on page 726.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class CreateSearchRules {
```

```
    public static void main(String[] args) {
        try{

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an OutputClient object
            OutputClient outClient = new OutputClient(myFactory);

            //Reference form data
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xml");
            Document inXMData = new Document (fileInputStream);

            //Define two text patterns
            Rule mortageRule = new Rule();
            mortageRule.setPattern("Mortgage");
            mortageRule.setForm("Mortgage.xdp");

            Rule automobileRule = new Rule();
            automobileRule.setPattern("Automobile");
            automobileRule.setForm("AutomobileLoan.xdp");

            //Add the Rules to a List object
            List<Rule> myList = new ArrayList<Rule>();
            myList.add(mortageRule);
            myList.add(automobileRule);

            //Define PDF run-time options which includes Search Rules
            PDFOutputOptionsSpec outputOptions = new PDFOutputOptionsSpec();
            outputOptions.setFileURI("C:\\Adobe\Loan.pdf");
            outputOptions.setRules(myList);
            outputOptions.setLookAhead(900);

            //Define rendering run-time options
            RenderOptionsSpec pdfOptions = new RenderOptionsSpec();
            pdfOptions.setCacheEnabled(new Boolean(true));

            //Create a PDF document based on multiple form designs
            OutputResult outputDocument = outClient.generatePDFOutput(
```

```
                TransformationFormat.PDF,
                "Loan.xdp",
                "C:\\Adobe",
                outputOptions,
                pdfOptions,
                inXMData
            );

            //Write the results of the operation to OutputLog.xml
            Document resultData = outputDocument.getStatusDoc();
            File myFile = new File("C:\\Adobe\OutputLog.xml");
            resultData.copyToFile(myFile);
            }
        catch (Exception ee)
                {
                    ee.printStackTrace();
                }
        }
}
```

Quick Start (SOAP mode): Transforming a PDF document using the Java API

The following Java code example transforms an interactive PDF document named *Loan.pdf* to a non-interactive PDF document named *NonInteractiveLoan.pdf*. (See "Flattening PDF Documents" on page 732.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-output-client.jar
    * 2. adobe--client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.output.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class TransformPDF {

    public static void main(String[] args) {

    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create an OutputClient object
        OutputClient outClient = new OutputClient(myFactory);

        //Reference an interactive PDF document to transform
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
        Document inPDFDoc = new Document (fileInputStream);
```

```
        //Transform the PDF document to a non-interactive PDF document
        Document transformedDocument = outClient.transformPDF(
                inPDFDoc,
                TransformationFormat.PDF,
                null,
                null,
                null);

        //Save the non-interactive PDF document
        File myFile = new File("C:\\Adobe\NonInteractiveLoan.pdf");
        transformedDocument.copyToFile(myFile);

    }catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }
}
```

PDF Utilities Service Java API Quick Start(SOAP)

The following Quick Starts are available for the PDF Utilities service.

"Quick Start (SOAP mode): Converting a PDF document to an XDP document using the Java API" on page 271

"Quick Start (SOAP mode): Converting an XDP document to a PDF document using the Java API" on page 273

"Quick Start (SOAP mode): Retrieving PDF document properties using the Java API" on page 275

"Quick Start (SOAP mode): Setting the save style for a PDF document using the Java API" on page 278

"Quick Start (SOAP mode): Sanitizing PDF documents" on page 282

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM forms are based on the Forms Server operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Converting a PDF document to an XDP document using the Java API

The following code example converts a PDF document to an XDP document. (See "Converting PDF Documents into XDP Documents" on page 997.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-pdfutility-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;
import com.adobe.livecycle.pdfutility.client.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ConvertPDFToXDP
```

```
{
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a PDF Utility client
            PDFUtilityServiceClient pdfUt = new PDFUtilityServiceClient(myFactory);

            // Specify a PDF document to convert to an XDP file
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document(fileInputStream);

            // Convert the PDF document to an XDP file
            Document myXDP = pdfUt.convertPDFtoXDP(inDoc);

            //Save the returned Document object as an XDP file
            File xdpFile = new File("C:\\Adobe\Loan.xdp");
            myXDP.copyToFile(xdpFile);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Converting an XDP document to a PDF document using the Java API

The following code example converts an XDP document to a PDF document. (See "Converting XDP Documents into PDF Documents" on page 999.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-pdfutility-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.pdfutility.client.*;
import java.util.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ConvertXDPToPDF
{
```

```
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a PDF Utility client
            PDFUtilityServiceClient pdfUt = new PDFUtilityServiceClient(myFactory);

            // Specify an XDP file to convert to a PDF document
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.xdp");
            Document inDoc = new Document(fileInputStream);

            // Convert the XDP file to a PDF document
            Document myPDF = pdfUt.convertXDPtoPDF(inDoc);

            //Save the returned Document object as a PDF file
            File pdfFile = new File("C:\\Adobe\Loan.pdf");
            myPDF.copyToFile(pdfFile);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieving PDF document properties using the Java API

The following code example determines whether the document is a PDF document and, if so, the earliest Acrobat version able to read it. (See "Retrieving PDF Document Properties" on page 1000.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-pdfutility-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.pdfutility.client.*;
import java.util.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RetrievePDFProperties
```

```
{
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a PDF Utility client
            PDFUtilityServiceClient pdfUt = new PDFUtilityServiceClient(myFactory);

            // Specify a document whose properties are retrieved
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document(fileInputStream);

            // Create a properties options specification
            PDFPropertiesOptionSpec optionsSpec = new PDFPropertiesOptionSpec();

            // Set the properties to be evaluated in the options specification.
            // In this example, the options specification will be used to determine
            // if the document is a PDF document, and if so,
            // which Acrobat version is required to read it.
            optionsSpec.setIsPDFDocument(true);
            optionsSpec.setQueryRequiredAcrobatVersion(true);

            // Perform the query and retrieve the document properties
```

```
            PDFPropertiesResult propertiesResult = pdfUt.getPDFProperties(inDoc, optionsSpec);

            // Inspect the result and determine whether the file is a PDF document
            if (propertiesResult.getIsPDFDocument().booleanValue())
            {
                System.out.println("Loan.pdf has been verified to be a PDF document.");

                // Determine the required Acrobat version for reading the document
                String acrobatVersion = propertiesResult.getRequiredAcrobatVersion();
                System.out.println("The required Acrobat version is: " + acrobatVersion);
            }
        }
        catch (Exception e)
        {
            System.out.println("Error occurred: " + e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Setting the save style for a PDF document using the Java API

The following code example sets the save mode for fast web viewing and then passes the PDF document to the Encryption service where it is encrypted. The encrypted PDF document that is saved for fast web viewing is saved as a PDF file named *FastWebViewLoan.pdf*. (See "Setting PDF Document Save Modes" on page 1002.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-pdfutility-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.encryption.client.EncryptionServiceClient;
import com.adobe.livecycle.encryption.client.PasswordEncryptionCompatability;
import com.adobe.livecycle.encryption.client.PasswordEncryptionOption;
import com.adobe.livecycle.encryption.client.PasswordEncryptionOptionSpec;
import com.adobe.livecycle.encryption.client.PasswordEncryptionPermission;
import com.adobe.livecycle.pdfutility.client.*;
import java.util.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class SaveDocument
{
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
```

```
            // Create a PDF Utility client
            PDFUtilityServiceClient pdfUt = new PDFUtilityServiceClient(myFactory);

            // Specify a document to be saved
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document(fileInputStream);

            // Specify the save option
            PDFUtilitySaveMode saveMode = new PDFUtilitySaveMode();
            saveMode.setSaveStyle("FAST_WEB_VIEW");
            Document outFastWebView = pdfUt.setSaveMode(inDoc, saveMode, true);

            //Pass the document that is saved as 'Fast
            //Web View'  to the Encryption service
            EncryptionServiceClient encryptClient = new EncryptionServiceClient(myFactory);

            //Create a PasswordEncryptionOptionSpec object that
            //stores encryption run-time values
            PasswordEncryptionOptionSpec passSpec = new PasswordEncryptionOptionSpec();

            //Specify the PDF document resource to encrypt
            passSpec.setEncryptOption(PasswordEncryptionOption.ALL);

            //Specify the permission associated with the password
            List encrypPermissions = new ArrayList();
            encrypPermissions.add(PasswordEncryptionPermission.PASSWORD_EDIT_EXTRACT);
            encrypPermissions.add(PasswordEncryptionPermission.PASSWORD_EDIT_FORM_FILL);
            passSpec.setPermissionsRequested(encrypPermissions);

            //Specify the Acrobat version
            passSpec.setCompatability(PasswordEncryptionCompatability.ACRO_7);

            //Specify the password values
            passSpec.setDocumentOpenPassword("OpenPassword");
            passSpec.setPermissionPassword("PermissionPassword");

            //Encrypt the PDF document
            Document encryptDoc = encryptClient.encryptPDFUsingPassword(inDoc,passSpec);

            //Save the encrypted document that is saved as FAST_WEB_VIEW
            //as a PDF file named FastWebViewLoan.pdf
            File pdfFile = new File("C:\\Adobe\FastWebViewLoan.pdf");
            encryptDoc.copyToFile(pdfFile);

            // Inspect the document?s save option
            PDFUtilitySaveMode verifySaveMode = pdfUt.getSaveMode(outFastWebView);
            String verifySaveStyle = verifySaveMode.getSaveStyle();
            System.out.println("The save mode is " + verifySaveStyle);
        }
        catch (Exception e)
        {
            System.out.println("Error occurred: " + e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Converting a document to a PDF/A-2b document using the Java API

The following Java code example converts a PDF document named *Loan.pdf* to a PDF/A-2b document that is saved as a PDF file named *LoanArchive.pdf*. (See "Converting Documents to PDF/A Documents" on page 990.)

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-docconverter-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * If you want to invoke a remote AEM Forms instance and there is a
 * firewall between the client application and AEM Forms, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include additional JAR files located in the following
 * path
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * For information about the SOAP
 * mode and the additional JAR files that need to be included,
 * see "Setting connection properties" in Programming
 * with AEM Forms
 *
 * For complete details about the location of the AEM Forms JAR files,
 * see "Including AEM Forms library files" in Programming
 * with AEM Forms
 */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.docconverter.client.DocConverterServiceClient;
import com.adobe.livecycle.docconverter.client.PDFAConversionOptionSpec;
import com.adobe.livecycle.docconverter.client.PDFAConversionResult;
import com.adobe.livecycle.docconverter.client.PDFAConversionOptionSpec.Compliance;

public class CreatePDFADocument {

    public static void main(String[] args) {
    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
```

```
entFactoryProperties.DSC_SOAP_PROTOCOL);
       connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a DocConverterServiceClient object
        DocConverterServiceClient docConverter = new DocConverterServiceClient(myFactory);

        //Reference a PDF document to convert to a PDF/A document
        FileInputStream myPDF = new FileInputStream("C:\\Adobe\\Loan.pdf");
        Document inDoc = new Document(myPDF);

        //Create a PDFAConversionOptionSpec object and set
        //tracking information
        PDFAConversionOptionSpec spec = new PDFAConversionOptionSpec();
        spec.setLogLevel("FINE");
        spec.setCompliance(Compliance.PDFA_2B);

        //Convert the PDF document to a PDF/A document
        PDFAConversionResult result =  docConverter.toPDFA(inDoc,spec);

        //Save the PDF/A file
        Document pdfADoc= result.getPDFADocument();
        File pdfAFile = new File("C:\\Adobe\\LoanArchive.pdf");
        pdfADoc.copyToFile(pdfAFile);
   }catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

Quick Start (SOAP mode): Sanitizing PDF documents

The following Java code example sanitizes a PDF document named *Loan.pdf*.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-docconverter-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * If you want to invoke a remote AEM Forms instance and there is a
 * firewall between the client application and AEM Forms, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include additional JAR files located in the following
 * path
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * For information about the SOAP
 * mode and the additional JAR files that need to be included,
 * see "Setting connection properties" in Programming
 * with AEM Forms
 *
 * For complete details about the location of the AEM Forms JAR files,
 * see "Including AEM Forms library files" in Programming
 * with AEM Forms
 */
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.pdfutility.client.PDFUtilityServiceClient;
import com.adobe.livecycle.pdfutility.client.SanitizationResult;
public class Sanitization {
    public static void main (String[] args){

        try {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

```
          //Create a ServiceClientFactory instance
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

          //Create a PDFUtilityServiceClient object
            PDFUtilityServiceClient pdfutility = new PDFUtilityServiceClient(myFactory);

          //Reference a PDF document to Sanitize
            FileInputStream myPDF;
            myPDF = new FileInputStream("C://loan.pdf");
            Document inDoc = new Document(myPDF);

            //Sanitize the document.
            SanitizationResult result = pdfutility.sanitize(inDoc);

            //Save the Sanitized document
            if(result.isSanitizationSuccessful()){
                Document pdfADoc= result.getDocument();
                File pdfAFile = new File("C://annotations_sanitized.pdf");
                pdfADoc.copyToFile(pdfAFile);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

LiveCycleProcess Java API(SOAP) Quick Start

The Java API(SOAP) Quick Start is available for processes. A *process instance* is an occurrence of a specific process that was started by an invocation method such as the Invocation API or from within Workspace.

"Quick Start (SOAP mode): Searching for Process Instances using the Java API" on page 284

"Quick Start (SOAP mode): Suspending process instances using the Java API" on page 287

"Quick Start (SOAP Mode): Starting suspended process instances using the Java API" on page 289

"Quick Start (SOAP mode): Terminating process instances using the Java API" on page 291

"Quick Start (SOAP mode): Purging process data using the Java API" on page 293

"Quick Start (SOAP Mode): Retrieving the status of a job using the Java API" on page 295

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick starts located in Programming with AEM Forms are based on the Forms if you are using another operating system, such as Unix, replace windows specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Searching for Process Instances using the Java API

The following Java code example searches for process instances that are based on the *MortgageLoan - Prebuilt* process.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.taskmanager.dsc.client.TaskManagerClientFactory;
import com.adobe.idp.taskmanager.dsc.client.TaskManagerQueryService;
import com.adobe.idp.taskmanager.dsc.client.query.ProcessInstanceRow;
import com.adobe.idp.taskmanager.dsc.client.query.ProcessSearchFilter;
import com.adobe.idp.workflow.client.ProcessManager;

/**
    * This Java Quick Start searches for all completed processes that are based on the
```

```
application
    * and tracks the time at which the processes where completed.
    *
    */
public class SearchingProcesses {

    public static void main(String[] args) {
        try{

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME, "tblue");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
            TaskManagerQueryService queryProcess =
TaskManagerClientFactory.getQueryManager(myFactory);

            ProcessSearchFilter processFilter = new ProcessSearchFilter();
            processFilter.setServiceName("MortgageLoan - Prebuilt");

            List allProcesses = queryProcess.processSearch(processFilter);

            //Create an Iterator object and iterate through
//           the List object
           Iterator iter = allProcesses.iterator();
           int i = 0 ;
           long processId=0 ;
           while (iter.hasNext()) {
               ProcessInstanceRow processInstance = (ProcessInstanceRow)iter.next();

          if (processInstance.getProcessInstanceStatus() ==
ProcessInstanceRow.STATUS_RUNNING){
```

```
            //Display the process d
            processId = processInstance.getProcessInstanceId();
            System.out.println("The process identifier is " +processId);
          }

            i++;
           }
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }



}
```

Quick Start (SOAP mode): Suspending process instances using the Java API

The following Java code example suspends a process instance. To successfully suspend a process instance, you require the process invocation identifier that can be obtained when invoking a long-lived process by using the Invocation API.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.um.api.infomodel.PrincipalSearchFilter;
import com.adobe.idp.workflow.client.ProcessManager;


public class SuspendProcesses {


    public static void main(String[] args) {
         try{

                //Set connection properties required to invoke AEM Forms
                Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
                connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME, "tblue");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

```
                  //Create a ServiceClientFactory object
                   ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

                   //Create a ProcessManager object
                   ProcessManager myProcessManager = new ProcessManager(myFactory);
                   myProcessManager.suspendProcess("1");
             }
               catch(Exception e)
                {
                      e.printStackTrace();
               }



    }

}
```

Quick Start (SOAP Mode): Starting suspended process instances using the Java API

The following Java code example starts a suspended process instance.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.workflow.client.ProcessManager;

public class StartProcess {

    public static void main(String[] args) {
        try{

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME, "tblue");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
```

```
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ProcessManager object
            ProcessManager myProcessManager = new ProcessManager(myFactory);

            //Start a suspended process instance
            myProcessManager.unSuspendProcess("5fae07190a242fb1010b2229ccad8a7e");
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Terminating process instances using the Java API

The following Java code example terminates a process instance with the identifier value of 756c22860a242fb101ec7a5bc0977fd6.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.workflow.client.ProcessManager;

public class TerminatingProcesses {

    public static void main(String[] args) {
        try{

            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME, "tblue");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

```
            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ProcessManager object
            ProcessManager myProcessManager = new ProcessManager(myFactory);


            myProcessManager.terminateProcess("sd");
            //Terminate a process instance
        //   myProcessManager.terminateProcess("756c22860a242fb101ec7a5bc0977fd6");
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Purging process data using the Java API

The following Java code purges data from a process named *SecureDocument*. A filter is used that specifies to purge data for those process instances where the process variable named *inValue* is greater than 200.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
```

```
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.workflow.client.ProcessManager;
import com.adobe.idp.workflow.dsc.type.*;

public class PurgeProcess
{
     public static void main(String[] args)
       {
          try
           {
       //Set connection properties required to invoke AEM Forms
             Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
             connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

             //Create a ServiceClientFactory instance
             ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

          //Create a ProcessManager object
          ProcessManager myProcessManager = new ProcessManager(myFactory);

          //Prepare parameters to use in the purge operation
             long age = 10;  // in seconds
             boolean includeChildren = false;// don't include children by default
             int status = 3;   // both completed and terminated by default
             short minor = 0;
             short major = 1;

            //Create the conditionFilter object to filter
```

```
            //out unwanted instances of the process
           ConditionFilter filter = new ConditionFilter("inValue", ConditionEnum.GREATER_THAN,
"200");

            //Delete process instances that contain a process
            //variable named inValue whose value
            //is greater than 200
            myProcessManager.purgeProcess(
             "SecureDocument",
             major,
             minor,
             status,
             age,
             filter,
             includeChildren);
      }
    catch (Exception e)
            {
              e.printStackTrace();
             }
        }
}
```

Quick Start (SOAP Mode): Retrieving the status of a job using the Java API

The following code example retrieves the status of 10 AEM Forms jobs. (See "Retrieving the Status of an AEM Forms Job" on page 1077.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-encryption-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
```

```
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.jobmanager.client.JobManager;
import com.adobe.idp.jobmanager.common.JobInstance;
import com.adobe.idp.jobmanager.common.JobInstanceFilter;


public class SearchForJobs {

    public static void main(String[] args) {

    //This function will upload a ceritificate to AEM Forms trust store
      try{

        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
        JobManager jobManager= new JobManager(myFactory);

        //Specify filter criteria
        JobInstanceFilter jobFilter = new JobInstanceFilter();
```

```
        jobFilter.setMaxObjects(10);

        //Retrieve the first 10 jobs
        List<JobInstance> allJobs = jobManager.getJobInstances(jobFilter);

        //Create an Iterator object and iterate through
        //the List object
        Iterator iter = allJobs.iterator();
        int i = 0 ;
        while (iter.hasNext()) {
            JobInstance JobInstance = (JobInstance)iter.next();
            System.out.println("The status of the job is " +JobInstance.getStatus() +". The
identifier value of the job is " +JobInstance.getId()+ ". The service on which the job is based
is " +JobInstance.getServiceName());
            i++;
        }

      }catch (Exception e) {
             e.printStackTrace();
            }

    }
}
```

Acrobat Reader DC extensions Service Java API Quick Start(SOAP)

The following Quick Starts are available for the Acrobat Reader DC Extensions service.

Quick Start (SOAP mode):Applying usage rights using the Java API

"Removing Usage Rights from PDF Documents" on page 755

"Quick Start (SOAP mode): Retrieving credential information using the Java API" on page 302

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Starts located in Programming with AEM Forms are based on the Forms server operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode):Applying usage rights using the Java API

The following Java code example applies usage rights to a PDF document named *Loan.pdf*. The rights-enabled PDF document is saved as a PDF file named *LoanUsageRights.pdf*. The following usage rights are applied to this PDF document: `enabledComments`, `enabledFormFillIn`, and `enabledDigitalSignatures`. (See "Applying Usage Rights to PDF Documents" on page 751.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-reader-extensions-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.readerextensions.client.*;
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ApplyUsageRightsSOAP{

    public static void main(String[] args) {
      try{

          //Set connection properties required to invoke AEM Forms using SOAP mode
          Properties connectionProps = new Properties();
```

```
connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory object
          ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

          //Create a ReaderExtensionsServiceClient object
          ReaderExtensionsServiceClient reClient = new
ReaderExtensionsServiceClient(myFactory);

          //Retrieve the PDF document to which to apply usage rights
          FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
          Document inputPDF = new Document(fileInputStream);

          //Create a UsageRight object and specify specific usage rights
          UsageRights useRight = new UsageRights();
          useRight.setEnabledDynamicFormFields(true);
          useRight.setEnabledComments(true);
          useRight.setEnabledFormFillIn(true);
          useRight.setEnabledDigitalSignatures(true);

          //Create a ReaderExtensionsOptions object
          ReaderExtensionsOptionSpec reOptions = new ReaderExtensionsOptionSpec();

          //Set the usage rights
          reOptions.setUsageRights(useRight);
          reOptions.setMessage("This is a Rights-Enabled PDF Document");

          //Apply usage rights to a PDF document
          Document rightsEnabledPDF = reClient.applyUsageRights(
            inputPDF,
            "RE2",
            null,
            reOptions);

          //Create a new PDF file that represents the rights-enabled PDF document
          File resultFile = new File("C:\\Adobe\LoanUsageRights.pdf");
          rightsEnabledPDF.copyToFile(resultFile);

        }catch (Exception e) {
             e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Removing usage rights from a PDF document using the Java API

The following Java code example removes usage rights from a rights-enabled PDF document named *LoanUsageRights.pdf*. (See "Removing Usage Rights from PDF Documents" on page 755.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-reader-extensions-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.readerextensions.client.*;
import java.util.*;
```

```
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RemoveUsageRights{

    public static void main(String[] args) {
      try{
          //Set connection properties required to invoke AEM Forms
          Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory object
          ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

          //Create a ReaderExtensionsServiceClient object
          ReaderExtensionsServiceClient reClient = new
ReaderExtensionsServiceClient(myFactory);

              //Retrieve a rights-enabled PDF document from
              //which to remove usage rights
          FileInputStream fileInputStream = new
FileInputStream("C:\\Adobe\LoanUsageRights.pdf");
          Document inputPDF = new Document(fileInputStream);

          //Remove usage rights from the PDF document
          Document rightsEnabledPDF = reClient.removeUsageRights(inputPDF);

          //Save the PDF document as a PDF file
          File resultFile = new File("C:\\Adobe\noUsageRightsLoan.pdf");
          rightsEnabledPDF.copyToFile(resultFile);
          System.out.println("Usage rights were removed from the document");

        }catch (Exception e) {
             e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieving credential information using the Java API

The following Java code example retrieves information about the credential that is used to apply usage-rights to a rights-enabled PDF document named *LoanUsageRights.pdf*. (See "Retrieving Credential Information" on page 758.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-reader-extensions-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.readerextensions.client.*;
import java.util.*;
```

```
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RetrieveCredentialInformation {

    public static void main(String[] args) {
      try{

          //Set connection properties required to invoke AEM Forms
          Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory object
          ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

          //Create a ReaderExtensionsServiceClient object
          ReaderExtensionsServiceClient reClient = new
ReaderExtensionsServiceClient(myFactory);

          //Retrieve a rights-enabled PDF document
          FileInputStream fileInputStream = new
FileInputStream("C:\\Adobe\LoanUsageRights.pdf");
          Document inputPDF = new Document(fileInputStream);

          //Retrieve credential information
          GetUsageRightsResult usageRightsResult = reClient.getDocumentUsageRights(inputPDF);

          //Get the date after which the credential is no longer valid
          Date endDate = usageRightsResult.getNotAfter();
```

```
            //Get the message displayed in Adobe Reader when the rights-enabled
            //document is opened
            String message = usageRightsResult.getMessage();

            //Get usage rights to see if the enableFormFillIn is enabled
            UsageRights myRights = usageRightsResult.getRights();
            boolean ans = myRights.isEnabledFormFillIn();

            if (ans==true)
              System.out.println("The enableFormFillIn usage right is enabled");
            else
              System.out.println("The enableFormFillIn usage right is not enabled");

          }catch (Exception e) {
               e.printStackTrace();
          }

    }
}
```

Repository Service API Quick Starts

The following Quick Starts are available for the AEM Forms Repository service.

"Quick Start (SOAP mode): Creating a folder using the Java API" on page 305

"Quick Start (SOAP mode): Writing a resource using the Java API" on page 307

"Quick Start (SOAP mode): Listing resources using the Java API" on page 309

"Quick Start (SOAP mode): Reading a resource using the Java API" on page 311

"Quick Start (SOAP mode): Updating a resource using the Java API" on page 313

"Quick Start (SOAP mode): Searching for resources using the Java API" on page 316

"Quick Start (SOAP mode): Creating relationships between resources using the Java API" on page 318

"Quick Start (SOAP mode): Locking a resource using the Java API" on page 321

"Quick Start (SOAP mode): Managing access control lists using the Java API" on page 323

"Quick Start (SOAP mode): Deleting a resource using the Java API" on page 325

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP

Applications/FormsApplication

Most AEM Forms repository service quick starts interact with an application named `Applications/FormsApplication`, as shown in the following illustration.

The folder FormsFolder is a location in the AEM Forms repository. You can, for example, programmatically add this folder to `Applications/FormsApplication`. (See "Quick Start (SOAP mode): Creating a folder using the Java API" on page 305.)

The path to a resource located in the AEM Forms repository is:

```
Applications/Application-name/Application-version/Folder.../Filename
```

Note: *You can browse the AEM Forms Repository by using a web browser. To browse the repository, enter the following URL into a web browser http://[server name]:[server port]/repository. You can verify quick start results by using a web browser. For example, if you add content to the AEM Forms Repository, you can see the content in a web browser.*

Note: *Applications/FormsApplication does not exist by default. To follow along with the quick starts, create this application by using Workbench. For information about creating an application using Workbench, see Getting started with process design.*

Quick Start (SOAP mode): Creating a folder using the Java API

The following Java code example creates a folder called *FormsFolder* in the following location `/Applications/FormsApplication/1.0/`. (See "Creating Folders" on page 1035.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
```

```
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.*;
import com.adobe.repository.infomodel.bean.*;

public class CreateFolder {

    public static void main(String[] args) {

        // This quick start creates a folder in the AEM Forms repository
        //Ensure that you create a AEM Forms application named FormsApplication using Workbench
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Create a RepositoryInfomodelFactoryBean needed for creating resources
            RepositoryInfomodelFactoryBean repositoryInfomodelFactory = new
RepositoryInfomodelFactoryBean(null);

            // Create a folder in a AEM Forms application named Application/FormsApplication
            ResourceCollection folder = repositoryInfomodelFactory.newResourceCollection(
                new Id(),
                new Lid(),
                "FormsFolder"
            );

            // Set the folder's description
```

```
            folder.setDescription("A folder to store forms");

            // Write the folder to the repository
            Resource newFolder =
repositoryClient.writeResource("/Applications/FormsApplication/1.0/", folder);

            // Retrieve the folder's identifier value
            String msg = "The identifier value of the new folder is" + newFolder.getId();

            // Print folder verification message
            System.out.println(msg);

        } catch (Exception e) {
                System.out.println(
                    "Exception thrown while trying to create the folder" +
                    e.getMessage()
                );
          }
    }
}
```

Quick Start (SOAP mode): Writing a resource using the Java API

The following Java code example writes a resource called *loan.xdp* in the repository. The resource is added to the `/Applications/FormsApplication/1.0/FormsFolder` location. (See "Writing Resources" on page 1037.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
```

```
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.Id;
import com.adobe.repository.infomodel.Lid;

import com.adobe.repository.infomodel.bean.RepositoryInfomodelFactoryBean;
import com.adobe.repository.infomodel.bean.Resource;
import com.adobe.repository.infomodel.bean.ResourceContent;

public class WriteFile {

    // This quick start writes Loan.xdp to Applications/FormsApplication/1.0/FormsFolder
    //Ensure that you create a AEM Forms application named FormsApplication using Workbench
    public static void main(String[] args) {

        try
        {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a ResourceRepositoryClient object
        ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

        //Specify the parent path
         String parentResourcePath = "/Applications/FormsApplication/1.0/FormsFolder";

        //Create a RepositoryInfomodelFactoryBean object
         RepositoryInfomodelFactoryBean infomodelFactory = new
RepositoryInfomodelFactoryBean(null);
```

```
        //Create a Resource object to add to the Repository
         Resource newResource = (Resource) infomodelFactory.newImage(
                            new Id(),
                            new Lid(),
                            "Loan.xdp");

        //Create a ResourceContent object that contains the content (file bytes)
        ResourceContent content = (ResourceContent) infomodelFactory.newResourceContent();

        //Create a Document that references an XDP file
        //to add to the Repository
        FileInputStream myForm = new FileInputStream("C:\\Adobe\Loan.xdp");
        Document form = new Document(myForm);

        //Set the description and the MIME type
        content.setDataDocument(form);
        content.setMimeType("application/vnd.adobe.xdp+xml");

        //Assign content to the Resource object
        newResource.setContent(content) ;

        //Set a description of the resource
        newResource.setDescription("An XDP file");

        //Commit to repository, and update resource
        //in memory (by assignment)
        Resource addResource = repositoryClient.writeResource(parentResourcePath,
newResource);

        //Get the description of the returned Resource object
        System.out.println("The description of the new resource is
"+addResource.getDescription());

        //Close the FileStream object
        myForm.close();

        } catch (Exception e) {

             e.printStackTrace();
           }
        }
}
```

Quick Start (SOAP mode): Listing resources using the Java API

The following Java code example lists resources that are located in `Applications/FormsApplication/1.0/FormsFolder`. (See "Listing Resources" on page 1041.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.bean.Resource;

//This quick start lists the content located in Applications/FormsApplication/1.0/FormsFolder
//Ensure that you create a AEM Forms application named Applications/FormsApplication using
Workbench
public class ListFiles {

    public static void main(String[] args) {

        try
        {
            //Set connection properties required to invoke AEM Forms
```

```
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // List all the files located in the
            String resourceFolderPath = "/Applications/FormsApplication/1.0/FormsFolder";

            // Retrieve the list of resources under the folder path
            List members = repositoryClient.listMembers(resourceFolderPath);

            // Print out the resources that were found
            System.out.println("The following resources were found:");
            for (int i = 0; i < members.size(); i++) {
                Resource r = (Resource)(members.get(i));
                System.out.println(
                    "Resource name: " +
                    r.getName() +
                    "  Resource Description: " +
                    r.getDescription()
                );
            }

        } catch (Exception e) {
             e.printStackTrace();
          }
    }
}
```

Quick Start (SOAP mode): Reading a resource using the Java API

The following Java code example reads a resource called *Loan.xdp* from the repository. The XDP file is located in `/Applications/FormsApplication/1.0/FormsFolder/`. (See "Reading Resources" on page 1043.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.bean.*;

//This quick start retrieves Loan.xdp from Applications/FormsApplication/1.0/FormsFolder
//Ensure that you create a AEM Forms application named FormsApplication using Workbench
public class ReadFile {


    public static void main(String[] args) {

        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();
```

```
connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Specify the path to the Loan.xdp
            String resourceUri =  "/Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";

            // Retrieve the XDP file
            Resource r = repositoryClient.readResource(resourceUri);

            // Print the resource verification message
            System.out.println(
                "Resource " +
                resourceUri +
                " was successfully retrieved." +
                "Resource content contains " +
                r.getContent().getDataDocument().length() +
                " bytes."
            );

        } catch (Exception e) {
            System.out.println(
                "Exception thrown while trying to read the file" +
                e.getMessage()
            );
        }
    }
}
```

Quick Start (SOAP mode): Updating a resource using the Java API

The following Java code example updates `/Applications/FormsApplication/1.0/FormsFolder` by modifying its description. (See "Updating Resources" on page 1045.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.bean.*;
import java.util.*;

//This quick start updates the description of Applications/FormsApplication/1.0/FormsFolder
//Ensure that you create a AEM Forms application named Applications/FormsApplication using
Workbench
public class UpdateResource {

    public static void main(String[] args) {

        // This example will update a resource in the AEM Forms repository
        try
        {
            //Set connection properties required to invoke AEM Forms
```

```
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Specify the URI of the resource to update
            String resourceUri =  "/Applications/FormsApplication/1.0/FormsFolder";

            // Retrieve the resource
            Resource resource = repositoryClient.readResource(resourceUri);

            // Update its description
            resource.setDescription("This folder stores XDP files");

            // Update the resource in the repository
            Resource updatedResource = repositoryClient.updateResource(
                resourceUri,
                resource,
                true
            );

            // Print the resource verification message
            System.out.println(
                "Resource " +
                resourceUri +
                "version " +
                updatedResource.getMajorVersion() +
                "." +
                updatedResource.getMinorVersion() +
                " was successfully updated."
            );

        } catch (Exception e) {
            System.out.println(
                "Exception thrown while trying to update the resource" +
                e.getMessage()
            );
        }
    }
}
```

Quick Start (SOAP mode): Searching for resources using the Java API

The following Java code example searches for Loan.xdp in `Applications/FormsApplication/1.0/FormsFolder`. (See "Searching for Resources" on page 1048.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.bean.*;
import com.adobe.repository.query.*;
import com.adobe.repository.query.sort.*;

//This quick start searches for Loan.xdp in Applications/FormsApplication/1.0/FormsFolder
//Ensure that you create a AEM Forms application named FormsApplication using Workbench
public class SearchResources {
```

```
    public static void main(String[] args) {

        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Specify the URI of the target folder
            String testFolderUri = "/Applications/FormsApplication/1.0/FormsFolder";

            // Specify the attribute name for which to search
            String name = "Loan.xdp";

            // Create a Query used in the search
            Query query = new Query();
            Query.Statement statement = new Query.Statement(
                Resource.ATTRIBUTE_NAME,
                Query.Statement.OPERATOR_BEGINS_WITH,
                name
            );
            statement.setNamespace(ResourceProperty.RESERVED_NAMESPACE_REPOSITORY);
            query.addStatement(statement);

            // Create the sort order used in the search
            SortOrder sortOrder = new SortOrder();
            SortOrder.Element element = new SortOrder.Element(Resource.ATTRIBUTE_NAME, true);
            sortOrder.addSortElement(element);

            // Search for the resources
            List listProperties = repositoryClient.searchProperties(
                testFolderUri,
                query,
                ResourceCollection.DEPTH_INFINITE,
```

```
                0,
                10,
                sortOrder
            );

            // Display the resources that were found
            System.out.println("The following resources were found:");
            for (int i = 0; i < listProperties.size(); i++) {
                Resource r = (Resource)(listProperties.get(i));
                System.out.println(r.getName());
            }

        } catch (Exception e) {
            System.out.println(
                "An exception occurred while attempting to search for resources." +
                e.getMessage()
            );
        }
    }
}
```

Quick Start (SOAP mode): Creating relationships between resources using the Java API

The following Java code example creates a relationship between two resources in the AEM Forms repository. (See "Creating Resource Relationships" on page 1051.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
```

```
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.*;
import com.adobe.repository.infomodel.bean.*;

public class CreateRelationship {

    public static void main(String[] args) {

        // This example creates a relationship between two resources in the AEM Forms repository.
        // First, two resources are created.
        // A dependence relationship between the two resources will then be established and
verified.
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new
ResourceRepositoryClient(myFactory);

            // Create a RepositoryInfomodelFactoryBean needed for creating resources
            RepositoryInfomodelFactoryBean repositoryInfomodelFactory = new
RepositoryInfomodelFactoryBean(null);
```

```
// Specify the URI of the target folder for writing the resource
String testFolderUri = "/Applications/FormsApplication/1.0/FormsFolder";

// Create the resources to be written to the folder
Resource testResource1 = repositoryInfomodelFactory.newResource(
    new Id(),
    new Lid(),
    "FormFolderA"
);

Resource testResource2 = repositoryInfomodelFactory.newResource(
    new Id(),
    new Lid(),
    "FormFolderB"
);

// Set the resources' descriptions
testResource1.setDescription("test resource1");
testResource2.setDescription("test resource2");

// Write the resources to the folder
repositoryClient.writeResource(testFolderUri, testResource1);
repositoryClient.writeResource(testFolderUri, testResource2);

// Retrieve the resources' URIs
String resourceUri1 = testFolderUri + "/" + testResource1.getName();
String resourceUri2 = testFolderUri + "/" + testResource2.getName();

// Retrieve the resources to verify that they were successfully written
Resource r1 = repositoryClient.readResource(resourceUri1);
Resource r2 = repositoryClient.readResource(resourceUri2);

// Create a relationship between the two resources
repositoryClient.createRelationship(
    resourceUri1,
    resourceUri2,
    Relation.TYPE_DEPENDANT_OF,
    true
);

// Verify the relationship
```

```
            List relations = repositoryClient.getRelated(
                resourceUri1,
                true,
                Relation.TYPE_DEPENDANT_OF
            );

            // Print the relationship
            for (int i = 0; i < relations.size(); i++) {
                Resource r = (Resource)(relations.get(i));
                System.out.println("Related resource: " + r.getName());
            }

        } catch (Exception e) {
            System.out.println(
                "Exception thrown while trying to create the relationship" +
                e.getMessage()
            );
        }
    }
}
```

Quick Start (SOAP mode): Locking a resource using the Java API

The following Java code example locks /Applications/FormsApplication/1.0/FormsFolder/Loan.xdp. (See "Locking Resources" on page 1054.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
```

```
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.bean.*;

public class LockFile {

    public static void main(String[] args) {

        // This example will lock and unlock a resource in the AEM Forms repository.
        try {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Specify the URI of the resource to lock
            String resourceUri = "/Applications/FormsApplication/1.0/FormsFolder/Loan.xdp";

            // Lock the resource
            repositoryClient.lockResource(
                resourceUri,
                Lock.SCOPE_EXCLUSIVE,
                Lock.DEPTH_ZERO
            );

            // Retrieve the locks on the resource
            List locks = repositoryClient.getLocks(resourceUri);
```

```
            // Print out the locks for the resource
            System.out.println("The following locks now exist for the resource:");
            for (int i = 0; i < locks.size(); i++) {
                Lock l = (Lock)(locks.get(i));
                System.out.println(
                    "Lock owner: " +
                    l.getOwnerUserId() +
                    "  Lock depth: " +
                    l.getDepth() +
                    " Lock scope: " +
                    l.getType()
                );
            }

            // Unlock the resource
            String lockToken = repositoryClient.unlockResource(resourceUri);

        } catch (Exception e) {
            System.out.println(
                "Exception thrown while trying to lock the file" +
                e.getMessage()
            );
        }
    }
}
```

Quick Start (SOAP mode): Managing access control lists using the Java API

The following Java code example reads and creates access control lists (ACLs) in the repository.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
```

```
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.bean.*;

public class UseACL {

    public static void main(String[] args) {

        // This example will read and create access control lists for resources in the AEM Forms
repository.
        try {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Specify the URI of the resource to be used
            String resourceUri = "/Applications/FormsApplication";

            // Retrieve the access control list for the resource
            AccessControlList acl = repositoryClient.readAccessControlList(resourceUri);

            // Retrieve a list of the users having access permissions
            List users = acl.getUsersWithPermissions();
```

```
            // Print out the list of users
            System.out.println("The following users have permissions:");
            for (int i = 0; i < users.size(); i++) {
                String user = (String)(users.get(i));
                System.out.println("User identifier: " + user);
            }

            // Set up a new access control list
            acl = new AccessControlList();

            // Retrieve a user identifier to be used in the access control list
            String userId = (String)(users.get(0));

            // Create traversal permissions for the user
            List permissions = new ArrayList();
            permissions.add(AccessControlEntry.READ_METADATA_USER_PERM);
            permissions.add(AccessControlEntry.READ_CONTENT_USER_PERM);
            acl.setPermissionsForUser(userId, permissions);

            // Set the access control list for the folder
            repositoryClient.writeAccessControlList(resourceUri, acl, true);

            // Print out confirmation message
            System.out.println("User " + userId + " has traversal permissions for the folder");

        } catch (Exception e) {
            System.out.println(
                "Exception thrown while trying to manage access control lists" +
                e.getMessage()
            );
        }
    }
}
```

Quick Start (SOAP mode): Deleting a resource using the Java API

The following Java code example deletes Loan.xdp from `Applications/FormsApplication/1.0/FormsFolder`. If this XDP file is not located in this folder, an exception is thrown. (See "Deleting Resources" on page 1058.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-repository-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. commons-code-1.3.jar
    * 7. jacorb.jar (use a different JAR file if the forms server is not deployed on JBoss)
    * 8. jnp-client.jar (use a different JAR file if the forms server is not deployed on JBoss)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.*;
import com.adobe.repository.infomodel.bean.*;
import com.adobe.repository.RepositoryException;
import com.adobe.idp.Document;


// This quick start deletes Loan.xdp from Applications/FormsApplication/1.0/FormsFolder
//If this XDP is not located in this folder, an exception is thrown
//Ensure that you create a AEM Forms application named FormsApplication using Workbench
public class DeleteResource {

    public static void main(String[] args) {
```

```
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create the service client factory
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a ResourceRepositoryClient object using the service client factory
            ResourceRepositoryClient repositoryClient = new ResourceRepositoryClient(myFactory);

            // Create a RepositoryInfomodelFactoryBean needed for creating resources
            RepositoryInfomodelFactoryBean repositoryInfomodelFactory = new
RepositoryInfomodelFactoryBean(null);

            // Specify the URI of the target folder from which the resource is deleted
            String testFolderUri = "/Applications/FormsApplication/1.0/FormsFolder";

            // Create the resource to be written to the folder
            Resource testResource = repositoryInfomodelFactory.newResource(
                new Id(),
                new Lid(),
                "Loan.xdp"
            );

            // Retrieve the resource's URI
            String resourceUri = testFolderUri + "/" + testResource.getName();
```

```
            // Retrieve the resource to verify that it exists
            Resource r = repositoryClient.readResource(resourceUri);

            // Print the resource verification message
            System.out.println(r.getName() +" is about to be deleted");

            // Delete the resource
            repositoryClient.deleteResource(resourceUri);

        } catch (Exception e) {
            System.out.println(
                "Exception thrown while trying to delete the resource" +
                e.getMessage()
            );
        }
    }
}
```

Document Security Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Rights Management service:

"Quick Start (SOAP mode): Creating a policy using the Java API" on page 329

Quick Start (SOAP mode): Modifying a policy using the Java API

"Quick Start (SOAP mode): Deleting a policy using the Java API" on page 333

Quick Start (SOAP mode): Applying a policy to a PDF document using the Java API

"Quick Start (SOAP mode): Removing a policy from a PDF document using the Java API" on page 337

"Quick Start (SOAP mode): Revoking a document using the Java API" on page 339

"Quick Start (SOAP mode): Reinstating access to a revoked document using the Java API" on page 344

"Quick Start (SOAP mode): Inspecting policy protected PDF documents using the Java API" on page 341

"Quick Start (SOAP mode): Creating a PDF Watermark using the Java API" on page 346

"Quick Start(SOAP mode): Modifying a watermark using the Java API" on page 353

"Quick Start (SOAP mode): Searching for events using the Java API" on page 355

Quick Start (SOAP mode): Applying a policy to a Word document using the Java API

"Quick Start (SOAP mode): Removing a policy from a Word document using the Java API" on page 359

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Start located in Programming with AEM Forms are based on the Forms server operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Creating a policy using the Java API

The following Java code example creates a new policy named *Allow Copy*. The policy set to which the policy is added is named *Global Policy Set*. This policy set exists by default. (See "Creating Policies" on page 832.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
```

```
import com.adobe.idp.um.api.infomodel.Principal;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;

public class CreatePolicy {

    public static void main(String[] args) {

        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a RightsManagementClient object
            RightsManagementClient rightsClient = new RightsManagementClient(myFactory);

            //Create a Policy object that represents the new policy
            Policy myPolicy = InfomodelObjectFactory.createPolicy();

            //Set policy attributes that are used by the policy
            myPolicy.setName("Allow Copy");
            myPolicy.setDescription("This policy enables users to copy information from the PDF
document");
            myPolicy.setPolicySetName("Global Policy Set");
            myPolicy.setOfflineLeasePeriod(30);
            myPolicy.setTracked(true);

            //Set the validity period to 30 days
            ValidityPeriod validityPeriod = InfomodelObjectFactory.createValidityPeriod();
            validityPeriod.setRelativeExpirationDays(30);
            myPolicy.setValidityPeriod(validityPeriod);

            //Create a PolicyEntry object
            PolicyEntry myPolicyEntry = InfomodelObjectFactory.createPolicyEntry();

            //Specify the permissions
            Permission onlinePermission =
InfomodelObjectFactory.createPermission(Permission.OPEN_ONLINE) ;
            Permission copyPermission =
InfomodelObjectFactory.createPermission(Permission.COPY);
```

```
            //Add permissions to the policy entry
            myPolicyEntry.addPermission(onlinePermission);
            myPolicyEntry.addPermission(copyPermission);

            //Create principal object
            Principal publisherPrincipal =
InfomodelObjectFactory.createSpecialPrincipal(InfomodelObjectFactory.PUBLISHER_PRINCIPAL);

            //Add a principal object to the policy entry
            myPolicyEntry.setPrincipal(publisherPrincipal);

            //Attach the policy entry to the policy
            myPolicy.addPolicyEntry(myPolicyEntry);

            //Register the policy
            PolicyManager policyManager = rightsClient.getPolicyManager();
            policyManager.registerPolicy(myPolicy,"Global Policy Set");
            }
        catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
}
```

Quick Start (SOAP mode): Modifying a policy using the Java API

The following Java code example modifies a policy named *Allow Copy* by setting the offline lease period to 40 days. (See "Modifying Policies" on page 839.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;

public class ModifyPolicySoap {

    public static void main(String[] args) {
        try
          {
                //Set connection properties required to invoke AEM Forms using SOAP mode
                Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
                connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

                //Create a ServiceClientFactory object
                ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

                //Create a RightsManagementClient object
                RightsManagementClient rightsClient = new RightsManagementClient(myFactory);

                //Create a policy manager instance
                PolicyManager policyManager = rightsClient.getPolicyManager();

                //Retrieve an existing policy
                Policy myPolicy = policyManager.getPolicy(
```

```
                    "Global Policy Set",
                    "Allow Copy") ;

                //Modify policy attributes
                myPolicy.setOfflineLeasePeriod(40);
                myPolicy.setTracked(true);

                //Set the validity period to 40 days
                ValidityPeriod validityPeriod = InfomodelObjectFactory.createValidityPeriod();
                validityPeriod.setRelativeExpirationDays(40);
                myPolicy.setValidityPeriod(validityPeriod);

                //Update the policy
                policyManager.updatePolicy(myPolicy) ;
                 }

        catch (Exception ee)
            {
             ee.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Deleting a policy using the Java API

The following Java code example deletes a policy named *Allow Copy*. (See "Deleting Policies" on page 842.)

```
/*
    * * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *   your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *   These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;

public class DeletePolicy {

    public static void main(String[] args) {
     try
      {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

```
        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
         RightsManagementClient rightsClient = new RightsManagementClient(myFactory);

         //Create a policy manager instance
         PolicyManager policyManager = rightsClient.getPolicyManager();

         //Delete the AllowCopy policy
         policyManager.deletePolicy("Global Policy Set", "Allow Copy") ;
          }
    catch (Exception ee)
        {
         ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Applying a policy to a PDF document using the Java API

The following Java code example applies a policy named *Allow Copy* to a PDF document named *Loan.pdf*. The policy set to which the policy is added is named *Global Policy Set*. The policy-protected document is saved as a PDF file named *PolicyProtectedLoanDoc.pdf*. (See "Applying Policies to PDF Documents" on page 844.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
```

```
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.RMSecureDocumentResult;
import com.adobe.livecycle.rightsmanagement.client.*;

public class ApplyPolicySoap {

    public static void main(String[] args) {
    try
     {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(factory);

        //Reference a PDF document to which a policy is applied
        FileInputStream is = new FileInputStream("C:\\Adobe\Loan.pdf");
        Document inPDF = new Document(is);

        //Create a Document Manager object
        DocumentManager  documentManager = rightsClient.getDocumentManager();

        //Apply a policy to the PDF document
        RMSecureDocumentResult rmSecureDocument =  documentManager.protectDocument(
            inPDF,
```

```
            "LoanPDF",
            "Global Policy Set",
            "Allow Copy",
            null,
            null,
            null);

        //Retrieve the policy-protected PDF document
        Document protectPDF = rmSecureDocument.getProtectedDoc();

        //Save the policy-protected PDF document
        File myFile = new File("C:\\Adobe\PolicyProtectedLoanDoc.pdf");
        protectPDF.copyToFile(myFile);
      }
    catch (Exception ee)
     {
        ee.printStackTrace();
     }
    }
}
```

Quick Start (SOAP mode): Removing a policy from a PDF document using the Java API

The following code example removes a policy from a PDF document named *PolicyProtectedLoanDoc.pdf*. The unsecured PDF document is saved as *unProtectedLoan.pdf*. (See "Removing Policies from PDF Documents" on page 848.)

```
/*
    * * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;


public class RemovePolicy {

    public static void main(String[] args) {

        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a RightsManagementClient object
            RightsManagementClient rightsClient = new RightsManagementClient(factory);
```

```
            //Reference a policy-protected PDF document from which to remove a policy
            FileInputStream is = new FileInputStream("C:\\Adobe\PolicyProtectedLoanDoc.pdf");
            Document inPDF = new Document(is);

            //Create a Document Manager object
            DocumentManager  documentManager = rightsClient.getDocumentManager();

            //Remove a policy from the policy-protected PDF document
            Document unsecurePDF =  documentManager.removeSecurity(inPDF);

            //Save the unsecured PDF document
            File myFile = new File("C:\\Adobe\UnProtectedLoan.pdf");
            unsecurePDF.copyToFile(myFile);
          }

         catch (Exception ee)
         {
          ee.printStackTrace();
         }
    }
}
```

Quick Start (SOAP mode): Revoking a document using the Java API

The following Java code example revokes a policy-protected document named *PolicyProtectedLoanDoc.pdf*. A revised PDF document is located at the following URL location `http://[server]:[port]/RightsManagement/UpdatedLoan.pdf`. (See "Revoking Access to Documents" on page 851.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
```

```
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

import java.net.URL;

import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;


public class RevokeDocument {

    public static void main(String[] args) {

        try
        {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
```

```
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(factory);

        //Reference a policy-protected PDF document to revoke
        FileInputStream is = new FileInputStream("C:\\Adobe\PolicyProtectedLoanDoc.pdf");
        Document inPDF = new Document(is);

        //Create a Document Manager object
        DocumentManager  documentManager = rightsClient.getDocumentManager();

        //Obtain the license identifier value of the policy-protected document
        String revokeLic = documentManager.getLicenseId(inPDF);

        //Create a LicenseManager object
        LicenseManager licManager = rightsClient.getLicenseManager();

        //Specify the URL to where an updated document is located
        URL myURL = new URL("http://[server]:[port]/RightsManagement/UpdatedLoan.pdf");

        //Revoke the policy-protected PDF document
        licManager.revokeLicense(revokeLic, License.DOCUMENT_REVISED, myURL);
        }
     catch (Exception ee)
       {
            ee.printStackTrace();
       }
    }
}
```

Quick Start (SOAP mode): Inspecting policy protected PDF documents using the Java API

The following Java code example inspects a policy-protected PDF document named *PolicyProtectedLoanDoc.pdf*. (See "Inspecting Policy Protected PDF Documents" on page 857.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
) * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.io.FileInputStream;
import java.util.Properties;
import java.util.Date;
import java.util.Calendar;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
```

```
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.*;

public class InspectDocument {

    public static void main(String[] args) {

         try
          {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a RightsManagementClient object
            RightsManagementClient rightsClient = new RightsManagementClient(factory);

            //Reference a policy-protected PDF document to inspect
            FileInputStream is = new FileInputStream("C:\\Adobe\PolicyProtectedLoanDoc.pdf");
            Document inPDF = new Document(is);

            //Create a Document Manager object
            DocumentManager  documentManager = rightsClient.getDocumentManager();

            //Inspect the policy-protected document
            RMInspectResult inspectResult = documentManager.inspectDocument(inPDF);

            //Get the document name
            String documentName = inspectResult.getDocName();
```

```
            //Get the name of the policy
            String policyName = inspectResult.getPolicyName();

            //Get the name of the document publisher
            String pubName = inspectResult.getPublisherName();

            //Display the name of the policy-protected document and the policy
            System.out.println("The policy protected document "+documentName +" is protected with
the policy "+policyName +". The name of the publisher is "+pubName+".");

              }
        catch (Exception ee)
            {
             ee.printStackTrace();
            }
    }
}
```

Quick Start (SOAP mode): Reinstating access to a revoked document using the Java API

The following Java code example reinstates access to a revoked PDF document named *PolicyProtectedLoanDoc.pdf*. (See "Reinstating Access to Revoked Documents" on page 854.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.io.FileInputStream;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;

public class ReinstateDocument {

    public static void main(String[] args) {

        try
        {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(factory);

        //Reference a revoked PDF document
        FileInputStream is = new FileInputStream("C:\\Adobe\PolicyProtectedLoanDoc.pdf");
        Document inPDF = new Document(is);
```

```
        //Create a Document Manager object
        DocumentManager  documentManager = rightsClient.getDocumentManager();

        //Obtain the license identifier value of the revoked PDF document
        String revokeLic = documentManager.getLicenseId(inPDF);

        //Create a LicenseManager object
        LicenseManager licManager = rightsClient.getLicenseManager();

        //Reinstate access to the revoked document
        licManager.unrevokeLicense(revokeLic);
        }
     catch (Exception ee)
       {
            ee.printStackTrace();
       }
    }
}
```

Quick Start (SOAP mode): Creating a PDF Watermark using the Java API

The following Java code example creates a new PDF watermark named 'Sample PDF Watermark'. This watermark contains a single element (See "Creating Watermarks" on page 861).

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 * These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
```

```
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */


import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Properties;
import com.adobe.edc.sdk.SDKException;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.RightsManagementClient;
import com.adobe.livecycle.rightsmanagement.client.WatermarkManager;
import com.adobe.livecycle.rightsmanagement.client.infomodel.InfomodelObjectFactory;
import com.adobe.livecycle.rightsmanagement.client.infomodel.PDRLException;
import com.adobe.livecycle.rightsmanagement.client.infomodel.Watermark2;
import com.adobe.livecycle.rightsmanagement.client.infomodel.Watermark2Element;

public class PDFWatermarksSOAPMode {
    public static void main(String[] args) {
        // Set connection properties required to invoke Adobe AEM Forms
        // using SOAP mode
        try {
            Properties connectionProps = new Properties();
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
                    "http://[server]:[port]/");
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
                    ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_SERVER_TYPE,
                    ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
                    "administrator");
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
                    "password");

            // Create a ServiceClientFactory object.
            ServiceClientFactory serviceClient = ServiceClientFactory
                    .createInstance(connectionProps);

            // Create a Document Security ServiceClient object.
            RightsManagementClient rmClient = new RightsManagementClient(
                    serviceClient);
```

```
            // Get the watermark manager which is used to add, delete or update
            // watermarks.
            WatermarkManager watermarkManager = rmClient.getWatermarkManager();

            // Registering and adding elements to the new watermarks.
            // Create a Watermark2 object using the InfomodelObjectFactory.
            Watermark2 newWatermark = InfomodelObjectFactory.createWatermark2();
            // Create a Watermark2Element object using the
            // InfomodelObjectFactory.
            Watermark2Element element1 = InfomodelObjectFactory
                    .createWatermark2Element();
            // Set the various properties such as name, description,custom text
            // and date.
            element1.setName("PDF element");
            element1.setDescription("This is a Sample PDF element.");
            // Set type of the watermark to Watermark2Element.TYPE_PDF.
            element1.setType(Watermark2Element.TYPE_PDF);
            // Create an IDf document form a PDF file.
            Document doc = new Document(new FileInputStream("C:\\Sample.pdf"));
            element1.setPDFContent(doc, "Watermark Doc");
            // Set the properties for this such rotation,opacity.
            element1.setOpacity(50);
            element1.setRotation(45);//45 degrees rotation.
            element1.setStartPage(5);
            element1.setStartPage(15);//Watermark appears only on pages 10 to 15.
            // Add it to the watermark.
            newWatermark.addWatermarkElement(element1);
            // Set the watermark name.
            newWatermark.setName("Sample PDF Watermark");
            // Register it.
            watermarkManager.registerWatermark2(newWatermark);
        } catch (PDRLException e) {
            System.out.println(e.getCause());
        } catch (SDKException e) {
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Creating a Text Watermark using the Java API

The following Java code example creates a new Text watermark named *Sample Text Watermark*. This watermark contains a single element.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */

import com.adobe.livecycle.rightsmanagement.client.RightsManagementClient;
import com.adobe.livecycle.rightsmanagement.client.WatermarkManager;
import com.adobe.livecycle.rightsmanagement.client.infomodel.InfomodelObjectFactory;
import com.adobe.livecycle.rightsmanagement.client.infomodel.PDRLException;
import com.adobe.livecycle.rightsmanagement.client.infomodel.Watermark2;
import com.adobe.livecycle.rightsmanagement.client.infomodel.Watermark2Element;
import com.adobe.edc.sdk.SDKException;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import java.util.Properties;

public class TextWatermarks {
    public static void main(String[] args) {
        try {
```

```
// Set connection properties required to invoke Adobe Document
// Services using SOAP mode
Properties connectionProps = new Properties();
connectionProps.setProperty(
        ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
        "http://[server]:[port]/");
connectionProps.setProperty(
        ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
        ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
connectionProps.setProperty(
        ServiceClientFactoryProperties.DSC_SERVER_TYPE,
        ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);
connectionProps.setProperty(
        ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
        "administrator");
connectionProps.setProperty(
        ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
        "password");

// Create a ServiceClientFactory object.
ServiceClientFactory serviceClient = ServiceClientFactory
        .createInstance(connectionProps);

// Create a Document Security ServiceClient object.
RightsManagementClient rmClient = new RightsManagementClient(
        serviceClient);
// Get the watermark manager which is used to add, delete or update
// watermarks.
WatermarkManager watermarkManager = rmClient.getWatermarkManager();

// Registering and adding elements to the new watermarks.
// Create a Watermark2 object using the InfomodelObjectFactory.
Watermark2 newWatermark = InfomodelObjectFactory.createWatermark2();
// Create a Watermark2Element object using the
// InfomodelObjectFactory.
Watermark2Element element1 = InfomodelObjectFactory
        .createWatermark2Element();
// Set the various properties such as name, description,custom text
// and date.
element1.setName("First element");
element1.setDescription("This is a Sample Text Watermark Element.");
element1.setUserNameIncluded(true);
element1.setShowOnPrint(false);// This element will not appear on
                               // print, but will only appear on
                               // screen.
// Set the type of the watermark element. It can either be
// Watermark2Element.TYPE_TEXT or Watermark2Element.TYPE_PDF.
element1.setType(Watermark2Element.TYPE_TEXT);
// Provide opacity, rotation page range and other such settings.
element1.setOpacity(50); // Opacity set to 50%.
element1.setEndPage(1);// The watermark will appear only on first
                       // page, start page is 1 by default.

// Create a new element.
Watermark2Element element2 = InfomodelObjectFactory
        .createWatermark2Element();
element2.setName("Second element");
```

```
            element2.setCustomText("Confidential");
            // Set type to Watermark2Element.TYPE_TEXT.
            element2.setType(Watermark2Element.TYPE_TEXT);
            // Provide opacity, rotation page range and other such settings.
            element2.setFontName("Times New Roman");
            element2.setFontSize(30);
            element2.setOpacity(30);// 30% opacity.
            element2.setRotation(45);// 45 degrees rotation.
            element2.setShowOnScreen(false);// This element will not appear on
                                            // screen, but will appear when we
                                            // print the document.

            // Add these elements to the watermark in the order in you want them
            // to be applied.
            newWatermark.addWatermarkElement(element1);// Will be applied first.
            newWatermark.addWatermarkElement(element2);// Will be applied on top
                                                       // of it.
            newWatermark.setName("Sample Text Watermark");
            watermarkManager.registerWatermark2(newWatermark);
        } catch (PDRLException e) {
            System.out.println(e.getCause());
        } catch (SDKException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Modifying a Text Watermark using the Java API

The following Java code example modifies a watermark named 'Sample Text Watermark' and sets the opacity of the first element to 100.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7. commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
```

```
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */


import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;

public class ModifyWatermarks {

    public static void main(String[] args) {
        try {
            // Set connection properties required to invoke AEM Forms using
            // SOAP mode
            Properties connectionProps = new Properties();
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
                    "http://[server]:[port]");
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
                    ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
                    "administrator");
            connectionProps.setProperty(
                    ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
                    "password");

            // Create a ServiceClientFactory instance
            ServiceClientFactory factory = ServiceClientFactory
                    .createInstance(connectionProps);

            // Create a RightsManagementClient object
            RightsManagementClient rightsClient = new RightsManagementClient(
                    factory);
```

```
            // Create a WatermarkManager object
            WatermarkManager myWatermarkManager = rightsClient
                    .getWatermarkManager();

            // Get the watermark to modify by name
            Watermark2 myWatermark = myWatermarkManager
                    .getWatermarkByName2("Sample Text Watermark");

            // Get the elements in the watermark.
            ArrayList<Watermark2Element> elements = myWatermark
                    .getWatermarkElements();
            // Iterate through the list and modify the opacity attribute of each
            // element.
            for (Iterator<Watermark2Element> iter = elements.iterator(); iter
                    .hasNext();) {
                Watermark2Element elem = iter.next();
                elem.setOpacity(100);
            }
            // Update the watermark
            myWatermarkManager.updateWatermark2(myWatermark);
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Quick Start(SOAP mode): Modifying a watermark using the Java API

The following Java code example modifies a watermark named *Confidential* by modifying the value of the `opacity` attribute to 80.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
```

```
    * The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;


public class ModifyWatermarks {

    public static void main(String[] args) {

    try
        {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(factory);
```

```
        //Create a WatermarkManager object
        WatermarkManager myWatermarkManager =  rightsClient.getWatermarkManager();

        //Get the watermark to modify by name
        Watermark myWatermark = myWatermarkManager.getWatermarkByName("Confidential");

        //Modify the opacity attribute
        myWatermark.setOpacity(80);

        //Update the watermark
        myWatermarkManager.updateWatermark(myWatermark);
        }

    catch (Exception ex)
        {
        ex.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Searching for events using the Java API

The following Java code example searches for the create policy event.

```
/*
    * * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
```

```
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.Event;
import com.adobe.livecycle.rightsmanagement.client.infomodel.EventSearchFilter;

public class SearchEvents {

    public static void main(String[] args) {

        try
        {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(factory);

        //Create a EventManager instance
        EventManager eventManager = rightsClient.getEventManager();

        //Create a EventSearchFilter object
        EventSearchFilter eventSearchFilter = new EventSearchFilter();

        //Search for the POLICY_CREATE_EVENT event
```

```
        eventSearchFilter.setEventCode(EventManager.POLICY_CREATE_EVENT);
        Event[] events = eventManager.searchForEvents(eventSearchFilter,20) ;

        //Retrieve information about each event
        int index = events.length;
        Calendar rightNow = Calendar.getInstance();

        for (int i=0; i<index;i++)
        {
            Event myEvent = events[i];

            Date myDate = myEvent.getTimestamp();
            rightNow.setTime(myDate);
            System.out.println("Policy Created on " + rightNow.getTime().toString());
        }
         }
    catch (Exception ee)
        {
         ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP): Applying a policy to a Word document using the Java API

The following Java code example applies a policy named *Allow Copy* to a Word document named *Loan.doc*. The policy set to which the policy is added is named *Global Policy Set*. The policy-protected document is saved as a DOC file named *PolicyProtectedLoanDoc.doc*. (See "Applying Policies to PDF Documents" on page 844.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
```

```
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.RMSecureDocumentResult;
import com.adobe.livecycle.rightsmanagement.client.*;

public class ApplyPolicyWordDocument {

    public static void main(String[] args) {
    try
     {
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(factory);
```

```
        //Reference a Word document to which a policy is applied
        FileInputStream is = new FileInputStream("C:\\Adobe\Loan.doc");
        Document inPDF = new Document(is);

        //Create a Document Manager object
        DocumentManager  documentManager = rightsClient.getDocumentManager();

        //Apply a policy to the Word document
        RMSecureDocumentResult rmSecureDocument=  documentManager.protectDocument(
            inPDF,
            "Loan.doc",
            "Global Policy Set",
            "Allow Copy",
            null,
            null,
            null);

        //Retrieve the policy-protected Word document
        Document protectPDF = rmSecureDocument.getProtectedDoc();

        //Save the policy-protected Word document
        File myFile = new File("C:\\PolicyProtectedLoanDoc.doc");
        protectPDF.copyToFile(myFile);
      }
    catch (Exception ee)
     {
        ee.printStackTrace();
     }
    }
}
```

Quick Start (SOAP mode): Removing a policy from a Word document using the Java API

The following code example removes a policy from a Word document named *PolicyProtectedLoanDoc.doc*. The unsecured Word document is saved as *unProtectedLoan.doc*. (See "Removing Policies from Word Documents" on page 876.)

```
/*
    * * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-rightsmanagement-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdkK/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
```

```
public class RemovePolicyWordDocument {

    public static void main(String[] args) {

        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a RightsManagementClient object
            RightsManagementClient rightsClient = new RightsManagementClient(factory);

            //Reference a policy-protected Word document from which to remove a policy
            FileInputStream is = new FileInputStream("C:\\PolicyProtectedLoanDoc.doc");
            Document inPDF = new Document(is);

            //Create a Document Manager object
            DocumentManager  documentManager = rightsClient.getDocumentManager();

            //Remove a policy from the policy-protected Word document
            Document unsecurePDF =  documentManager.removeSecurity(inPDF);

            //Save the unsecured Word document
            File myFile = new File("C:\\Adobe\UnProtectedLoan.doc");
            unsecurePDF.copyToFile(myFile);
         }

        catch (Exception ee)
        {
         ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Creating an abstract policy using the Java API

The following Java code example creates a new abstract policy named AllowCopy. The policy set to which the policy is added is named Global Policy Set. This policy set exists by default. (See Creating Policies.)

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */


import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;

public class CreateAbstractPolicySoap {
```

```
    public static void main(String args[]) {

    try{

     //Set connection properties required to invoke forms server using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClientFa
ctoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "Jboss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(myFactory);

        AbstractPolicy abstractPolicy = InfomodelObjectFactory.createAbstractPolicy();

        abstractPolicy.setName("AllowCopy");
       abstractPolicy.setDescription("This abstract policy helps users to create policy that copy
information from the PDF document");
        abstractPolicy.setPolicySetName("Global Policy Set");

        abstractPolicy.setOfflineLeasePeriod(30);
        abstractPolicy.setTracked(true);
        abstractPolicy.setEncryptAttachmentsOnly(false);

        ValidityPeriod validityPeriod = InfomodelObjectFactory.createValidityPeriod();
        validityPeriod.setRelativeExpirationDays(30);

        abstractPolicy.setValidityPeriod(validityPeriod);

        //Adding publisher permissions.
        AbstractPolicyEntry userPolicyEntry = InfomodelObjectFactory.createAbstractPolicyEntry();

        Permission onlinePermission =
InfomodelObjectFactory.createPermission(Permission.OPEN_ONLINE);
        Permission copyPermission = InfomodelObjectFactory.createPermission(Permission.COPY);

        userPolicyEntry.addPermission(onlinePermission);
```

```
        userPolicyEntry.addPermission(copyPermission);

        abstractPolicy.addAbstractPolicyEntry(userPolicyEntry);

        AbstractPolicyManager abstractPolicyManager = rightsClient.getAbstractPolicyManager();

        abstractPolicyManager.registerAbstractPolicy(abstractPolicy, "Global Policy Set");

        AbstractPolicy abstractPolicy1 = abstractPolicyManager.getAbstractPolicy("Global Policy
Set","AllowCopy");

        System.out.println("The Abstract Policy was successfully created:" +
abstractPolicy1.getName());

        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Modifying an abstract policy using the Java API

The following Java code example modifies an abstract policy named AllowCopy. The policy set in which the policy is modfied is named Global Policy Set. This policy set exists by default. (See Creating Policies.)

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
```

```
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.client.infomodel.*;

public class ModifyingAbstractPolicySoap {

    public static void main(String args[]) {

    try{

     //Set connection properties required to invoke forms server using SOAP mode
         Properties connectionProps = new Properties();
       connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
       connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "Jboss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory object
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(myFactory);

      AbstractPolicyManager abstractPolicyManager = rightsClient.getAbstractPolicyManager();

      AbstractPolicy abstractPolicy = abstractPolicyManager.getAbstractPolicy("Global Policy
Set","AllowCopy");
```

```
        //Modify policy attributes
        abstractPolicy.setOfflineLeasePeriod(40);
        abstractPolicy.setTracked(true);

        //Set the validity period to 40 days
        ValidityPeriod validityPeriod = InfomodelObjectFactory.createValidityPeriod();
        validityPeriod.setRelativeExpirationDays(40);
        abstractPolicy.setValidityPeriod(validityPeriod);

        abstractPolicyManager.updateAbstractPolicy(abstractPolicy);

        System.out.println("The Abstract Policy was updated:" + abstractPolicy.getName());

        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Deleting an abstract policy using the Java API

The following Java code example deletes an abstract policy named AllowCopy. The policy set from which the policy is deleted is named Global Policy Set. This policy set exists by default. (See Creating Policies.)

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
```

```
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming with AEM Forms
 * with forms server
 */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;

public class DeleteAbstractPolicySoap {

    public static void main(String args[]) {

    try{

     //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
       connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
       connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "Jboss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory object
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a RightsManagementClient object
        RightsManagementClient rightsClient = new RightsManagementClient(myFactory);

       AbstractPolicyManager abstractPolicyManager = rightsClient.getAbstractPolicyManager();

        abstractPolicyManager.deleteAbstractPolicy("Global Policy Set", "AllowCopy");

        System.out.println("The Abstract Policy was deleted:");

        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Protect a PDF in Statement Workflow for an Existing User, using the Java API

The following Java code example demonstrates the method to protect a Document in Statement Workflow, for an existing User.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.RMSecureDocumentResult;
import com.adobe.edc.common.dto.PublishLicenseDTO;
```

```
public class protectStatementWorkFlowExistingUserSoap {

    public static void main(String args[]) {

    try{

       //Set connection properties required to invoke forms server using SOAP mode
            Properties connectionProps = new Properties();
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "Jboss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

             //Create a ServiceClientFactory instance
           ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

             //Create a RightsManagementClient object
             RightsManagementClient rightsClient = new RightsManagementClient(factory);

             DocumentManager documentManager = rightsClient.getDocumentManager();

             //Reference a PDF document to which a policy is applied
             FileInputStream is = new FileInputStream("C:\\Adobe\\Sample.pdf");
             Document inPDF = new Document(is);

             //Get the License for existing user
              PublishLicenseDTO publishLicense =
documentManager.getPublishLicenseForUser("DefaultDom","wblue");

             //protect the PDF document using license
             RMSecureDocumentResult rmSecureDocument = documentManager.protectDocument(inPDF,
publishLicense);

             //Retrieve the policy-protected PDF document
             Document protectedDocument = rmSecureDocument.protectedDoc;

             //Save the policy-protected PDF document
             String outputFile = "C:\\Adobe\\PolicyProtectedSample.pdf";
             File myFile = new File(outputFile);
             protectedDocument.copyToFile(myFile);

             System.out.println("Protected the PDF With policy");

         }catch(Exception ex){
             ex.printStackTrace();
         }

}

}
```

Quick Start (SOAP mode): Protect a PDF in Statement Workflow for a new User, using the Java API

The following Java code example demonstrates how you can protect a document in Statement Workflow. This is a two-step process:

- A new User, License and Policy are created.
- The User is associated with the License and Policy, and the document is Protected.

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-rightsmanagement-client.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. activation.jar (required for SOAP mode)
 * 5. axis.jar (required for SOAP mode)
 * 6. commons-codec-1.3.jar (required for SOAP mode)
 * 7.  commons-collections-3.1.jar  (required for SOAP mode)
 * 8. commons-discovery.jar (required for SOAP mode)
 * 9. commons-logging.jar (required for SOAP mode)
 * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
 * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
 * 12. jaxrpc.jar (required for SOAP mode)
 * 13. log4j.jar (required for SOAP mode)
 * 14. mail.jar (required for SOAP mode)
 * 15. saaj.jar (required for SOAP mode)
 * 16. wsdl4j.jar (required for SOAP mode)
 * 17. xalan.jar (required for SOAP mode)
 * 18. xbean.jar (required for SOAP mode)
 * 19. xercesImpl.jar (required for SOAP mode)
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * SOAP required JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * If you want to invoke a remote forms server instance and there is a
 * firewall between the client application and forms server, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include these additional JAR files
 *
 * For information about the SOAP
 * mode, see "Setting connection properties" in Programming
 * with forms server
 */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
```

```
import com.adobe.livecycle.rightsmanagement.client.*;
import com.adobe.livecycle.rightsmanagement.RMSecureDocumentResult;

import com.adobe.idp.um.api.infomodel.PrincipalSearchFilter;
import com.adobe.idp.um.api.infomodel.PrincipalReference;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;
import com.adobe.edc.common.dto.PublishLicenseDTO;
import com.adobe.idp.um.api.infomodel.User;
import com.adobe.idp.um.api.impl.UMBaseLibrary;

public class protectStatementWorkFlowSoap {

    public static void main(String args[]) {

    try{

       //Set connection properties required to invoke forms server using SOAP mode
            Properties connectionProps = new Properties();
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "Jboss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

             //Create a ServiceClientFactory instance
           ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);

             //Create a RightsManagementClient object
             RightsManagementClient rightsClient = new RightsManagementClient(factory);

             DirectoryManagerServiceClient directoryManagerServiceClient = new
DirectoryManagerServiceClient(factory);

             DocumentManager documentManager = rightsClient.getDocumentManager();

          AbstractPolicyManager abstractPolicyManager = rightsClient.getAbstractPolicyManager();

             //Reference a PDF document to which a policy is applied
             FileInputStream is = new FileInputStream("C:\\Adobe\\Sample.pdf");
             Document inPDF = new Document(is);

             //Create user
             String userName = "wblue";
             User user = UMBaseLibrary.createUser(userName, "DefaultDom", userName);
             user.setCommonName(userName);
             user.setGivenName(userName);
             user.setFamilyName("User");
             directoryManagerServiceClient.createLocalUser(user, "password");

             //Ensure that the user was added
             //Create a PrincipalSearchFilter to find the user by ID
             List userList = new ArrayList<PrincipalReference>();
```

```
            PrincipalSearchFilter psf = new PrincipalSearchFilter();
            psf.setUserIdAbsolute(user.getUserid());
            psf.setRetrieveOnlyActive();
            List p = directoryManagerServiceClient.findPrincipalReferences(psf);
            PrincipalReference principal = (PrincipalReference)(p.get(0));
            userList.add(principal);

            //Create Policy From AbstractPolicy "test2"
            String newPolicyId = abstractPolicyManager.createPolicyFromAbstractPolicy("Global
Policy Set", "PolicyFromAbstractPolicy_AllowCopy","Global Policy Set", "AllowCopy", userList);

            System.out.println("Created policy from abstract policy: " + newPolicyId);

            //Create License for the Policy
            PublishLicenseDTO publishLicense = documentManager.createLicense(newPolicyId,
user.getUserid(), "DefaultDom");

            //get the license id from license object
            String licID = publishLicense.getLicenseId();

            //Associate User with License and Policy
            documentManager.associateUserWithLicenseAndPolicy("DefaultDom", user.getUserid(),
licID, newPolicyId);

            //protect the PDF document using license
            RMSecureDocumentResult rmSecureDocument = documentManager.protectDocument(inPDF,
publishLicense);

            //Retrieve the policy-protected PDF document
            Document protectedDocument = rmSecureDocument.protectedDoc;

            //Save the policy-protected PDF document
            String outputFile = "C:\\Adobe\\PolicyProtected"+ user.getUserid()+".pdf";
            File myFile = new File(outputFile);
            protectedDocument.copyToFile(myFile);

            System.out.println("Protected the PDF With policy");

        }catch(Exception ex){
            ex.printStackTrace();
        }

}

}
```

Signature Service Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the Signature service:

"Quick Start (SOAP mode): Adding a signature field to a PDF document using the Java API" on page 373

"Quick Start (SOAP mode): Retrieving signature field names using the Java API" on page 375

"Quick Start (SOAP mode): Modifying a signature field using the Java API" on page 377

"Quick Start (SOAP mode): Digitally signing a PDF document using the Java API" on page 380

"Quick Start (SOAP mode): Digitally signing a XFA-based Form using the Java API" on page 382

"Quick Start (SOAP mode): Certifying a PDF document using the Java API" on page 386

"Quick Start (SOAP mode): Verifying a digital signature using the Java API" on page 389

"Quick Start (SOAP mode): Verifying multiple digital signatures using the Java API" on page 393

"Quick Start (SOAP mode): Removing a digital signature using the Java API" on page 396

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick Start located in Programming with AEM Forms are based on the Forms server being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Adding a signature field to a PDF document using the Java API

The following Java code example adds a signature field named *SignatureField1* to a PDF document that is based on a PDF file named*Loan.pdf*. The PDF document that contains the new signature field is saved as a PDF file named *LoanSig.pdf*. (See "Adding Signature Fields" on page 880.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7.  commons-collections-3.1.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
```

```
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class AddSignatureFieldSOAP {

    public static void main(String[] args) {

        try
        {
         //Set connection properties required to invoke AEM Forms using SOAP mode
         Properties connectionProps = new Properties();
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
         connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

          //Create a ServiceClientFactory instance
          ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

          //Create a SignatureServiceClient object
          SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

          //Specify a PDF document to which a signature field is added
          FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
          Document inDoc = new Document (fileInputStream);

          //Specify the name of the signature field
          String fieldName = "SignatureField1";
```

```
            //Create a  PositionRectangle object that specifies
            //the signature fields location
            PositionRectangle post = new  PositionRectangle(193,47,133,12);

            //Specify the page number that will contain the signature field
            java.lang.Integer pageNum = new java.lang.Integer(1);

            //Add a signature field to the PDF document
            Document sigFieldPDF = signClient.addSignatureField(
              inDoc,
              fieldName,
              pageNum,
              post,
              null,
              null);

            //Save the PDF document that contains the signature field
            File outFile = new File("C:\\Adobe\LoanSig.pdf");
            sigFieldPDF.copyToFile(outFile);
              }

          catch (Exception ee)
              {
                  ee.printStackTrace();
              }
    }
}
```

Quick Start (SOAP mode): Retrieving signature field names using the Java API

The following Java code example retrieves the names of signature fields located in a PDF document named *LoanSig.pdf*. (See "Retrieving Signature Field Names" on page 884.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
```

```
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class GetSignatureFieldsSOAP {

public static void main(String[] args) {
    try
    {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a SignatureServiceClient object
        SignatureServiceClient signClient = new SignatureServiceClient(myFactory);
```

```
        //Specify a PDF document that contains signature fields
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSig.pdf");
        Document inDoc = new Document (fileInputStream);

        //Retrieve the name of the document's signature fields
        List fieldNames = signClient.getSignatureFieldList(inDoc);

        //Obtain the name of each signature field by iterating through the
        //List object
        Iterator iter = fieldNames.iterator();
        int i = 0 ;
        String fieldName="";
        while (iter.hasNext()) {
            PDFSignatureField signatureField = (PDFSignatureField)iter.next();
            fieldName = signatureField.getName();
            System.out.println("The name of the signature field is " +fieldName);
            i++;
            }
        }
        catch (Exception ee)
        {
            ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Modifying a signature field using the Java API

The following Java code example modifies a signature field named SignatureField1 by locking all fields in the form when a signature is applied to the signature field and ensuring that no changes are allowed. After the Signature service returns the PDF document that contains the modified signature field, the PDF document is saved as a PDF file named LoanSig.pdf. (This example overwrites the PDF file that is passed to the Signature service.) (See "Modifying Signature Fields" on page 887.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
```

```
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ModifySignatureFieldSOAP {

    public static void main(String[] args) {

    try
    {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
```

```
        //Create a SignatureServiceClient object
        SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

        //Specify a PDF document that contains a signature field to modify
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSig.pdf");
        Document inDoc = new Document (fileInputStream);

        //Specify the name of the signature field
        String fieldName = "SignatureField1";

        //Create a PDFSignatureFieldProperties
        PDFSignatureFieldProperties fieldProperties = new PDFSignatureFieldProperties();

         //Create a PDFSeedValueOptionSpec object that stores
         //seed value dictionary information.
         PDFSeedValueOptionSpec seedOptionsSpec = new PDFSeedValueOptionSpec();

         //Disallow changes to the PDF document. Any change to the document invalidates
         //the signature
         seedOptionsSpec.setMdpValue(MDPPermissions.NoChanges);

         //Create a FieldMDPOptionSpec object that stores
         //signature field lock dictionary information.
         FieldMDPOptionSpec fieldMDPOptionsSpec = new FieldMDPOptionSpec();

         //Lock all fields in the PDF document
         fieldMDPOptionsSpec.setAction(FieldMDPAction.ALL);

         //Set dictionary information
         fieldProperties.setSeedValue(seedOptionsSpec);
         fieldProperties.setFieldMDP(fieldMDPOptionsSpec);

         //Modify the signature field
         Document modSignatureField =
signClient.modifySignatureField(inDoc,fieldName,fieldProperties);

         //Save the PDF document that contains modified signature field
         File file = new File("C:\\Adobe\LoanSig.pdf");
         modSignatureField.copyToFile(file);

    }
    catch (Exception ee)
      {
        ee.printStackTrace();
      }
    }
}
```

Quick Start (SOAP mode): Digitally signing a PDF document using the Java API

The following Java code example digitally signs a PDF document that is based on a PDF file named *LoanSig.pdf*. The alias that is specified for the security credential is secure, and revocation checking is performed. Because no CRL or OCSP server information is specified, the server information is obtained from the certificate used to digitally sign the PDF document. The signed document is saved as a PDF file named *LoanSigned.pdf*. (See "Digitally Signing PDF Documents" on page 892.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
```

```
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.livecycle.signatures.pki.client.types.common.HashAlgorithm;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class SignDocumentSOAP {

public static void main(String[] args) {

    try
    {
      //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

      //Create a ServiceClientFactory instance
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

      //Create a SignatureServiceClient object
      SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

      //Specify a PDF document to sign
      FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSig.pdf");
      Document credDoc = new Document (fileInputStream);

      //Specify the name of the signature field
      String fieldName = "SignatureField1";

      //Create a Credential object
      Credential myCred = Credential.getInstance("secure");

      //Specify the reason to sign the document
      String reason = "The document was reviewed";

      //Specify the location of the signer
      String location  = "New York HQ";

      //Specify contact information
      String contactInfo = "Tony Blue";

      //Create a PDFSignatureAppearanceOptions object
      //and show date information
      PDFSignatureAppearanceOptionSpec appear = new  PDFSignatureAppearanceOptionSpec();
      appear.setShowDate(true);
      appear.setShowReason(true);
```

```
      //Set revocation checking to false
      java.lang.Boolean revCheck = new Boolean(true);

      //Create an OCSPOptionSpec object to pass to the sign method
          OCSPOptionSpec ocspSpec = new OCSPOptionSpec();

          //Create a CRLOptionSpec object to pass to the sign method
          CRLOptionSpec crlSpec = new CRLOptionSpec();

          //Create a TSPOptionSpec object to pass to the sign method
          TSPOptionSpec tspSpec = new TSPOptionSpec();

      //Sign the PDF document
      Document signedDoc = signClient.sign(
        credDoc,
        fieldName,
        myCred,
        HashAlgorithm.SHA1,
        reason,
        location,
        contactInfo,
        appear,
        revCheck,
        ocspSpec,
            crlSpec,
            tspSpec);

      //Save the signed PDF document
      File outFile = new File("C:\\Adobe\LoanSigned.pdf");
      signedDoc.copyToFile (outFile);
      }

    catch (Exception ee)
      {
        ee.printStackTrace();
      }
      }
}
```

Quick Start (SOAP mode): Digitally signing a XFA-based Form using the Java API

The following Java code example signs an interactive form that is rendered by the Forms service. The `com.adobe.idp.Document` instance that is returned by the Forms service is passed to the Signature service. The signed interactive form is saved as a PDF file named *LoanXFASigned.pdf*.

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;

import com.adobe.livecycle.formsservice.client.FormsResult;
import com.adobe.livecycle.formsservice.client.FormsServiceClient;
import com.adobe.livecycle.formsservice.client.PDFFormRenderSpec;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.livecycle.signatures.pki.client.types.common.HashAlgorithm;
```

```
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class SignXFAFormsSOAP {

public static void main(String[] args) {

    try
    {

     //Set connection properties required to invoke AEM Forms using SOAP mode
     Properties connectionProps = new Properties();
     connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
     connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
     connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
     connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

     //Create a ServiceClientFactory instance
     ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

     //Get the XFA form to sign
     Document myForm = GetForm(myFactory);

    //Sign the XFA form
    SignXFA(myForm,myFactory);
    }

    catch (Exception ee)
      {
        ee.printStackTrace();
      }
      }


//Creates an interactive PDF form based on a XFA form
private static Document GetForm(ServiceClientFactory myFactory)
{

    try
    {
    //Create a FormsServiceClient object
    FormsServiceClient formsClient = new FormsServiceClient(myFactory);

    //Specify a PDF document to sign
    FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSigXFA.pdf");
    Document xfaForm = new Document (fileInputStream);

    //Retrieve form data
    FileInputStream cData = new FileInputStream("C:\\Adobe\Loan.xml");
    Document oInputData = new Document(cData);
```

```
    //Cache the PDF form
    PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
    pdfFormRenderSpec.setGenerateServerAppearance(true);

    //Invoke the renderPDFForm2 method
    FormsResult formOut = formsClient.renderPDFForm2(
            xfaForm,               //formQuery
            oInputData, //inDataDoc
            pdfFormRenderSpec,      //PDFFormRenderSpec
            null,           //urlSpec
            null        //attachments
            );

    //Create a Document object that stores form data
    Document myForm = formOut.getOutputContent();
    return myForm;
}

    catch (Exception ee)
      {
        ee.printStackTrace();
      }
    return null;

}

//Sign the PDF document
private static void SignXFA(Document doc, ServiceClientFactory myFactory)
{
    try
    {

    //Create a SignatureServiceClient object
     SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

     //Specify the name of the signature field
     String fieldName = "SignatureField1";

     //Create a Credential object
     Credential myCred = Credential.getInstance("secure");

     //Specify the reason to sign the document
     String reason = "The document was reviewed";

     //Specify the location of the signer
     String location  = "New York HQ";

     //Specify contact information
     String contactInfo = "Tony Blue";

     //Create a PDFSignatureAppearanceOptions object
     //and show date information
     PDFSignatureAppearanceOptionSpec appear = new  PDFSignatureAppearanceOptionSpec();
     appear.setShowDate(true);
     appear.setShowReason(true);
```

```
      //Set revocation checking to false
      java.lang.Boolean revCheck = new Boolean(true);

      //Create an OCSPOptionSpec object to pass to the sign method
      OCSPOptionSpec ocspSpec = new OCSPOptionSpec();

      //Create a CRLOptionSpec object to pass to the sign method
      CRLOptionSpec crlSpec = new CRLOptionSpec();

      //Create a TSPOptionSpec object to pass to the sign method
      TSPOptionSpec tspSpec = new TSPOptionSpec();

      //Sign the PDF document
       Document signedDoc = signClient.sign(
         doc,
         fieldName,
         myCred,
         HashAlgorithm.SHA1,
         reason,
         location,
         contactInfo,
         appear,
         revCheck,
         ocspSpec,
         crlSpec,
         tspSpec);

       //Save the signed PDF document
       File outFile = new File("C:\\Adobe\LoanXFASigned.pdf");
       signedDoc.copyToFile (outFile);
}

    catch (Exception ee)
      {
        ee.printStackTrace();
      }
     }
}
```

Quick Start (SOAP mode): Certifying a PDF document using the Java API

The following Java code example certifies a PDF document that is based on a PDF file named *LoanSig.pdf*. The alias that is specified for the security credential is secure, and revocation checking is not performed. The certified document is saved as a PDF file named *LoanCertified.pdf*. (See "Certifying PDF Documents" on page 904.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.livecycle.signatures.pki.client.types.common.HashAlgorithm;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
```

```
public class CertifyDocumentSOAP {

    public static void main(String[] args) {
    try
    {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

      //Create a ServiceClientFactory instance
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

      //Create a SignatureServiceClient object
      SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

      //Specify a PDF document to certify
      FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSig.pdf");
      Document credDoc = new Document (fileInputStream);

      //Specify the name of the signature field
      String fieldName = "SignatureField1";

      //Create a Credential object
      Credential myCred = Credential.getInstance("secure");

      //Specify the reason to sign the document
      String reason = "The document was reviewed";

      //Specify the location of the signer
      String location  = "My company";

      //Specify contact information
          String contactInfo = "New York, New York";

      //Create a PDFSignatureAppearanceOptions object and show date information
      PDFSignatureAppearanceOptionSpec appear = new  PDFSignatureAppearanceOptionSpec();
      appear.setShowDate(true);

      //Set revocation checking to false
          java.lang.Boolean revCheck = new Boolean(false);

          //Set locking to false
          java.lang.Boolean lockField = new Boolean(false);

          //Specify a legalAttestation value
          String msg = "Any change to this document will invalidate the certificate";

          //Create objects to pass to the certify method
```

```
            OCSPOptionSpec ocspSpec = new OCSPOptionSpec();
            CRLOptionSpec crlSpec = new CRLOptionSpec();
            TSPOptionSpec tspSpec = new TSPOptionSpec();

            //Certify the PDF document
            Document signedDoc = signClient.certify(
              credDoc,
              fieldName,
              myCred,
              HashAlgorithm.SHA1,
              reason,
              location,
              contactInfo,
              MDPPermissions.NoChanges,
              msg,
              appear,
              revCheck,
              lockField,
              ocspSpec,
              crlSpec,
              tspSpec);

        //Save the signed PDF document
            File outFile = new File("C:\\Adobe\LoanCertified.pdf");
            signedDoc.copyToFile (outFile);
        }

    catch (Exception ee)
        {
        ee.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Verifying a digital signature using the Java API

The following Java code example verifies a digital signature that is located in a signed PDF document that is based on a PDF file named LoanSigned.pdf. The verification time is set to current time and the revocation checking option is set to best effort. (See "Verifying Digital Signatures" on page 911.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7.  commons-collections-3.1.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.livecycle.signatures.pki.client.types.common.RevocationCheckStyle;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class VerifySignatureSOAP{

    public static void main(String[] args) {

    try
```

```
    {
      //Set connection properties required to invoke AEM Forms using SOAP mode
      Properties connectionProps = new Properties();
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

      //Create a ServiceClientFactory instance
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

      //Create a SignatureServiceClient object
      SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

      //Specify a PDF document that contains a digital signature
      FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSigned.pdf");
      Document inDoc = new Document (fileInputStream);

      //Specify the name of the signature field
      String fieldName = "SignatureField1";

      //Create a PKIOptions object that contains PKI run-time options
      PKIOptions pkiOptions = new PKIOptions();
      pkiOptions.setVerificationTime(VerificationTime.CURRENT_TIME);
      pkiOptions.setRevocationCheckStyle(RevocationCheckStyle.BestEffort);

      //Verify the digital signature
      PDFSignatureVerificationInfo  signInfo = signClient.verify2(
        inDoc,
        fieldName,
        pkiOptions,
        null);

      //Get the Signature Status
      SignatureStatus sigStatus = signInfo.getStatus();
      String myStatus="";

      //Determine the status of the signature
      if (sigStatus == SignatureStatus.DynamicFormSignatureUnknown)
             myStatus = "The signatures located in the dynamic PDF form are unknown";
         else if (sigStatus == SignatureStatus.DocumentSignatureUnknown)
             myStatus = "The signatures located in the PDF document are unknown";
         else if (sigStatus == SignatureStatus.CertifiedDynamicFormSignatureTamper)
             myStatus = "The signatures located in a certified PDF form are valid";
         else if (sigStatus == SignatureStatus.SignedDynamicFormSignatureTamper)
             myStatus = "The signatures located in a signed dynamic PDF form are valid";
         else if (sigStatus == SignatureStatus.CertifiedDocumentSignatureTamper)
             myStatus = "The signatures located in a certified PDF document are valid";
         else if (sigStatus == SignatureStatus.SignedDocumentSignatureTamper)
             myStatus = "The signatures located in a signed PDF document are valid";
         else if (sigStatus == SignatureStatus.SignatureFormatError)
```

```
            myStatus = "The format of a signature in a signed document is invalid";
        else if (sigStatus == SignatureStatus.DynamicFormSigNoChanges)
            myStatus = "No changes were made to the signed dynamic PDF form";
        else if (sigStatus == SignatureStatus.DocumentSigNoChanges)
            myStatus = "No changes were made to the signed PDF document";
        else if (sigStatus == SignatureStatus.DynamicFormCertificationSigNoChanges)
            myStatus = "No changes were made to the certified dynamic PDF form";
        else if (sigStatus == SignatureStatus.DocumentCertificationSigNoChanges)
            myStatus = "No changes were made to the certified PDF document";
        else if (sigStatus == SignatureStatus.DocSigWithChanges)
            myStatus = "There were changes to a signed PDF document";
       else if (sigStatus == SignatureStatus.CertificationSigWithChanges)
            myStatus = "There were changes made to the PDF document.";

     //Get the signature type
     SignatureType sigType = signInfo.getSignatureType();
     String myType = "";

     if (sigType.getType() == PDFSignatureType.AUTHORSIG)
            myType="Certification";
     else if(sigType.getType() == PDFSignatureType.RECIPIENTSIG)
            myType="Recipient";

     //Get the identity of the signer
     IdentityInformation signerId = signInfo.getSigner();
     String signerMsg = "";

    if (signerId.getStatus() == IdentityStatus.UNKNOWN)
        signerMsg = "Identity Unknown";
    else if (signerId.getStatus() == IdentityStatus.TRUSTED)
        signerMsg = "Identity Trusted";
    else if (signerId.getStatus() == IdentityStatus.NOTTRUSTED)
        signerMsg = "Identity Not Trusted";

    //Get the Signature properties returned by the Signature service
    SignatureProperties sigProps = signInfo.getSignatureProps();
    String signerName =  sigProps.getSignerName();

   System.out.println("The status of the signature is: "+myStatus +". The signer identity is
"+signerMsg +". The signature type is "+myType +". The name of the signer is "+signerName+".");
    }
    catch (Exception ee)
    {
        ee.printStackTrace();
    }
     }
}
```

Quick Start (SOAP mode): Verifying multiple digital signatures using the Java API

The following Java code example verifies multiple digital signatures that are located in a signed PDF document that is based on a PDF file named LoanAllSigs.pdf. The verification time is set to current time and the revocation checking option is set to best effort. (See "Verifying Multiple Digital Signatures" on page 916.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    * These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;
import java.io.FileInputStream;
```

```
import com.adobe.livecycle.signatures.client.*;
import com.adobe.livecycle.signatures.client.types.*;
import com.adobe.livecycle.signatures.pki.client.types.common.RevocationCheckStyle;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class VerifyAllSignaturesSOAP{

    public static void main(String[] args) {

    try
    {
      //Set connection properties required to invoke AEM Forms using SOAP mode
      Properties connectionProps = new Properties();
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
      connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

      //Create a ServiceClientFactory instance
      ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

      //Create a SignatureServiceClient object
      SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

      //Specify a PDF document that contains multiple digital signatures
      FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanAllSigs.pdf");
      Document inDoc = new Document (fileInputStream);

      //Create a PKIOptions object that contains PKI run-time options
      PKIOptions pkiOptions = new PKIOptions();
      pkiOptions.setVerificationTime(VerificationTime.CURRENT_TIME);
      pkiOptions.setRevocationCheckStyle(RevocationCheckStyle.BestEffort);

      //Verify all digital signatures that are located in a PDF document
      PDFDocumentVerificationInfo  allSig = signClient.verifyPDFDocument(
        inDoc,
        pkiOptions,
        null);

      //Get a list of all signatures that are located in the PDF document
      List allSignatures = allSig.getVerificationInfos();

    //Create an Iterator object and iterate through
    //the List object
    Iterator<PDFSignatureVerificationInfo> iter = allSignatures.iterator();

    while (iter.hasNext()) {
          PDFSignatureVerificationInfo signInfo = (PDFSignatureVerificationInfo)iter.next();
```

```
            //Get the Signature Status
               SignatureStatus sigStatus = signInfo.getStatus();
               String myStatus="";

             //Determine the status of the signature
               if (sigStatus == SignatureStatus.DynamicFormSignatureUnknown)
                   myStatus = "The signatures located in the dynamic PDF form are unknown";
               else if (sigStatus == SignatureStatus.DocumentSignatureUnknown)
                   myStatus = "The signatures located in the PDF document are unknown";
               else if (sigStatus == SignatureStatus.CertifiedDynamicFormSignatureTamper)
                   myStatus = "The signatures located in a certified PDF form are valid";
               else if (sigStatus == SignatureStatus.SignedDynamicFormSignatureTamper)
                  myStatus = "The signatures located in a signed dynamic PDF form are valid";
               else if (sigStatus == SignatureStatus.CertifiedDocumentSignatureTamper)
                   myStatus = "The signatures located in a certified PDF document are valid";
               else if (sigStatus == SignatureStatus.SignedDocumentSignatureTamper)
                   myStatus = "The signatures located in a signed PDF document are valid";
               else if (sigStatus == SignatureStatus.SignatureFormatError)
                   myStatus = "The format of a signature in a signed document is invalid";
               else if (sigStatus == SignatureStatus.DynamicFormSigNoChanges)
                   myStatus = "No changes were made to the signed dynamic PDF form";
               else if (sigStatus == SignatureStatus.DocumentSigNoChanges)
                   myStatus = "No changes were made to the signed PDF document";
               else if (sigStatus == SignatureStatus.DynamicFormCertificationSigNoChanges)
                   myStatus = "No changes were made to the certified dynamic PDF form";
               else if (sigStatus == SignatureStatus.DocumentCertificationSigNoChanges)
                   myStatus = "No changes were made to the certified PDF document";
               else if (sigStatus == SignatureStatus.DocSigWithChanges)
                   myStatus = "There were changes to a signed PDF document";
              else if (sigStatus == SignatureStatus.CertificationSigWithChanges)
                   myStatus = "There were changes made to the PDF document.";

               //Get the signature type
              SignatureType sigType = signInfo.getSignatureType();
              String myType = "";

              if (sigType.getType() == PDFSignatureType.AUTHORSIG)
                  myType="Certification";
              else if(sigType.getType() == PDFSignatureType.RECIPIENTSIG)
                  myType="Recipient";

              //Get the Signature properties returned by the Signature service
              SignatureProperties sigProps = signInfo.getSignatureProps();
              String signerName =  sigProps.getSignerName();

            System.out.println("The status of the signature is: "+myStatus +". The signature
type is "+myType +". The name of the signer is "+signerName+".");
        }
    }
    catch (Exception ee)
    {
        ee.printStackTrace();
    }
     }
}
```

Quick Start (SOAP mode): Removing a digital signature using the Java API

The following Java code example removes a digital signature from a signature field named *SignatureField1*. The name of the PDF file that contain the signature field is *LoanSigned.pdf*. (See "Removing Digital Signatures" on page 921.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-signatures-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. adobe-utilities.jar
    * 5. jboss-client.jar (use a different JAR file if the forms server is not deployed
    * on JBoss)
    * 6. activation.jar (required for SOAP mode)
    * 7. axis.jar (required for SOAP mode)
    * 8. commons-codec-1.3.jar (required for SOAP mode)
    * 9.  commons-collections-3.1.jar  (required for SOAP mode)
    * 10. commons-discovery.jar (required for SOAP mode)
    * 11. commons-logging.jar (required for SOAP mode)
    * 12. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 13. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 14. jaxrpc.jar (required for SOAP mode)
    * 15. log4j.jar (required for SOAP mode)
    * 16. mail.jar (required for SOAP mode)
    * 17. saaj.jar (required for SOAP mode)
    * 18. wsdl4j.jar (required for SOAP mode)
    * 19. xalan.jar (required for SOAP mode)
    * 20. xbean.jar (required for SOAP mode)
    * 21. xercesImpl.jar (required for SOAP mode)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * The adobe-utilities.jar file is located in the following path:
    * <install directory>/sdk/client-libs/jboss
    *
    * The jboss-client.jar file is located in the following path:
    * <install directory>/jboss/bin/client
    *
    * SOAP required JAR files are located in the following path:
    * <install directory>/sdk/client-libs/thirdparty
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include these additional JAR files
    *
    * For information about the SOAP
    * mode, see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;
import java.io.File;
import java.io.FileInputStream;
import com.adobe.livecycle.signatures.client.*;
```

```
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ClearSignatureFieldSOAP {

    public static void main(String[] args) {

    try
    {
        //Set connection properties required to invoke AEM Forms using SOAP mode
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory instance
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a SignatureServiceClient object
        SignatureServiceClient signClient = new SignatureServiceClient(myFactory);

        //Specify a PDF document that contains the signature to remove
        FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\LoanSigned.pdf");
        Document inDoc = new Document (fileInputStream);

        //Specify the name of the signature field
        String fieldName = "SignatureField1";

        //Clear the signature field
        Document outPDF = signClient.clearSignatureField(inDoc,fieldName);

        //Save the PDF document
        File outFile = new File("C:\\Adobe\Loan.pdf");
        outPDF.copyToFile(outFile);
        }

    catch (Exception ee)
        {
        ee.printStackTrace();
        }
    }
}
```

Task Manager Service Java API Quick Start(SOAP)

The following Quick Starts are available for the Task Manager service.

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *You cannot search for tasks assigned to users by using the web service API. The reason is because you cannot invoke the* `taskList` *method, which is a necessary method call to perform this task.*

Note: *Quick Start located in Programming with AEM Forms are based on the Forms server operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Assigning tasks using the Java API

The following Java code example assigns a task to a user named Tony Blue.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
```

```
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.taskmanager.dsc.client.task.TaskManager;
import com.adobe.idp.taskmanager.dsc.client.*;
import com.adobe.idp.um.api.infomodel.PrincipalSearchFilter;
import com.adobe.idp.um.api.infomodel.User;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;

public class AssignTask {

    public static void main(String[] args) {

        try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"tblue");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a TaskManager object
```

```
        TaskManager myTaskManager = TaskManagerClientFactory.getTaskManager(myFactory);

        //Get the user identifer by calling
        //a user-defined method
        String userID = getUserId(myFactory);

        //Forward task to another user
        myTaskManager.forwardTask(343,userID);
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }


    //This method returns the identifier value of tony blue
     static private String getUserId(ServiceClientFactory myFactory){
      String oid = "";
      try{

          //Create a DirectoryManagerServiceClient object
          DirectoryManagerServiceClient dirClient = new
DirectoryManagerServiceClient(myFactory);

         //Find a local user
         PrincipalSearchFilter psf = new PrincipalSearchFilter();
         psf.setUserId("tblue");
         List principalList = dirClient.findPrincipals(psf);
         Iterator pit = principalList.iterator();

         User testUser = null;
         if (pit.hasNext())
         {
            //Obtain the principals object identifier
            testUser = (User)(pit.next());
         }
         oid = testUser.getOid();
        }

          catch(Exception e)
          {
              e.printStackTrace();
          }
              return oid;
      }
}
```

Quick Start (SOAP mode): Locking tasks using the Java API

The following Java code example locks a task that corresponds to the task identifier value of 2.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.taskmanager.dsc.client.task.TaskManager;
import com.adobe.idp.taskmanager.dsc.client.*;

public class LockTask {

    public static void main(String[] args) {
```

```
    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"tblue");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a TaskManager object
        TaskManager myTaskManager = TaskManagerClientFactory.getTaskManager(myFactory);

        //Lock the task that corresponds to task identifier 2
        myTaskManager.lockTask(2);
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieving tasks assigned to users using the Java API

The following Java code example retrieves all tasks that are assigned to a user named *tony blue*. Notice that this user is specified in the connection properties. Information about returned tasks, such as its identifier value and description, is displayed.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.taskmanager.dsc.client.query.StatusFilter;
import com.adobe.idp.taskmanager.dsc.client.query.TaskFilter;
import com.adobe.idp.taskmanager.dsc.client.query.TaskRow;
import com.adobe.idp.taskmanager.dsc.client.*;

public class RetrieveTaskInfo {

    public static void main(String[] args) {
```

```
        try{
             //Set connection properties required to invoke AEM Forms
               Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
               connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

             //Create a ServiceClientFactory object
             ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

             //Create a TaskManagerQueryService object
             TaskManagerQueryService queryManager =
TaskManagerClientFactory.getQueryManager(myFactory);

             //Define search criteria by performing a search on
             //Assigned tasks (tasks assigned to the user specified
             //in connection properties)
             TaskFilter filter = queryManager.newTaskFilter();
             StatusFilter sf = filter.newStatusFilter();
             sf.addStatus(StatusFilter.assigned);
             filter.setStatusFiltering(sf);

             //Perform the search
             List result = queryManager.taskList(filter);

             //Create an Iterator object and iterate through
             //the List object
             Iterator iter = result.iterator();
             int i = 0 ;

             while (iter.hasNext()) {

                 TaskRow myTask = (TaskRow)iter.next();

                 //Get the task identifier value
                 long taskId = myTask.getTaskId();

                 //Get the status of the task
                 long taskStatus = myTask.getTaskStatus();
```

```
                //Get the name of process on which this task is based
                String processName = myTask.getProcessName();

                //Get the task description
                String taskDes = myTask.getDescription();

                System.out.println("The task identifier is "+taskId +"\n"+
                "The status of the task is "+taskStatus +"\n"+
                "The name of the process on which the task is based is "+processName +"\n"+
                "The task description is "+taskDes);
                i++ ;
                 }
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieving form data from tasks using the Java API

The following Java code example retrieves form data from a task with the identifier value of 304. Form data is written to an XML file named *FormData.xml* located at C:\Adobe.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
```

```
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.io.File;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

import com.adobe.idp.taskmanager.dsc.client.*;
import com.adobe.idp.taskmanager.dsc.client.task.FormInstance;
import com.adobe.idp.taskmanager.dsc.client.task.TaskInfo;
import com.adobe.idp.taskmanager.dsc.client.task.TaskManager;

public class RetrieveFormData {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
              Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
              connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a TaskManager object
            TaskManager myTaskManager = TaskManagerClientFactory.getTaskManager(myFactory);

            //Retrieve information about task 304
            long taskId = 304;
            TaskInfo tInfo = myTaskManager.getTaskInfo(taskId);
```

```
            //Retrieve the form instance associated with task 304
            FormInstance[] fi = tInfo.getTaskItems();
            long formInstanceId = fi[0].getFormInstanceId();
            FormInstance newfi = myTaskManager.getFormInstanceForTask(taskId, formInstanceId,
true);

            //Get data located in the form and
            //write the data to FormData.xml
            Document doc = newfi.getDocument();
            File myTestFile = new File("C:\\Adobe\FormData.xml");
            doc.copyToFile(myTestFile);
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Modifying form data using the Java API

The following Java code example updates a form with data that is located in the *FormData.xml* file.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
```

```
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

import com.adobe.idp.taskmanager.dsc.client.*;
import com.adobe.idp.taskmanager.dsc.client.task.FormInstance;
import com.adobe.idp.taskmanager.dsc.client.task.SaveTaskResult;
import com.adobe.idp.taskmanager.dsc.client.task.TaskManager;

public class SetFormData {

    public static void main(String[] args) {
    try{
        //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"tblue");
        connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

        //Create a ServiceClientFactory object
        ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);

        //Create a TaskManager object
        TaskManager myTaskManager = TaskManagerClientFactory.getTaskManager(myFactory);

        //Specify form data that is used to update the form
        FileInputStream myData = new FileInputStream("C:\\Adobe\FormData.xml");
        Document doc = new Document(myData);
        InputStream in = doc.getInputStream();
        byte[] formarray = new byte[in.available()];
        in.read(formarray);
```

```
        //Get an empty form instance
        FormInstance newForm = myTaskManager.getEmptyForm();
        newForm.setTemplatePath("C:\\Adobe\Mortgage.xdp");
        newForm.setXFAData(formarray);
        newForm.setDocument(doc);

        //Save the modified form
        SaveTaskResult result = myTaskManager.save(4, newForm);
        System.out.println("ActionFromData= "+result.getActionFromData());
        System.out.println("task id= "+result.getTaskId());
        }

    catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieving file attachments from tasks using the Java API

The following Java code example retrieves file attachments. Each file attachment is saved as a TXT file.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
```

```
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */

import java.io.File;
import java.util.*;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.taskmanager.dsc.client.*;
import com.adobe.idp.taskmanager.dsc.client.task.TaskManager;

public class RetrieveFileAttachments
    {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
              Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
              connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a TaskManager object
            TaskManager myTaskManager = TaskManagerClientFactory.getTaskManager(myFactory);

            //Retrieve file attachments associated with the task
            List fileAttachments = myTaskManager.getAttachmentListForTask(322);
```

```
            //Create an Iterator object and iterate through
            //the List object
            Iterator iter = fileAttachments.iterator();
            int i = 0 ;
            while (iter.hasNext()) {
                 Document fileAttachment= (Document)iter.next();
                 File myFile = new File("C:\\FileAtt" +i+".txt");
                 fileAttachment.copyToFile(myFile);
                 i++ ;
              }
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Retrieving task information using the Java API

The following Java code example retrieves all tasks that are based on a process named *MortgageLoan - Prebuilt*. The status of each returned task is checked to ensure that it is a completed task. Information such as the name of the user who completed the task and the date that the task was completed is retrieved and displayed.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-taskmanager-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    * 20. adobe-workflow-client-sdk.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.taskmanager.dsc.client.query.TaskRow;
import com.adobe.idp.taskmanager.dsc.client.query.TaskSearchFilter;
import com.adobe.idp.taskmanager.dsc.client.task.ParticipantInfo;
import com.adobe.idp.taskmanager.dsc.client.task.TaskInfo;
import com.adobe.idp.taskmanager.dsc.client.task.TaskManager;
import com.adobe.idp.taskmanager.dsc.client.*;
import com.adobe.idp.um.api.infomodel.Principal;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;

public class RetrievingTasks {

    public static void main(String[] args) {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClie
ntFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
          connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a TaskManagerQueryService object
            TaskManagerQueryService queryManager =
TaskManagerClientFactory.getQueryManager(myFactory);
```

```
            //Create a TaskManager object
            TaskManager taskManager = TaskManagerClientFactory.getTaskManager(myFactory);

            //Define search criteria by performing a search on
            //completed tasks
            TaskSearchFilter filter = new TaskSearchFilter();
            filter.setServiceName("MortgageLoan - Prebuilt");
            filter.setAdminIgnoreAllAcls(true);

            //Perform the search on tasks
            List result = queryManager.taskSearch(filter);

            //Create an Iterator object and iterate through
            //the List object
            Iterator iter = result.iterator();
            int i = 0 ;
            while (iter.hasNext()) {
                TaskRow myTask = (TaskRow)iter.next();

                //Make sure that the task is completed- 100 represents
                //a completed task
                if (myTask.getTaskStatus()== 100)
                {
                    //Get the name of the user who completed the task
                    long taskId = myTask.getTaskId();
                    TaskInfo taskInfo= taskManager.getTaskInfo(taskId);
                    ParticipantInfo user = taskInfo.getAssignedTo();
                    String userId = user.getSpecifiedUserId();
                    String userName = getUserName(myFactory, userId);

                    //Get the name of the process
                    String processName = myTask.getProcessName();

                    //Get the completion time
                    Date completionTime = myTask.getCompleteTime();

                    //Display task information
                    System.out.println("The task identifier is "+taskId +"\n"+
                    "The name of the user who completed the task is "+ userName +"\n"+
                    "The name of the process on which the task is based is "+ processName+"\n"+
                    "The completion time is "+ completionTime.getDate());
                    i++ ;
                    }
                 }
            }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
```

```
    //This method accepts a user Id and returns the corresponding user name
    static private String getUserName(ServiceClientFactory myFactory, String userId){
         String userName = "";
         try{
             //Create a DirectoryManagerServiceClient object
             DirectoryManagerServiceClient dirClient = new
DirectoryManagerServiceClient(myFactory);

            //Find a local user
            Principal prin = dirClient.findPrincipal(userId);
            userName = prin.getCanonicalName();
           }
         catch(Exception e)
         {
             e.printStackTrace();
         }
         return userName;
      }
    }
```

XMP Utilities Service Java API Quick Start(SOAP)

The following Quick Starts are available for the XMP Utilities service.

"Quick Start (SOAP mode): Exporting XMP metadata using the Java API" on page 414

"Quick Start (SOAP mode): Importing XMP metadata using the Java API" on page 416

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick starts located in Programming with AEM forms are based on the Forms server if you are using another operating system, such as UNIX, replace windows-specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Exporting XMP metadata using the Java API

The following code example retrieves, inspects, and saves XMP metadata. (See "Exporting Metadata from PDF Documents" on page 1011.)

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-pdfutility-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.xmputility.*;
import com.adobe.livecycle.xmputility.client.*;
import java.util.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ExportMetadata
{
```

```
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a XMP Utility client
            XMPUtilityServiceClient xmpUt = new XMPUtilityServiceClient(factory);

            // Specify a PDF document whose metadata is to be exported
            FileInputStream fileInputStream = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document(fileInputStream);

            // Export the XMP metadata object
            XMPUtilityMetadata myXmp = xmpUt.exportMetadata(inDoc);

            // Inspect the XMP metadata object (retrieve the document?s author in this case)
            String name = myXmp.getAuthor();
            System.out.println("The document?s author is " + name);

            // Export the XMP metadata to an XML file
            Document outDoc = xmpUt.exportXMP(inDoc);
            File xmpFile = new File("c:\\LoanMetaData.xml");
            outDoc.copyToFile(xmpFile);
        }
        catch (Exception e)
        {
            System.out.println("Error occurred: " + e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Importing XMP metadata using the Java API

The following code example imports XMP metadata and saves the new PDF file to disk. The PDF document is based on a PDF file named Loan.pdf. The XML document that contains the metadata to import into the PDF document is based on an XML file named *LoanMetaData.xml*. For information about this XML file, see "Importing Metadata into PDF Documents" on page 1007.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-pdfutility-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    * 4. activation.jar (required for SOAP mode)
    * 5. axis.jar (required for SOAP mode)
    * 6. commons-codec-1.3.jar (required for SOAP mode)
    * 7. commons-collections-3.2.jar  (required for SOAP mode)
    * 8. commons-discovery.jar (required for SOAP mode)
    * 9. commons-logging.jar (required for SOAP mode)
    * 10. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 11. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 12. jaxrpc.jar (required for SOAP mode)
    * 13. log4j.jar (required for SOAP mode)
    * 14. mail.jar (required for SOAP mode)
    * 15. saaj.jar (required for SOAP mode)
    * 16. wsdl4j.jar (required for SOAP mode)
    * 17. xalan.jar (required for SOAP mode)
    * 18. xbean.jar (required for SOAP mode)
    * 19. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import com.adobe.livecycle.xmputility.*;
import com.adobe.livecycle.xmputility.client.*;
import java.util.*;
import java.io.*;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ImportMetadata
{
```

```
    public static void main(String[] args)
    {
        try
        {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory instance
            ServiceClientFactory factory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a XMP Utility client
            XMPUtilityServiceClient xmpUt = new XMPUtilityServiceClient(factory);

            //Specify a PDF document into which XMP metadata is imported
            FileInputStream filePDF = new FileInputStream("C:\\Adobe\Loan.pdf");
            Document inDoc = new Document(filePDF);

            //Specify an XML file containing XMP metadata to import
            FileInputStream fileXML = new FileInputStream("C:\\Adobe\LoanMetaData.xml");
            Document xmpDoc = new Document(fileXML );

            //Import the XMP metadata
            Document outDoc = xmpUt.importXMP(inDoc, xmpDoc);

            //Inspect the XMP metadata object (retrieve the document?s author in this case)
            XMPUtilityMetadata myXmp = xmpUt.exportMetadata(outDoc);
            String name = myXmp.getAuthor();
            System.out.println("The document?s author is " + name);

            //Save the PDF document containing the new metadata
            File pdfFile = new File("c:\\Adobe\LoanWithMetadata.pdf");
            outDoc.copyToFile(pdfFile);
        }
        catch (Exception e)
        {
            System.out.println("Error occurred: " + e.getMessage());
        }
    }
}
```

User Manager Java API Quick Start(SOAP)

Java API Quick Start(SOAP) is available for the User Manager API.

AEM Forms operations can be performed using the AEM Forms strongly-typed API and the connection mode should be set to SOAP.

Note: *Quick start located in Programming with AEM forms are based on the Document if you are using another operating system, such as Unix, replace Windows-specific paths with paths supported by the applicable operating system. Likewise, if you are using another J2EE application server, then ensure that you specify valid connection properties. (See "Setting connection properties" on page 500.)*

Quick Start (SOAP mode): Adding users using the Java API

The following code example adds a user named Wendy Blue to AEM Forms. (See "Adding Users" on page 1015.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
```

```
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;
import com.adobe.idp.um.api.infomodel.impl.*;
import com.adobe.idp.um.api.infomodel.*;

public class AddUser {

    public static void main(String[] args) {
        try {
            //Set connection properties required to invoke AEM Forms
               Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
               connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create an DirectoryManagerServiceClient object
            DirectoryManagerServiceClient dmClient = new
DirectoryManagerServiceClient(myFactory);

            //Create a User object and populate its attributes
```

```
            UserImpl u = new UserImpl();
            u.setDomainName("DefaultDom");
            u.setUserid("wblue");
            u.setCanonicalName("wblue");
            u.setPrincipalType("USER");
            u.setGivenName("Wendy");
            u.setFamilyName("Blue");
            u.setLocale(Locale.CANADA);
            u.setTimezone(TimeZone.getDefault());
            u.setDisabled(false);

            //Add the User to the system using the DirectoryManagerServiceClient
            dmClient.createLocalUser(u,"password");

            //Ensure that the user was added
            //Create a PrincipalSearchFilter to find the user by ID
            PrincipalSearchFilter psf = new PrincipalSearchFilter();
            psf.setUserId("wblue");
            List<User> principalList = dmClient.findPrincipals(psf);
            Iterator<User> pit = principalList.iterator();
            if(pit.hasNext()){
                User theUser = pit.next();
                System.out.println("User ID: " + theUser.getUserid());
                System.out.println("User name: " + theUser.getGivenName() +" "+
theUser.getFamilyName());
                System.out.println("User Domain: " + theUser.getDomainName());
                System.out.println("is user disabled?: " + theUser.isDisabled());
            }

        }catch (Exception e) {
            e.printStackTrace();
        }
;
    }

}
```

Quick Start (SOAP mode): Deleting users using the Java API

The following code example deletes a user named Wendy Blue from AEM Forms. (See "Deleting Users" on page 1019.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.um.api.infomodel.PrincipalSearchFilter;
import com.adobe.idp.um.api.infomodel.User;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;

public class DeleteUser {
```

```
    public static void main(String[] args) {
        try {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            // Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create a DirectoryManagerServiceClient object
            DirectoryManagerServiceClient dm = new DirectoryManagerServiceClient(myFactory);

            //Find the target user by the user ID value
            PrincipalSearchFilter psf = new PrincipalSearchFilter();
            psf.setUserId("wblue");
            List<User> principalList = dm.findPrincipals(psf);
            Iterator<User> pit = principalList.iterator();

            //Delete the user
            while(pit.hasNext()){
                User targetUser = pit.next();
                dm.deleteLocalUser(targetUser.getOid());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Managing users and groups using the Java API

The following code example finds a local user and the local group to which the user belongs. (See "Managing Users and Groups" on page 1023.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM FOrms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.*;

import com.adobe.idp.um.api.infomodel.*;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ManageUsersAndGroupsTest
{
    public static void main(String[] args) {
```

```
        try {
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create an DirectoryManagerServiceClient object
            DirectoryManagerServiceClient dirClient = new
DirectoryManagerServiceClient(myFactory);

            // Find a local user
            PrincipalSearchFilter psf = new PrincipalSearchFilter();
            psf.setUserId("wblue");
            List principalList = dirClient.findPrincipals(psf);
            Iterator pit = principalList.iterator();
            String oid = "";
            User testUser = null;
            if (pit.hasNext())
            {
                // Obtain the principal's object identifier
                testUser = (User)(pit.next());
            }

            // Find the local group
            Set groupMemberships = testUser.getGroupMemberships();
            Iterator git = groupMemberships.iterator();
            Group localGroup = null;
            if (git.hasNext())
            {
                // Obtain the group to which the user belongs
                localGroup = (Group)(git.next());
            }

            // Determine the domain and the group to which the local user belongs
            String verifyCanonicalName = testUser.getCanonicalName();
            Domain verifyDomain = dirClient.findDomain(testUser.getDomainName());
```

```
            String verifyDomainName = verifyDomain.getDomainName();
            Group verifyGroup = dirClient.getDomainAsGroup(verifyDomainName);
            String verifyGroupName = verifyGroup.getCanonicalName();

            // Print the uniquely identifying information about the user
            System.out.println("User name: " + verifyCanonicalName);
            System.out.println("Group name: " + verifyGroupName);
            System.out.println("Domain names should match: " + verifyDomainName + ", "+
testUser.getDomainName());

        }
        catch (Exception e)
        {
            System.out.println("Error occurred: " + e.getMessage());
        }
    }
}
```

Quick Start (SOAP mode): Managing roles and permissions using the Java API

The following code example assigns the Services User role to a principal, prints the roles the principal has, and subsequently removes the role from the principal. Two services are invoked for this quick start: the DirectoryManager service and the AuthorizationManager service.(See "Managing Roles and Permissions" on page 1026.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
```

```
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;

import com.adobe.idp.um.api.infomodel.*;
import com.adobe.livecycle.usermanager.client.AuthorizationManagerServiceClient;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class ManageRolesAndPermissionsTest
{
    public static void main(String[] args) {
        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            // Create an AuthorizationManagerServiceClient object
            AuthorizationManagerServiceClient amClient = new
AuthorizationManagerServiceClient(myFactory);

            // Retrieve a principal
            DirectoryManagerServiceClient dirClient = new
DirectoryManagerServiceClient(myFactory);
```

```
PrincipalSearchFilter psf = new PrincipalSearchFilter();
psf.setUserId("wblue");
List principalList = dirClient.findPrincipals(psf);
Iterator pit = principalList.iterator();
String oid = "";
if (pit.hasNext())
{
    // Obtain the principal's object identifier
    oid = ((User)pit.next()).getOid();
    String[] principalOids = new String[1];
    principalOids[0] = oid;

    //Obtain the roles to be assigned
    RoleSearchFilter rsf = new RoleSearchFilter();
    rsf.setRoleName("Services User");
    List roleList = amClient.findRoles(rsf);
    Iterator rit = roleList.iterator();
    String roleId1 = "";
    if (rit.hasNext())
    {
        // Obtain the role identifier
        roleId1 = ((Role)rit.next()).getId();

        // Assign the role to the principal
        amClient.assignRole(roleId1, principalOids);
    }
    else
    {
        System.out.println("Role not found");
    }

    // Determine which roles the principal has
    Set roleSet = amClient.findRolesForPrincipal(oid);

    // Print the roles the principal has
    Iterator it = roleSet.iterator();
    Role r = null;
    System.out.println("Roles:");
    while (it.hasNext())
    {
```

```
                    r = ((Role)it.next());
                    System.out.println(r.getName());
                }

                // Remove a role from the principal
                //amClient.unassignRole(roleId1, principalOids);
            }
            else
            {
                System.out.println("Principal not found");
            }

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quick Start (SOAP mode): Programmatically synchronizing users using the Java API

The following Java code example synchronizes users by using the User Management APIs. (See "Programmatically Synchronizing Users" on page 1031.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    * 19. adobe-usermanager-util-client.jar
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class
path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
```

```
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;
import com.adobe.idp.um.api.DirectoryManager;
import com.adobe.idp.um.api.infomodel.DirectorySyncInfo;
import com.adobe.idp.um.dsc.util.client.UserManagerUtilServiceClient;

public class SynchDomain {

    public static void main(String[] args) {
        try {

            //Set connection properties required to invoke AEM Forms
               Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
               connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a UserManagerUtilServiceClient object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
            UserManagerUtilServiceClient umutil = new  UserManagerUtilServiceClient(myFactory);

            //Specify the set of enterprise domains to synchronize
```

```
            Set<String> domainNames = new HashSet<String>();
            domainNames.add("adobe3");

            //Perform the synchronization operation on the set of enterprise domains specified
above
            umutil.scheduleSynchronization(domainNames);

            //In case the synchronization needs to be performed on all the registered enterprise
domains use this method umutil.scheduleSynchronization();

            DirectoryManager dm = new DirectoryManagerServiceClient(myFactory);
            Map<String, DirectorySyncInfo> synchStatus =
dm.getDirectorySyncStatus(domainNames);

            String domainName = "adobe3";
            DirectorySyncInfo di = synchStatus.get(domainName);
            if(di.getSyncStatus() == DirectorySyncInfo.SYNCSTATUS_COMPLETED){
                        System.out.println("Directory synch for domain "+domainName+" is
complete");

            }

        }catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Quick Start (SOAP mode): Adding users using the Java API

The following code example adds a user named Wendy Blue to AEM Forms. (See "Adding Users" on page 1015.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
```

```
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;
import com.adobe.idp.um.api.infomodel.impl.*;
import com.adobe.idp.um.api.infomodel.*;

public class AddUser {

    public static void main(String[] args) {
        try {
            //Set connection properties required to invoke AEM Forms
               Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
               connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
           connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);
```

```
            //Create an DirectoryManagerServiceClient object
            DirectoryManagerServiceClient dmClient = new
DirectoryManagerServiceClient(myFactory);

            //Create a User object and populate its attributes
            UserImpl u = new UserImpl();
            u.setDomainName("DefaultDom");
            u.setUserid("wblue");
            u.setCanonicalName("wblue");
            u.setPrincipalType("USER");
            u.setGivenName("Wendy");
            u.setFamilyName("Blue");
            u.setLocale(Locale.CANADA);
            u.setTimezone(TimeZone.getDefault());
            u.setDisabled(false);

            //Add the User to the system using the DirectoryManagerServiceClient
            dmClient.createLocalUser(u,"password");

            //Ensure that the user was added
            //Create a PrincipalSearchFilter to find the user by ID
            PrincipalSearchFilter psf = new PrincipalSearchFilter();
            psf.setUserId("wblue");
            List<User> principalList = dmClient.findPrincipals(psf);
            Iterator<User> pit = principalList.iterator();
            if(pit.hasNext()){
                User theUser = pit.next();
                System.out.println("User ID: " + theUser.getUserid());
                System.out.println("User name: " + theUser.getGivenName() +" "+
theUser.getFamilyName());
                System.out.println("User Domain: " + theUser.getDomainName());
                System.out.println("is user disabled?: " + theUser.isDisabled());
            }

        }catch (Exception e) {
            e.printStackTrace();
        }
;
    }

}
```

Quick Start (SOAP mode): Creating Groups using the Java API

The following code example creates a group named AdobeGroup to AEM Forms. (See “Creating Groups” on page 1021.)

```
/*
    * This Java Quick Start uses the SOAP mode and contains the following JAR files
    * in the class path:
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
    *
    *  The JBoss files must be kept in the jboss\client folder. You can copy the client folder to
    *  your local development environment and then include the 3 JBoss JAR files in your class path
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    *
    * <install directory>/jboss/bin/client
    *
    * If you want to invoke a remote forms server instance and there is a
    * firewall between the client application and the server, then it is
    * recommended that you use the SOAP mode. When using the SOAP mode,
    * you have to include additional JAR files located in the following
    * path
    * <install directory>/sdk/client-libs/thirdparty
    *
    * For information about the SOAP
    * mode and the additional JAR files that need to be included,
    * see "Setting connection properties" in Programming
    * with AEM Forms
    *
    * For complete details about the location of the AEM Forms JAR files,
    * see "Including AEM Forms Java library files" in Programming
    * with AEM Forms
    */
import java.util.List;
import java.util.Properties;

import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.um.api.infomodel.Group;
import com.adobe.idp.um.api.infomodel.Principal;
import com.adobe.idp.um.api.infomodel.PrincipalSearchFilter;
import com.adobe.idp.um.api.infomodel.PrincipalReference;
import com.adobe.idp.um.api.infomodel.impl.GroupImpl;
```

```
import com.adobe.livecycle.usermanager.client.DirectoryManagerServiceClient;

public class AddGroup {

    public static void main(String[] args) {
         try {

        //Set connection properties that are required to invoke AEM Forms
        Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClient
FactoryProperties.DSC_SOAP_PROTOCOL);
                connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

                //Create a ServiceClientFactory object
                ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

                //Create an DirectoryManagerServiceClient object
                DirectoryManagerServiceClient dmClient = new
DirectoryManagerServiceClient(myFactory);

                //Specify the group and domain name
                String groupName = "AdobeGroup";
                String domainName = "TestDomain";

                //Check whether the group exists
                String groupOid = checkGroupExist(groupName, domainName,dmClient);

                //The group exists
                if(groupOid != null){
                    System.out.println("The group exists");
                    return;
                }

                //The group does not exist
                String groupCanonicalName = groupName;

                GroupImpl group = new GroupImpl();
                group.setCanonicalName(groupCanonicalName);
                group.setDomainName(domainName);
                group.setGroupType(Group.GROUPTYPE_PRINCIPALS);
                group.setLocal(true);
                group.setPrincipalType(Principal.PRINCIPALTYPE_GROUP);

                groupOid = dmClient.createLocalGroup(group);
                System.out.println("Sample group created with name "+groupName);
```

```
         }catch (Exception e) {
                e.printStackTrace();
            }

        }

    /**
        * Search for a group located in the specified domain
        */
       private static String checkGroupExist(String groupName, String domainName,
DirectoryManagerServiceClient directoryManager){
        try {
        PrincipalSearchFilter psf = new PrincipalSearchFilter();
           psf.setCommonName(groupName);
           psf.setSpecificDomainName(domainName);

           //By default the filter causes like search unless you are using the absolute version
           //Setting this ensures that search is exact
           psf.setMatchExactCriteria(true);

           //By default search returns obsolete groups also. Set this to ensure that
           //only active groups are returned
           psf.setRetrieveOnlyActive();

           //PrincipalReference are lightweight group objects and searching for them is better
performance.
           //If you do not require any other group attribute then use this
           //mode of search
           List<PrincipalReference> result = directoryManager.findPrincipalReferences(psf);
           if(result.isEmpty()){
        System.out.println("Sample group with name "+groupName +" does not exist");
                return null;
           }else{
                String oid = result.get(0).getOid();
                System.out.println("Sample group with name "+groupName +" already exists");
                return oid;
           }
        }catch (Exception e) {
                e.printStackTrace();
           }
        return "";
       }
}
```

Quick Start (SOAP mode) Managing Preferences Nodes

The following Java code models managing of Preferences Nodes by using the User Management APIs. (See "Programmatically managing the Preferences Nodes" on page 1141)

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
    * 3. activation.jar (required for SOAP mode)
    * 4. axis.jar (required for SOAP mode)
    * 5. commons-codec-1.3.jar (required for SOAP mode)
    * 6. commons-collections-3.2.jar  (required for SOAP mode)
    * 7. commons-discovery.jar (required for SOAP mode)
    * 8. commons-logging.jar (required for SOAP mode)
    * 9. dom3-xml-apis-2.5.0.jar (required for SOAP mode)
    * 10. jaxen-1.1-beta-9.jar (required for SOAP mode)
    * 11. jaxrpc.jar (required for SOAP mode)
    * 12. log4j.jar (required for SOAP mode)
    * 13. mail.jar (required for SOAP mode)
    * 14. saaj.jar (required for SOAP mode)
    * 15. wsdl4j.jar (required for SOAP mode)
    * 16. xalan.jar (required for SOAP mode)
    * 17. xbean.jar (required for SOAP mode)
    * 18. xercesImpl.jar (required for SOAP mode)
*
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * If you want to invoke a remote AEM Forms instance and there is a
 * firewall between the client application and AEM Forms, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include additional JAR files located in the following
 * path
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * For information about the SOAP
 * mode and the additional JAR files that need to be included,
 * see "Setting connection properties" in Programming
 * with AEM Forms
 *
 * For complete details about the location of the AEM Forms JAR files,
 * see "Including AEM Forms library files" in Programming
 * with AEM Forms
 */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.um.api.UMException;
import com.adobe.livecycle.usermanager.client.PreferenceManagerServiceClient;


public class ManagePreferences {

    public static void main(String[] args) {
    //Set connection properties required to invoke AEM Forms
```

```
        Properties connectionProps = new Properties();
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

    //Create a PreferenceManagerServiceClient object
    ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);
    PreferenceManagerServiceClient pmutil = new  PreferenceManagerServiceClient(factory);

    //get the preference map for a particular node
    String path = "/Adobe/LiveCycle/Config/UM/CommonNameOrder";
    Map<String, String> map;
    try {
        map = pmutil.getPreferences(path);
        for(String str:map.keySet()) {
            //assert on the key as "ReverseOrder"
            //assert on the value[map.get(str)] as "false"
        }
    } catch (UMException e) {
        e.printStackTrace();
    }

    // set preferences by editing a particular key/value pair of a Node.
    String path = "/Adobe/LiveCycle/Config/UM/CommonNameOrder";
    Map<String, String> map = new HashMap<String, String>();
    map.put("ReverseOrder", "true");
    try {
        pmutil.setPreferences(path, map);
        Map<String, String> map1 = pmutil.getPreferences(path);
        for(String str:map1.keySet()) {
            //assert on the key as "ReverseOrder"
            //assert on the value[map.get(str)] as "true"
        }
    } catch (UMException e) {
        e.printStackTrace();
    }
    }
}
```

Quick Start (SOAP mode): Programmatically managing the Preferences Nodes using the Java API

The following Java code models managing of Preferences Nodes by using the User Management APIs (See "Programmatically managing the Preferences Nodes" on page 1141)

```
/*
 * This Java Quick Start uses the SOAP mode and contains the following JAR files
 * in the class path:
 * 1. adobe-livecycle-client.jar
 * 2. adobe-usermanager-client.jar
 *
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/common
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/jboss
 *
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/jboss/bin/client
 *
 * If you want to invoke a remote AEM Forms instance and there is a
 * firewall between the client application and AEM FOrms, then it is
 * recommended that you use the SOAP mode. When using the SOAP mode,
 * you have to include additional JAR files located in the following
 * path
 * <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs/thirdparty
 *
 * For information about the SOAP
 * mode and the additional JAR files that need to be included,
 * see "Setting connection properties" in Programming
 * with AEM Forms
 *
 * For complete details about the location of the AEM Forms JAR files,
 * see "Including AEM FOrms library files" in Programming
 * with AEM FOrms
 */

import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.um.api.UMException;
import com.adobe.livecycle.usermanager.client.PreferenceManagerServiceClient;


public class ManagePreferences {

    public static void main(String[] args) {
    //Set connection properties required to invoke AEM Forms
        Properties connectionProps = new Properties();
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
    connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

    //Create a PreferenceManagerServiceClient object
    ServiceClientFactory factory = ServiceClientFactory.createInstance(connectionProps);
    PreferenceManagerServiceClient pmutil = new  PreferenceManagerServiceClient(factory);
```

```
    //get the preference map for a particular node
    String path = "/Adobe/LiveCycle/Config/UM/CommonNameOrder";
    Map<String, String> map;
    try {
        map = pmutil.getPreferences(path);
        for(String str:map.keySet()) {
            //assert on the key as "ReverseOrder"
            //assert on the value[map.get(str)] as "false"
        }
    } catch (UMException e) {
        e.printStackTrace();
    }

    // set preferences by editing a particular key/value pair of a Node.
    String path = "/Adobe/LiveCycle/Config/UM/CommonNameOrder";
    Map<String, String> map = new HashMap<String, String>();
    map.put("ReverseOrder", "true");
    try {
        pmutil.setPreferences(path, map);
        Map<String, String> map1 = pmutil.getPreferences(path);
        for(String str:map1.keySet()) {
            //assert on the key as "ReverseOrder"
            //assert on the value[map.get(str)] as "true"
        }
    } catch (UMException e) {
        e.printStackTrace();
    }
}
}
```

Chapter 3: Invoking AEM Forms using APIs

Invoking AEM Forms using APIs

Adobe Experience Manager Forms is J2EE-based enterprise software that consists of services that operate within a shared infrastructure. Service operations typically consume or produce documents. By using AEM Forms, you can combine forms workflow with electronic forms, document security, and document generation in an integrated and cohesive set of services. These services can be accessed from inside and outside the firewall.

Client applications can programmatically invoke AEM Forms services using a Java API, web services, Remoting, and REST. Using administration console, you can configure a service to expose an endpoint that lets AEM Forms services by programmatically invoked. By default, most services are pre-configured to expose Remoting, Java, and web service endpoints.

Your business requirements determine which invocation method to use. For example, using the Java API, you can integrate AEM Forms functionality into your Java enterprise applications, such as Java Entity and Message beans. Likewise, you can integrate AEM Forms functionality into .NET projects (or other projects developed with development environments that support web service standards) using web services.

Services require a service container to run, similar to how Enterprise JavaBeans™ (EJBs) require a J2EE container. AEM Forms includes only one implementation of a service container. The service container is responsible for managing the lifetime of a service, including deploying it and ensuring that all requests are sent to the correct service. It also manages documents that a service consumes or produces.

Note: *Programming with AEM forms does not include information on how to invoke AEM Forms using Watched Folders or e-mail.*

Understanding AEM Forms Processes

A common use case is for a set of AEM Forms services to operate on a single document. You can send a request to the service container by creating a process using Workbench. A process represents a business process that you are automating. For information about creating processes, see Using Workbench.

Once a process is activated, it becomes a service and can be invoked like other services. One difference between a standard service, such as the Encryption service and a service that originated from a process, is that the latter has one operation that performs many actions. In contrast, an standard service has many operations. Each operation typically performs one action, such as applying a policy to a document or encrypting a document.

Processes can be short-lived or long-lived. A short-lived process is an operation that is performed synchronously and on the same execution thread from which it was invoked. Short-lived operations are comparable to the standard behavior found in most programming languages, where a client application calls a method and waits for a return value.

However, there are situations where a process cannot be completed synchronously due to factors such as these:

- A process can span a significant amount of time.
- A process can span organizational boundaries.

- A process needs external input in order for it to finish. For example, consider a situation where a form is sent to a manager who is out of the office. In this situation, the process is not complete until the manager returns and fills out the form.

  These types of processes are known as long-lived processes. A long-lived process is performed asynchronously, allowing for systems to interact as resources permit and allowing for the tracking and monitoring of the operation. When a long-lived process is invoked, AEM Forms creates an invocation identifier value as part of a record that tracks the long-lived process status. The record is stored in the AEM Forms database. You can purge long-lived process records when they are no longer required. (See "Purging Process Data" on page 1073.)

  Note: *AEM Forms does not create a record when a short-lived process is invoked.*

  Using the invocation identifier value, you can track the status of the long-lived process. For example, you can use the process invocation identifier value to perform Process Manager operations such as terminating a running process instance. (See "Terminating Process Instances" on page 1071.)

Short lived process example

The following illustration is an example of a short-lived process named *MyApplication/EncryptDocument*.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code examples that discuss how to invoke this process, create a process named* `MyApplication/EncryptDocument` *using Workbench. (See Using Workbench.)*

When this short-lived process is invoked, it performs the following actions:

1. Obtains the unsecured PDF document that is passed to the process as an input value.
2. Encrypts the PDF document with a password. The name of the input parameter for this process is `inDoc` and the data type is document.
3. Saves the password-encrypted PDF document as a PDF file to the local file system. This process returns the encrypted PDF document as an output value. The name of the output parameter for this process is `outDoc` and the data type is document.

   This process is completed synchronously on the same execution thread from which it was invoked. The name of this short-lived process is `MyApplication/EncryptDocument` and its operation is `invoke`.

   Note: *Typically a short-lived process consists of more than three actions. You create a process using Workbench. (See Using Workbench.)*

   *Programming with AEM forms*describes the following ways in which you can programmatically invoke this short-lived process:

   - "Invoking a short-lived process by passing an unsecure document using Remoting" on page 449 (Using a Flex application)
   - "Invoking a short-lived process using the Invocation API" on page 512 (Java Invocation API)
   - "Invoking AEM Forms using Base64 encoding" on page 525 (web service example)
   - "Invoking AEM Forms using MTOM" on page 529 (web service example)
   - "Invoking AEM Forms using SwaRef" on page 531 (web service example)
   - "Invoking AEM Forms using BLOB data over HTTP" on page 533 (web service example)
   - "Invoking AEM Forms using DIME" on page 536 (web service example)
   -

Long-lived process example

The following illustration is an example of a long-lived process.

This process is invoked when an applicant submits a loan form. The process is not complete until a loan officer approves or rejects the loan request. The name of this long-lived process is *FirstAppSolution/PreLoanProcess* and its operation is `invoke_Async`. This process must be invoked asynchronously. For information about programmatically invoking this long-lived process, see "Invoking Human-Centric Long-Lived Processes" on page 560.

Note: *This process can be created by following the tutorial specified in Creating Your First AEM Forms Application.*

Service container

AEM Forms services located in the service container (including standard servces such as the Encryption service, long-lived, and short-lived processes) can be invoked using various providers, such as an EJB provider. An EJB provider enables AEM Forms services to be invoked over RMI/IIOP. A web service provider exposes services as web services (WSDL Generation) using standards such as SOAP/HTTP and SOAP/JMS.

The following table describes the different ways in which you can programmatically invoke AEM Forms services.

Invocation method	Description
Remote integration	Remote integration provides the ability for Flex clients to invoke service operations. (See "Invoking AEM Forms using Remoting" on page 444.)
Java API	A Java API can invoke an AEM Forms service. The Java API is organized into client libraries and the Java Invocation API. (See "Invoking AEM Forms using the Java API" on page 490.)
Web services	AEM Forms supports web service standards such as SOAP/HTTP. A service can be exposed as a web service, with the WSDL complying to web service standards defined by W3C. A service can be invoked from any web service stack, including the .NET Framework and Sun™ Web Services SDK. (See "Invoking AEM Forms using Web Services" on page 514.)
REST requests	AEM Forms supports REST requests. A service can be invoked directly from an HTML page. (See "Invoking AEM Forms using REST Requests" on page 553.)

The following illustration provides a visual representation of the different ways in which AEM Forms services can be programmatically invoked.

Note: *In addition to using the AEM Forms SDK to create client applications that can invoke AEM Forms services, you can also create components that can be deployed to the service container. For example, you can create a Bank component that contains custom data types that can be used in processes. That is, you can create a data type such as* `com.adobe.idp.BankAccount`*. You can then create* `com.adobe.idp.BankAccount` *instances in your client applications. (See Creating Components That Use Custom Data Types.)*

The service container provides the following functionality:

- Allows AEM Forms services to be invoked using different methods. You can configure a service by setting endpoints so that it can be invoked using all methods: Remoting, the Java API, web services, and REST. (See "Programmatically Managing Endpoints" on page 1111.)
- Converts a message into a normalized format called an invocation request. An invocation request is sent from a client application (or another service) to a service located in the service container. An invocation request contains information such as the name of the service to invoke and data values that are required to perform the operation. Many services require a document to perform an operation. Therefore, an invocation request usually contains a document, which can be PDF data, XDP data, XML data, and so on.

- Routes invocation requests to appropriate services (the name of the service to invoke is part of the invocation request).
- Performs tasks such as determining whether the caller has permission to invoke the specified service operation. The invocation request must contain a valid AEM forms user name and password.

  There are different ways to send an invocation request to a service. As well, there are different ways to send required input values to the service. For example, assume that you use the Java API to invoke a service that requires a PDF document. The corresponding Java method contains a parameter that accepts a PDF document. In this situation, the data type of the parameter is `com.adobe.idp.Document`. (See "Passing data to AEM Forms services using the Java API" on page 505.)

  If you invoke a service using watched folders, then an invocation request is sent when you place a file in a configured watched folder. If you invoke a service using e-mail, then an invocation request is sent to a service when an e-mail message arrives in a configured inbox.

  The service container sends back an invocation response once the operation is performed. An invocation response contains information such as the operation results. For example, if the operation modifies a PDF document, then the invocation response contains the modified PDF document. If the operation was unsuccessful, then the invocation response contains an error message.

  An invocation response can be retrieved in the same way in which an invocation request is sent. That is, if the invocation request is sent using the Java API, then an invocation response can be retrieved using the Java API. Assume, for example, that an operation modifies a PDF document. You can retrieve the modified PDF document by getting the return value of the Java method that invoked the service.

  When a long-lived process is invoked, an invocation response contains an identifier value that is associated with the invocation request. Using this identifier value, you can check the status of the process at a later time. For example, consider the MortgageLoan long-lived service. Using the identifier value, you can check to determine whether the process successfully completed. (See "Invoking Human-Centric Long-Lived Processes" on page 560.)

  The following diagram shows a client application (that uses the Java API) invoking a service.

  When a client application invokes a service, three events occur:

  1. A client application sends an invocation request to a service.
  2. The service performs the operation that is specified in the invocation request.
  3. The service container returns an invocation response to the client application.

See also

"Understanding AEM Forms Processes" on page 441

"Invoking AEM Forms using Remoting" on page 444

"Invoking AEM Forms using the Java API" on page 490

"Invoking AEM Forms using Web Services" on page 514

"Invoking Human-Centric Long-Lived Processes" on page 560

"Invoking AEM Forms using REST Requests" on page 553

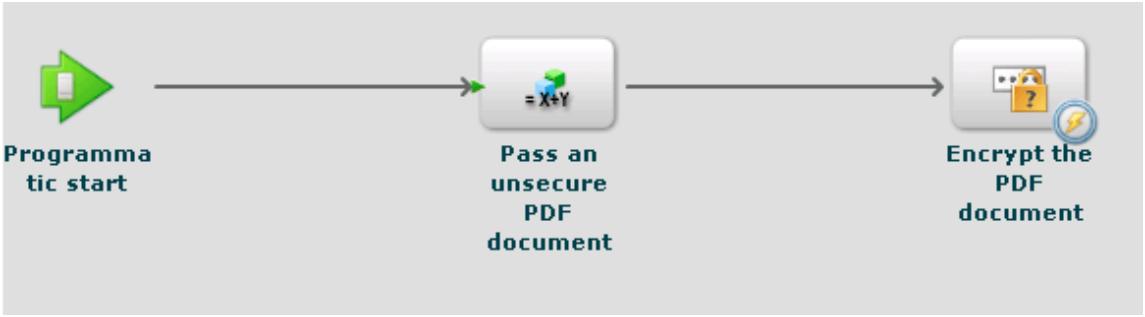
Invoking AEM Forms using Remoting

Processes created in Workbench can be invoked by using Remoting. That is, you can invoke a AEM Forms process from a client application built with Flex. This feature is based on Data Services.

Note: *When using Remoting, it is recommended that you invoke processes that were created in Workbench as opposed to AEM Forms services. However, it is possible to invoke AEM Forms services directly. (See Encrypting PDF documents using Remoting located on AEM Forms Developer Center.)*

Note: *If a AEM Forms service is not configured to allow anonymous access, requests from a Flex client result in a web browser challenge. The user must enter user name and password credentials.*

The following AEM Forms short-lived process, named `MyApplication/EncryptDocument`, can be invoked using Remoting. (For information about this process such as its input and output values, see .)



Note: *To invoke an AEM Forms process using a Flex application, ensure that a remoting endpoint is enabled. By default, a remoting endpoint is enabled when you deploy a process.*

When this process is invoked, it performs the following actions:

1. Obtains the unsecured PDF document that is passed as an input value. This action is based on the `SetValue` operation. The name of the input parameter is `inDoc` and its data type is `document`. (The `document` data type is an available data type from within Workbench.)
2. Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The name of the output value for this process is `outDoc` and represents the password-encrypted PDF document. The data type of outDoc is `document`.
3. Saves the password-encrypted PDF document as a PDF file to the local file system. This action is based on the `WriteDocument` operation.

Note: *The `MyApplication/EncryptDocument` process is not based on an existing AEM Forms process. To following along with the code examples, create a process named `MyApplication/EncryptDocument` using Workbench.*

Note: *For information about using Remoting to invoke a long-lived process, see "Invoking Human-Centric Long-Lived Processes" on page 560.*

See also

"Including the AEM Forms Flex library file" on page 446

"Handling documents with Remoting" on page 446

"Invoking a short-lived process by passing an unsecure document using Remoting" on page 449

"Authenticating client applications built with Flex" on page 451

"Passing secure documents to invoke processes using Remoting" on page 456

"Invoking custom component services using Remoting" on page 463

"Creating a client application built with Flex that invokes a human-centric long-lived process" on page 576

Creating Flash Builder applications that perform SSO authentication using HTTP tokens

For information on how to display process data in a Flex graph control, see Displaying AEM Forms process data in Flex graphs.

Besure to place the crossdomain.xml file in the proper place. For example, assuming that you deployed AEM Forms on JBoss, place this file in the following location: <install_directory>\Adobe_Experience_Manager_forms\jboss\server\lc_turnkey\deploy\jboss-web.deployer\ROOT.war.

Including the AEM Forms Flex library file

To programmatically invoke AEM Forms processes using Remoting, add the adobe-remoting-provider.swc file to your Flex project's class path. This SWC file is located in the following location:

- *<install_directory>\Adobe_Experience_Manager_forms\sdk\misc\DataServices\Client-Libraries*

  where *<install_directory>* is the directory where AEM Forms is installed.

See also

"Invoking AEM Forms using Remoting" on page 444

"Handling documents with Remoting" on page 446

"Invoking a short-lived process by passing an unsecure document using Remoting" on page 449

"Authenticating client applications built with Flex" on page 451

Handling documents with Remoting

One of the most important non-primitive Java types used in AEM Forms is the `com.adobe.idp.Document` class. A document is commonly required to invoke a AEM Forms operation. It is primarily a PDF document, but can contain other document types such as SWF, HTML, XML, or a DOC file. (See "Passing data to AEM Forms services using the Java API" on page 505.)

A client application built with Flex cannot directly request a document. For example, you cannot launch Adobe Reader to request a URL that produces a PDF file. Requests for document types, such as PDF and Microsoft Word documents, return a result that is a URL. It is the client's responsibility to display the contents of the URL. The Document Management service helps generate the URL and content type information. Requests for XML documents return the full XML document in the result.

Passing a document as an input parameter

A client application built with Flex cannot pass a document directly to a AEM Forms process. Instead, the client application uses an instance of the `mx.rpc.livecycle.DocumentReference` ActionScript class to pass input parameters to an operation that expects a `com.adobe.idp.Document` instance. A Flex client application has several options for setting up a `DocumentReference` object:

- When the document is on the server and its file location is known, set the `DocumentReference` object's `referenceType` property to `REF_TYPE_FILE`. Set the `fileRef` property to the location of the file, as the following example shows:

```
...
var docRef:DocumentReference = new DocumentReference();
docRef.referenceType=DocumentReference.REF_TYPE_FILE;
docRef.fileRef = "C:/install/adobe/cs2/How to Uninstall.pdf";
...
```

- When the document is on the server and you know its URL, set the `DocumentReference` object's `referenceType` property to `REF_TYPE_URL`. Set the `url` property to the URL, as the following example shows:

```
...
var docRef:DocumentReference = new DocumentReference();
docRef.referenceType=DocumentReference.REF_TYPE_URL;
docRef.url = "http://companyserver:8080/DocumentManager/116/7855";
...
```

- To create a `DocumentReference` object from a text string in the client application, set the `DocumentReference` object's `referenceType` property to `REF_TYPE_INLINE`. Set the `text` property to the text to include in the object, as the following example shows:

```
...
var docRef:DocumentReference = new DocumentReference();
docRef.referenceType=DocumentReference.REF_TYPE_INLINE;
docRef.text = "Text for my document";
// Optionally, you can override the server's default character set
// if necessary:
// docRef.charsetName=CharacterSetName
...
```

- When the document is not on the server, use the Remoting upload servlet to upload a document to AEM Forms. New in AEM Forms is the ability to upload secure documents. When uploading a secure document, you have to use a user who has the *Document Upload Application User* role. Without this role, the user cannot upload a secure document. It is recommended that you use single sign on to upload a secure document. (See "Passing secure documents to invoke processes using Remoting" on page 456.)

  Note: *if AEM Forms is configured to allow unsecure documents to be uploaded, you can use a user that does not have the Document Upload Application User role to upload a document. A user can also have the Document Upload permission. However, if AEM Forms is configured to only allow secure documents, then ensure that the user has the Document Upload Application User role or Document Upload permission. (See "Configuring AEM Forms to accept secure and unsecure documents" on page 458.)*

  You use standard Flash upload capabilities for the designated upload URL: `http://SERVER:PORT/remoting/lcfileupload`. You can then use the `DocumentReference` object wherever an input parameter of type `Document` is expected

```
private function startUpload():void
{
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener("uploadCompleteData", completeHandler);
try
{
    var success:Boolean = fileRef.browse();
}

catch (error:Error)
{
    trace("Unable to browse for files.");
}
}


private function selectHandler(event:Event):void {
var request:URLRequest = new
URLRequest("http://SERVER:PORT/remoting/lcfileupload")
try
    {
    fileRef.upload(request);
    }

catch (error:Error)
    {
    trace("Unable to upload file.");
    }
}

private function completeHandler(event:DataEvent):void
{
    var params:Object = new Object();
    var docRef:DocumentReference = new DocumentReference();
    docRef.url = event.data as String;
    docRef.referenceType = DocumentReference.REF_TYPE_URL;
}
```

The Remoting Quick Start uses the Remoting upload servlet to pass a PDF file to the `MyApplication/EncryptDocument` process. (See "Invoking a short-lived process by passing an unsecure document using Remoting" on page 449.)

Passing a document back to a client application

A client application receives an object of type `mx.rpc.livecycle.DocumentReference` for a service operation that returns an `com.adobe.idp.Document` instance as an output parameter. Because a client application deals with ActionScript objects and not Java, you cannot pass a Java-based Document object back to a Flex client. Instead, the server generates a URL for the document and passes the URL back to the client. The `DocumentReference` object's `referenceType` property specifies whether the content is in the `DocumentReference` object or must be retrieved from a URL in the `DocumentReference.url` property. The `DocumentReference.contentType` property specifies the type of document.

See also

"Invoking AEM Forms using Remoting" on page 444

"Including the AEM Forms Flex library file" on page 446

Invoking a short-lived process by passing an unsecure document using Remoting

To invoke a AEM Forms process from an application built with Flex, perform the following tasks:

1. Create a `mx:RemoteObject` instance.
2. Create a `ChannelSet` instance.
3. Pass required input values.
4. Handle return values.

Note: *This section discusses how to invoke a AEM Forms process and upload a document when AEM Forms is configured to upload unsecure documents. For information about how to invoke AEM Forms processes and upload secure documents and how to configure AEM Forms to accept secure and unsecure documents, see "Passing secure documents to invoke processes using Remoting" on page 456.*

Creating a mx:RemoteObject instance

You create a `mx:RemoteObject` instance to invoke a AEM Forms process created in Workbench. To create a `mx:RemoteObject` instance, specify the following values:

- **id:** The name of the `mx:RemoteObject` instance that represents the process to invoke.
- **destination:** The name of the AEM Forms process to invoke. For example, to invoke the `MyApplication/EncryptDocument` process, specify `MyApplication/EncryptDocument`.
- **result:** The name of the Flex method that handles the result.

Within the `mx:RemoteObject` tag, specify a `<mx:method>` tag that specifies the name of the process's invocation method. Typically, the name of a Forms invocation method is `invoke`.

The following code example creates a `mx:RemoteObject` instance that invokes the `MyApplication/EncryptDocument` process.

```
<mx:RemoteObject id="EncryptDocument" destination="MyApplication/EncryptDocument"
result="resultHandler(event);">
         <mx:method name="invoke" result="handleExecuteInvoke(event)"/>
     </mx:RemoteObject>
```

Create a Channel to AEM Forms

A client application can invoke AEM Forms by specifying a Channel in MXML or ActionScript, as the following ActionScript example shows. The Channel must be an `AMFChannel`, `SecureAMFChannel`, `HTTPChannel`, or `SecureHTTPChannel`.

```
...
private function refresh():void{
    var cs:ChannelSet= new ChannelSet();
    cs.addChannel(new AMFChannel("my-amf",
        "http://yourlcserver:8080/remoting/messagebroker/amf"));
    EncryptDocument.setCredentials("administrator", "password");
    EncryptDocument.channelSet = cs;
}
...
```

Assign the `ChannelSet` instance to the `mx:RemoteObject` instance's `channelSet` field (as shown in the previous code example). Generally, you import the channel class in an import statement rather than specifying the fully qualified name when you invoke the `ChannelSet.addChannel` method.

Passing input values

A process created in Workbench can take zero or more input parameters and return an output value. A client application passes input parameters within an `ActionScript` object with fields that correspond to parameters that belong to the AEM Forms process. The short-lived process, named `MyApplication/EncryptDocument`, requires one input parameter named `inDoc`. The name of the operation exposed by the process is `invoke` (the default name for a short-lived process). (See "Invoking AEM Forms using Remoting" on page 444.)

The following code example passes a PDF document to the `MyApplication/EncryptDocument` process:

```
...
var params:Object = new Object();

//Document is an instance of DocumentReference
//that store an unsecured PDF document
params["inDoc"] = pdfDocument;

// Invoke an operation synchronously:
EncryptDocument.invoke(params);
...
```

In this code example, `pdfDocument` is a `DocumentReference` instance that contains an unsecured PDF document. For information about a `DocumentReference`, see "Handling documents with Remoting" on page 446.

Invoking a specific version of a service

You can invoke a specific version of a Forms service by using a `_version` parameter in the invocation's parameter map. For example, to invoke version 1.2 of the `MyApplication/EncryptDocument` service:

```
var params:Object = new Object();
params["inDoc"] = pdfDocument;
params["_version"] = "1.2"
var token:AsyncToken = echoService.echoString(params);
```

The `version` parameter must be a string containing a single period. The values to the left, major version, and right, minor version, of the period must be integers. If this parameter is not specified, the head active version is invoked.

Handling return values

AEM Forms process output parameters are deserialized into ActionScript objects from which the client application extracts specific parameters by name, as the following example shows. (The output value of the `MyApplication/EncryptDocument` process is named `outDoc`.)

```
...
var res:Object = event.result;
var docRef:DocumentReference = res["outDoc"] as DocumentReference;
...
```

Invoking the MyApplication/EncryptDocument process

You can invoke the `MyApplication/EncryptDocument` process by performing the following steps:

1 Create a `mx:RemoteObject` instance through either ActionScript or MXML. (See)

2 Set up a `ChannelSet` instance to communicate with AEM Forms, and associate it with the `mx:RemoteObject` instance. (See .)

3 Call the ChannelSet's `login` method or the service's `setCredentials` method to specify the user identifier value and password. (See "Using single sign-on" on page 454.)

4 Populate an `mx.rpc.livecycle.DocumentReference` instance with an unsecured PDF document to pass to the `MyApplication/EncryptDocument` process. (See "Passing a document as an input parameter" on page 446.)

5 Encrypt the PDF document by calling the `mx:RemoteObject` instance's `invoke` method. Pass the `Object` that contains the input parameter (which is the unsecured PDF document). (See .)

6 Retrieve the password-encrypted PDF document that is returned from the process. (See .)

"Quick Start: Invoking a short-lived process by passing an unsecure document using (Deprecated for AEM forms) AEM Forms Remoting" on page 225

Authenticating client applications built with Flex

There are several ways that AEM forms user Manager can authenticate a Remoting request from a Flex application, including AEM Forms single sign-on through the central login service, basic authentication, and custom authentication. When neither single sign-on nor anonymous access is enabled, a Remoting request results in either basic authentication (the default) or custom authentication.

Basic authentication relies on standard J2EE basic authentication from the web application container. For basic authentication, an HTTP 401 error causes a browser challenge. That means that when you attempt to connect to a Forms application by using RemoteObject, and have not yet logged in from the Flex application, the browser prompts you for a user name and password.

For custom authentication, the server sends a fault to the client to indicate that authentication is required.

Note: *For information about performing authentication using HTTP tokens, see Creating Flash Builder applications that perform SSO authentication using HTTP tokens.*

Using custom authentication

You enable custom authentication in administration console by changing the authentication method from Basic to Custom on the remoting endpoint. If you use custom authentication, your client application calls the `ChannelSet.login` method to log in and the `ChannelSet.logout` method to log out.

Note: *In the previous release of AEM Forms, you sent credentials to a destination by calling the `RemoteObject.setCredentials` method. The `setCredentials` method did not actually pass the credentials to the server until the first attempt by the component to connect to the server. Therefore, if the component issued a fault event, you could not be certain if the fault happened because of an authentication error, or for another reason. The `ChannelSet.login` method connects to the server when you call it so that you can handle an authentication issue immediately. Although you can continue to use the `setCredentials` method, it is recommended that you use the `ChannelSet.login` method.*

Because multiple destinations can use the same channels, and corresponding ChannelSet object, logging in to one destination logs the user in to any other destination that uses the same channel or channels. If two components apply different credentials to the same ChannelSet object, the last credentials applied are used. If multiple components use the same authenticated ChannelSet object, calling the `logout` method logs all components out of the destinations.

The following example uses the `ChannelSet.login` and `ChannelSet.logout` methods with a RemoteObject control. This application performs the following actions:

- Creates a `ChannelSet` object in the `creationComplete` handler that represents the channels used by the `RemoteObject` component

- Passes credentials to the server by calling the `ROLogin` function in response to a Button click event
- Uses the RemoteObject component to send a String to the server in response to a Button click event. The server returns the same String back to the RemoteObject component
- Uses the result event of the RemoteObject component to display the String in a TextArea control
- Logs out of the server by calling the `ROLogout` function in response to a Button click event

```
<?xml version="1.0"?>
<!-- security/SecurityConstraintCustom.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
    height="100%" creationComplete="creationCompleteHandler();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.messaging.config.ServerConfig;
            import mx.rpc.AsyncToken;
            import mx.rpc.AsyncResponder;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.messaging.ChannelSet;

            // Define a ChannelSet object.
            public var cs:ChannelSet;

            // Define an AsyncToken object.
            public var token:AsyncToken;

            // Initialize ChannelSet object based on the
            // destination of the RemoteObject component.
            private function creationCompleteHandler():void {
                if (cs == null)
                cs = ServerConfig.getChannelSet(remoteObject.destination);
            }

            // Login and handle authentication success or failure.
            private function ROLogin():void {
                // Make sure that the user is not already logged in.
                if (cs.authenticated == false) {
                    token = cs.login("sampleuser", "samplepassword");
                    // Add result and fault handlers.
                    token.addResponder(new AsyncResponder(LoginResultEvent,
                    LoginFaultEvent));
                }
            }

            // Handle successful login.
            private function LoginResultEvent(event:ResultEvent,
                token:Object=null):void  {
                    switch(event.result) {
                        case "success":
                            authenticatedCB.selected = true;
                            break;
                            default:
                    }
                }
```

```
            // Handle login failure.
            private function LoginFaultEvent(event:FaultEvent,
                token:Object=null):void {
                    switch(event.fault.faultCode) {
                        case "Client.Authentication":
                            default:
                            authenticatedCB.selected = false;
                            Alert.show("Login failure: " + event.fault.faultString);
                }
            }

            // Logout and handle success or failure.
            private function ROLogout():void {
                // Add result and fault handlers.
                token = cs.logout();
                token.addResponder(new
                    AsyncResponder(LogoutResultEvent,LogoutFaultEvent));
            }

            // Handle successful logout.
            private function LogoutResultEvent(event:ResultEvent,
                token:Object=null):void {
                    switch (event.result) {
                        case "success":
                            authenticatedCB.selected = false;
                            break;
                            default:
                }
            }

            // Handle logout failure.
            private function LogoutFaultEvent(event:FaultEvent,
                token:Object=null):void {
                    Alert.show("Logout failure: " + event.fault.faultString);
            }
            // Handle message recevied by RemoteObject component.
            private function resultHandler(event:ResultEvent):void {
                ta.text += "Server responded: "+ event.result + "\n";
            }

            // Handle fault from RemoteObject component.
            private function faultHandler(event:FaultEvent):void {
                ta.text += "Received fault: " + event.fault + "\n";
            }
        ]]>
</mx:Script>
<mx:HBox>
```

```
        <mx:Label text="Enter a text for the server to echo"/>
        <mx:TextInput id="ti" text="Hello World!"/>
        <mx:Button label="Login"
            click="ROLogin();"/>
        <mx:Button label="Echo"
            enabled="{authenticatedCB.selected}"
            click="remoteObject.echo(ti.text);"/>
        <mx:Button label="Logout"
            click="ROLogout();"/>
        <mx:CheckBox id="authenticatedCB"
            label="Authenticated?"
            enabled="false"/>
    </mx:HBox>
    <mx:TextArea id="ta" width="100%" height="100%"/>

    <mx:RemoteObject id="remoteObject"
        destination="myDest"
        result="resultHandler(event);"
        fault="faultHandler(event);"/>
</mx:Application>
```

The `login` and `logout` methods return an AsyncToken object. Assign event handlers to the AsyncToken object for the result event to handle a successful call, and for the fault event to handle a failure.

Using single sign-on

AEM forms users can connect to multiple AEM Forms web applications to perform a task. As users move from one web application to another, it is not efficient to require them to log in separately to each web application. The AEM Forms single sign-on mechanism lets users log in once, and then access any AEM Forms web application. Because AEM Forms developers can create client applications for use with AEM Forms, they must also be able to take advantage of the single sign-on mechanism.

Each AEM Forms web application is packaged in its own Web Archive (WAR) file, which is then packaged as part of an Enterprise Archive (EAR) file. Because an application server does not allow the sharing of session data across different web applications, AEM Forms uses HTTP cookies to store authentication information. Authentication cookies enable a user to log in to a Forms application, and then connect to other AEM Forms web applications. This technique is known as single sign-on.

AEM Forms developers write client applications to extend the functionality of form Guides (deprecated) and to customize Workspace. For example, a Workspace application can start a process. The client application then uses a remoting endpoint to retrieve data from the Forms service.

When an AEM Forms service is invoked using (Deprecated for AEM forms) AEM Forms Remoting, the client application passes the authentication cookie as part of the request. Because the user has already authenticated, no additional login is required to make a connection from the client application to the AEM Forms service.

Note: *If a cookie is invalid or missing, there is no implicit redirect to a login page. Therefore, you can still call an anonymous service.*

You can bypass the AEM Forms single sign-on mechanism by writing a client application that logs in and logs out on its own. If you bypass the single sign-on mechanism, you can use either basic or custom authentication with your application.

Because this mechanism does not use the AEM Forms single sign-on mechanism, no authentication cookie is written to the client. Login credentials are stored in the `ChannelSet` object for the remoting channel. Therefore, any `RemoteObject` calls you make over the same `ChannelSet` are made in the context of those credentials.

Setting up single sign-on in AEM Forms

To use single sign-on in AEM Forms, install the forms workflow component, which includes the centralized login service. After a user successfully logs in, the centralized login service returns an authentication cookie to the user. Every subsequent request to a Forms web applications contains the cookie. If the cookie is valid, the user is considered to be authenticated and does not have to log in again.

Writing a client application that uses single sign-on

When you take advantage of the single sign-on mechanism, you expect users to log in by using the centralized login service before starting a client application. That is, a client application does not log in through the browser or by calling the `ChannelSet.login` method.

If you are using the AEM Forms single sign-on mechanism, configure the Remoting endpoint to use custom authentication, not basic. Otherwise, when using basic authentication, an authentication error causes a browser challenge, which you do not want the user to see. Instead, your application detects the authentication error and then displays a message instructing the user to log in using the centralized login service.

A client application accesses AEM Forms through a remoting endpoint by using the `RemoteObject` component, as the following example shows.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
       backgroundColor="#FFFFFF">

      <mx:Script>
         <![CDATA[

           import mx.controls.Alert;
           import mx.rpc.events.FaultEvent;

           // Prompt user to login on a fault.
           private function faultHandler(event:FaultEvent):void
           {
            if(event.fault.faultCode=="Client.Authentication")
            {
                Alert.show(
                    event.fault.faultString + "\n" +
                    event.fault.faultCode + "\n" +
                    "Please login to continue.");
            }
        }
         ]]>
      </mx:Script>

      <mx:RemoteObject id="srv"
          destination="product"
          fault="faultHandler(event);"/>

      <mx:DataGrid
          width="100%" height="100%"
          dataProvider="{srv.getProducts.lastResult}"/>

      <mx:Button label="Get Data"
          click="srv.getProducts();"/>

</mx:Application>
```

Logging in as a new user while the Flex application is still running

An application built with Flex includes the authentication cookie with every request to an AEM Forms service. For performance reasons, AEM Forms does not validate the cookie on every request. However, AEM Forms does detect when an authentication cookie is replaced with another authentication cookie.

For example, you start a client application and while the application is active, you use the centralized login service to log out. Next, you can log in as a different user. Logging in as a different user replaces the existing authentication cookie with an authentication cookie for the new user.

On the next request from the client application, AEM Forms detects that the cookie has changed, and logs out the user. Therefore, the first request after a cookie change fails. All subsequent requests are made in the context of the new cookie and are successful.

Logging out

To log out of AEM Forms and invalidate a session, the authentication cookie must be deleted from the client's computer. Because the purpose of single sign-on is to allow a user to log in once, you do not want a client application to delete the cookie. This action effectively logs out the user.

Therefore, calling the `RemoteObject.logout` method in a client application generates an error message on the client specifying that the session is not logged out. Instead, the user can use the centralized login service to log out and delete the authentication cookie.

Logging out while the Flex application is still running

You can start a client application built with Flex and use the centralized login service to log out. As part of the logout process, the authentication cookie is deleted. If a remoting request is made without a cookie, or with an invalid cookie, the user session is invalidated. This action is in effect a logout. The next time the client application attempts to connect to an AEM Forms service, the user is requested to log in.

See also

"Invoking AEM Forms using Remoting" on page 444

"Handling documents with Remoting" on page 446

"Including the AEM Forms Flex library file" on page 446

"Invoking a short-lived process by passing an unsecure document using Remoting" on page 449

"Passing secure documents to invoke processes using Remoting" on page 456

Passing secure documents to invoke processes using Remoting

You can pass secure documents to AEM Forms when invoking a process that requires one or more documents. By passing a secure document, you are protecting business information and confidential documents. In this situation, a document can refer to a PDF document, an XML document, a Word document, and so on. Passing a secure document to AEM Forms from a client application written in Flex is required when AEM Forms is configured to allow secure documents. (See "Configuring AEM Forms to accept secure and unsecure documents" on page 458.)

When passing a secure document, use single sign-on and specify a AEM forms user who has the *Document Upload Application User* role. Without this role, the user cannot upload a secure document. You can programmatically assign a role to a user. (See "Managing Roles and Permissions" on page 1026.)

Note: *When you create a new role and you want members of that role to upload secure documents, ensure that you specify the Document Upload permission.*

AEM Forms supports an operation named `getFileUploadToken` that returns a token that is passed to the upload servlet. The `DocumentReference.constructRequestForUpload` method requires a URL to AEM Forms along with the token returned by the `LC.FileUploadAuthenticator.getFileUploadToken` method. This method returns a `URLRequest` object that is used in the invocation to the upload servlet. The following code demonstrates this application logic.

```
    ...
        private function startUpload():void
        {
            fileRef.addEventListener(Event.SELECT, selectHandler);
            fileRef.addEventListener("uploadCompleteData", completeHandler);
            try
            {
        var success:Boolean = fileRef.browse();
            }
            catch (error:Error)
            {
                trace("Unable to browse for files.");
            }

        }

         private function selectHandler(event:Event):void
            {
            var authTokenService:RemoteObject = new
RemoteObject("LC.FileUploadAuthenticator");
            authTokenService.addEventListener("result", authTokenReceived);
            authTokenService.channelSet = cs;
            authTokenService.getFileUploadToken();
            }

        private function authTokenReceived(event:ResultEvent):void
            {
            var token:String = event.result as String;
            var request:URLRequest =
DocumentReference.constructRequestForUpload("http://localhost:8080", token);

            try
            {
         fileRef.upload(request);
            }
            catch (error:Error)
            {
            trace("Unable to upload file.");
            }
            }

        private function completeHandler(event:DataEvent):void
        {

            var params:Object = new Object();
            var docRef:DocumentReference = new DocumentReference();
            docRef.url = event.data as String;
            docRef.referenceType = DocumentReference.REF_TYPE_URL;
        }
        ...
```

)

Configuring AEM Forms to accept secure and unsecure documents

You can use administration console to specify whether documents are secure when passing a document from a Flex client application to a AEM Forms process. By default, AEM Forms is configured to accept secure documents. You can configure AEM Forms to accept secure documents by performing the following steps:

1 Log in to administration console.

2 Click **Settings**.

3 Click **Core System Settings.**

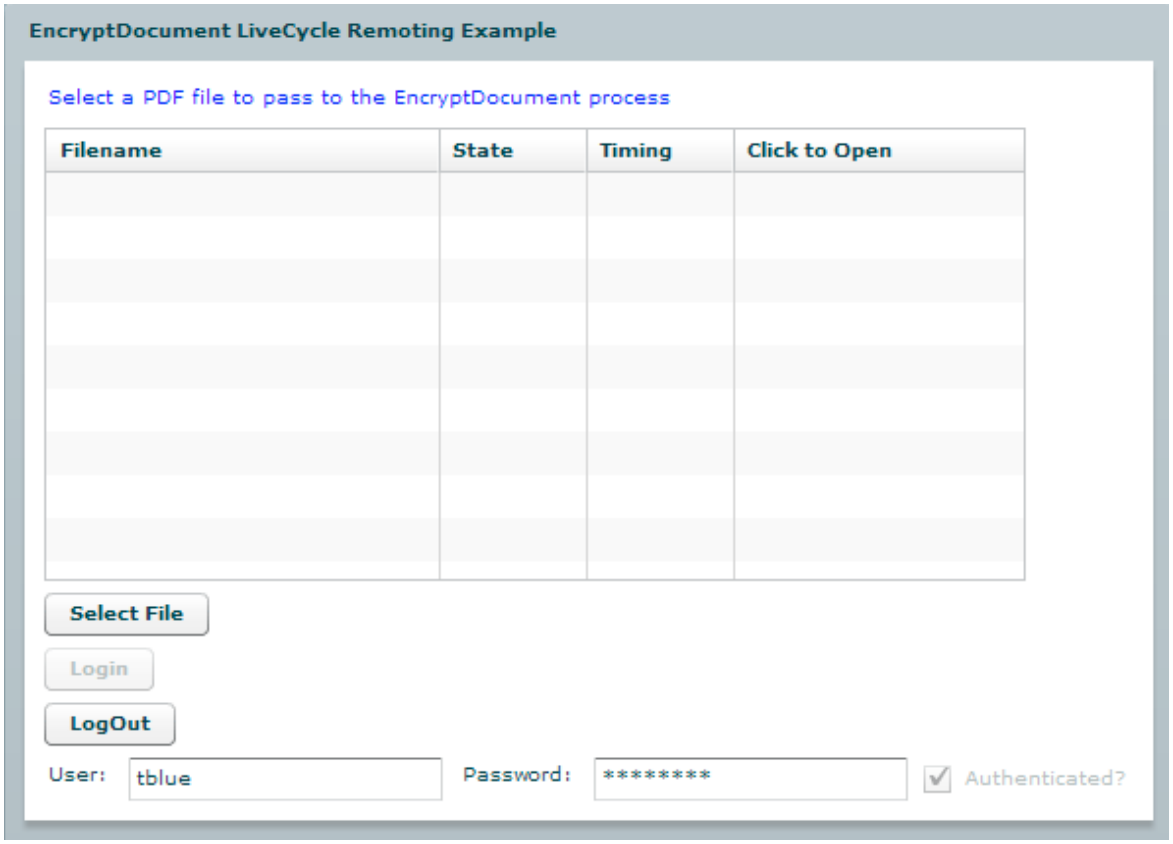
4 Click Configurations.

5 Ensure that the Allow non secured document upload from Flex applications option is unselected.

Note: *To configure AEM Forms to accept unsecure documents, select the Allow non secured document upload from Flex applications option. Then restart an application or service to ensure that the setting takes effect.*

Quick Start: Invoking a short-lived process by passing a secure document using Remoting

The following code example invokes the `MyApplication/EncryptDocument` . A user must login to click the Select File button that is used to upload a PDF file and invoke the process. That is, once the user is authenticated, the Select File button is enabled. The following illustration shows the Flex client application after a user is authenticated. Notice that the Authenticated CheckBox is enabled.



if AEM Forms is configured to only allow secure documents to be uploaded and the user not have the *Document Upload Application User* role, then an exception is thrown. If the user does have this role, then the file is uploaded and the process is invoked.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
     creationComplete="initializeChannelSet();">
       <mx:Script>
       <![CDATA[
     import mx.rpc.livecycle.DocumentReference;
     import flash.net.FileReference;
     import flash.net.URLRequest;
     import flash.events.Event;
     import flash.events.DataEvent;
     import mx.messaging.ChannelSet;
     import mx.messaging.channels.AMFChannel;
     import mx.rpc.events.ResultEvent;
     import mx.collections.ArrayCollection;
     import mx.rpc.AsyncToken;
     import mx.controls.Alert;
     import mx.rpc.events.FaultEvent;
     import mx.rpc.AsyncResponder;

     // Classes used in file retrieval
     private var fileRef:FileReference = new FileReference();
     private var docRef:DocumentReference = new DocumentReference();
     private var parentResourcePath:String = "/";
     private var now1:Date;
     private var serverPort:String = "hiro-xp:8080";

     // Define a ChannelSet object.
     public var cs:ChannelSet;

     // Define an AsyncToken object.
     public var token:AsyncToken;

      // Holds information returned from AEM Forms
     [Bindable]
     public var progressList:ArrayCollection = new ArrayCollection();


     // Handles a successful login
    private function LoginResultEvent(event:ResultEvent,
        token:Object=null):void  {
            switch(event.result) {
                case "success":
                    authenticatedCB.selected = true;
                    btnFile.enabled = true;
                    btnLogout.enabled = true;
                    btnLogin.enabled = false;
                        break;
                    default:
                }
            }


// Handle login failure.
```

```
private function LoginFaultEvent(event:FaultEvent,
    token:Object=null):void {
    switch(event.fault.faultCode) {
                case "Client.Authentication":
                        default:
                        authenticatedCB.selected = false;
                        Alert.show("Login failure: " + event.fault.faultString);
                }
            }


     // Set up channel set to invoke AEM Forms
     private function initializeChannelSet():void {
       cs = new ChannelSet();
       cs.addChannel(new AMFChannel("remoting-amf", "http://" + serverPort +
"/remoting/messagebroker/amf"));
       EncryptDocument2.channelSet = cs;
     }

    // Call this method to upload the file.
     // This creates a file picker and lets the user select a PDF file to pass to the
EncryptDocument process.
     private function uploadFile():void {
       fileRef.addEventListener(Event.SELECT, selectHandler);
       fileRef.addEventListener(DataEvent.UPLOAD_COMPLETE_DATA,completeHandler);
       fileRef.browse();
     }

     // Gets called for selected file. Does the actual upload via the file upload servlet.
     private function selectHandler(event:Event):void {
            var authTokenService:RemoteObject = new
RemoteObject("LC.FileUploadAuthenticator");
        authTokenService.addEventListener("result", authTokenReceived);
        authTokenService.channelSet = cs;
        authTokenService.getFileUploadToken();
     }

    private function authTokenReceived(event:ResultEvent):void
    {
    var token:String = event.result as String;
    var request:URLRequest = DocumentReference.constructRequestForUpload("http://hiro-
xp:8080", token);

    try
    {
         fileRef.upload(request);
    }
    catch (error:Error)
    {
        trace("Unable to upload file.");
    }
}

     // Called once the file is completely uploaded.
     private function completeHandler(event:DataEvent):void {

       // Set the docRef's url and referenceType parameters
```

```
    docRef.url = event.data as String;
    docRef.referenceType=DocumentReference.REF_TYPE_URL;
    executeInvokeProcess();
  }

 //This method invokes the EncryptDocument process
  public function executeInvokeProcess():void {
     //Create an Object to store the input value for the EncryptDocument process
       now1 = new Date();

     var params:Object = new Object();
     params["inDoc"]=docRef;

     // Invoke the EncryptDocument process
     var token:AsyncToken;
     token = EncryptDocument2.invoke(params);
     token.name = name;
  }

  // AEM Forms  login method
  private function ROLogin():void {
     // Make sure that the user is not already logged in.

     //Get the User and Password
     var userName:String = txtUser.text;
     var pass:String = txtPassword.text;

    if (cs.authenticated == false) {
         token = cs.login(userName, pass);

     // Add result and fault handlers.
     token.addResponder(new AsyncResponder(LoginResultEvent,LoginFaultEvent));
             }
         }

  // This method handles a successful process invocation
  public function handleResult(event:ResultEvent):void
  {
       //Retrieve information returned from the service invocation
      var token:AsyncToken = event.token;
      var res:Object = event.result;
      var dr:DocumentReference = res["outDoc"] as DocumentReference;
      var now2:Date = new Date();

       // These fields map to columns in the DataGrid
      var progObject:Object = new Object();
      progObject.filename = token.name;
      progObject.timing = (now2.time - now1.time).toString();
      progObject.state = "Success";
      progObject.link = "<a href='" + dr.url + "'> open </a>";
      progressList.addItem(progObject);
  }

  // Prompt user to login on a fault.
   private function faultHandler(event:FaultEvent):void
        {
         if(event.fault.faultCode=="Client.Authentication")
```

```
            {
                Alert.show(
                    event.fault.faultString + "\n" +
                    event.fault.faultCode + "\n" +
                    "Please login to continue.");
            }
           }

      // AEM Forms  logout method
    private function ROLogout():void {
        // Add result and fault handlers.
        token = cs.logout();
        token.addResponder(new AsyncResponder(LogoutResultEvent,LogoutFaultEvent));
    }

    // Handle successful logout.
    private function LogoutResultEvent(event:ResultEvent,
        token:Object=null):void {
        switch (event.result) {
        case "success":
                authenticatedCB.selected = false;
                btnFile.enabled = false;
                btnLogout.enabled = false;
                btnLogin.enabled = true;
                break;
                default:
            }
    }

    // Handle logout failure.
    private function LogoutFaultEvent(event:FaultEvent,
            token:Object=null):void {
            Alert.show("Logout failure: " + event.fault.faultString);
    }

        private function resultHandler(event:ResultEvent):void {
        // Do anything else here.
        }
       ]]>

     </mx:Script>
     <mx:RemoteObject id="EncryptDocument" destination="MyApplication/EncryptDocument"
result="resultHandler(event);">
          <mx:method name="invoke" result="handleResult(event)"/>
     </mx:RemoteObject>

      <!--//This consists of what is displayed on the webpage-->
     <mx:Panel id="lcPanel" title="EncryptDocument  (Deprecated for AEM forms) AEM Forms
Remoting Example"
          height="25%" width="25%" paddingTop="10" paddingLeft="10" paddingRight="10"
          paddingBottom="10">
        <mx:Label width="100%" color="blue"
               text="Select a PDF file to pass to the EncryptDocument process"/>
       <mx:DataGrid x="10" y="0" width="500" id="idProgress" editable="false"
          dataProvider="{progressList}" height="231" selectable="false" >
         <mx:columns>
           <mx:DataGridColumn headerText="Filename" width="200" dataField="filename"
```

```
editable="false"/>
           <mx:DataGridColumn headerText="State" width="75" dataField="state"
editable="false"/>
           <mx:DataGridColumn headerText="Timing" width="75" dataField="timing"
editable="false"/>
           <mx:DataGridColumn headerText="Click to Open" dataField="link" editable="false" >
            <mx:itemRenderer>
               <mx:Component>
                  <mx:Text x="0" y="0" width="100%" htmlText="{data.link}"/>
               </mx:Component>
            </mx:itemRenderer>
           </mx:DataGridColumn>
         </mx:columns>
       </mx:DataGrid>
       <mx:Button label="Select File" click="uploadFile()"  id="btnFile" enabled="false"/>
       <mx:Button label="Login" click="ROLogin();" id="btnLogin"/>
       <mx:Button label="LogOut" click="ROLogout();" enabled="false" id="btnLogout"/>
       <mx:HBox>
        <mx:Label text="User:"/>
        <mx:TextInput id="txtUser" text=""/>
        <mx:Label text="Password:"/>
        <mx:TextInput id="txtPassword" text="" displayAsPassword="true"/>
        <mx:CheckBox id="authenticatedCB"
            label="Authenticated?"
            enabled="false"/>
    </mx:HBox>
     </mx:Panel>
</mx:Application>
```

See also

"Invoking AEM Forms using Remoting" on page 444

"Handling documents with Remoting" on page 446

"Including the AEM Forms Flex library file" on page 446

"Invoking a short-lived process by passing an unsecure document using Remoting" on page 449

"Authenticating client applications built with Flex" on page 451

Invoking custom component services using Remoting

You can invoke services located in a custom component using Remoting. For example, consider the Bank component that contains the Customer service. You can invoke operations that belong to the Customer service using a client application written in Flex. Before you can execute the quick start associated with this section, you have to create the Bank custom component. (See Creating Components That Use Custom Data Types.)

The Customer service exposes an operation named `createCustomer`. This discussion describes how to create a Flex client application that invokes the Customer service and creates a customer. This operation requires a complex object of type `com.adobe.livecycle.sample.customer.Customer` that represents the new customer. The following illustration shows the client application that invokes the Customer service and creates a new customer. The `createCustomer` operation returns a customer identifier value. The identifier value is displayed in the Customer Identifier text box.

The following table lists the controls that are part of this client application.

Control name	Description
txtFirst	Specifies the customer's first name.
txtLast	Specifies the customer's last name.
txtPhone	Specifies the customer's phone number.
txtStreet	Specifies the customer's street name.
txtState	Specifies the customer's state.
txtZIP	Specifies the customer's zip code.
txtCity	Specifies the customer's city.
txtCustId	Specifies the customer identifier value to which the new account belongs. This text box is populated by the return value of the Customer service's `createCustomer` operation.

Mapping AEM Forms complex data types

Some AEM Forms operations require complex data types as input values. These complex data types define run-time values used by the operation. For example, the Customer service's `createCustomer` operation requires a `Customer` instance that contains run-time values required by the service. Without the complex type, the Customer service throws an exception and does not perform the operation.

When invoking an AEM Forms service, create ActionScript objects that map to required AEM Forms complex types. For each complex data type that an operation requires, create a separate ActionScript object.

In the ActionScript class, use the `RemoteClass` metadata tag to map to the AEM Forms complex type. For example, when invoking the Customer service's `createCustomer` operation, create an ActionScript class that maps to `com.adobe.livecycle.sample.customer.Customer` data type.

The following ActionScript class named Customer shows how to map to the AEM Forms data type `com.adobe.livecycle.sample.customer.Customer`.

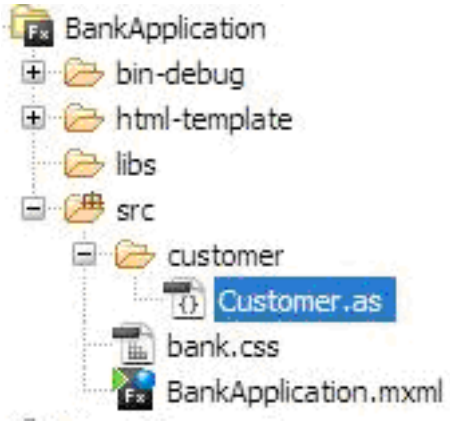
```
package customer

{
    [RemoteClass(alias="com.adobe.livecycle.sample.customer.Customer")]
    public class Customer
    {
            public var name:String;
            public var street:String;
            public var city:String;
            public var state:String;
            public var phone:String;
            public var zip:int;
        }
}
```

The fully qualified data type of the AEM Forms complex type is assigned to the alias tag.

The ActionScript class's fields match the fields that belong to the AEM Forms complex type. The six fields located in the Customer ActionScript class match the fields that belong to `com.adobe.livecycle.sample.customer.Customer`. (See Defining the Customer class.)

A good way to determine the field names that belong to a Forms complex type is to view a service's WSDL in a web browser. A WSDL specifies a service's complex types and the corresponding data members. The following WSDL is used for the Customer service: http://[yourServer]:[yourPort]/soap/services/CustomerService?wsdl.

The Customer ActionScript class belongs to a package named customer. It is recommended that you place all ActionScript classes that map to complex AEM Forms data types in their own package. Create a folder in the Flex project's src folder and place the ActionScript file in the folder, as shown in the following illustration.



Quick Start: Invoking the Customer custom service using Remoting

The following code example invokes the Customer service and creates a new customer. When you run this code example, ensure that you fill out all text boxes. Also, ensure that you create the Customer.as file that maps to `com.adobe.livecycle.sample.customer.Customer`.

Note: *Before you can execute this quick start, you have to create and deploy the Bank custom component. (See Creating Components That Use Custom Data Types.)*

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
backgroundColor="#B1ABAB">

<mx:Script>
          <![CDATA[

     import flash.net.FileReference;
     import flash.net.URLRequest;
     import flash.events.Event;
     import flash.events.DataEvent;
     import mx.messaging.ChannelSet;
     import mx.messaging.channels.AMFChannel;
     import mx.rpc.events.ResultEvent;
     import mx.collections.ArrayCollection;
     import mx.rpc.AsyncToken;
     import mx.managers.CursorManager;
     import mx.rpc.remoting.mxml.RemoteObject;


     // Custom class that corresponds to an input to the
     // AEM Forms encryption method
     import customer.Customer;

     // Classes used in file retrieval
     private var fileRef:FileReference = new FileReference();
     private var parentResourcePath:String = "/";
     private var serverPort:String = "hiro-xp:8080";
     private var now1:Date;
     private var fileName:String;

     // Prepares parameters for encryptPDFUsingPassword method call
     public function executeCreateCustomer():void
     {

       var cs:ChannelSet= new ChannelSet();
    cs.addChannel(new AMFChannel("remoting-amf", "http://" + serverPort +
"/remoting/messagebroker/amf"));

    customerService.setCredentials("administrator", "password");
    customerService.channelSet = cs;

    //Create a Customer object required to invoke the Customer service's
    //createCustomer operation
    var myCust:Customer = new Customer();

    //Get values from the user of the Flex application
    var fullName:String = txtFirst.text +" "+txtLast.text ;
    var Phone:String = txtPhone.text;
```

```
    var Street:String = txtStreet.text;
    var State:String = txtState.text;
    var Zip:int = parseInt(txtZIP.text);
    var City:String = txtCity.text;

    //Populate Customer fields
    myCust.name = fullName;
    myCust.phone = Phone;
    myCust.street= Street;
    myCust.state= State;
    myCust.zip = Zip;
    myCust.city = City;

    //Invoke the Customer service's createCustomer operation
    var params:Object = new Object();
       params["inCustomer"]=myCust;
    var token:AsyncToken;
       token = customerService.createCustomer(params);
       token.name = name;
     }

     private function handleResult(event:ResultEvent):void
     {
         // Retrieve the information returned from the service invocation
         var token:AsyncToken = event.token;
         var res:Object = event.result;
         var custId:String = res["CustomerId"] as String;

         //Assign to the custId to the text box
         txtCustId.text = custId;
     }


     private function resultHandler(event:ResultEvent):void
     {

     }
          ]]>
</mx:Script>
<mx:RemoteObject id="customerService" destination="CustomerService"
result="resultHandler(event);">
<mx:method name="createCustomer" result="handleResult(event)"/>
</mx:RemoteObject>


<mx:Style source="../bank.css"/>
    <mx:Grid>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="New Customer" fontSize="16" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
```

```
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="First Name:" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtFirst"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:Button label="Create Customer" id="btnCreateCustomer"
click="executeCreateCustomer()"/>
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="Last Name" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtLast"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="Phone" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtPhone"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="Street" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtStreet"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="State" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtState"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="ZIP" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtZIP"/>
```

```
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                    <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="City" fontSize="12" fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtCity"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                            <mx:GridRow width="100%" height="100%">
                        <mx:GridItem width="100%" height="100%">
                            <mx:Label text="Customer Identifier" fontSize="12"
fontWeight="bold"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                            <mx:TextInput styleName="textField" id="txtCustId"
editable="false"/>
                        </mx:GridItem>
                        <mx:GridItem width="100%" height="100%">
                        </mx:GridItem>
                    </mx:GridRow>
                </mx:Grid>
</mx:Application>
```

Style sheet

This quick start contains a style sheet named *bank.css*. The following code represents the style sheet that is used.

```
/* CSS file */
global
{
        backgroundGradientAlphas: 1.0, 1.0;
        backgroundGradientColors: #525152,#525152;
        borderColor: #424444;
        verticalAlign: middle;
        color: #FFFFFF;
        font-size:12;
        font-weight:normal;
}

ApplicationControlBar
{
        fillAlphas: 1.0, 1.0;
        fillColors: #393839, #393839;
}

.textField
{
        backgroundColor: #393839;
        background-disabled-color: #636563;
}
```

```
.button
{
        fillColors: #636563, #424242;
}

.dropdownMenu
{
        backgroundColor: #DDDDDD;
        fillColors: #636563, #393839;
        alternatingItemColors: #888888, #999999;
}

.questionLabel
{

}

ToolTip
{
      backgroundColor: black;
      backgroundAlpha: 1.0;
      cornerRadius: 0;
      color: white;
}

DateChooser
{
      cornerRadius: 0; /* pixels */
      headerColors: black, black;
      borderColor: black;
      themeColor: black;
      todayColor: red;
      todayStyleName: myTodayStyleName;
      headerStyleName: myHeaderStyleName;
      weekDayStyleName: myWeekDayStyleName;
      dropShadowEnabled: true;
}

.myTodayStyleName
{
      color: white;
}

.myWeekDayStyleName
{
      fontWeight: normal;
}

.myHeaderStyleName
{
      color: red;
      fontSize: 16;
      fontWeight: bold;
}
```

See also

"Invoking AEM Forms using Remoting" on page 444

"Handling documents with Remoting" on page 446

"Including the AEM Forms Flex library file" on page 446

"Invoking a short-lived process by passing an unsecure document using Remoting" on page 449

"Authenticating client applications built with Flex" on page 451

"Passing secure documents to invoke processes using Remoting" on page 456

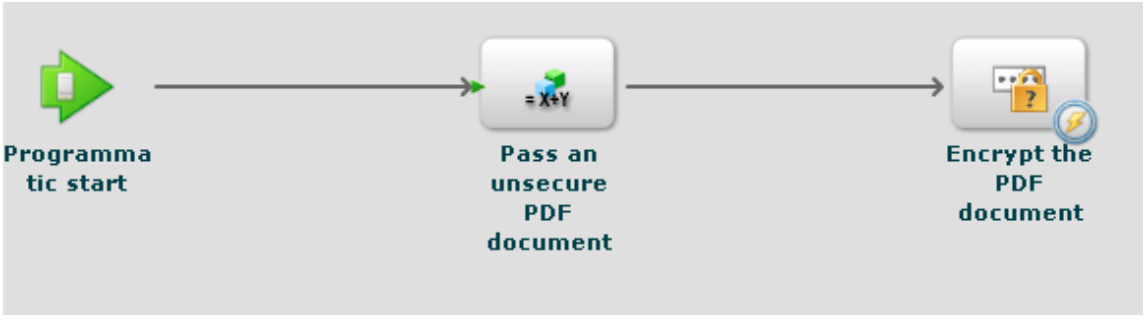
Creating Flash Builder applications that perform SSO authentication using HTTP tokens

You can create a client application using Flash Builder that performs single-sign on (SSO) authentication using HTTP tokens. Assume, for example, that you create a web-based application using Flash Builder. Next assume that the application contains different views, where each view invokes a different AEM Forms operation. Instead of authenticating a user for each Forms operation, you can create a login page that lets a user authenticate once. Once authenticated, a user is able to invoke multiple operations without having to authenticate again. For example, if a user has logged into Workspace (or another Forms application), the user would not need to authenticate again.

Although the client application contains required application logic to perform SSO authentication, AEM forms user Management performs the actual user authentication. To authenticate a user using HTTP tokens, the client application invokes the Authentication Manager service's `authenticateWithHTTPToken` operation. User Management is able to authenticate users using a HTTP token. For subsequent remoting or web service calls to AEM Forms, you do not have to pass credentials for authentication.

Note: *Before reading this section, it is recommended that you are familair with Invoking AEM Forms using Remoting. (See Invoking AEM Forms using Remoting.)*

The following AEM Forms short-lived process, named `MyApplication/EncryptDocument`, is invoked after a user is authenticated using SSO. (For information about this process such as its input and output values, see .)



Note: *This process is not based on an existing AEM Forms process. To follow along with the code examples that discuss how to invoke this process, create a process named* `MyApplication/EncryptDocument` *using workbench. (See Using Workbench.)*

The client application built using Flash Builder interacts with the User Manager's security servlet configured at `/um/login` and `/um/logout`. That is, the client application sends a request to the `/um/login` URL during startup to determine the status of the user. Then User Manager responds with the user status. The client application and the User Manager security servlet communicate using HTTP.

Request format

The security servlet requires the following input variables:

- `um_no_redirect` - This value must be `true`. This variable accompanies all the requests made to the User Manager security servlet. It also helps the security servlet differentiate the incoming request coming from a flex client or other web applications.
- `j_username` - This value is the login identifier value of the user as provided in the login form.
- `j_password` - This value is the corresponding password of the user as provided in the login form.

The `j_password` value is only required for credential requests. If the password value is not specified, then the security servlet checks to determine if the account you are using is already authenticated. If so, you are able to proceed; however, the security servlet does not authenticate you again.

Note: *For proper handling of i18n, ensure that these values are in POST form.*

Response format

The security servlet configured at `/um/login` responds by using the `URLVariables` format. In this format, the output of the content type is text/plain. The output contains name value pairs separated by an ampersand (&) character. The response contains the following variables:

- `authenticated` - The value is either `true` or `false`.
- `authstate` - This value can contain one of the following values:
  - `CREDENTIAL_CHALLENGE` - This state indicates that User Manager is not able to determine the user's identity through any means. In order for authentication to occur, the user's username and password is required.
  - `SPNEGO_CHALLENGE` - This state is treated the same as `CREDENTIAL_CHALLENGE`.
  - `COMPLETE` - This state indicates that User Manager is able to authenticate the user.
  - `FAILED` - This state indicates that User Manager was not able to authenticate the user. As a response to this state, the flex client can show an error message to the user.
  - `LOGGED_OUT` - This state indicates that the user has successfully logged out.
- `assertionid` - If the state was `COMPLETE` then it contains the user's `assertionId` value. A client application can obtain the `AuthResult` for the user.

Login process

When a client application starts, you can make a POST request to the `/um/login` security servlet. For example, `http://<your_serverhost>:<your_port>/um/login?um_no_redirect=true`. When the request reaches the User Manager security servlet, it performs the following steps:

1. It looks for a cookie named `lcAuthToken`. If the user has already logged in to another Forms application, then this cookie is present. If the cookie is found, then its content is validated.
2. If Header based SSO is enabled, then the servlet looks for configured headers to determine the user's identity.
3. If SPNEGO is enabled, then the servlet tries to initiate SPNEGO and tries to determine the user's identity.

If the security servlet locates a valid token that matches a user, the security servlet lets you proceed and responds with `authstate=COMPLETE`. Otherwise the security servlet responds with `authstate=CREDENTIAL_CHALLENGE`. The following list explains these values:

- `Case authstate=COMPLETE`: Indicates that the user is authenticated and the `assertionid` value contains the assertion identifier for the user. At this stage, the client application can connect to AEM Forms. The servlet configured for that URL can obtain the `AuthResult` for the user by invoking the `AuthenticationManager.authenticate(HttpRequestToken)` method. The `AuthResult` instance can create the user manager context and store it in the session.
- `Case authstate=CREDENTIAL_CHALLENGE`: Indicates that the security servlet requires the user's credentials. As a response, the client application can display the login screen to the user and send the obtained credential to the security servlet (for example, `http://<your_serverhost>:<your_port>/um/login?um_no_redirect=true&j_username=administrator&j_password=password)`. If authentication is successful, then the security servlet responds with `authstate=COMPLETE`.

If the authentication is still not successful, then the security servlet responds with `authstate=FAILED`. To respond to this value, the client application can display a message to obtain the credentials again.

Note: *While `authstate=CREDENTIAL_CHALLENGE`, it's recommended that client send the obtained credential to the security servlet in a POST form.*

Logout process

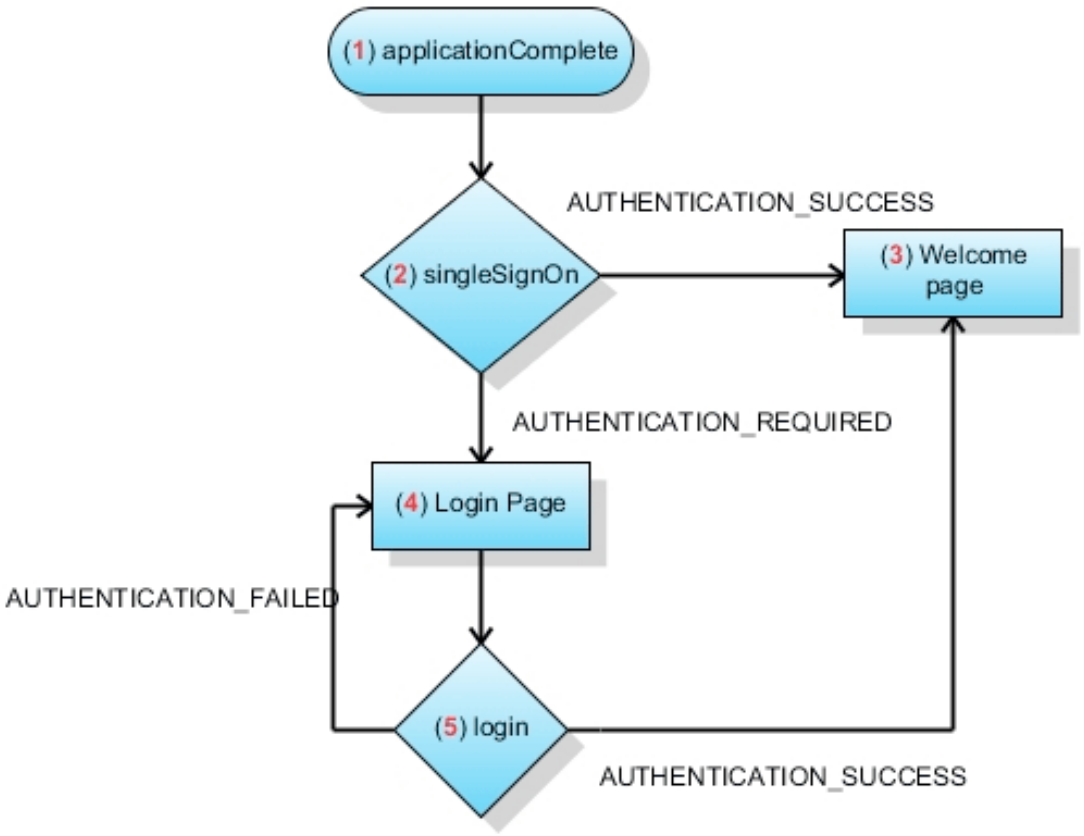
When a client application logs out, you can send a request to the following URL:

```
http://<your_serverhost>:<your_port>/um/logout?um_no_redirect=true
```

On receiving this request, the User Manager security servlet deletes the `lcAuthToken` cookie and responds with `authstate=LOGGED_OUT`. After the client application receives this value, the application can perform cleanup tasks.

Creating a client application that authenticates AEM forms users using SSO

To demonstrate how to create a client application that performs SSO authentication, an example client application is created. The following illustration shows the steps that the client application performs to authenticate a user using SSO.



The previous illustration describes the application flow that occurs when the client application starts.

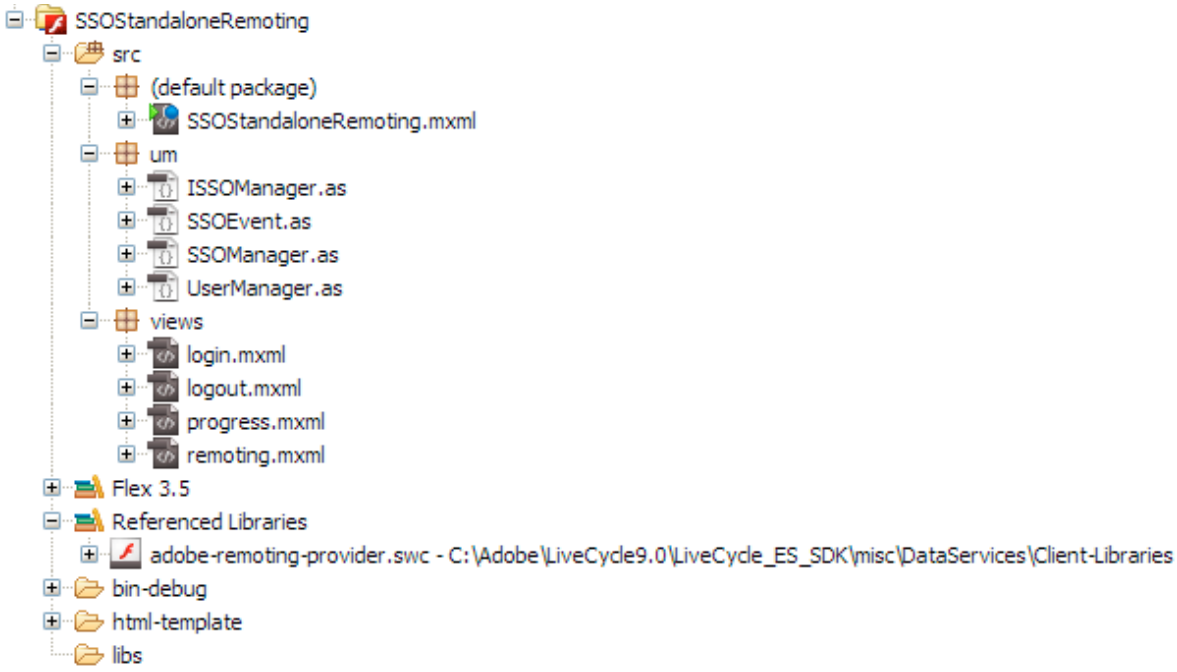
1. The client application triggers the `applicationComplete` event.
2. The call to `ISSOManager.singleSignOn` is made. The client application sends a request to the User Manager security servlet.
3. If the security servlet authenticates the user, then `ISSOManager` dispatches `SSOEvent.AUTHENTICATION_SUCCESS`. As a response, the client application shows the main page. In this example, the main page invokes the AEM Forms short-lived process named MyApplication/EncryptDocument.
4. If the security servlet cannot determine if the user is valid, then the application requests user credentials again. The `ISSOManager` class dispatches the `SSOEvent.AUTHENTICATION_REQUIRED` event. The client application displays the login page.
5. The credentials provided in the login page are sent to the `ISSOManager.login` method. If the authentication is successful, then it leads to step 3. Otherwise the `SSOEvent.AUTHENTICATION_FAILED` event is triggered. The client application displays the login page and an appropriate error message.

Creating the client application

The client application consists of the following files:

- `SSOStandalone.mxml`: The main MXML file that represents the client application. (See Creating the SSOStandalone.mxml file.)
- `um/ISSOManager.as`: Expose operations related to Single Sign On (SSO). (See Creating the ISSOManager.as file.)
- `um/SSOEvent.as`: The `SSOEvent` is dispatched for SSO related events. (See Creating the SSOEvent.as file.)
- `um/SSOManager.as`: Manages the SSO related operations and dispatches appropriate events. (See Creating the SSOManager.as file.)
- `um/UserManager.as`: Contains application logic that invokes the Authentication Manager service using its WSDL. (See Creating the UserManager.as file.)
- `views/login.mxml`: Represents the login screen. (See Creating the login.mxml file.)
- `views/logout.mxml`: Represents the logout screen. (See Creating the logout.mxml file.)
- `views/progress.mxml`: Represents a progress view. (See Creating the progress.mxml file.)
- `views/remoting.mxml`: Represents the view that invokes AEM Forms short-lived process named MyApplication/EncryptDocument using remoting. (See Creating the remoting.mxml file.)

The following illustration provides a visual representation of the client application.



Note: *Notice that there are two packages named um and views. When creating the client application, ensure that you place the files in their proper packages. Also, ensure that you add the adobe-remoting-provider.swc file to your project's class path. (See Including the AEM Forms Flex library file.)*

Creating the SSOStandalone.mxml file

The following code represents the SSOStandalone.mxml file.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                applicationComplete="initApp()"
                height="400" width="550"
                xmlns:v="views.*"
                backgroundColor="#EDE8F0" viewSourceURL="srcview/index.html">
    <mx:Script>
        <![CDATA[
            import mx.utils.URLUtil;
            import um.SSOEvent;
            import mx.core.UIComponent;
            import um.SSOManager;
            import mx.rpc.events.ResultEvent;
            import mx.utils.ObjectUtil;
            import mx.controls.Alert;

            [Bindable]
            private var _serverURL:String;

            private var _ssoManager:SSOManager;

            private var _progress:UIComponent;

            private var _loginPage:UIComponent;

            private function initApp():void{
                _serverURL = determineServerUrl();
                _ssoManager = new SSOManager(_serverURL);

                _ssoManager.addEventListener(SSOEvent.AUTHENTICATION_FAILED,loginHandler);
                _ssoManager.addEventListener(SSOEvent.AUTHENTICATION_SUCCESS,loginHandler);
                _ssoManager.addEventListener(SSOEvent.AUTHENTICATION_REQUIRED,loginHandler);
                _ssoManager.addEventListener(SSOEvent.LOGOUT_COMPLETE,loginHandler);
                _ssoManager.addEventListener(SSOEvent.AUTHENTICATION_FAULT,loginHandler);

                trace("[Main] Add the required event handlers for authentication");
                _ssoManager.singleSignOn();

                showBusy();
            }

            private function determineServerUrl():String
            {
                var s:String ;
                var appUrl:String = Application.application.url;
                var givenUrl:String  = ExternalInterface.call("serverUrl.toString");
                trace("[Main] Application url ["+appUrl+"] Given url ["+givenUrl+"]");
                if(appUrl != null && appUrl.search("^http") != -1){
                    s = appUrl;
                }
                if(s == null){
                    s = givenUrl;
                }
                if(s== null){
                    s = "http://hiro-xp:8080/";
                }
```

```
                s = URLUtil.getFullURL(s,"/");
                trace("[Main] Would be using ["+s+"] as serverUrl");
                return s;
            }

            private function loginHandler(event:SSOEvent):void
            {
                trace("[Main] Handling event "+event.type);
                switch(event.type)
                {
                    case SSOEvent.AUTHENTICATION_FAILED:
                        viewContent.selectedChild = login;
                        login.showLoginFailed();
                        break;
                    case SSOEvent.AUTHENTICATION_SUCCESS:
                        viewContent.selectedChild = remoting;
                        break;
                    case SSOEvent.AUTHENTICATION_REQUIRED:
                        viewContent.selectedChild = login;
                        break;
                    case SSOEvent.LOGOUT_COMPLETE:
                        viewContent.selectedChild = logout;
                        break;
                    case SSOEvent.AUTHENTICATION_FAULT:
                        Alert.show("Error doing authentication. Root error
["+event.rootEvent+"]","Authentication Fault",Alert.OK);
                }
            }

            public function get ssoManager():SSOManager
            {
                return _ssoManager;
            }

            public function showBusy():void
            {
                viewContent.selectedChild = progress;
            }

            public function get serverUrl():String
            {
                return _serverURL;
            }

        ]]>
    </mx:Script>
    <mx:ViewStack x="0" y="0" id="viewContent" >
        <v:login id="login" />
        <v:remoting id="remoting"  />
        <v:progress id="progress" />
        <v:logout id="logout"/>
    </mx:ViewStack>
</mx:Application>
```

Creating the ISSOManager.as file

The following code represents the ISSOManager.as file.

```
package um
{
    import flash.events.IEventDispatcher;

    /**
     * The <code>ISSOManager</code> expose operations related to Single Sign On (SSO) in AEM Forms
     * environment. The application should register appropriate <code>SSOEvent</code> handlers
prior
     * to calling any of the following operations
     */
    public interface ISSOManager extends IEventDispatcher
    {
        /**
         * Tries to validate whether the user has an already existing session or not (SSO
Scenarios). The application
         * may call this method during the initialization. In general this call would lead to one
of the
         * following events getting dispatched
         * <ul>
         *    <li>SSOEvent.AUTHENTICATION_SUCCESS - If a SSO session was found and valid
         *    <li>SSOEvent.AUTHENTICATION_REQUIRED - No SSO session was found and as such
authentication is required in
         *        the form of username and password.
         *    <li>SSOEvent.AUTHENTICATION_FAULT - Some error has occured while connecting to the
server
         * </ul>
         */
        function singleSignOn():void;

        /**
         * Authenticates the user using username and password. It may lead to one of the following
events
         * <ul>
         *   <li>SSOEvent.AUTHENTICATION_SUCCESS - The authentication is successful and a session
is established
         *    <li>SSOEvent.AUTHENTICATION_FAILED - Authentication has failed
         * </ul>
         */
        function login(username:String, password:String):void;

        /**
         * Terminates the current session and logs out the user.
         */
        function logout():void;

        /**
         * Get the assertionId for the logged in user
         */
        function get assertionId():String;
    }
}
```

Creating the SSOEvent.as file

The following code represents the SSOEvent.as file.

```
package um
{
    import flash.events.Event;

    /**
     * The <code>SSOEvent</code> is dispatched for SSO related events
     */
    public class SSOEvent extends Event
    {
        /**
         * This type of event would be dispatched when the Authentication process is successful.
Authentication
         * might have been done with SSO or username and password. As a response to this event the
application
         * can show the welcome page to the user
         * The application may want to perform specific check for permission/role so as to verify
the user is allowed.
         * So as a response to this event the application would do those checks and then only show
the welcome page
         */
        public static const AUTHENTICATION_SUCCESS:String = "authenticationSuccess";

        /**
         * This type of event would be dispatched when authentication fails using the username,
password.
         * As a response to this type of event an application can show an error message to the user.
         * This event would only happen when authentication is done using username and password
and NOT in
         * SSO case.
         */
        public static const AUTHENTICATION_FAILED:String = "authenticationFailed";

        /**
         * This type of event would be dispatched when authentication using SSO is not achieved.
And due to
         * that we require the user's username and password for authentication. As a response to
this event
         * the application can show the login page to the user.
         */
        public static const AUTHENTICATION_REQUIRED:String = "authenticationRequired";

        /**
         * This type of event would be dispatched when logout is complete. As a response to this
event the
         * application may show a logout page informing the user that he has been logged out. Or
the application
         * can take the user back to login page
         */
        public static const LOGOUT_COMPLETE:String = "logoutComplete";

        /**
         * This type of event would be dispatched when ever there is a problem in doing
Authentication. The root cause
         * can be obtained from the <code>rootEvent</code>.
```

```
         */
        public static const AUTHENTICATION_FAULT:String = "authenticationFault";

        private var _rootEvent:Event;

        public function SSOEvent(type:String, rootEvent:Event=null)
        {
            super(type,true,false);
            _rootEvent = rootEvent;
        }

        /**
         * The root event. If current event type is <code>AUTHENTICATION_FAULT</code> then it would
be an
         * <code>IOErrorEvent</code> in other cases it would be complete event. Its basic use is
to extract the root
         * cause in case of an authentication fault.
         */
        public function get rootEvent():Event
        {
            return _rootEvent;
        }
    }
}
```

Creating the SSOManager.as file

The following code represents the SSOManager.as file.

```
package um
{
    import flash.events.Event;
    import flash.events.EventDispatcher;
    import flash.events.IOErrorEvent;
    import flash.external.ExternalInterface;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.net.URLVariables;

    import mx.utils.ObjectUtil;

    /**
     * Manages the SSO related operations and dispatches appropriate events. It would connect to
the UM Filter/Servlet
     * at <code>um/login</code> The UM response would be of form of url encoded variables. It would
look for
     * <code>authstate</code> value in the response and depending on that it would proceed.
     *
     * <p>If there is an IO_Error while initial attempt to UM then it would assume it as a 401
response. And it would
     * be assumed that SPNEGO based authenticatin is not working and therefore user would be shown
a login page.
     */
    public class SSOManager extends EventDispatcher implements ISSOManager
    {
        private static const SSO_URL:String = "um/login";
```

```
        private static const SSO_LOGOUT_URL:String = "um/logout";
        private static const AUTH_COOKIE_NAME:String = "lcAuthToken";

        private var _serverUrl:String;
        private var _assertionId:String;

        /**
         * Constructs an SSOManager with the given server url.
         *
         * @param serverUrl - The uri of the server to connect to. it must be without any context
path e.g
         * http://localhost:8080/. The SSOManager would directly append the path of UM exposed SSO
url to it
         * for its operations
         */
        public function SSOManager(serverUrl:String)
        {
            _serverUrl = serverUrl;
        }

        public function singleSignOn():void
        {
            sendRequest(SSO_URL,true);
        }

        public function login(username:String, password:String):void
        {
            sendRequest(SSO_URL,false,
                function(request:URLRequest,vars:URLVariables):void
                {
                    vars.j_username = username;
                    vars.j_password = password;
                }
            );
        }

        public function logout():void
        {
            sendRequest(SSO_LOGOUT_URL);
        }

        public function get assertionId():String
        {
            return _assertionId;
        }



        /**
         * Connects to the UM security service.
         */
        private function sendRequest(relativeUrl:String,authenticationRequest:Boolean=false,
requestProcessor:Function=null):void
        {
            var loader:URLLoader = new URLLoader();
            loader.dataFormat = URLLoaderDataFormat.VARIABLES;
            var request:URLRequest = new URLRequest(_serverUrl + relativeUrl);
```

```
    trace("[SSOmanager] Contacting ["+request.url+"]");
    var vars:URLVariables = new URLVariables();
    vars.um_no_redirect = "true";
    request.data = vars;
    if(requestProcessor != null){
        requestProcessor(request,vars);
    }

    loader.addEventListener(Event.COMPLETE,authHandler);
    //if its an authentication request then only treat io error as a possible 401
    //for others treat them as faults
    if(authenticationRequest){
        loader.addEventListener(IOErrorEvent.IO_ERROR,httpAuthenticationHandler);
    }else{
        loader.addEventListener(IOErrorEvent.IO_ERROR,authFaultHandler);
    }
    trace("[SSOmanager] Sending request "+ ObjectUtil.toString(request));
    loader.load(request);
}

private function authHandler(event:Event):void
{
    var loader:URLLoader = URLLoader(event.target);
    var response:URLVariables = URLVariables(loader.data);
    trace("[SSOmanager] Processing response ["+ObjectUtil.toString(response)+"]");
    handleAuthResult(response["authstate"],response);
}

/**
 * Handles the IOErrorEvent. Flash would dispatch IOEvent in response to HTTP 401.
 * There is no way to distinguish it from the genuine IOError.
 */
private function httpAuthenticationHandler(event:IOErrorEvent):void
{
    trace("[SSOmanager] Processing IOErrorEvent ["+ObjectUtil.toString(event)+"]");
    handleAuthResult("CREDENTIAL_CHALLENGE");

}

/**
 * Dispatches appropriate <code>SSOEvent</code> on the basis of the <code>authstate</code>
 * value of the response.
 * The response is url encoded in for of
 * <pre>
 * authenticated=false&authstate=SPNEGO_CHALLENGE
 * </pre>
 * Depending on <code>authstate</code> the SSOEvent is dispatched
 */
private function handleAuthResult(authState:String,response:URLVariables = null):void
{
    trace("[SSOmanager] processing state "+authState);
    switch(authState)
    {
        case "FAILED"  :
            dispatchEvent(new SSOEvent(SSOEvent.AUTHENTICATION_FAILED));
            break;
        case "COMPLETE" :
```

```
                    _assertionId = response ? response["assertionid"] : null;
                    dispatchEvent(new SSOEvent(SSOEvent.AUTHENTICATION_SUCCESS));
                    break;
                case "CREDENTIAL_CHALLENGE" :
                    dispatchEvent(new SSOEvent(SSOEvent.AUTHENTICATION_REQUIRED));
                    break;
                case "LOGGED_OUT" :
                    dispatchEvent(new SSOEvent(SSOEvent.LOGOUT_COMPLETE));
                    break;
                default:
                    dispatchEvent(new SSOEvent(SSOEvent.AUTHENTICATION_REQUIRED));
                    break;
            }
        }

        private function authFaultHandler(event:Event):void
        {
            dispatchEvent(new SSOEvent(SSOEvent.AUTHENTICATION_FAULT,event));
        }

    }
}
```

Creating the UserManager.as file

The following code represents the UserManager.as file.

```
package um
{
    import flash.events.Event;
    import mx.rpc.soap.WebService;
    import mx.rpc.soap.Operation;
    import mx.rpc.IResponder;
    import mx.rpc.events.FaultEvent;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.soap.LoadEvent;

    public class UserManager
    {
        private var _ssoManager:ISSOManager;
        private var _serverUrl:String;

        public function UserManager(ssoManager:ISSOManager,serverUrl:String)
        {
            _serverUrl = serverUrl;
            _ssoManager = ssoManager;
        }

        public function retrieveAssertion(responder:IResponder):String
        {
            var assertionId:String = _ssoManager.assertionId;
            if(!assertionId)
            {
                trace("[UserManager] AssertionId not found");
                return null;
            }
```

```
            var ws:WebService = new WebService();
            var wsdl:String =
_serverUrl+'soap/services/AuthenticationManagerService?wsdl&lc_version=8.2.1';
            ws.loadWSDL(wsdl);
            ws.addEventListener(LoadEvent.LOAD,
                function(event:Event):void
                {
                    trace("[UserManager] WSDL loaded");
                    var authenticate:Operation = ws.authenticateWithHttpToken as Operation;
                    authenticate.resultFormat = "e4x";
                    authenticate.addEventListener(ResultEvent.RESULT,
                        function(event:Event):void
                        {
                            responder.result(event);
                        }
                    );
                    authenticate.send({assertionId:assertionId});
                }
            );

            ws.addEventListener(FaultEvent.FAULT,
                function(event:Event):void
                {
                    responder.fault(event);
                }
            );
            return null;
        }
    }
}
```

Creating the login.mxml file

The following code represents the login.mxml file.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml" width="500" height="400">
    <mx:Script>
        <![CDATA[
            import mx.core.Application;
            public function showLoginFailed():void
            {
                loginMessage.text = "Username or Password incorrect";
            }

            private function doLogin():void
            {
                Application.application.ssoManager.login(j_username.text,j_password.text);
                Application.application.showBusy();
            }

        ]]>
    </mx:Script>

    <mx:VBox height="113" width="244" x="128" y="144" horizontalAlign="center"
verticalGap="10">
        <mx:HBox width="100%">
            <mx:HBox width="100%" verticalAlign="middle" horizontalAlign="center" height="32">
                <mx:Label text="Username" fontWeight="bold"/>
                <mx:TextInput id="j_username"/>
            </mx:HBox>
        </mx:HBox>
        <mx:HBox width="100%" height="33" horizontalAlign="center" horizontalGap="10"
verticalAlign="middle">
            <mx:Label text="Password" fontWeight="bold"/>
            <mx:TextInput displayAsPassword="true" id="j_password"/>
        </mx:HBox>
        <mx:Button label="Login" click="doLogin()"/>
    </mx:VBox>
    <mx:Text x="128" y="122" id="loginMessage" width="230" height="14"/>
    <mx:Label x="154" y="65" text="AEM Forms SSO Demo" fontFamily="Georgia" fontSize="20"
color="#0A0A0A"/>
</mx:Canvas>
```

Creating the logout.mxml file

The following code represents the logout.mxml file.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml" width="500" height="400">
    <mx:Label x="97" y="188" text="You have successfully logged out from the application"/>

</mx:Canvas>
```

Creating the progress.mxml file

The following code represents the progress.mxml file.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Label x="151" y="141" text="Wait...."/>
    <mx:SWFLoader source="LoadingCircle.swf" width="50" height="50" horizontalCenter="0"
verticalCenter="0"/>
</mx:Canvas>
```

Creating the remoting.mxml file

The following code represents the remoting.mxml file that invokes the `MyApplication/EncryptDocument` process. Because a document is passed to the process, application logic responsible for passing a secure document to AEM Forms is located in this file. (See Passing secure documents to invoke processes using Remoting.)

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml" width="664" height="400"
creationComplete="initializeChannelSet()" xmlns:views="views.*">
    <mx:Script>

        <![CDATA[

            import mx.rpc.livecycle.DocumentReference;
            import flash.net.FileReference;
            import flash.net.URLRequest;
            import flash.events.Event;
            import flash.events.DataEvent;
            import mx.messaging.ChannelSet;
            import mx.messaging.channels.AMFChannel;
            import mx.rpc.events.ResultEvent;
            import mx.collections.ArrayCollection;
            import mx.rpc.AsyncToken;
            import um.UserManager;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.core.Application;
            import mx.rpc.Responder;
            import mx.utils.ObjectUtil;

            // Classes used in file retrieval
            private var fileRef:FileReference = new FileReference();
            private var docRef:DocumentReference = new DocumentReference();
            private var parentResourcePath:String = "/";
            //private var serverPort:String = "[server]:[port]";
            private var serverPort:String = "[server]:[port]";
            private var now1:Date;
            private var userManager:UserManager;

            // Define a ChannelSet object.
            public var cs:ChannelSet;

            // Holds information returned from AEM Forms
            [Bindable]
            public var progressList:ArrayCollection = new ArrayCollection();


            // Set up channel set to invoke AEM Forms.
            // This must be done before calling any service or process, but only
            // once for the entire application.
```

```
            private function initializeChannelSet():void {
                cs = new ChannelSet();
                cs.addChannel(new AMFChannel("remoting-amf", "http://" + serverPort +
"/remoting/messagebroker/amf"));
                EncryptDocument.channelSet = cs;

            //Get the user that is authenticated
            userManager = new
UserManager(Application.application.ssoManager,Application.application.serverUrl);
            userManager.retrieveAssertion(
                    new mx.rpc.Responder(
                        function(event:ResultEvent):void
                        {
                            var name:String =
XML(event.currentTarget.lastResult)..*::authenticatedUser.*::userid.text();
                            username.text = "Welcome "+name;
                        },
                        function(event:FaultEvent):void
                        {
                            mx.controls.Alert.show(event.fault.faultString,'Error')
                        }
                    )
                );

            }

            // Call this method to upload the file.
            // This creates a file picker and lets the user select a PDF file to pass to the
EncryptDocument process.
            private function uploadFile():void {
                fileRef.addEventListener(Event.SELECT, selectHandler);
                fileRef.addEventListener(DataEvent.UPLOAD_COMPLETE_DATA,completeHandler);
                fileRef.browse();
            }

            // Gets called for selected file. Does the actual upload via the file upload servlet.
            private function selectHandler(event:Event):void {
                var authTokenService:RemoteObject = new
RemoteObject("LC.FileUploadAuthenticator");
                authTokenService.addEventListener("result", authTokenReceived);
                authTokenService.channelSet = cs;
                authTokenService.getFileUploadToken();
            }

            private function authTokenReceived(event:ResultEvent):void
            {
                var token:String = event.result as String;
                var request:URLRequest =
DocumentReference.constructRequestForUpload("http://hiro-xp:8080", token);

                try
                {
                    fileRef.upload(request);
                }
                catch (error:Error)
                {
                    trace("Unable to upload file.");
```

```
    }
}

// Called once the file is completely uploaded.
private function completeHandler(event:DataEvent):void {

    // Set the docRefs url and referenceType parameters
    docRef.url = event.data as String;
    docRef.referenceType=DocumentReference.REF_TYPE_URL;
    executeInvokeProcess();
}

//This method invokes the EncryptDocument process
public function executeInvokeProcess():void {
    //Create an Object to store the input value for the EncryptDocument process
    now1 = new Date();

    var params:Object = new Object();
    params["inDoc"]=docRef;

    // Invoke the EncryptDocument process
    var token:AsyncToken;
    token = EncryptDocument.invoke(params);
    token.name = name;
}


// This method handles a successful conversion invocation
public function handleResult(event:ResultEvent):void
{

    //Retrieve information returned from the service invocation
    var token:AsyncToken = event.token;
    var res:Object = event.result;
    var dr:DocumentReference = res["outDoc"] as DocumentReference;
    var now2:Date = new Date();

    // These fields map to columns in the DataGrid
    var progObject:Object = new Object();
    progObject.filename = token.name;
    progObject.timing = (now2.time - now1.time).toString();
    progObject.state = "Success";
    progObject.link = "<a href='" + dr.url + "'> open </a>";
    progressList.addItem(progObject);
}


private function resultHandler(event:ResultEvent):void {
// Do anything else here.

}

private function logout():void
{
    Application.application.ssoManager.logout();
    Application.application.showBusy();
}
```

```
        ]]>

    </mx:Script>

    <mx:RemoteObject id="EncryptDocument" destination="MyApplication/EncryptDocument"
result="resultHandler(event);">
            <mx:method name="invoke" result="handleResult(event)"/>
    </mx:RemoteObject>


    <!--//This consists of what is displayed on the webpage-->
    <mx:Panel id="lcPanel" title="EncryptDocument  (Deprecated for AEM forms) AEM Forms Remoting
Example"
          height="25%" width="25%" paddingTop="10" paddingLeft="10" paddingRight="10"
          paddingBottom="10">
        <mx:Label width="100%" color="blue"
                  id="username"/>

        <mx:DataGrid x="10" y="0" width="500" id="idProgress" editable="false"
                         dataProvider="{progressList}" height="231" selectable="false" >
        <mx:columns>
                <mx:DataGridColumn headerText="Filename" width="200" dataField="filename"
editable="false"/>
                <mx:DataGridColumn headerText="State" width="75" dataField="state"
editable="false"/>
                <mx:DataGridColumn headerText="Timing" width="75" dataField="timing"
editable="false"/>
                <mx:DataGridColumn headerText="Click to Open" dataField="link" editable="false"
>
                <mx:itemRenderer>

                        <mx:Component>
                        <mx:Text x="0" y="0" width="100%" htmlText="{data.link}"/>
                        </mx:Component>
                    </mx:itemRenderer>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
    <mx:Button label="Select File" click="uploadFile()" />
    <mx:Button label="Logout"  click="logout()" />
    </mx:Panel>
</mx:Canvas>
```

Additional Information

The following sections provide additional details that describe the communication between the client application and the User Manager security servlet.

A new authentication occurs

In this situation, the user attempts to log in from a client application to AEM Forms for the first time. (no previous session involving the user exists.) In the `applicationComplete` event, the `SSOManager.singleSignOn` method is invoked that sends a request to the User Manager.

```
GET /um/login?um%5Fno%5Fredirect=true HTTP/1.1
```

The User Manager security servlet responds with the following value:

```
HTTP/1.1 200 OK

authenticated=false&authstate=CREDENTIAL_CHALLENGE
```

As response to this value, a `SSOEvent.AUTHENTICATION_REQUIRED` value is dispatched. As a result, the client application displays a login screen to the user. The credentials are submitted back to the User Manager security servlet.

```
GET /um/login?um%5Fno%5Fredirect=true&j%5Fusername=administrator&j%5Fpassword=password
HTTP/1.1
```

The User Manager security servlet responds with the following value:

```
HTTP/1.1 200 OK
Set-Cookie: lcAuthToken=53630BC8-F6D4-F588-5D5B-4668EFB2EC7A; Path=/
authenticated=true&authstate=COMPLETE&assertionid=53630BC8-F6D4-F588-5D5B-4668EFB2EC7A
```

As a result, `authstate=COMPLETE the SSOEvent.AUTHENTICATION_SUCCESS` is dispatched. The client application can perform further processing if necessary. For example, a log that tracks the date and time that the user was authenticated can be created.

The user is already authenticated

In this situation, the user has already logged in to AEM Forms and then navigates to the client application. The client application connects to the User Manager security servlet during startup.

```
GET /um/login?um%5Fno%5Fredirect=true HTTP/1.1
Cookie: JSESSIONID=A4E0BCC2DD4BCCD3167C45FA350BD72A; lcAuthToken=53630BC8-F6D4-F588-5D5B-
4668EFB2EC7A
```

Because the user is already authenticated, the User Manager cookie is present and is sent to the User Manager security servlet. The servlet then gets the `assertionId` value and verifies whether it is valid. If it is valid, then `authstate=COMPLETE` is returned. Otherwise `authstate=CREDENTIAL_CHALLENGE` is returned. The following is a typical response:

```
HTTP/1.1 200 OK
      authenticated=true&authstate=COMPLETE&assertionid=53630BC8-F6D4-F588-5D5B-4668EFB2EC7A
```

In this situation, the user is not shown a login screen and instead directly taken to a Welcome Screen.

Invoking AEM Forms using the Java API

AEM Forms can be invoked by using the AEM Forms Java API. When using the AEM Forms Java API, you can use either the Invocation API or Java client libraries. Java client libraries are available for services such as the Rights Management service. These strongly typed APIs let you develop Java applications that invoke AEM Forms.

The Invocation API are classes that are located in the `com.adobe.idp.dsc` package. Using these classes, you can send an invocation request directly to a service and handle an invocation response that is returned. Use the Invocation API to invoke short-lived or long-lived processes that were created by using Workbench.

The recommended way to programmatically invoke a service is to use a Java client library that corresponds to the service as opposed to the Invocation API. For example, to invoke the Encryption service, use the Encryption service client library. To perform an Encryption service operation, invoke a method that belongs to the Encryption service client object. You can encrypt a PDF document with a password by invoking the `EncryptionServiceClient` object's `encryptPDFUsingPassword` method.

The Java API supports the following features:

- RMI transport protocol for remote invocation
- VM transport for local invocation
- SOAP for remote invocation
- Different authentication, such as user name and password
- Synchronous and asynchronous invocation requests

Adobe Developer website

The Adobe Developer website contains the following articles that discuss invoking AEM Forms services using the Java API:

Using Java servlets to invoke AEM Forms processes

Invoking the AEM Forms Distiller API from Java

See also

"Including AEM Forms Java library files" on page 491

"Invoking Human-Centric Long-Lived Processes" on page 560

"Invoking AEM Forms using Web Services" on page 514

"Setting connection properties" on page 500

"Passing data to AEM Forms services using the Java API" on page 505

"Invoking a service using a Java client library" on page 511

"Invoking a short-lived process using the Invocation API" on page 512

"Creating a Java web application that invokes a human-centric long-lived process" on page 561

Including AEM Forms Java library files

To programmatically invoke a AEM Forms service by using the Java API, include required library files (JAR files) in your Java project's classpath. The JAR files that you include in your client application's classpath depend on several factors:

- The AEM Forms service to invoke. A client application can invoke one or more services.
- The mode in which you want to invoke a AEM Forms service. You can use the EJB or SOAP mode. (See "Setting connection properties" on page 500.)
- The J2EE application server on which AEM Forms is deployed.

Service-specific JAR files

The following table lists the JAR files that are required to invoke AEM Forms services.

File	Description	Location
adobe-livecycle-client.jar	Must always be included in a Java client application's class path.	<*install directory*>/sdk/client-libs/common
adobe-usermanager-client.jar	Must always be included in a Java client application's class path.	<*install directory*>/sdk/client-libs/common
adobe-utilities.jar	Must always be included in a Java client application's class path.	<*install directory*>/sdk//client-libs/<app server>
adobe-applicationmanager-client-sdk.jar	Required to invoke the Application Manager service.	<*install directory*>/sdk/client-libs/common
adobe-assembler-client.jar	Required to invoke the Assembler service.	<*install directory*>/sdk/client-libs/common
adobe-backup-restore-client-sdk.jar	Required to invoke the Backup and Restore service API.	<*install directory*>/sdk/client-libs/common
adobe-barcodedforms-client.jar	Required to invoke the barcoded forms service.	<*install directory*>/sdk/client-libs/common
adobe-convertpdf-client.jar	Required to invoke the Convert PDF service.	<*install directory*>/sdk/client-libs/common
adobe-distiller-client.jar	Required to invoke the Distiller service.	<*install directory*>/sdk/client-libs/common
adobe-docconverter-client.jar	Required to invoke the DocConverter service.	<*install directory*>/sdk/client-libs/common
adobe-contentservices-client.jar	Required to invoke the Document Management service.	<*install directory*>/sdk/client-libs/common
adobe-encryption-client.jar	Required to invoke the Encryption service.	<*install directory*>/sdk/client-libs/common
adobe-forms-client.jar	Required to invoke the Forms service.	<*install directory*>/sdk/client-libs/common
adobe-formdataintegration-client.jar	Required to invoke the Form Data Integration service.	<*install directory*>/sdk/client-libs/common
adobe-generatepdf-client.jar	Required to invoke the Generate PDF service.	<*install directory*>/sdk/client-libs/common
adobe-generate3dpdf-client.jar	Required to invoke the Generate 3D PDF service.	<*install directory*>/sdk/client-libs/common
adobe-jobmanager-client-sdk.jar	Required to invoke the Job Manager service.	<*install directory*>/sdk/client-libs/common
adobe-output-client.jar	Required to invoke the Output service.	<*install directory*>/sdk/client-libs/common
adobe-pdfutility-client.jar	Required to invoke the PDF Utilities or XMP Utilities service.	<*install directory*>/sdk/client-libs/common
adobe-reader-extensions-client.jar	Required to invoke the Acrobat Reader DC extensions service.	<*install directory*>/sdk/client-libs/common
adobe-repository-client.jar commons-codec-1.3.jar	Required to invoke the Repository service.	<*install directory*>/sdk/client-libs/common <*install directory*>/sdk/client-libs\thirdparty

File	Description	Location
• adobe-rightsmanagement-client.jar • namespace.jar • jaxb-api.jar • jaxb-impl.jar • jaxb-libs.jar • jaxb-xjc.jar • relaxngDatatype.jar • xsdlib.jar	Required to invoke the Rights Management service. If AEM Forms is deployed on JBoss, include all these files.	<*install directory*>/sdk/client-libs/common JBoss-specific lib directory
adobe-signatures-client.jar	Required to invoke the Signature service.	<*install directory*>/sdk/client-libs/common
adobe-taskmanager-client-sdk.jar	Required to invoke the Task Manager service.	<*install directory*>/sdk/client-libs/common
adobe-truststore-client.jar	Required to invoke the Trust Store service.	<*install directory*>/sdk/client-libs/common

Connection mode and J2EE application JAHawR files

The following table lists the JAR files that are dependant upon the connection mode and the J2EE application server on which AEM Forms is deployed.

File	Description	Location
• activation.jar • axis.jar • commons-codec-1.3.jar • commons-collections-3.1.jar • commons-discovery.jar • commons-logging.jar • dom3-xml-apis-2.5.0.jar • jaxen-1.1-beta-9.jar • jaxrpc.jar • log4j.jar • mail.jar • saaj.jar • wsdl4j.jar • xalan.jar • xbean.jar • xercesImpl.jar	if AEM Forms is invoked using the SOAP mode, include these JAR files.	*<install directory>*/sdk/client-libs/thirdparty
jbossall-client.jar	if AEM Forms is deployed on JBoss Application Server, include this JAR file. The size of jbossall-client.jar is significantly lesser in JBoss 5.1.0. This is because JBoss 4.2.1 jbossall-client.jar packages all the classes required to connect to JBoss 4.2.1 server. However, JBoss 5.1.0 jbossall-client.jar does not package any classes. This jar file contains a classpath reference to various client jar files used by the JBoss client applications. When writing AEM Forms API-based programs where AEM Forms is running on JBoss 5.2.1, you must place jbossall-client.jar and all the referenced jars in a single folder and then include jbossall-client.jar in the classpath. Required classes will not be found by the classloader if jbossall-client.jar and the referenced jars are not co-located.	JBoss client lib directory If you deploy your client application on the same J2EE application server, you do not need to include this file.

File	Description	Location
• jacorb.jar • jnp-client.jar	if AEM Forms is deployed on JBoss Application Server, include this JAR file.	JBoss client lib directory If you deploy your client application on the same J2EE application server, you do not need to include this file.
wlclient.jar	if AEM Forms is deployed on BEA WebLogic Server®, then include this JAR file.	WebLogic-specific lib directory If you deploy your client application on the same J2EE application server, you do not need to include this file.
• com.ibm.ws.admin.client_6.1.0.jar • com.ibm.ws.webservices.thinclient_6.1.0.jar	• if AEM Forms is deployed on WebSphere Application Server, include these JAR files. • (com.ibm.ws.webservices.thinclient_6.1.0.jar is required for web service invocation).	WebSphere-specific lib directory (*[WAS_HOME]*/runtimes) If you deploy your client application on the same J2EE application server, you do not have to include these files.

Invoking scenarios

The following table specifies invoking scenarios and lists the required JAR files to successfully invoke AEM Forms.

Services	Invocation mode	J2EE application server	Required JAR files
Forms service	EJB	JBoss	• adobe-livecycle-client.jar • adobe-usermanager-client.jar • jbossall-client.jar • jacorb.jar • jnp-client.jar • adobe-forms-client.jar

Services	Invocation mode	J2EE application server	Required JAR files
Forms service Acrobat Reader DC extensions service Signature service	EJB	JBoss	• adobe-livecycle-client.jar • adobe-usermanager-client.jar • jbossall-client.jar • jacorb.jar • jnp-client.jar • adobe-forms-client.jar • adobe-reader-extensions-client.jar • adobe-signatures-client.jar
Forms service	SOAP	WebLogic	• adobe-livecycle-client.jar • adobe-usermanager-client.jar • wlclient.jar • activation.jar • axis.jar • commons-codec-1.3.jar • commons-collections-3.1.jar • commons-discovery.jar • commons-logging.jar • dom3-xml-apis-2.5.0.jar • jai_imageio.jar • jaxen-1.1-beta-9.jar • jaxrpc.jar • log4j.jar • mail.jar • saaj.jar • wsdl4j.jar • xalan.jar • xbean.jar • xercesImpl.jar • adobe-forms-client.jar

Services	Invocation mode	J2EE application server	Required JAR files
Forms service Acrobat Reader DC extensions service Signature service	SOAP	WebLogic	• adobe-livecycle-client.jar • adobe-usermanager-client.jar • wlclient.jar • activation.jar • axis.jar • commons-codec-1.3.jar • commons-collections-3.1.jar • commons-discovery.jar • commons-logging.jar • dom3-xml-apis-2.5.0.jar • jai_imageio.jar • jaxen-1.1-beta-9.jar • jaxrpc.jar • log4j.jar • mail.jar • saaj.jar • wsdl4j.jar • xalan.jar • xbean.jar • xercesImpl.jar • adobe-forms-client.jar • adobe-reader-extensions-client.jar • adobe-signatures-client.jar

Upgrading JAR files

If you are upgrading from LiveCycle to AEM Forms, it is recommeded that you include the AEM Forms JAR files in your Java project's class path. For example, if you are using services such as the Rights Management service, you will encounter a compatibility issue if you do not include AEM Forms JAR files in your class path.

Assuming that you are upgrading to AEM Forms. To use a Java application that invokes the Rights Management service, include the AEM Forms versions of the following JAR files:

- adobe-rightsmanagement-client.jar
- adobe-livecycle-client.jar
- adobe-usermanager-client.jar

See also

"Invoking AEM Forms using the Java API" on page 490

"Setting connection properties" on page 500

"Passing data to AEM Forms services using the Java API" on page 505

"Invoking a service using a Java client library" on page 511

Setting connection properties

You set connection properties to invoke AEM Forms when using the Java API. When setting connection properties, specify whether to invoke services remotely or locally, and also specify the connection mode and authentication values. Authentication values are required if service security is enabled. However, if service security is disabled, it is not necessary to specify authentication values. (See "Disabling Service Security" on page 1083.)

The connection mode can either be SOAP or EJB mode. The EJB mode uses the RMI/IIOP protocol, and the performance of the EJB mode is better than the performance of the SOAP mode. The SOAP mode is used to eliminate a J2EE application server dependency or when a firewall is located between AEM Forms and the client application. The SOAP mode uses the https protocol as the underlying transport and can communicate across firewall boundaries. If neither a J2EE application server dependency or a firewall is an issue, it is recommended that you use the EJB mode.

To successfully invoke a AEM Forms service, set the following connection properties:

- **DSC_DEFAULT_EJB_ENDPOINT:** If you are using the EJB connection mode, this value represents the URL of the J2EE application server on which AEM Forms is deployed. To remotely invoke AEM Forms, specify the J2EE application server name on which AEM Forms is deployed. If your client application is located on the same J2EE application server, then you can specify `localhost`. Depending on which J2EE application server AEM Forms is deployed on, specify one of the following values:
  - JBoss: `jnp://<ServerName>:1099 (default port)`
  - WebSphere: `iiop://<ServerName>:2809 (default port)`
  - WebLogic: `t3://<ServerName>:7001 (default port)`
- **DSC_DEFAULT_SOAP_ENDPOINT**: If you are using the SOAP connection mode, this value represents the endpoint to where an invocation request is sent. To remotely invoke AEM Forms, specify the J2EE application server name on which AEM Forms is deployed. If your client application is located on the same J2EE application server, you can specify `localhost` (for example, `http://localhost:8080`.)
  - The port value `8080` is applicable if the J2EE application is JBoss. If the J2EE application server is IBM® WebSphere®, use port `9080`. Likewise, if the J2EE application server is WebLogic, use port `7001`. (These values are default port values. If you change the port value, use the applicable port number.)
- **DSC_TRANSPORT_PROTOCOL**: If you are using the EJB connection mode, specify `ServiceClientFactoryProperties.DSC_EJB_PROTOCOL` for this value. If you are using the SOAP connection mode, specify `ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL`.
- **DSC_SERVER_TYPE**: Specifies the J2EE application server on which AEM Forms is deployed. Valid values are `JBoss`, `WebSphere`, `WebLogic`.
  - If you set this connection property to `WebSphere`, the `java.naming.factory.initial` value is set to `com.ibm.ws.naming.util.WsnInitCtxFactory`.
  - If you set this connection property to `WebLogic`, the `java.naming.factory.initial` value is set to `weblogic.jndi.WLInitialContextFactory`.
  - Likewise, if you set this connection property to `JBoss`, the `java.naming.factory.initial` value is set to `org.jnp.interfaces.NamingContextFactory`.

- You can set the `java.naming.factory.initial` property to a value that meets your requirements if you do not want to use the default values.

Note: *Instead of using a string to set the* `DSC_SERVER_TYPE` *connection property, you can use a static member of the* `ServiceClientFactoryProperties` *class. The following values can be used:* `ServiceClientFactoryProperties.DSC_WEBSPHERE_SERVER_TYPE`*,* `ServiceClientFactoryProperties.DSC_WEBLOGIC_SERVER_TYPE`*, or* `ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE`*.*

- **DSC_CREDENTIAL_USERNAME:** Specifies the AEM forms user name. For a user to sucessfully invoke a AEM Forms service, they need the Services User role. A user can also have another role that includes the Service Invoke permission. Otherwise, an exception is thrown when they attempt to invoke a service. If service security is disabled, it is not necessary to specify this connection property. (See "Disabling Service Security" on page 1083.)
- **DSC_CREDENTIAL_PASSWORD:** Specifies the corresponding password value. If service security is disabled, it is not necessary to specify this connection property.
- **DSC_REQUEST_TIMEOUT:** The default request timeout limit for the SOAP request is 1200000 milliseconds (20 minutes). Sometime, a request can require longer time to complete the operation. For example, a SOAP request that retrieves a large set of records can require a longer timeout limit. You can use the `ServiceClientFactoryProperties.DSC_REQUEST_TIMEOUT` to increase the request call timeout limit for the SOAP requests.

  Note: *Only SOAP-based invocations support the DSC_REQUEST_TIMEOUT property.*

To set connection properties, perform the following tasks:

1. Create a `java.util.Properties` object by using its constructor.
2. To set the `DSC_DEFAULT_EJB_ENDPOINT` connection property, invoke the `java.util.Properties` object's `setProperty` method and pass the following values:
   - The `ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT` enumeration value
   - A string value that specifies the URL of the J2EE application server that hosts AEM Forms

   Note: *If you are using the SOAP connection mode, specify the* `ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT` *enumeration value instead of the* `ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT` *enumeration value.*
3. To set the `DSC_TRANSPORT_PROTOCOL` connection property, invoke the `java.util.Properties` object's `setProperty` method and pass the following values:
   - The `ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL` enumeration value
   - The `ServiceClientFactoryProperties.DSC_EJB_PROTOCOL` enumeration value

   Note: *If you are using the SOAP connection mode, specify the* `ServiceClientFactoryProperties.DSC_SOAP_PROTOCOL` *enumeration value instead of the* `ServiceClientFactoryProperties.DSC_EJB_PROTOCOL` *enumeration value.*
4. To set the `DSC_SERVER_TYPE` connection property, invoke the `java.util.Properties` object's `setProperty` method and pass the following values:
   - The `ServiceClientFactoryProperties.DSC_SERVER_TYPE` enumeration value
   - A string value that specifies the J2EE application server that hosts AEM Forms (for example, if AEM Forms is deployed on JBoss, specify `JBoss`).
     - ❖ To set the `DSC_CREDENTIAL_USERNAME` connection property, invoke the `java.util.Properties` object's `setProperty` method and pass the following values:

- The `ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME` enumeration value
- A string value that specifies the user name required to invoke AEM Forms
  - ❖ To set the `DSC_CREDENTIAL_PASSWORD` connection property, invoke the `java.util.Properties` object's `setProperty` method and pass the following values:
- The `ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD` enumeration value
- A string value that specifies the corresponding password value

Setting the EJB connection mode

The following Java code example sets connection properties to invoke AEM Forms deployed on JBoss and using the EJB connection mode.

```
Properties ConnectionProps = new Properties();
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT,
"jnp://localhost:1099");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_EJB_PROTOCOL);
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

Setting the EJB connection mode for WebLogic

The following Java code example sets connection properties to invoke AEM Forms deployed on WebLogic and using the EJB connection mode.

```
Properties ConnectionProps = new Properties();
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT,
"t3://localhost:7001");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_EJB_PROTOCOL);
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "WebLogic");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

Setting the EJB connection mode for WebSphere

The following Java code example sets connection properties to invoke AEM Forms deployed on WebSphere and using the EJB connection mode.

```
Properties ConnectionProps = new Properties();
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT,
"iiop://localhost:2809");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_EJB_PROTOCOL);
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "WebSphere");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

Setting the SOAP connection mode

The following Java code example sets connection properties in SOAP mode to invoke AEM Forms deployed on JBoss.

```
Properties ConnectionProps = new Properties();
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://localhost:8080");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
```

Note: *If you select the SOAP connection mode, ensure to include additional JAR files in your client application's class path.*

Setting connection properties when service security is disabled

The following Java code example sets connection properties required to invoke AEM Forms deployed on JBoss Application Server and when service security is disabled. (See "Disabling Service Security" on page 1083.)

```
Properties ConnectionProps = new Properties();
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT,
"jnp://localhost:1099");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_EJB_PROTOCOL);
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
```

Note: *All Java Quick Starts associated with Programming with AEM Forms show both EJB and SOAP connection settings.*

Setting the SOAP connection mode with custom request timeout limit

```
Properties ConnectionProps = new Properties();
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://localhost:8080");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceCli
entFactoryProperties.DSC_SOAP_PROTOCOL);
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE, "JBoss");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");
ConnectionProps.setProperty(ServiceClientFactoryProperties.DSC_REQUEST_TIMEOUT, "1800000");
// Request timeout limit 30 Minutes
```

Using a Context object to invoke AEM Forms

You can use a `com.adobe.idp.Context` object to invoke a AEM Forms service with an authenticated user (the `com.adobe.idp.Context` object represents an authenticated user). When using a `com.adobe.idp.Context` object, you do not need to set the `DSC_CREDENTIAL_USERNAME` or `DSC_CREDENTIAL_PASSWORD` properties. You can obtain a `com.adobe.idp.Context` object when authenicating users by using the `AuthenticationManagerServiceClient` object's `authenticate` method.

The `authenticate` method returns an `AuthResult` object that contains the results of the authentication. You can create a `com.adobe.idp.Context` object by invoking its constructor. Then invoke the `com.adobe.idp.Context` object's `initPrincipal` method and pass the `AuthResult` object, as shown in the following code:

```
Context myCtx = new Context();
myCtx.initPrincipal(authResult);
```

Instead of setting the `DSC_CREDENTIAL_USERNAME` or `DSC_CREDENTIAL_PASSWORD` properties, you can invoke the `ServiceClientFactory` object's `setContext` method and pass the `com.adobe.idp.Context` object. When using a AEM forms user to invoke a service, ensure that they have the role named `Services User` that is required to invoke a AEM Forms service.

The following code example shows how to use a `com.adobe.idp.Context` object within connection settings that are used to create an `EncryptionServiceClient` object.

```
//Authenticate a user and use the Context object within connection settings
// Authenticate the user
String username = "wblue";
String password = "password";
AuthResult authResult = authClient.authenticate(username, password.getBytes());

//Set a Content object that represents the authenticated user
//Use the Context object to invoke the Encryption service
Context myCtx = new Context();
myCtx.initPrincipal(authResult);

//Set connection settings
Properties connectionProps = new Properties();
connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT,
"jnp://hiro-xp:1099");
connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,
ServiceClientFactoryProperties.DSC_EJB_PROTOCOL);
connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
ServiceClientFactoryProperties.DSC_JBOSS_SERVER_TYPE);

//Create a ServiceClientFactory object
ServiceClientFactory myFactory = ServiceClientFactory.createInstance(connectionProps);
myFactory.setContext(myCtx);

//Create an EncryptionServiceClient object
EncryptionServiceClient encryptClient  = new EncryptionServiceClient(myFactory);
```

Note: *For complete details about authenticating a user, see "Authenticating Users" on page 1028.*

Invoking scenarios

The following invoking scenarios are discussed in this section:

- A client application running in its own Java virtual machine (JVM) invokes a stand-alone AEM Forms instance.
- A client application running in its own JVM invokes clustered AEM Forms instances.

Client application invoking a stand-alone AEM Forms instance

The following diagram shows a client application running in its own JVM and invoking a stand-alone AEM Forms instance.

In this scenario, a client application is running in its own JVM and invokes AEM Forms services.

Note: *This scenario is the invoking scenario on which all Quick Starts are based.*

Client application invoking clustered AEM Forms instances

The following diagram shows a client application running in its own JVM and invoking AEM Forms instances located in a cluster.

This scenario is similar to a client application invoking a stand-alone AEM Forms instance. However, the provider URL is different. If a client application wants to connect to a specific J2EE application server, the application must change the URL to reference the specific J2EE application server.

Referencing a specific J2EE application server is not recommended because the connection between the client application and AEM Forms is terminated if the application server stops. It is recommended that the provider URL reference a cell-level JNDI manager, instead of a specific J2EE application server.

Client applications that use the SOAP connection mode can use the HTTP load balancer port for the cluster. Client applications that use the EJB connection mode can connect to the EJB port of a specific J2EE application server. This action handles the Load Balancing between cluster nodes.

WebSphere

The following example shows the contents of a jndi.properties file that is used to connect to AEM Forms that is deployed on WebSphere.

```
java.naming.factory.initial=com.ibm.websphere.naming.
WsnInitialContextFactory
java.naming.provider.url=corbaloc::appserver1:9810,:appserver2:9810
```

WebLogic

The following example shows the contents of a jndi.properties file that is used to connect to AEM Forms that is deployed on WebLogic.

```
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.provider.url=t3://appserver1:8001, appserver2:8001
```

JBoss

The following example shows the contents of a jndi.properties file that is used to connect to AEM Forms that is deployed on JBoss.

```
java.naming.factory.initial= org.jnp.interfaces.NamingContextFactory
java.naming.provider.url= jnp://appserver1:1099, appserver2:1099,
appserver3:1099
```

Note: *Consult your administrator to determine the J2EE application server name and port number.*

See also

"Including AEM Forms Java library files" on page 491

"Passing data to AEM Forms services using the Java API" on page 505

"Invoking a service using a Java client library" on page 511

Passing data to AEM Forms services using the Java API

AEM Forms service operations typically consume or produce PDF documents. When you invoke a service, sometimes it is necessary to pass a PDF document (or other document types such as XML data) to the service. Likewise sometimes it is necessary to handle a PDF document that is returned from the service. The Java class that enables you to pass data to and from AEM Forms services is `com.adobe.idp.Document`.

AEM Forms services do not accept a PDF document as other data types, such as a `java.io.InputStream` object or a byte array. A `com.adobe.idp.Document` object can also be used to pass other types of data, such as XML data, to services.

A `com.adobe.idp.Document` object is a Java serializable type, so it can be passed over an RMI call. The receiving side can be collocated (same host, same class loader), local (same host, different class loader), or remote (different host). Passing of document content is optimized for each case. For example, if the sender and receiver are located on the same host, the content is passed over a local file system. (In some cases, documents can be passed in memory.)

Depending on the `com.adobe.idp.Document` object size, the data is carried within the `com.adobe.idp.Document` object or stored on the server's file system. Any temporary storage resources occupied by the `com.adobe.idp.Document` object are removed automatically upon the `com.adobe.idp.Document` disposal. (See "Disposing Document objects" on page 510.)

Sometimes it is necessary to know the content type of a `com.adobe.idp.Document` object before you can pass it to a service. For example, if an operation requires a specific content type, such as `application/pdf`, it is recommended that you determine the content type. (See "Determining the content type of a document" on page 510.)

The `com.adobe.idp.Document` object attempts to determine the content type using the supplied data. If the content type cannot be retrieved from the data supplied (for example, when the data was supplied as a byte array), set the content type. To set the content type, invoke the `com.adobe.idp.Document` object's `setContentType` method. (See "Determining the content type of a document" on page 510)

If collateral files reside on the same file system, creating a `com.adobe.idp.Document` object is faster. If collateral files reside on remote file systems, a copy operation must be done, which affects performance.

An application can contain both `com.adobe.idp.Document` and `org.w3c.dom.Document` data types. However, ensure that you fully qualify the `org.w3c.dom.Document` data type. For information about converting a `org.w3c.dom.Document` object to a `com.adobe.idp.Document` object, see "Quick Start (SOAP mode): Prepopulating Forms with Flowable Layouts using the Java API" on page 190.

Note: *To prevent a memory leak in WebLogic while using a `com.adobe.idp.Document` object, read the document information in chunks of 2048 bytes or less. For example, the following code reads the document information in chunks of 2048 bytes:*

```
    // Set up the chunk size to prevent a potential memory leak
    int buffSize = 2048;

    // Determine the total number of bytes to read
    int docLength = (int) inDoc.length();
    byte [] byteDoc = new byte[docLength];

    // Set up the reading position
    int pos = 0;

    // Loop through the document information, 2048 bytes at a time
    while (docLength > 0) {
  // Read the next chunk of information
        int toRead = Math.min(buffSize, docLength);
        int bytesRead = inDoc.read(pos, byteDoc, pos, toRead);

        // Handle the exception in case data retrieval failed
        if (bytesRead == -1) {

            inDoc.doneReading();
            inDoc.dispose();
            throw new RuntimeException("Data retrieval failed!");

        }

         // Update the reading position and number of bytes remaining
         pos += bytesRead;
         docLength -= bytesRead;

    }

    // The document information has been successfully read
    inDoc.doneReading();
    inDoc.dispose();
```

See also

"Invoking AEM Forms using the Java API" on page 490

"Setting connection properties" on page 500

Creating documents

Create a `com.adobe.idp.Document` object before you invoke a service operation that requires a PDF document (or other document types) as an input value. The `com.adobe.idp.Document` class provides constructors that enable you to create a document from the following content types:

- A byte array
- An existing `com.adobe.idp.Document` object
- A `java.io.File` object
- A `java.io.InputStream` object
- A `java.net.URL` object

Creating a document based on a byte array

The following code example creates a `com.adobe.idp.Document` object that is based on a byte array.

Creating a Document object that is based on a byte array

```
Document myPDFDocument = new Document(myByteArray);
```

Creating a document based on another document

The following code example creates a `com.adobe.idp.Document` object that is based on another `com.adobe.idp.Document` object.

Creating a Document object that is based on another document

```
//Create a Document object based on a byte array
InputStream is = new FileInputStream("C:\\Map.pdf");
int len = is.available();
byte [] myByteArray = new byte[len];
int i = 0;
while (i < len) {
      i += is.read(myByteArray, i, len);
}
Document myPDFDocument = new Document(myByteArray);

//Create another Document object
Document anotherDocument = new Document(myPDFDocument);
```

Creating a document based on a file

The following code example creates a `com.adobe.idp.Document` object that is based on a PDF file named *map.pdf*. This file is located in the root of the C hard drive. This constructor attempts to set the MIME content type of the `com.adobe.idp.Document` object using the filename extension.

The `com.adobe.idp.Document` constructor that accepts a `java.io.File` object also accepts a Boolean parameter. By setting this parameter to `true`, the `com.adobe.idp.Document` object deletes the file. This action means that you do not have to remove the file after passing it to the `com.adobe.idp.Document` constructor.

Setting this parameter to `false` means that you retain ownership of this file. Setting this parameter to `true` is more efficient. The reason is because the `com.adobe.idp.Document` object can move the file directly to the local managed area instead of copying it (which is slower).

Creating a Document object that is based on a PDF file

```
//Create a Document object based on the map.pdf source file
File mySourceMap = new File("C:\\map.pdf");
Document myPDFDocument = new Document(mySourceMap,true);
```

Creating a document based on an InputStream object

The following Java code example creates a `com.adobe.idp.Document` object that is based on a `java.io.InputStream` object.

Creating a document based on an InputStream object

```
//Create a Document object based on an InputStream object
InputStream is = new FileInputStream("C:\\Map.pdf");
Document myPDFDocument = new Document(is);
```

Creating a document based on content accessible from an URL

The following Java code example creates a `com.adobe.idp.Document` object that is based on a PDF file named *map.pdf*. This file is located within a web application named `WebApp` that is running on `localhost`. This constructor attempts to set the `com.adobe.idp.Document` object's MIME content type using the content type returned with the URL protocol.

The URL supplied to the `com.adobe.idp.Document` object is always read at the side where the original `com.adobe.idp.Document` object is created, as shown in this example:

```
Document doc = new Document(new java.net.URL("file:c:/temp/input.pdf"));
```

The c:/temp/input.pdf file must be located on the client computer (not on the server computer). The client computer is where the URL is read and where the `com.adobe.idp.Document` object was created.

Creating a document based on content accessible from an URL

```
//Create a Document object based on a java.net.URL object
URL myURL = new URL("http", "localhost", 8080,"/WebApp/map.pdf");

//Create another Document object
Document myPDFDocument = new Document(myURL);
```

See also

"Invoking AEM Forms using the Java API" on page 490

"Setting connection properties" on page 500

Handling returned documents

Service operations that return a PDF document (or other data types such as XML data) as an output value return a `com.adobe.idp.Document` object. After you receive a `com.adobe.idp.Document` object, you can convert it to the following formats:

- A `java.io.File` object
- A `java.io.InputStream` object
- A byte array

The following line of code converts a `com.adobe.idp.Document` object to a `java.io.InputStream` object. Assume that `myPDFDocument` represents a `com.adobe.idp.Document` object:

```
java.io.InputStream resultStream = myDocument.getInputStream();
```

Likewise, you can copy the contents of a `com.adobe.idp.Document` to a local file by performing the following tasks:

1. Create a `java.io.File` object.
2. Invoke the `com.adobe.idp.Document` object's `copyToFile` method and pass the `java.io.File` object.

The following code example copies the contents of a `com.adobe.idp.Document` object to a file named *AnotherMap.pdf*.

Copying the contents of a document object to a file

```
File outFile = new File("C:\\AnotherMap.pdf");
myDocument.copyToFile (outFile);
```

See also

"Invoking AEM Forms using the Java API" on page 490

"Setting connection properties" on page 500

Determining the content type of a document

Determine the MIME type of a `com.adobe.idp.Document` object by invoking the `com.adobe.idp.Document` object's `getContentType` method. This method returns a string value that specifies the content type of the `com.adobe.idp.Document` object. The following table describes the different content types that AEM Forms returns.

MIME type	Description
`application/pdf`	PDF document
`application/vnd.adobe.xdp+xml`	XML Data Packaging (XDP), which is used for exported XML Forms Architecture (XFA) forms
`text/xml`	Bookmarks, attachments, or other XML documents
`application/vnd.fdf`	Forms Data Format (FDF), which is used for exported Acrobat forms
`application/vnd.adobe.xfdf`	XML Forms Data Format (XFDF), which is used for exported Acrobat forms
`application/rdf+xml`	Rich data format and XML
`application/octet-stream`	Generic data format
`NULL`	Unspecified MIME type

The following code example determines the content type of a `com.adobe.idp.Document` object.

Determining the content type of a Document object

```
//Determine the content type of the Document object
String ct = myDocument.getContentType();
System.out.println("The content type of the Document object is " +ct);
```

See also

"Invoking AEM Forms using the Java API" on page 490

"Setting connection properties" on page 500

Disposing Document objects

When you no longer require a `Document` object, it is recommended that you dispose of it by invoking its `dispose` method. Each `Document` object consumes a file descriptor and as much as 75 MB of RAM space on your application's host platform. If a `Document` object is not disposed, then the Java Garage collection process disposes it. However, by disposing of it sooner by using the `dispose` method, you can free the memory occupied by the `Document` object.

See also

"Invoking AEM Forms using the Java API" on page 490

"Including AEM Forms Java library files" on page 491

"Invoking a service using a Java client library" on page 511

Invoking a service using a Java client library

AEM Forms service operations can be invoked by using a service's strongly typed API, which is known as a Java client library. A *Java client library* is a set of concrete classes that provide access to services deployed in the service container. You instantiate a Java object that represents the service to invoke instead of creating an `InvocationRequest` object by using the Invocation API. The Invocation API is used to invoke processes, such as long-lived processes, created in Workbench. (See "Invoking Human-Centric Long-Lived Processes" on page 560.)

To perform a service operation, invoke a method that belongs to the Java object. A Java client library contains methods that typically map one-to-one with service operations. When using a Java client library, set required connection properties. (See "Setting connection properties" on page 500.)

After you set connection properties, create a `ServiceClientFactory` object that is used to instantiate a Java object that lets you invoke a service. Each service that has a Java client library has a corresponding client object. For example, to invoke the Repository service, create a `ResourceRepositoryClient` object by using its constructor and passing the `ServiceClientFactory` object. The `ServiceClientFactory` object is responsible for maintaining connection settings that are required to invoke AEM Forms services.

Although obtaining a `ServiceClientFactory` is typically fast, some overhead is involved when the factory is first used. This object is optimized for reuse and therefore, when possible, use the same `ServiceClientFactory` object when you are creating multiple Java client objects. That is, do not create a separate `ServiceClientFactory` object for each client library object that you create.

There is a User Manager setting that controls the lifetime of the SAML assertion that is inside the `com.adobe.idp.Context` object that affects the `ServiceClientFactory` object. This setting controls all authentication context lifetimes throughout AEM Forms, including all invocations performed by using the Java API. By default, the time period in which a `ServiceCleintFactory` object can be used is two hours.

Note: *To explain how to invoke a service by using the Java API, the Repository service's `writeResource` operation is invoked. This operation places a new resource into the repository.*

You can invoke the Repository service by using a Java client library and by performing the following steps:

1. Include client JAR files, such as the adobe-repository-client.jar, in your Java project's class path. For information about the location of these files, see "Including AEM Forms Java library files" on page 491.
2. Set connection properties that are required to invoke a service.
3. Create a `ServiceClientFactory` object by invoking the `ServiceClientFactory` object's static `createInstance` method and passing the `java.util.Properties` object that contains connection properties.
4. Create a `ResourceRepositoryClient` object by using its constructor and passing the `ServiceClientFactory` object. Use the `ResourceRepositoryClient` object to invoke Repository service operations.
5. Create a `RepositoryInfomodelFactoryBean` object by using its constructor and pass `null`. This object lets you create a `Resource` object that represents the content that is added to the repository.
6. Create a `Resource` object by invoking the `RepositoryInfomodelFactoryBean` object's `newImage` method and passing the following values:
   - A unique ID value by specifying `new Id()`.
   - A unique UUID value by specifying `new Lid()`.
   - The name of the resource. You can specify the file name of the XDP file.

   Cast the return value to `Resource`.

7 Create a `ResourceContent` object by invoking the `RepositoryInfomodelFactoryBean` object's `newImage` method and casting the return value to `ResourceContent`. This object represents the content that is added to the repository.

8 Create a `com.adobe.idp.Document` object by passing a `java.io.FileInputStream` object that stores the XDP file to add to the repository. (See "Creating a document based on an InputStream object" on page 508.)

9 Add the content of the `com.adobe.idp.Document` object to the `ResourceContent` object by invoking the `ResourceContent` object's `setDataDocument` method. Pass the `com.adobe.idp.Document` object.

10 Set the MIME type of the XDP file to add to the repository by invoking the `ResourceContent` object's `setMimeType` method and passing `application/vnd.adobe.xdp+xml`.

11 Add the content of the `ResourceContent` object to the `Resource` object by invoking the `Resource` object 's `setContent` method and passing the `ResourceContent` object.

12 Add a description of the resource by invoking the `Resource` object 's `setDescription` method and passing a string value that represents a description of the resource.

13 Add the form design to the repository by invoking the `ResourceRepositoryClient` object's `writeResource` method and passing the following values:

- A string value that specifies the path to the resource collection that contains the new resource
- The `Resource` object that was created

See also

"Quick Start (SOAP mode): Writing a resource using the Java API" on page 307

"Invoking AEM Forms using the Java API" on page 490

"Including AEM Forms Java library files" on page 491

Invoking a short-lived process using the Invocation API

You can invoke a short-lived process using the Java Invocation API. When you invoke a short-lived process using the Invocation API, you pass required parameter values by using a `java.util.HashMap` object. For each parameter to pass to a service, invoke the `java.util.HashMap` object's `put` method and specify the name-value pair that is required by the service in order to perform the specified operation. Specify the exact name of the parameters that belong to the short-lived process.

Note: *For information about invoking a long-lived process, see "Invoking Human-Centric Long-Lived Processes" on page 560.*

The discussion here is about using Invocation API to invoke the following AEM Forms short-lived process named `MyApplication/EncryptDocument`.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code example, create a process named `MyApplication/EncryptDocument` using Workbench. (See Using Workbench.)*

When this process is invoked, it performs the following actions:

1 Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.

2 Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

Invoke the MyApplication/EncryptDocument short-lived process using the Java invocation API

Invoke the `MyApplication/EncryptDocument` short-lived process using the Java invocation API:

1. Include client JAR files, such as the adobe-livecycle-client.jar, in your Java project's class path. (See "Including AEM Forms Java library files" on page 491.)
2. Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
3. Create a `ServiceClient` object by using its constructor and passing the `ServiceClientFactory` object. A `ServiceClient` object lets you invoke a service operation. It handles tasks such as locating, dispatching, and routing invocation requests.
4. Create a `java.util.HashMap` object by using its constructor.
5. Invoke the `java.util.HashMap` object's `put` method for each input parameter to pass to the long-lived process. Because the `MyApplication/EncryptDocument` short-lived process requires one input parameter of type `Document`, you only have to invoke the `put` method once, as shown in the following example.

```
//Create a Map object to store the parameter value for inDoc
Map params = new HashMap();
InputStream inFile = new FileInputStream("C:\\Adobe\Loan.pdf");
Document inDoc = new Document(inFile);
params.put("inDoc", inDoc);
```

6. Create an `InvocationRequest` object by invoking the `ServiceClientFactory` object's `createInvocationRequest` method and passing the following values:
   - A string value that specifies the name of the long-lived process to invoke. To invoke the `MyApplication/EncryptDocument` process, specify `MyApplication/EncryptDocument`.
   - A string value that represents the process operation name. Typically the name of a short-lived process operation is `invoke`.
   - The `java.util.HashMap` object that contains the parameter values that the service operation requires.
   - A Boolean value that specifies `true`, which creates a synchronous request (this value is applicable to invoke a short-lived process).
7. Send the invocation request to the service by invoking the `ServiceClient` object's `invoke` method and passing the `InvocationRequest` object. The `invoke` method returns an `InvocationReponse` object.

   Note: *A long-lived process can be invoked by passing the value `false` as the fourth parameter of the `createInvocationRequest` method. Passing the value `false` creates an asynchronous request.*
8. Retrieve the process's return value by invoking the `InvocationReponse` object's `getOutputParameter` method and passing a string value that specifies the name of the output parameter. In this situation, specify `outDoc` (`outDoc` is the name of the output parameter for the `MyApplication/EncryptDocument` process). Cast the return value to `Document`, as shown in the following example.

```
InvocationResponse response = myServiceClient.invoke(request);
Document encryptDoc = (Document) response.getOutputParameter("outDoc");
```

9. Create a `java.io.File` object and ensure that the file extension is .pdf.
10. Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that was returned by the `getOutputParameter` method.

See also

"Quick Start: Invoking a short-lived process using the Invocation API" on page 219

"Invoking Human-Centric Long-Lived Processes" on page 560

"Including AEM Forms Java library files" on page 491

Invoking AEM Forms using Web Services

Most AEM Forms services in the service container are configured to expose a web service, with full support for web service definition language (WSDL) generation. That is, you can create proxy objects that consume the native SOAP stack of an AEM Forms service. As a result, AEM Forms services can exchange and process the following SOAP messages:

- **SOAP request**: Sent to a Forms service by a client application requesting an action.
- **SOAP response**: Sent to a client application by a Forms service after a SOAP request is processed.

Using web services, you can perform the same AEM Forms services operations that you can by using the Java API. A benefit of using web services to invoke AEM Forms services is that you can create a client application in a development environment that supports SOAP. A client application is not bound to a specific development environment or programming language. For example, you can create a client application using Microsoft Visual Studio .NET and C# as the programming language.

AEM Forms services are exposed over the SOAP protocol and are WSI Basic Profile 1.1 compliant. Web Services Interoperability (WSI) is an open standards organization that promotes web service interoperability across platforms. For information, see http://www.ws-i.org/.

AEM Forms supports the following web service standards:

- **Encoding**: Supports only document and literal encoding (which is the preferred encoding according to the WSI Basic Profile). (See "Invoking AEM Forms using Base64 encoding" on page 525.)
- **MTOM**: Represents a way to encode attachments with SOAP requests. (See "Invoking AEM Forms using MTOM" on page 529.)
- **SwaRef**: Represents another way to encode attachments with SOAP requests. (See "Invoking AEM Forms using SwaRef" on page 531.)
- **SOAP with attachments**: Supports both MIME and DIME (Direct Internet Message Encapsulation). These protocols are standard ways of sending attachments over SOAP. Microsoft Visual Studio .NET applications use DIME. (See "Invoking AEM Forms using Base64 encoding" on page 525.)
- **WS-Security**: Supports a user name password token profile, which is a standard way of sending user names and passwords as part of the WS Security SOAP header. AEM Forms also supports HTTP basic authentication. (See Passing credentials using WS-Security headers.)

To invoke AEM Forms services using a web service, typically you create a proxy library that consumes the service WSDL. The *Invoking AEM Forms using Web Services* section uses JAX-WS to create Java proxy classes to invoke services. (See "Creating Java proxy classes using JAX-WS" on page 521.)

You can retrieve a service WDSL by specifying the following URL definition (items in brackets are optional):

```
http://<your_serverhost>:<your_port>/soap/services/<service_name>?wsdl[&version=<version>][&
async=true|false][lc_version=<lc_version>]
```

where:

- *your_serverhost* represents the IP address of the J2EE application server hosting AEM Forms.
- *your_port* represents the HTTP port that the J2EE application server uses.
- *service_name* represents the service name.

- *version* represents the target version of a service (the latest service version is used by default).
- `async` specifies the value `true` to enable additional operations for asynchronous invocation (`false` by default).
- *lc_version* represents the version of AEM Forms that you want to invoke.

The following table lists service WSDL definitions (assuming that AEM Forms is deployed on the local host and the post is 8080).

Service	WSDL definition
Assembler	`http://localhost:8080/soap/services/ AssemblerService?wsdl`
Back and Restore	`http://localhost:8080/soap/services/BackupService?wsdl`
barcoded forms	`http://localhost:8080/soap/services/ BarcodedFormsService?wsdl`
Convert PDF	`http://localhost:8080/soap/services/ ConvertPDFService?wsdl`
Distiller	`http://localhost:8080/soap/services/ DistillerService?wsdl`
DocConverter	`http://localhost:8080/soap/services/DocConverterService?WSDL`
DocumentManagement	`http://localhost:8080/soap/services/DocumentManagementService?WSDL`
Encryption	`http://localhost:8080/soap/services/EncryptionService?wsdl`
Forms	`http://localhost:8080/soap/services/FormsService?wsdl`
Form Data Integration	`http://localhost:8080/soap/services/FormDataIntegration?wsdl`
Generate PDF	`http://localhost:8080/soap/services/ GeneratePDFService?wsdl`
Generate 3D PDF	`http://localhost:8080/soap/services/Generate3dPDFService?WSDL`
Output	`http://localhost:8080/soap/services/ OutputService?wsdl`
PDF Utilities	`http://localhost:8080/soap/services/ PDFUtilityService?wsdl`
Acrobat Reader DC extensions	`http://localhost:8080/soap/services/ ReaderExtensionsService?wsdl`
Repository	`http://localhost:8080/soap/services/ RepositoryService?wsdl`
Rights Management	`http://localhost:8080/soap/services/ RightsManagementService?wsdl`
Signature	`http://localhost:8080/soap/services/ SignatureService?wsdl`
XMP Utilities	`http://localhost:8080/soap/services/ XMPUtilityService?wsdl`

AEM Forms Process WSDL definitions

You must specify the Application name and the Process name within the WSDL definition to access a WSDL that belongs to a process created in Workbench. Assume that the name of the application is `MyApplication` and the name of the process is `EncryptDocument`. In this situation, specify the following WSDL definition:

```
http://localhost:8080/soap/services/MyApplication/EncryptDocument?wsdl
```

Note: *For information about the example `MyApplication/EncryptDocument` short-lived process, see .*

Note: *An Application can contain folder(s). In this case, specify the folder name(s) in the WSDL definition:*

```
http://localhost:8080/soap/services/MyApplication/[<folderA>/.../<folderZ>/]EncryptDocument?wsdl
```

Accessing new functionality using web services

New AEM Forms service functionality can be accessed using web services. For example, in AEM Forms, the ability to encode attachments using MTOM is introduced. (See "Invoking AEM Forms using MTOM" on page 529.)

To access new functionality introduced in AEM Forms, specify the `lc_version` attribute in the WSDL definition. For example, to access new service functionality (including MTOM support), specify the following WSDL definition:

```
http://localhost:8080/soap/services/MyApplication/EncryptDocument?wsdl&lc_version=9.0.1
```

Note: *When setting the `lc_version` attribute, ensure that you use three digits. For example, 9.0.1 is equal to version 9.0.*

Web service BLOB data type

AEM Forms service WSDLs define many data types. One of the most important data types exposed in a web service is a `BLOB` type. This data type maps to the `com.adobe.idp.Document` class when working with AEM Forms Java APIs. (See "Passing data to AEM Forms services using the Java API" on page 505.)

A `BLOB` object sends and retrieves binary data (for example, PDF files, XML data, and so on) to and from AEM Forms services. The `BLOB` type is defined in a service WSDL as follows:

```
<complexType name="BLOB">
    <sequence>
        <element maxOccurs="1" minOccurs="0" name="contentType"
            type="xsd:string"/>
        <element maxOccurs="1" minOccurs="0" name="binaryData"
            type="xsd:base64Binary"/>
        <element maxOccurs="1" minOccurs="0" name="attachmentID"
            type="xsd:string"/>
        <element maxOccurs="1" minOccurs="0" name="remoteURL"
            type="xsd:string"/>
        <element maxOccurs="1" minOccurs="0" name="MTOM"
            type="xsd:base64Binary"
            xmime:expectedContentTypes="*/*"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/>
        <element maxOccurs="1" minOccurs="0" name="swaRef"
            type="tns1:swaRef"/>
        <element maxOccurs="1" minOccurs="0" name="attributes"
            type="impl:MyMapOf_xsd_string_To_xsd_anyType"/>
    </sequence>
</complexType>
```

The `MTOM` and `swaRef` fields are supported only in AEM Forms. You can use those new fields only if you specify a URL that includes the `lc_version` property.

Supplying BLOB objects in service requests

If an AEM Forms service operation requires a `BLOB` type as an input value, create an instance of the `BLOB` type in your application logic. (Many of the web service quick starts located in *Programming with AEM forms* show how to work with a BLOB data type.)

Assign values to fields that belong to the `BLOB` instance as follows:

- **Base64**: To pass data as text encoded in a Base64 format, set the data in the `BLOB.binaryData` field and set the data type in the MIME format (for example `application/pdf`) in the `BLOB.contentType` field. (See "Invoking AEM Forms using Base64 encoding" on page 525.)
- **MTOM**: To pass binary data in an MTOM attachment, set the data in the `BLOB.MTOM` field. This setting attaches the data to the SOAP request using the Java JAX-WS framework or the SOAP framework's native API. (See "Invoking AEM Forms using MTOM" on page 529.)
- **SwaRef**: To pass binary data in an WS-I SwaRef attachment, set the data in the `BLOB.swaRef` field. This setting attaches the data to the SOAP request using the Java JAX-WS framework. (See "Invoking AEM Forms using SwaRef" on page 531.)

- **MIME or DIME attachment**: To pass data in a MIME or DIME attachment, attach the data to the SOAP request using the SOAP framework's native API. Set the attachment identifier in the `BLOB.attachmentID` field. (See "Invoking AEM Forms using Base64 encoding" on page 525.)
- **Remote URL**: If data is hosted on a web server and accessible over an HTTP URL, set the HTTP URL in the `BLOB.remoteURL` field. (See "Invoking AEM Forms using BLOB data over HTTP" on page 533.)

Accessing data in BLOB objects returned from services

The transmission protocol for returned `BLOB` objects depends on several factors, which are considered in the following order, stopping when the main condition is satisfied:

1. **Target URL specifies transmission protocol**. If the target URL specified at the SOAP invocation contains the parameter `blob="`*BLOB_TYPE*", then *BLOB_TYPE* determines the transmission protocol. *BLOB_TYPE* is a placeholder for base64, dime, mime, http, mtom, or swaref.
2. **Service SOAP endpoint is Smart**. If the following conditions are true, then the output documents are returned using the same transmission protocol as the input documents:
   - Service's SOAP endpoint parameter Default Protocol For Output Blob Objects is set to Smart.

     For each service with a SOAP endpoint, the administration console allows you to specify the transmission protocol for any returned blobs. (See administration help.)
   - AEM Forms service takes one or more documents as input.
3. **Service SOAP endpoint is not Smart**. The configured protocol determines the document transmission protocol, and the data is returned in the corresponding `BLOB` field. For example, if the SOAP endpoint is set to DIME, then the returned blob is in the `blob.attachmentID` field regardless of the transmission protocol of any input document.
4. **Otherwise**. If a service does not take the document type as input, then the output documents are returned in the `BLOB.remoteURL` field over the HTTP protocol.

As described in the first condition, you can ensure the transmission type for any returned documents by extending the SOAP endpoint URL with a suffix as follows:

```
http://<your_serverhost>:<your_port>/soap/services/<service
name>?blob=base64|dime|mime|http|mtom|swaref
```

Here is the correlation between transmission types and the field from which you obtain the data:

- **Base64 format**: Set the `blob` suffix to `base64` to return the data in the `BLOB.binaryData` field.
- **MIME or DIME attachment**: Set the `blob` suffix to `DIME` or `MIME` to return the data as a corresponding attachment type with the attachment identifier returned in the `BLOB.attachmentID` field. Use the SOAP framework's proprietary API to read the data from the attachment.
- **Remote URL**: Set the `blob` suffix to `http` to keep the data on the application server and return the URL pointing to the data in the `BLOB.remoteURL` field.
- **MTOM or SwaRef**: Set the `blob` suffix to `mtom` or `swaref` to return the data as a corresponding attachment type with the attachment identifier returned in the `BLOB.MTOM` or `BLOB.swaRef` fields. Use the SOAP framework's native API to read the data from the attachment.

Important: *It is recommended that you do not exceed 30 MB when populating a `BLOB` object by invoking its `setBinaryData` method. Otherwise, there is a possibility that an `OutOfMemory` exception occurs.*

Important: *JAX WS-based applications that use the MTOM transmission protocol are limited to 25MB of sent and received data. This limitation is due to a bug in JAX-WS. If the combined size of your sent and received files exceeds 25MB, use the the SwaRef transmission protocol instead of the MTOM one. Otherwise, there is a possibility of an* `OutOfMemory` *exception.*

MTOM transmission of base64-encoded byte arrays

In addition to the `BLOB` object, the MTOM protocol supports any byte-array parameter or byte-array field of a complex type. This means that client SOAP frameworks supporting MTOM can send any `xsd:base64Binary` element as an MTOM attachment (instead of a base64-encoded text). AEM Forms SOAP endpoints can read this type of byte-array encoding. However, the AEM Forms service always returns a byte-array type as a base64-encoded text. The output byte-array parameters do not support MTOM.

AEM Forms services that return a large amount of binary data use the Document/BLOB type rather than the byte-array type. The Document type is much more efficient for transmitting large amounts of data.

Web service data types

The following table lists Java data types and shows the corresponding web service data type.

Java data type	Web service data type
`java.lang.byte[]`	`xsd:base64Binary`
`java.lang.Boolean`	`xsd:boolean`

`java.util.Date`	The `DATE` type, which is defined in a service WSDL as follows:
	`<complexType name="DATE">`
	`<sequence>`
	`<element maxOccurs="1" minOccurs="0" name="date" type="xsd:dateTime" />`
	`<element maxOccurs="1" minOccurs="0" name="calendar" type="xsd:dateTime" />`
	`</sequence>`
	`</complexType>`
	If a AEM Forms service operation takes a `java.util.Date` value as input, the SOAP client application must pass the date in the `DATE.date` field. Setting the `DATE.calendar` field in this case causes a runtime exception. If the service returns a `java.util.Date`, the date is retuned in the `DATE.date` field.

Java data type	Web service data type
java.util.Calendar	The DATE type, which is defined in a service WSDL as follows: <complexType name="DATE"> <sequence> <element maxOccurs="1" minOccurs="0" name="date" type="xsd:dateTime" /> <element maxOccurs="1" minOccurs="0" name="calendar" type="xsd:dateTime" /> </sequence> </complexType> If a AEM Forms service operation takes a java.util.Calendar value as input, the SOAP client application must pass the date in the DATE.caledendar field. Setting the DATE.date field in this case causes a run-time exception. If the service returns a java.util.Calendar, then the date is returned in the DATE.calendar field.
java.math.BigDecimal	xsd:decimal
com.adobe.idp.Document	BLOB
java.lang.Double	xsd:double
java.lang.Float	xsd:float
java.lang.Integer	xsd:int
java.util.List	MyArrayOf_xsd_anyType
java.lang.Long	xsd:long
java.util.Map	The apachesoap:Map, which is defined in a service WSDL as follows: <schema elementFormDefault="qualified" targetNamespace="http://xml.apache.org/xml-soap" xmlns="http://www.w3.org/2001/XMLSchema"> <complexType name="mapItem"> <sequence> <element name="key" nillable="true" type="xsd:anyType"/> <element name="value" nillable="true" type="xsd:anyType"/> </sequence> </complexType> <complexType name="Map"> <sequence> <element maxOccurs="unbounded" minOccurs="0" name="item" type="apachesoap:mapItem"/> </sequence> </complexType> </schema> The Map is represented as a sequence of key/value pairs.
java.lang.Object	xsd:anyType
java.lang.Short	xsd:short

Java data type	Web service data type
`java.lang.String`	`xsd:string`

`org.w3c.dom.Document`	The XML type, which is defined in a service WSDL as follows:
	`<complexType name="XML">`
	`<sequence>`
	`<element maxOccurs="1" minOccurs="0" name="document" type="xsd:string" />`
	`<element maxOccurs="1" minOccurs="0" name="element" type="xsd:string" />`
	`</sequence>`
	`</complexType>`
	If an AEM Forms service operation accepts an `org.w3c.dom.Document` value, pass the XML data in the `XML.document` field.
	Setting the `XML.element` field causes a runtime exception. If the service returns an `org.w3c.dom.Document`, then the XML data is returned in the `XML.document` field.

`org.w3c.dom.Element`	The XML type, which is defined in a service WSDL as follows:
	`<complexType name="XML">`
	`<sequence>`
	`<element maxOccurs="1" minOccurs="0" name="document" type="xsd:string" />`
	`<element maxOccurs="1" minOccurs="0" name="element" type="xsd:string" />`
	`</sequence>`
	`</complexType>`
	If an AEM Forms service operation takes an `org.w3c.dom.Element` as input, pass the XML data in the `XML.element` field.
	Setting the `XML.document` field causes a runtime exception. If the service returns an `org.w3c.dom.Element`, then the XML data is retuned in the `XML.element` field.

Adobe Developer website

The Adobe Developer website contains the following article that discusses invoking AEM Forms services using the web service API:

Creating form rendering ASP.NET applications

Invoking web services using custom components

Note: *Invoking web services using custom components describes how to create a AEM Forms component that invokes third party web services.*

Creating Java proxy classes using JAX-WS

You can use JAX-WS to convert a Forms service WSDL to Java proxy classes. These classes enable you to invoke AEM Forms services operations. Apache Ant lets you create a build script that generates Java proxy classes by referencing a AEM Forms service WSDL. You can generate JAX-WS proxy files by performing the following steps:

1 Install Apache Ant on the client computer. (See http://ant.apache.org/bindownload.cgi.)

- Add the bin directory to your class path.
- Set the `ANT_HOME` environment variable to the directory where you installed Ant.

2 Install JDK 1.6 or later.

- Add the JDK bin directory to your class path.
- Add the JRE bin directory to your class path. This bin is located in the [*JDK_INSTALL_LOCATION*]/jre directory.
- Set the `JAVA_HOME` environment variable to the directory where you installed the JDK.

JDK 1.6 includes the wsimport program used in the build.xml file. JDK 1.5 does not include that program.

3 Install JAX-WS on the client computer. (See Java API for XML Web Services.)

4 Use JAX-WS and Apache Ant to generate Java proxy classes. Create an Ant build script to accomplish this task. The following script is a sample Ant build script named build.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<project basedir="." default="compile">

<property name="port" value="8080" />
<property name="host" value="localhost" />
<property name="username" value="administrator" />
<property name="password" value="password" />
<property name="tests" value="all" />

<target name="clean" >
       <delete dir="classes" />
</target>

<target name="wsdl" depends="clean">
       <mkdir dir="classes"/>
       <exec executable="wsimport" failifexecutionfails="false" failonerror="true"
resultproperty="foundWSIMPORT">
           <arg line="-keep -d classes
http://${host}:${port}/soap/services/EncryptionService?wsdl&amp;lc_version=9.0.1"/>
       </exec>
       <fail unless="foundWSIMPORT">
          !!! Failed to execute JDK's wsimport tool. Make sure that JDK 1.6 (or later) is
on your PATH !!!
       </fail>
</target>
```

```
<target name="compile" depends="clean, wsdl" >
     <javac destdir="./classes" fork="true" debug="true">
        <src path="./src"/>
     </javac>
</target>

<target name="run">
     <java classname="Client" fork="yes" failonerror="true" maxmemory="200M">
        <classpath>
          <pathelement location="./classes"/>
        </classpath>
        <arg value="${port}"/>
        <arg value="${host}"/>
        <arg value="${username}"/>
        <arg value="${password}"/>
        <arg value="${tests}"/>
     </java>
</target>
</project>
```

Within this Ant build script, notice that the `url` property is set to reference the Encryption service WSDL running on localhost. The `username` and `password` properties must be set to a valid AEM forms user name and password. Notice that the URL contains the `lc_version` attribute. Without specifying the `lc_version` option, you cannot invoke new AEM Forms service operations.

Note: *Replace `EncryptionService` with the AEM Forms service name that you want to invoke using Java proxy classes. For example, to create Java proxy classes for the Rights Management service, specify:*

```
http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1
```

5 Create a BAT file to execute the Ant build script. The following command can be located within a BAT file that is responsible for executing the Ant build script:

```
ant -buildfile "build.xml" wsdl
```

Place the ANT build script in the C:\Program Files\Java\jaxws-ri\bin directory. The script writes the JAVA files to the ./classes folder. The script generates JAVA files that can invoke the service.

6 Package the JAVA files into a JAR file. If you are working on Eclipse, follow these steps:

- Create a new Java project that is used to package the proxy JAVA files into a JAR file.
- Create a source folder in the project.
- Create a `com.adobe.idp.services` package in the Source folder.
- Select the `com.adobe.idp.services` package and then import the JAVA files from the adobe/idp/services folder into the package.
- If necessary, create an `org/apache/xml/xmlsoap` package in the Source folder.
- Select the source folder and then import the JAVA files from the org/apache/xml/xmlsoap folder.
- Set the Java compiler's compliance level to 5.0 or greater.
- Build the project.
- Export the project as a JAR file.
- Import this JAR file in a client project's class path. In addition, import all of the JAR files located in <Install Directory>\Adobe\Adobe_Experience_Manager_forms\sdk\client-libs\thirdparty.

Note: *All Java web service quick starts (except for the Forms service) located in Programming with AEM forms create Java proxy files using JAX-WS. In addition, all Java web service quick starts, use SwaRef. (See "Invoking AEM Forms using SwaRef" on page 531.)*

See also

"Creating Java proxy classes using Apache Axis" on page 523

"Invoking AEM Forms using Base64 encoding" on page 525

"Invoking AEM Forms using BLOB data over HTTP" on page 533

"Invoking AEM Forms using SwaRef" on page 531

Creating Java proxy classes using Apache Axis

You can use the Apache Axis WSDL2Java tool to convert a Forms service into Java proxy classes. These classes enable you to invoke Forms service operations. Using Apache Ant, you can generate Axis library files from a service WSDL. You can download Apache Axis at the URL http://ws.apache.org/axis/.

Note: *The web service quick starts associated with the Forms service use Java proxy classes created using Apache Axis. The Forms web service quick starts also use Base64 as the encoding type. (See "Forms Service API Quick Starts" on page 153.)*

You can generate Axis Java library files by performing the following steps:

1. Install Apache Ant on the client computer. It is available at http://ant.apache.org/bindownload.cgi.
   - Add the bin directory to your class path.
   - Set the `ANT_HOME` environment variable to the directory where you installed Ant.
2. Install Apache Axis 1.4 on the client computer. It is available at http://ws.apache.org/axis/.
3. Set up the class path to use the Axis JAR files in your web service client, as described in the Axis installation instructions at http://ws.apache.org/axis/java/install.html.
4. Use the Apache WSDL2Java tool in Axis to generate Java proxy classes. Create an Ant build script to accomplish this task. The following script is a sample Ant build script named build.xml:

```
<?xml version="1.0"?>
<project name="axis-wsdl2java">

<path id="axis.classpath">
<fileset dir="C:\axis-1_4\lib" >
    <include name="**/*.jar" />
</fileset>
</path>

<taskdef resource="axis-tasks.properties" classpathref="axis.classpath" />

<target name="encryption-wsdl2java-client" description="task">
<axis-wsdl2java
    output="C:\JavaFiles"
    testcase="false"
    serverside="false"
    verbose="true"
    username="administrator"
    password="password"
    url="http://localhost:8080/soap/services/EncryptionService?wsdl&amp;lc_version=9.0.1"
>
</axis-wsdl2java>
</target>

</project>
```

Within this Ant build script, notice that the `url` property is set to reference the Encryption service WSDL running on localhost. The `username` and `password` properties must be set to a valid AEM forms user name and password.

5 Create a BAT file to execute the Ant build script. The following command can be located within a BAT file that is responsible for executing the Ant build script:

```
ant -buildfile "build.xml" encryption-wsdl2java-client
```

The JAVA files are written to the C:\JavaFiles folder as specified by the `output` property. To successfully invoke the Forms service, import these JAVA files into your class path.

By default, these files belong to a Java package named `com.adobe.idp.services`. It is recommended that you place these JAVA files into a JAR file. Then import the JAR file into your client application's class path.

Note: *There are different ways to put .JAVA files into a JAR. One way is using a Java IDE like Eclipse. Create a Java project and create a* `com.adobe.idp.services` *package (all .JAVA files belong to this package). Next import all the .JAVA files into the package. Finally, export the project as a JAR file.*

6 Amend the URL in the `EncryptionServiceLocator` class to specify the encoding type. For example, to use base64, specify `?blob=base64` to ensure that the `BLOB` object returns binary data. That is, in the `EncryptionServiceLocator` class, locate the following line of code:

```
http://localhost:8080/soap/services/EncryptionService;
```

and change it to:

```
http://localhost:8080/soap/services/EncryptionService?blob=base64;
```

7 Add the following Axis JAR files to your Java project's class path:

- activation.jar
- axis.jar
- commons-codec-1.3.jar

- commons-collections-3.1.jar
- commons-discovery.jar
- commons-logging.jar
- dom3-xml-apis-2.5.0.jar
- jai_imageio.jar
- jaxen-1.1-beta-9.jar
- jaxrpc.jar
- log4j.jar
- mail.jar
- saaj.jar
- wsdl4j.jar
- xalan.jar
- xbean.jar
- xercesImpl.jar

These JAR files are in the *[install directory]*/Adobe/Adobe Experience Manager Forms/sdk/lib/thirdparty directory.

See also

"Creating Java proxy classes using JAX-WS" on page 521

"Invoking AEM Forms using Base64 encoding" on page 525

"Invoking AEM Forms using BLOB data over HTTP" on page 533

Invoking AEM Forms using Base64 encoding

You can invoke a AEM Forms service using Base64 encoding. Base64 encoding encodes attachments that are sent with a web service invocation request. That is, `BLOB` data is Base64 encoded, not the entire SOAP message.

"Invoking AEM Forms using Base64 encoding" discusses invoking the following AEM Forms short-lived process named `MyApplication/EncryptDocument` by using Base64 encoding.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code example, create a process named* `MyApplication/EncryptDocument` *using Workbench. (See Using Workbench.)*

When this process is invoked, it performs the following actions:

1. Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.
2. Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

Creating a .NET client assembly that uses Base64 encoding

You can create a .NET client assembly to invoke a Forms service from a Microsoft Visual Studio .NET project. To create a .NET client assembly that uses base64 encoding, perform the following steps:

1. Create a proxy class based on an AEM Forms invocation URL.
2. Create a Microsoft Visual Studio .NET project that produces the .NET client assembly.

Creating a proxy class

You can create a proxy class that is used to create the .NET client assembly by using a tool that accompanies Microsoft Visual Studio. The name of the tool is wsdl.exe and it is located in the Microsoft Visual Studio installation folder. To create a proxy class, open the command prompt and navigate to the folder that contains the wsdl.exe file. For more information about the wsdl.exe tool, see the *MSDN Help*.

Enter the following command at the command prompt:

```
wsdl http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?WSDL&lc_version=9.0.1
```

By default, this tool creates a CS file in the same folder that is based on the name of the WSDL. In this situation, it creates a CS file named *EncryptDocumentService.cs*. You use this CS file to create a proxy object that lets you invoke the service that was specified in the invocation URL.

Amend the URL in the proxy class to include `?blob=base64` to ensure that the `BLOB` object returns binary data. In the proxy class, locate the following line of code:

```
"http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument";
```

and change it to:

```
"http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=base64";
```

The *Invoking AEM Forms using Base64 Encoding* section uses `MyApplication/EncryptDocument` as an example. If you are creating a .NET client assembly for another Forms service, ensure that you replace `MyApplication/EncryptDocument` with the name of the service.

Developing the .NET client assembly

Create a Visual Studio Class Library project that produces a .NET client assembly. The CS file that you created using wsdl.exe can be imported into this project. This project produces a DLL file (the .NET client assembly) that you can use in other Visual Studio .NET projects to invoke a service.

1. Start Microsoft Visual Studio .NET.
2. Create a Class Library project and name it DocumentService.
3. Import the CS file that you created using wsdl.exe.
4. In the **Project** menu, select **Add Reference**.
5. In the Add Reference dialog box, select **System.Web.Services.dll**.
6. Click **Select** and then click **OK**.
7. Compile and build the project.

Note: *This procedure creates a .NET client assembly named DocumentService.dll that you can use to send SOAP requests to the* `MyApplication/EncryptDocument` *service.*

Important: *Make sure that you added* `?blob=base64` *to the URL in the proxy class that is used to create the .NET client assembly. Otherwise, you cannot retrieve binary data from the* `BLOB` *object.*

Referencing the .NET client assembly

Place your newly created .NET client assembly on the computer where you are developing your client application. After you place the .NET client assembly in a directory, you can reference it from a project. Also reference the `System.Web.Services` library from your project. If you do not reference this library, you cannot use the .NET client assembly to invoke a service.

1. In the **Project** menu, select **Add Reference**.

2 Click the **.NET** tab.

3 Click **Browse** and locate the DocumentService.dll file.

4 Click **Select** and then click **OK**.

Invoking a service using a .NET client assembly that uses Base64 encoding

You can invoke the `MyApplication/EncryptDocument` service (which was built in Workbench) using a .NET client assembly that uses Base64 encoding. To invoke the `MyApplication/EncryptDocument` service, perform the following steps:

1 Create a Microsoft .NET client assembly that consumes the `MyApplication/EncryptDocument` service WSDL.

2 Create a client Microsoft .NET project. Reference the Microsoft .NET client assembly in the client project. Also reference `System.Web.Services`.

3 Using the Microsoft .NET client assembly, create a `MyApplication_EncryptDocumentService` object by invoking its default constructor.

4 Set the `MyApplication_EncryptDocumentService` object's `Credentials` property with a `System.Net.NetworkCredential` object. Within the `System.Net.NetworkCredential` constructor, specify a AEM forms user name and the corresponding password. Set authentication values to enable your .NET client application to successfully exchange SOAP messages with AEM Forms.

5 Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document pass to the `MyApplication/EncryptDocument` process.

6 Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the PDF document and the mode in which to open the file.

7 Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.

8 Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.

9 Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.

10 Invoke the `MyApplication/EncryptDocument` process by invoking the `MyApplication_EncryptDocumentService` object's `invoke` method and passing the `BLOB` object that contains the PDF document. This process returns an encrypted PDF document within a `BLOB` object.

11 Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the password-encrypted document.

12 Create a byte array that stores the data content of the `BLOB` object returned by the `MyApplicationEncryptDocumentService` object's `invoke` method. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.

13 Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

14 Write the byte array contents to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

"Quick Start: Invoking a service using base64 in a Microsoft .NET project" on page 222

Invoking a service using Java proxy classes and Base64 encoding

You can invoke an AEM Forms service using Java proxy classes and Base64. To invoke the `MyApplication/EncryptDocument` service using Java proxy classes, perform the following steps:

1 Create Java proxy classes using JAX-WS that consumes the `MyApplication/EncryptDocument` service WSDL. Use the following WSDL endpoint:

```
http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?WSDL&lc_version=9.0.1
```

Note: *Replace `hiro-xp` with the IP address of the J2EE application servier hosting AEM Forms.*

2 Package the Java proxy classes created using using JAX-WS into a JAR file.

3 Include the Java proxy JAR file and the JAR files located in the following path:

<Install Directory>\Adobe\Adobe_Experience_Manager_forms\sdk\client-libs\thirdparty

into your Java client project's class path.

4 Create a `MyApplicationEncryptDocumentService` object by using its constructor.

5 Create a `MyApplicationEncryptDocument` object by invoking the `MyApplicationEncryptDocumentService` object's `getEncryptDocument` method.

6 Set the connection values required to invoke AEM Forms by assigning values to the following data members:

- Assign the WSDL endpoint and the encoding type to the `javax.xml.ws.BindingProvider` object's `ENDPOINT_ADDRESS_PROPERTY` field. To invoke the `MyApplication/EncryptDocument` service using Base64 encoding, specify the following URL value:

  ```
  http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=base64
  ```

- Assign the AEM forms user to the `javax.xml.ws.BindingProvider` object's `USERNAME_PROPERTY` field.
- Assign the corresponding password value to the `javax.xml.ws.BindingProvider` object's `PASSWORD_PROPERTY` field.

The following code example shows this application logic:

```
//Set connection values required to invoke AEM Forms
String url = "http://hiro-
xp:8080/soap/services/MyApplication/EncryptDocument?blob=base64";
String username = "administrator";
String password = "password";
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.USERNAME_PROPERTY, username);
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, password);
```

7 Retrieve the PDF document to send to the `MyApplication/EncryptDocument` process by creating a `java.io.FileInputStream` object by using its constructor. Pass a string value that specifies the location of the PDF document.

8 Create a byte array and populate it with the contents of the `java.io.FileInputStream` object.

9 Create a `BLOB` object by using its constructor.

10 Populate the `BLOB` object by invoking its `setBinaryData` method and passing the byte array. The `BLOB` object's `setBinaryData` is the method to call when using Base64 encoding. (See .)

11 Invoke the `MyApplication/EncryptDocument` process by invoking the `MyApplicationEncryptDocument` object's `invoke` method. Pass the `BLOB` object that contains the PDF document. The invoke method returns a `BLOB` object that contains the encrypted PDF document.

12 Create a byte array that contains the encrypted PDF document by invoking the `BLOB` object's `getBinaryData` method.

13 Save the encrypted PDF document as a PDF file. Write the byte array to a file.

See also

"Quick Start: Invoking a service using Java proxy files and Base64 encoding" on page 223

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Invoking AEM Forms using MTOM

You can invoke AEM Forms services by using the web service standard MTOM. This standard defines how binary data, such as a PDF document, is transmitted over the Internet or intranet. A feature of MTOM is the use of the `XOP:Include` element. This element is defined in the XML Binary Optimized Packaging (XOP) specification to reference the binary attachments of a SOAP message.

The discussion here is about using MTOM to invoke the following AEM Forms short-lived process named `MyApplication/EncryptDocument`.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code example, create a process named* `MyApplication/EncryptDocument` *using Workbench. (See Using Workbench.)*

When this process is invoked, it performs the following actions:

1 Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.

2 Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

Note: *MTOM support was added in AEM Forms, version 9.*

Important: *JAX WS-based applications that use the MTOM transmission protocol are limited to 25MB of sent and received data. This limitation is due to a bug in JAX-WS. If the combined size of your sent and received files exceeds 25MB, use the the SwaRef transmission protocol instead of the MTOM one. Otherwise, there is a possibility of an* `OutOfMemory` *exception.*

The discussion here is about using MTOM withthin a Microsoft .NET project to invoke AEM Forms services. The .NET framework used is 3.5, and the development environment is Visual Studio 2008. If you have Web Service Enhancements (WSE) installed on your development computer, remove it. The .NET 3.5 framework supports a SOAP framework named Windows Communication Foundation (WCF). When invoking AEM Forms by using MTOM, only WCF (not WSE) is supported.

Creating a .NET project that invokes a service using MTOM

You can create a Microsoft .NET project that can invoke a AEM Forms service using web services. First, create a Microsoft .NET project by using Visual Studio 2008. To invoke a AEM Forms service, create a Service Reference to the AEM Forms service that you want to invoke within your project. When you create a Service Reference, specify a URL to the AEM Forms service:

```
http://localhost:8080/soap/services/MyApplication/EncryptDocument?WSDL&lc_version=9.0.1
```

Replace `localhost` with the IP address of the J2EE application server hosting AEM Forms. Replace `MyApplication/EncryptDocument` with the name of the AEM Forms service to invoke. For example, to invoke a Rights Management operation, specify:

```
http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1
```

The `lc_version` option ensures that AEM Forms functionality, such as MTOM, is available. Without specifying the `lc_version` option, you cannot invoke AEM Forms using MTOM.

After you create a Service Reference, data types associated with the AEM Forms service are available for use within your .NET project. To create a .NET project that invokes an AEM Forms service, perform the following steps:

1 Create a .NET project using Microsoft Visual Studio 2008.

2 In the **Project** menu, select **Add Service Reference**.

3 In the **Address** dialog box, specify the WSDL to the AEM Forms service. For example,

```
http://localhost:8080/soap/services/MyApplication/EncryptDocument?WSDL&lc_version=9.0.1
```

4 Click **Go** and then click **OK**.

Invoking a service using MTOM in a .NET project

Consider the `MyApplication/EncryptDocument` process that accepts an unsecured PDF document and returns a password-encrypted PDF document. To invoke the `MyApplication/EncryptDocument` process (which was built in Workbench) by using MTOM, perform the following steps:

1 Create a Microsoft .NET project.

2 Create a `MyApplication_EncryptDocumentClient` object by using its default constructor.

3 Create a `MyApplication_EncryptDocumentClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service and the encoding type:

```
http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=mtom
```

You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, ensure that you specify`?blob=mtom`.

Note: *Replace `hiro-xp` with the IP address of the J2EE application servier hosting AEM Forms.*

4 Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptDocumentClient.Endpoint.Binding` data member. Cast the return value to `BasicHttpBinding`.

5 Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` data member to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.

6 Enable basic HTTP authentication by performing the following tasks:

- Assign the AEM forms user name to the data member `MyApplication_EncryptDocumentClient.ClientCredentials.UserName.UserName`.
- Assign the corresponding password value to the data member `MyApplication_EncryptDocumentClient.ClientCredentials.UserName.Password`.
- Assign the constant value `HttpClientCredentialType.Basic` to the data member `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the data member `BasicHttpBindingSecurity.Security.Mode`.

The following code example shows these tasks.

```
//Enable BASIC HTTP authentication
encryptProcess.ClientCredentials.UserName.UserName = "administrator";
encryptProcess.ClientCredentials.UserName.Password = "password";
b.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
b.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
b.MaxReceivedMessageSize = 4000000;
b.MaxBufferSize = 4000000;
b.ReaderQuotas.MaxArrayLength = 4000000;
```

7 Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document to pass to the `MyApplication/EncryptDocument` process.

8 Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the PDF document and the mode in which to open the file.

9 Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.

10 Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.

11 Populate the `BLOB` object by assigning its `MTOM` data member with the contents of the byte array.

12 Invoke the `MyApplication/EncryptDocument` process by invoking the `MyApplication_EncryptDocumentClient` object's `invoke` method. Pass the `BLOB` object that contains the PDF document. This process returns an encrypted PDF document within a `BLOB` object.

13 Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the secured PDF document.

14 Create a byte array that stores the data content of the `BLOB` object that was returned by the `invoke` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.

15 Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

16 Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

Note: *Most AEM Forms service operations have a MTOM quick start. You can view these quick starts in a service's corresponding quick start section. For example, to see the Output quick start section, see "Output Service Java API Quick Start(SOAP)" on page 239.*

See also

"Quick Start: Invoking a service using MTOM in a .NET project" on page 236

"Accessing multiple services using web services" on page 548

"Creating an ASP.NET web application that invokes a human-centric long-lived process" on page 569

Invoking AEM Forms using SwaRef

You can invoke AEM Forms services using SwaRef. The content of the `wsi:swaRef` XML element is sent as an attachment inside a SOAP body that stores the reference to the attachment. When invoking a Forms service by using SwaRef, create Java proxy classes by using the Java API for XML Web Services (JAX-WS). (See Java API for XML Web Services.)

The discussion here is about invoking the following Forms short-lived process named `MyApplication/EncryptDocument` by using SwaRef.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code example, create a process named* `MyApplication/EncryptDocument` *using Workbench. (See Using Workbench.)*

When this process is invoked, it performs the following actions:

1 Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.

2 Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

Note: *SwaRef support added in AEM Forms*

The discussion below is about how to invoke Forms services by using SwaRef within a Java client application. The Java application uses proxy classes created by using JAX-WS.

Invoke a service using JAX-WS library files that use SwaRef

To invoke the `MyApplication/EncryptDocument` process by using Java proxy files created using JAX-WS and SwaRef, perform the following steps:

1 Create Java proxy classes using JAX-WS that consumes the `MyApplication/EncryptDocument` service WSDL. Use the following WSDL endpoint:

```
http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?WSDL&lc_version=9.0.1
```

For information, see "Creating Java proxy classes using JAX-WS" on page 521.

Note: *Replace* `hiro-xp` *with the IP address of the J2EE application server hosting AEM Forms.*

2 Package the Java proxy classes created using using JAX-WS into a JAR file.

3 Include the Java proxy JAR file and the JAR files located in the following path:

<Install Directory>\Adobe\Adobe_Experience_Manager_forms\sdk\client-libs\thirdparty

into your Java client project's class path.

4 Create a `MyApplicationEncryptDocumentService` object by using its constructor.

5 Create a `MyApplicationEncryptDocument` object by invoking the `MyApplicationEncryptDocumentService` object's `getEncryptDocument` method.

6 Set the connection values required to invoke AEM Forms by assigning values to the following data members:

- Assign the WSDL endpoint and the encoding type to the `javax.xml.ws.BindingProvider` object's `ENDPOINT_ADDRESS_PROPERTY` field. To invoke the `MyApplication/EncryptDocument` service using SwaRef encoding, specify the following URL value:

  ```
  http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=swaref
  ```

- Assign the AEM forms user to the `javax.xml.ws.BindingProvider` object's `USERNAME_PROPERTY` field.
- Assign the corresponding password value to the `javax.xml.ws.BindingProvider` object's `PASSWORD_PROPERTY` field.

The following code example shows this application logic:

```
//Set connection values required to invoke AEM Forms
String url = "http://hiro-
xp:8080/soap/services/MyApplication/EncryptDocument?blob=swaref";
String username = "administrator";
String password = "password";
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.USERNAME_PROPERTY, username);
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, password);
```

7 Retrieve the PDF document to send to the `MyApplication/EncryptDocument` process by creating a `java.io.File` object by using its constructor. Pass a string value that specifies the location of the PDF document.

8 Create a `javax.activation.DataSource` object by using the `FileDataSource` constructor. Pass the `java.io.File` object.

9 Create a `javax.activation.DataHandler` object by using its constructor and passing the `javax.activation.DataSource` object.

10 Create a `BLOB` object by using its constructor.

11 Populate the `BLOB` object by invoking its `setSwaRef` method and passing the `javax.activation.DataHandler` object.

12 Invoke the `MyApplication/EncryptDocument` process by invoking the `MyApplicationEncryptDocument` object's `invoke` method and passing the `BLOB` object that contains the PDF document. The invoke method returns a `BLOB` object that contains an encrypted PDF document.

13 Populate a `javax.activation.DataHandler` object by invoking the `BLOB` object's `getSwaRef` method.

14 Convert the `javax.activation.DataHandler` object to a `java.io.InputSteam` instance by invoking the `javax.activation.DataHandler` object's `getInputStream` method.

15 Write the `java.io.InputSteam` instance to a PDF file that represents the encrypted PDF document.

Note: *Most AEM Forms service operations have a SwaRef quick start. You can view these quick starts in a service's corresponding quick start section. For example, to see the Output quick start section, see "Output Service Java API Quick Start(SOAP)" on page 239.*

See also

"Quick Start: Invoking a service using SwaRef in a Java project" on page 238

Invoking AEM Forms using BLOB data over HTTP

You can invoke AEM Forms services using web services and passing BLOB data over HTTP. Passing BLOB data over HTTP is an alternative technique instead of using base64 encoding, DIME, or MIME. For example, you can pass data over HTTP in a Microsoft .NET project that uses Web Service Enhancement 3.0, which does not support DIME or MIME. When using BLOB data over HTTP, input data is uploaded before the AEM Forms service is invoked.

"Invoking AEM Forms using BLOB Data over HTTP" discusses invoking the following AEM Forms short-lived process named `MyApplication/EncryptDocument` by passing BLOB data over HTTP.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code example, create a process named `MyApplication/EncryptDocument` using Workbench. (See Using Workbench.)*

When this process is invoked, it performs the following actions:

1 Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.

2 Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

Note: *It is recommended that you be familiar with Invoking AEM Forms using SOAP. (See "Invoking AEM Forms using Web Services" on page 514.)*

Creating a .NET client assembly that uses data over HTTP

To create a client assembly that uses data over HTTP, follow the process specified in "Invoking AEM Forms using Base64 encoding" on page 525. However, amend the URL in the proxy class to include `?blob=http` instead of `?blob=base64`. This action ensures that data is passed over HTTP. In the proxy class, locate the following line of code:

```
"http://localhost:8080/soap/services/MyApplication/EncryptDocument";
```

and change it to:

```
"http://localhost:8080/soap/services/MyApplication/EncryptDocument?blob=http";
```

Referencing the .NET clienMyApplication/EncryptDocumentt assembly

Place your new .NET client assembly on the computer where you are developing your client application. After you place the .NET client assembly in a directory, you can reference it from a project. Reference the `System.Web.Services` library from your project. If you do not reference this library, you cannot use the .NET client assembly to invoke a service.

1 In the **Project** menu, select **Add Reference**.

2 Click the **.NET** tab.

3 Click **Browse** and locate the DocumentService.dll file.

4 Click **Select** and then click **OK**.

Invoking a service using a .NET client assembly that uses BLOB data over HTTP

You can invoke the `MyApplication/EncryptDocument` service (which was built in Workbench) using a .NET client assembly that uses data over HTTP. To invoke the `MyApplication/EncryptDocument` service, perform the following steps:

1 Create the .NET client assembly.

2 Reference the Microsoft .NET client assembly. Create a client Microsoft .NET project. Reference the Microsoft .NET client assembly in the client project. Also reference `System.Web.Services`.

3 Using the Microsoft .NET client assembly, create a `MyApplication_EncryptDocumentService` object by invoking its default constructor.

4 Set the `MyApplication_EncryptDocumentService` object's `Credentials` property with a `System.Net.NetworkCredential` object. Within the `System.Net.NetworkCredential` constructor, specify a AEM forms user name and the corresponding password. Set authentication values to enable your .NET client application to successfully exchange SOAP messages with AEM Forms.

5 Create a `BLOB` object by using its constructor. The `BLOB` object is used to pass data to the `MyApplication/EncryptDocument` process.

6 Assign a string value to the `BLOB` object's `remoteURL` data member that specifies the URI location of a PDF document to pass to the `MyApplication/EncryptDocument` service.

7 Invoke the `MyApplication/EncryptDocument` process by invoking the `MyApplication_EncryptDocumentService` object's `invoke` method and passing the `BLOB` object. This process returns an encrypted PDF document within a `BLOB` object.

8 Create a `System.UriBuilder` object by using its constructor and passing the value of the returned `BLOB` object's `remoteURL` data member.

9 Convert the `System.UriBuilder` object to a `System.IO.Stream` object. (The C# Quick Start that follows this list illustrates how to perform this task.)

10 Create a byte array and populate it with the data located in the `System.IO.Stream` object.

11 Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

12 Write the byte array contents to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

Invoking a service using Java proxy classes and BLOB data over HTTP

You can invoke an AEM Forms service using Java proxy classes and BLOB data over HTTP. To invoke the `MyApplication/EncryptDocument` service using Java proxy classes, perform the following steps:

1 Create Java proxy classes using JAX-WS that consumes the `MyApplication/EncryptDocument` service WSDL. Use the following WSDL endpoint:

```
http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?WSDL&lc_version=9.0.1
```

For information, see "Creating Java proxy classes using JAX-WS" on page 521.

Note: *Replace `hiro-xp` with the IP address of the J2EE application server hosting AEM Forms.*

2 Package the Java proxy classes created using using JAX-WS into a JAR file.

3 Include the Java proxy JAR file and the JAR files located in the following path:

<Install Directory>\Adobe\Adobe_Experience_Manager_forms\sdk\client-libs\thirdparty

into your Java client project's class path.

4 Create a `MyApplicationEncryptDocumentService` object by using its constructor.

5 Create a `MyApplicationEncryptDocument` object by invoking the `MyApplicationEncryptDocumentService` object's `getEncryptDocument` method.

6 Set the connection values required to invoke AEM Forms by assigning values to the following data members:

- Assign the WSDL endpoint and the encoding type to the `javax.xml.ws.BindingProvider` object's `ENDPOINT_ADDRESS_PROPERTY` field. To invoke the `MyApplication/EncryptDocument` service using BLOB over HTTP encoding, specify the following URL value:

  ```
  http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=http
  ```

- Assign the AEM forms user to the `javax.xml.ws.BindingProvider` object's `USERNAME_PROPERTY` field.
- Assign the corresponding password value to the `javax.xml.ws.BindingProvider` object's `PASSWORD_PROPERTY` field.

The following code example shows this application logic:

```
//Set connection values required to invoke AEM Forms
String url = "http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=http";
String username = "administrator";
String password = "password";
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.USERNAME_PROPERTY, username);
((BindingProvider)
encryptDocClient).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, password);
```

7 Create a `BLOB` object by using its constructor.

8 Populate the `BLOB` object by invoking its `setRemoteURL` method. Pass a string value that specifies the URI location of a PDF document to pass to the `MyApplication/EncryptDocument` service.

9 Invoke the `MyApplication/EncryptDocument` process by invoking the `MyApplicationEncryptDocument` object's `invoke` method and passing the `BLOB` object that contains the PDF document. This process returns an encrypted PDF document within a `BLOB` object.

10 Create a byte array to store the data stream that represents the encrypted PDF document. Invoke the `BLOB` object's `getRemoteURL` method (use the `BLOB` object returned by the `invoke` method).

11 Create a `java.io.File` object by using its constructor. This object represents the encrypted PDF document.

12 Create a `java.io.FileOutputStream` object by using its constructor and passing the `java.io.File` object.

13 Invoke the `java.io.FileOutputStream` object's `write` method. Pass the byte array that contains the data stream that represents the encrypted PDF document.

More Help topics

"Quick Start: Invoking a service using BLOB data over HTTP in a Java project" on page 232

"Quick Start: Invoking a service using BLOB data over HTTP in a .NET project" on page 234

Invoking AEM Forms using DIME

You can invoke AEM Forms services using SOAP with attachments. AEM Forms supports both MIME and DIME web service standards. DIME lets you send binary attachments, such as PDF documents, along with invocation requests instead of encoding the attachment. The *Invoking AEM Forms using DIME* section discusses invoking the following AEM Forms short-lived process named `MyApplication/EncryptDocument` using DIME.

When this process is invoked, it performs the following actions:

1 Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.

2 Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

This process is not based on an existing AEM Forms process. To follow along with the code examples, create a process named `MyApplication/EncryptDocument`using Workbench. (See Using Workbench.)

Note: *Invoking AEM Forms service operations using DIME is deprecated. It is recommended that you use MTOM. (See "Invoking AEM Forms using MTOM" on page 529.)*

"Creating a .NET project that uses DIME" on page 537
"Creating Apache Axis Java proxy classes that use DIME" on page 539

Creating a .NET project that uses DIME

To create a .NET project that can invoke a Forms service using DIME, perform the following tasks:

- Install Web Services Enhancements 2.0 on your development computer.
- From within your .NET project, create a web reference to the FormsAEM Forms service.

Installing Web Services Enhancements 2.0

Install Web Services Enhancements 2.0 on your development computer and integrate it with Microsoft Visual Studio .NET. You can download Web Services Enhancements 2.0 from the Microsoft Download Center.

From this web page, search for Web Services Enhancements 2.0 and download it onto your development computer. This download places a file named Microsoft WSE 2.0 SPI.msi on your computer. Run the installation program and follow the online directions.

Note: *Web Services Enhancements 2.0 supports DIME. The supported version of Microsoft Visual Studio is 2003 when working with Web Services Enhancements 2.0. Web Services Enhancements 3.0 does not support DIME; however, it supports MTOM.*

Creating a web reference to an AEM Forms service

After you install Web Services Enhancements 2.0 on your development computer and create a Microsoft .NET project, create a web reference to the Forms service. For example, to create a web reference to the `MyApplication/EncryptDocument` process and assuming that Forms is installed on the local computer, specify the following URL:

```
http://localhost:8080/soap/services/MyApplication/EncryptDocument?WSDL
```

After you create a web reference, the following two proxy data types are available for you to use within your .NET project: `EncryptDocumentService` and `EncryptDocumentServiceWse`. To invoke the `MyApplication/EncryptDocument` process using DIME, use the `EncryptDocumentServiceWse` type.

Note: *Before creating a web reference to the Forms service, ensure that you reference Web Services Enhancements 2.0 in your project. (See "Installing Web Services Enhancements 2.0".)*

Reference the WSE library

1. In the Project menu, select Add Reference.
2. In the Add Reference dialog box, select Microsoft.Web.Services2.dll.
3. Select System.Web.Services.dll.
4. Click Select and then click OK.

Create a web reference to a Forms service

1. In the Project menu, select Add Web Reference.
2. In the URL dialog box, specify the URL to the Forms service.
3. Click Go and then click Add Reference.

Note: *Ensure that you enable your .NET project to use the WSE library. From within the Project Explorer, right-click the project name and select Enable WSE 2.0. Ensure that the check box on the dialog box that appears is selected.*

Invoking a service using DIME in a .NET project

You can invoke a Forms service using DIME. Consider the `MyApplication/EncryptDocument` process that accepts an unsecured PDF document and returns a password-encrypted PDF document. To invoke the `MyApplication/EncryptDocument` process using DIME, perform the following steps:

1. Create a Microsoft .NET project that enables you to invoke a Forms service using DIME. Ensure that you include Web Services Enhancements 2.0 and create a web reference to the AEM Forms service.
2. After setting a web reference to the `MyApplication/EncryptDocument` process, create an `EncryptDocumentServiceWse` object by using its default constructor.
3. Set the `EncryptDocumentServiceWse` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the AEM forms user name and password value.
4. Create a `Microsoft.Web.Services2.Dime.DimeAttachment` object by using its constructor and passing the following values:
   - A string value that specifies a GUID value. You can obtain a GUID value by invoking the `System.Guid.NewGuid.ToString` method.
   - A string value that specifies the content type. Because this process requires a PDF document, specify `application/pdf`.
   - A `TypeFormat` enumeration value. Specify `TypeFormat.MediaType`.
   - A string value that specifies the location of the PDF document to pass to the AEM Forms process.
5. Create a `BLOB` object by using its constructor.
6. Add the DIME attachment to the `BLOB` object by assigning the `Microsoft.Web.Services2.Dime.DimeAttachment` object's `Id` data member value to the `BLOB` object's `attachmentID` data member.
7. Invoke the `EncryptDocumentServiceWse.RequestSoapContext.Attachments.Add` method and pass the `Microsoft.Web.Services2.Dime.DimeAttachment` object.
8. Invoke the `MyApplication/EncryptDocument` process by invoking the `EncryptDocumentServiceWse` object's `invoke` method and passing the `BLOB` object that contains the DIME attachment. This process returns an encrypted PDF document within a `BLOB` object.
9. Obtain the attachment identifier value by getting the value of the returned `BLOB` object's `attachmentID` data member.
10. Iterate through the attachments located in `EncryptDocumentServiceWse.ResponseSoapContext.Attachments` and use the attachment identifier value to obtain the encrypted PDF document.
11. Obtain a `System.IO.Stream` object by getting the value of the `Attachment` object's `Stream` data member.
12. Create a byte array and pass that byte array to the `System.IO.Stream` object's `Read` method. This method populates the byte array with a data stream that represents the encrypted PDF document.
13. Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents a PDF file location. This object represents the encrypted PDF document.
14. Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
15. Write the contents of the byte array to the PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

"Quick Start: Invoking a service using DIME in a .NET project" on page 228

Creating Apache Axis Java proxy classes that use DIME

You can use the Apache Axis WSDL2Java tool to convert a service WSDL into Java proxy classes so that you can invoke service operations. Using Apache Ant, you can generate Axis library files from a AEM Forms service WSDL that lets you invoke the service. (See "Creating Java proxy classes using Apache Axis" on page 523.)

The Apache Axis WSDL2Java tool generates JAVA files that contain methods that are used to send SOAP requests to a service. SOAP requests received by a service are decoded by the Axis-generated libraries and turned back into the methods and arguments.

To invoke the `MyApplication/EncryptDocument` service (which was built in Workbench) using Axis-generated library files and DIME, perform the following steps:

1. Create Java proxy classes that consume the `MyApplication/EncryptDocument` service WSDL using Apache Axis. (See "Creating Java proxy classes using Apache Axis" on page 523.)
2. Include the Java proxy classes into your class path.
3. Create a `MyApplicationEncryptDocumentServiceLocator` object by using its constructor.
4. Create a `URL` object by using its constructor and passing a string value that specifies the AEM Forms service WSDL definition. Ensure that you specify `?blob=dime` at the end of the SOAP endpoint URL. For example, use

   `http://hiro-xp:8080/soap/services/MyApplication/EncryptDocument?blob=dime.`
5. Create an `EncryptDocumentSoapBindingStub` object by invoking its constructor and passing the `MyApplicationEncryptDocumentServiceLocator` object and the `URL` object.
6. Set the AEM forms user name and password value by invoking the `EncryptDocumentSoapBindingStub` object's `setUsername` and `setPassword` methods.

   ```
   encryptionClientStub.setUsername("administrator");
   encryptionClientStub.setPassword("password");
   ```
7. Retrieve the PDF document to send to the `MyApplication/EncryptDocument` service by creating a `java.io.File` object. Pass a string value that specifies the PDF document location.
8. Create a `javax.activation.DataHandler` object by using its constructor and passing a `javax.activation.FileDataSource` object. The `javax.activation.FileDataSource` object can be created by using its constructor and passing the `java.io.File` object that represents the PDF document.
9. Create an `org.apache.axis.attachments.AttachmentPart` object by using its constructor and passing the `javax.activation.DataHandler` object.
10. Attach the attachment by invoking the `EncryptDocumentSoapBindingStub` object's `addAttachment` method and passing the `org.apache.axis.attachments.AttachmentPart` object.
11. Create a `BLOB` object by using its constructor. Populate the `BLOB` object with the attachment identifier value by invoking the `BLOB` object's `setAttachmentID` method and passing the attachment identifier value. This value can be obtained by invoking the `org.apache.axis.attachments.AttachmentPart` object's `getContentId` method.
12. Invoke the `MyApplication/EncryptDocument` process by invoking the `EncryptDocumentSoapBindingStub` object's `invoke` method. Pass the `BLOB` object that contains the DIME attachment. This process returns an encrypted PDF document within a `BLOB` object.
13. Obtain the attachment identifier value by invoking the returned `BLOB` object's `getAttachmentID` method. This method returns a string value that represents the identifier value of the returned attachment.
14. Retrieve the attachments by invoking the `EncryptDocumentSoapBindingStub` object's `getAttachments` method. This method returns an array of `Objects` that represent the attachments.
15. Iterate through the attachments (the `Object` array) and use the attachment identifier value to obtain the encrypted PDF document. Each element is an `org.apache.axis.attachments.AttachmentPart` object.

16 Obtain the `javax.activation.DataHandler` object associated with the attachment by invoking the `org.apache.axis.attachments.AttachmentPart` object's `getDataHandler` method.

17 Obtain a `java.io.FileStream` object by invoking the `javax.activation.DataHandler` object's `getInputStream` method.

18 Create a byte array and pass that byte array to the `java.io.FileStream` object's `read` method. This method populates the byte array with a data stream that represents the encrypted PDF document.

19 Create a `java.io.File` object by using its constructor. This object represents the encrypted PDF document.

20 Create a `java.io.FileOutputStream` object by using its constructor and passing the `java.io.File` object.

21 Invoke the `java.io.FileOutputStream` object's `write` method and pass the byte array that contains the data stream that represents the encrypted PDF document.

See also

"Quick Start: Invoking a service using DIME in a Java project" on page 230

Using SAML-based authentication

AEM Forms supports various web service authentication modes when invoking services. One authentication mode is specifying both a user name and password value using a basic authorization header in the web service call. AEM Forms also supports SAML assertion-based authentication. When a client application invokes an AEM Forms service using a web service, the client application can provide authentication information in one of the following ways:

- Passing credentials as part of Basic Authorization
- Passing username token as part of WS-Security header
- Passing a SAML assertion as part of WS-Security header
- Passing Kerberos token as part of WS-Security header

AEM Forms does not support standard certificate-based authentication but it does support certificate-based authentication in a different form.

Note: *The web service quick starts in Programming with AEM Forms specify user name and password values to perform authorization.*

The identity of AEM forms users can be represented through a SAML assertion signed using a secret key. The following XML code shows an example of a SAML assertion.

```
<Assertion xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
    xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol"
    AssertionID="fd4bd0c87302780e0d9bbfa8726d5bc0" IssueInstant="2008-04-17T13:47:00.720Z"
Issuer="LiveCycle"
    MajorVersion="1" MinorVersion="1">
    <Conditions NotBefore="2008-04-17T13:47:00.720Z" NotOnOrAfter="2008-04-17T15:47:00.720Z">
    </Conditions>
    <AuthenticationStatement
        AuthenticationInstant="2008-04-17T13:47:00.720Z"
        AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:unspecified">
        <Subject>
            <NameIdentifier NameQualifier="DefaultDom">administrator</NameIdentifier>
            <SubjectConfirmation>
                <ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:sender-
vouches</ConfirmationMethod>
            </SubjectConfirmation>
        </Subject>
    </AuthenticationStatement>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"></ds:CanonicalizationMethod>
            <ds:SignatureMethodAlgorithm="http://www.w3.org/2000/09/xmldsig#hmac-
sha1"></ds:SignatureMethod>
            <ds:Reference URI="#fd4bd0c87302780e0d9bbfa8726d5bc0">
                <ds:Transforms>
                    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"></ds:Transform>
                    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                        <ec:InclusiveNamespacesxmlns:ec="http://www.w3.org/2001/10/xml-exc-
c14n#"
                            PrefixList="code ds kind rw saml samlp typens #default">
                        </ec:InclusiveNamespaces>
                    </ds:Transform>
                </ds:Transforms>
                <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></ds:DigestMethod>
                <ds:DigestValue>hVrtqjWr+VzaVUIpQx0YI9lIjaY=</ds:DigestValue>
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>UMbBb+cUcPtcWDCIhXes4n4FxfU=</ds:SignatureValue>
    </ds:Signature>
</Assertion>
```

This example assertion is issued for an administrator user. This assertion contains the following noticeable items:

- It is valid for certain duration.
- It is issued for a particular user.
- It is digitally signed. So any modification done to it would break the signature.
- It can be presented to AEM Forms as a token of user's identity similar to user name and password.

A client application can retrieve the assertion from any AEM Forms AuthenticationManager API which returns an `AuthResult` object. You can obtain an `AuthResult` instance by performing one of the following two methods:

- Authenticating the user using any of the authenticate methods exposed by AuthenticationManager API. Typically, one would use the user name and password; however, you can also use the certificate authentication.
- Using the `AuthenticationManager.getAuthResultOnBehalfOfUser` method. This method lets a client application get an `AuthResult` object for any AEM forms user.

a AEM forms user can be authenticated using a SAML token that is obtained. This SAML assertion (xml fragment) can be send as part of the WS-Security header with the web service call for user authentication. Typically, a client application has authenticated a user but has not stored the user credentials. (Or the user has logged on to that client through a mechanism other than using a user name and password.) In this situation, the client application has to invoke AEM Forms and impersonate a specific user which is allowed to invoke AEM Forms.

To impersonate a specific user, invoke the `AuthenticationManager.getAuthResultOnBehalfOfUser` method using a web service. This method returns an `AuthResult` instance which contains the SAML assertion for that user.

Next, use that SAML assertion to invoke any service that requires authentication. This action involves sending the assertion as part of the SOAP header. When a web service call is made with this assertion, AEM Forms identifies the user as the one represented by that assertion. That is, the user specified in the assertion is the user who is invoking the service.

Using Apache Axis classes and SAML-based authentication

You can invoke an AEM Forms service by Java proxy classes that were created using the Axis library. (See "Creating Java proxy classes using Apache Axis" on page 523.)

When using AXIS that uses SAML-based authentication, register the request and response handler with Axis. Apache Axis invokes the handler before sending an invocation request to AEM Forms. To register a handler, create a Java class that extends `org.apache.axis.handlers.BasicHandler`.

Create an AssertionHandler with Axis

The following Java class, named `AssertionHandler.java`, shows an example of a Java class that extends `org.apache.axis.handlers.BasicHandler`.

```
public class AssertionHandler extends BasicHandler {
       public void invoke(MessageContext ctx) throws AxisFault {
           String assertion = (String) ctx.getProperty(LC_ASSERTION);

           //no assertion hence nothing to insert
           if(assertion == null) return;

           try {
               MessageElement samlElement = new MessageElement(convertToXML(assertion));
               SOAPHeader header = (SOAPHeader) ctx.getRequestMessage().getSOAPHeader();
               //Create the wsse:Security element which would contain the SAML element
               SOAPElement wsseHeader = header.addChildElement("Security", "wsse", WSSE_NS);
               wsseHeader.appendChild(samlElement);
              //remove the actor attribute as in LC we do not specify any actor. This would not
remove the actor attribute though
               //it would only remove it from the soapenv namespace
               wsseHeader.getAttributes().removeNamedItem("actor");
           } catch (SOAPException e) {
            throw new AxisFault("Error occured while adding the assertion to the SOAP Header",e);
           }
       }
}
```

Register the handler

To register a handler with Axis, create a client-config.wsdd file. By default, Axis looks for a file with this name. The following XML code is an example of a client-config.wsdd file. See Axis documentation for more information.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <transport name="http" pivot="java:org.apache.axis.transport.http.HTTPSender"/>
     <globalConfiguration >
      <requestFlow >
       <handler type="java:com.adobe.idp.um.example.AssertionHandler" />
      </requestFlow >
     </globalConfiguration >
</deployment>
```

Invoke an AEM Forms service

The following code example invokes an AEM Forms service using SAML-based authentication.

```
public class ImpersonationExample {
       . . .
       public void  authenticateOnBehalf(String superUsername,String password,
               String canonicalName,String domainName) throws UMException, RemoteException{
           ((org.apache.axis.client.Stub) authenticationManager).setUsername(superUsername);
           ((org.apache.axis.client.Stub) authenticationManager).setPassword(password);

           //Step 1 - Invoke the Auth manager api to get an assertion for the user to be
impersonated
           AuthResult ar = authenticationManager.getAuthResultOnBehalfOfUser(canonicalName,
domainName, null);
           String assertion = ar.getAssertion();
           //Step 2 - Setting the assertion here to be picked later by the AssertionHandler.
Note that stubs are not threadSafe
           //hence should not be reused. For this simple example we have made them instance
variable but care should be taken
           //regarding the thread safety
           ((javax.xml.rpc.Stub)
authorizationManager)._setProperty(AssertionHandler.LC_ASSERTION, assertion);
       }

       public Role findRole(String roleId) throws UMException, RemoteException{
           //This api would be invoked under bob's user rights
           return authorizationManager.findRole(roleId);
       }

       public static void main(String[] args) throws Exception {
           ImpersonationExample ie = new ImpersonationExample("http://localhost:5555");
           //Get the SAML assertion for the user to impersonate and store it in stub
           ie.authenticateOnBehalf(
               "administrator", //The Super user which has the required impersonation permission
                   "password", // Password of the super user as referred above
                   "bob", //Cannonical name of the user to impersonate
                   "testdomain" //Domain of the user to impersonate
                   );

           Role r = ie.findRole("BASIC_ROLE_ADMINISTRATOR");
           System.out.println("Role "+r.getName());
       }
}
```

Using a .NET client assembly and SAML-based authentication

You can invoke a Forms service by using a .NET client assembly and SAML-based authentication. To do so, you must use the Web Service Enhancements 3.0 (WSE). For information about creating a .NET client assembly that uses WSE, see "Creating a .NET project that uses DIME" on page 537.

Note: *The DIME section uses WSE 2.0. To use SAML-based authentication, follow the same instructions that are specified in the DIME topic. However, replace WSE 2.0 with WSE 3.0. Install Web Services Enhancements 3.0 on your development computer and integrate it with Microsoft Visual Studio .NET. You can download Web Services Enhancements 3.0 from the Microsoft Download Center.*

The WSE architecture uses Policies, Assertions, and SecurityToken data types. Briefly, for a web service call, specify a policy. A policy can have multiple assertions. Each assertion can contain filters. A filter is invoked at certain stages in a web service call and, at that time, they can modify the SOAP request. For full details, see the Web Service Enhancements 3.0 documentation.

Create the Assertion and Filter

The following C# code example creates filter and assertion classes. This code example creates a SamlAssertionOutputFilter. This filter is invoked by the WSE framework before the SOAP request is sent to AEM Forms.

```
class LCSamlPolicyAssertion : Microsoft.Web.ServicES4.Design.PolicyAssertion
{
       public override Microsoft.Web.ServicES4.SoapFilter
CreateClientOutputFilter(FilterCreationContext context)
       {
          return new SamlAssertionOutputFilter();
       }
       . . .
}


class SamlAssertionOutputFilter : SendSecurityFilter
{
       public override void SecureMessage(SoapEnvelope envelope, Security security)
       {
          // Get the SamlToken from the SessionState
          SamlToken samlToken =
envelope.Context.Credentials.UltimateReceiver.GetClientToken<SamlToken>();
          security.Tokens.Add(samlToken);
       }
}
```

Create the SAML Token

Create a class to represent the SAML assertion. The main task that this class performs is convert data values from string to xml and preserve white space. This assertion xml is later imported into the SOAP request.

```
class SamlToken : SecurityToken
{
       public const string SAMLAssertion = "http://docs.oasis-open.org/wss/oasis-wss-saml-
token-profile-1.1#SAMLV1.1";
       private XmlElement _assertionElement;

       public SamlToken(string assertion)
            : base(SAMLAssertion)
       {
          XmlDocument xmlDoc = new XmlDocument();
          //The white space has to be preserved else the digital signature would get broken
          xmlDoc.PreserveWhitespace = true;
          xmlDoc.LoadXml(assertion);
          _assertionElement = xmlDoc.DocumentElement;
        }

        public override XmlElement GetXml(XmlDocument document)
        {
           return (XmlElement)document.ImportNode(_assertionElement, true);
        }
       . . .
}
```

Invoke an AEM Forms service

The following C# code example invokes a Forms service by using SAML-based authentication.

```
public class ImpersonationExample
{
       . . .
       public void AuthenticateOnBehalf(string superUsername, string password, string
canonicalName, string domainName)
       {
           //Create a policy for UsernamePassword Token
           Policy usernamePasswordPolicy = new Policy();
           usernamePasswordPolicy.Assertions.Add(new UsernameOverTransportAssertion());

           UsernameToken token = new UsernameToken(superUsername, password,
PasswordOption.SendPlainText);
           authenticationManager.SetClientCredential(token);
           authenticationManager.SetPolicy(usernamePasswordPolicy);

           //Get the SAML assertion for impersonated user
           AuthClient.AuthenticationManagerService.AuthResult ar
               = authenticationManager.getAuthResultOnBehalfOfUser(canonicalName, domainName,
null);
           System.Console.WriteLine("Received assertion " + ar.assertion);

           //Create a policy for inserting SAML assertion
           Policy samlPolicy = new Policy();
           samlPolicy.Assertions.Add(new LCSamlPolicyAssertion());
           authorizationManager.SetPolicy(samlPolicy);
           //Set the SAML assertion obtained previously as the token
           authorizationManager.SetClientCredential(new SamlToken(ar.assertion));
       }

       public Role findRole(string roleId)
       {
           return authorizationManager.findRole(roleId);
       }

       static void Main(string[] args)
       {
           ImpersonationExample ie = new ImpersonationExample("http://localhost:5555");
           ie.AuthenticateOnBehalf(
                "administrator", //The Super user which has the required impersonation permission
                "password", // Password of the super user as referred above
                "bob", //Cannonical name of the user to impersonate
                "testdomain" //Domain of the user to impersonate
                );

        Role r = ie.findRole("BASIC_ROLE_ADMINISTRATOR");
           System.Console.WriteLine("Role "+r.name);
    }
}
```

Related considerations when using web services

Sometimes issues occur when invoking certain AEM Forms services operations by using web services. The objective of this discussion is to identify those issues and provide a solution, if one is available.

Invoking service operations asynchronously

If you attempt to asynchronously invoke an AEM Forms service operation, such as the Generate PDF's `htmlToPDF` operation, a `SoapFaultException` occurs. To resolve this issue, create a custom-binding XML file that maps the `ExportPDF_Result` element and other elements into different classes. The following XML represents a custom binding file.

```
<bindings
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:jxb="http://java.sun.com/xml/ns/jaxb" jxb:version="1.0"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

wsdlLocation="http://localhost:8080/soap/services/GeneratePDFService?wsdl&amp;async=true&amp
;lc_version=9.0.0"
       xmlns="http://java.sun.com/xml/ns/jaxws">
       <enableAsyncMapping>false</enableAsyncMapping>
       <package name="external_customize.client"/>
       <enableWrapperStyle>true</enableWrapperStyle>
       <bindings
node="/wsdl:definitions/wsdl:types/xsd:schema[@targetNamespace='http://adobe.com/idp/service
s']/xsd:element[@name='ExportPDF_Result']">
           <jxb:class name="ExportPDFAsyncResult">
           </jxb:class>
       </bindings>
       <bindings
node="/wsdl:definitions/wsdl:types/xsd:schema[@targetNamespace='http://adobe.com/idp/service
s']/xsd:element[@name='CreatePDF_Result']">
           <jxb:class name="CreatePDFAsyncResult">
           </jxb:class>
       </bindings>
       <bindings
node="/wsdl:definitions/wsdl:types/xsd:schema[@targetNamespace='http://adobe.com/idp/service
s']/xsd:element[@name='HtmlToPDF_Result']">
           <jxb:class name="HtmlToPDFAsyncResult">
           </jxb:class>
       </bindings>
       <bindings
node="/wsdl:definitions/wsdl:types/xsd:schema[@targetNamespace='http://adobe.com/idp/service
s']/xsd:element[@name='OptimizePDF_Result']">
           <jxb:class name="OptimizePDFAsyncResult">
           </jxb:class>
       </bindings>
       <!--bindings
node="//wsdl:portType[@name='GeneratePDFService']/wsdl:operation[@name='HtmlToPDF_Result']">
           <jxb:class name="HtmlToPDFAsyncResult"/>
       </bindings-->
</bindings>
```

Use this XML file when creating Java proxy files by using JAX-WS. (See "Creating Java proxy classes using JAX-WS" on page 521.)

Reference this XML file when executing the JAX-WS tool (wsimport.exe) by using the `-b` command line option. Update the `wsdlLocation` element in the binding XML file to specify the URL of AEM Forms.

To ensure that asynchronous invocation works, modify the end point URL value and specify `async=true`. For example, for Java proxy files that are created with JAX-WS, specify the following for the `BindingProvider.ENDPOINT_ADDRESS_PROPERTY`.

```
http://server:port/soap/services/ServiceName?wsdl&async=true&lc_version=9.0.0
```

The following list specifies other services that need a custom binding file when invoked asynchronously:

- PDFG3D
- Task Manager
- Application Manager
- Directory Manager
- Distiller
- Rights Management
- Document Management

Differences in J2EE application servers

Sometimes a proxy library created using a specific J2EE application server does not successfully invoke AEM Forms that is hosted on a different J2EE application server. Consider a proxy library that is generated using AEM Forms that is deployed on WebSphere. This proxy library cannot successfully invoke AEM Forms services that are deployed on the JBoss Application Server.

Some AEM Forms complex data types, such as `PrincipalReference`, are defined differently when AEM Forms is deployed on WebSphere as compared to the JBoss Application Server. Differences in the JDKs used by the different J2EE application services are the reason why there are differences in WSDL definitions. As a result, use proxy libraries that are generated from the same J2EE application server.

Accessing multiple services using web services

Due to namespace conflicts, data objects cannot be shared between multiple service WSDLs. Different services can share data types and, therefore the services share the definition of these types in the WSDLs. For example, you cannot add two .NET client assemblies that contain a `BLOB` data type to the same .NET client project. If you attempt to do so, a compile error occurs.

The following list specifies data types that cannot be shared between multiple service WSDLs:

- `User`
- `Principals`
- `PrincipalReference`
- `Groups`
- `Roles`
- `BLOB`

To avoid this problem, it is recommended that you fully-qualify the data types. For example, consider a .NET application that references both the Forms service and Signature service using a service reference. Both service references will contain a `BLOB` class. To use a `BLOB` instance, fully-qualify the `BLOB` object when you declare it. This approach is shown in the following code example. For information about this code exampe, see "Digitally Signing Interactive Forms" on page 898.

The following C# code example signs an interactive form that is rendered by the Forms service. The client application has two service references. The `BLOB` instance that is associated with the Forms service belongs to the `SignInteractiveForm.ServiceReference2` namespace. Likewise, the `BLOB` instance that is associated with the Signature service belongs to the `SignInteractiveForm.ServiceReference1` namespace. The signed interactive form is saved as a PDF file named *LoanXFASigned.pdf*.

```
???/**
    * Ensure that you create a .NET project that uses
    * MS Visual Studio 2008 and version 3.5 of the .NET
    * framework. This is required to invoke a
    * AEM Forms service using MTOM.
    *
    * For information, see "Invoking AEM Forms using MTOM" in Programming with AEM forms
    */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using System.IO;

//A reference to the Signature service
using SignInteractiveForm.ServiceReference1;

//A reference to the Forms service
using SignInteractiveForm.ServiceReference2;

namespace SignInteractiveForm
{
       class Program
       {
           static void Main(string[] args)
           {
               try
               {
                   //Because BLOB objects are used in both service references
                   //it is necessary to fully-qualify the BLOB objects

                   //Retrieve the form -- invoke the Forms service
                   SignInteractiveForm.ServiceReference2.BLOB formData = GetForm();

                   //Create a BLOB object associated with the Signature service
                   SignInteractiveForm.ServiceReference1.BLOB sigData = new
SignInteractiveForm.ServiceReference1.BLOB();

                   //Transfer the byte stream from one Forms BLOB object to the
                   //Signature BLOB object
                   sigData.MTOM = formData.MTOM;

                   //Sign the Form -- invoke the Signature service
                   SignForm(sigData);
               }
               catch (Exception ee)
               {
                   Console.WriteLine(ee.Message);
               }
           }

           //Creates an interactive PDF form based on a XFA form - invoke the Forms service
           private static SignInteractiveForm.ServiceReference2.BLOB GetForm()
           {

               try
```

```
            {
                //Create a FormsServiceClient object
                FormsServiceClient formsClient = new FormsServiceClient();
                formsClient.Endpoint.Address = new
System.ServiceModel.EndpointAddress("http://hiro-
xp:8080/soap/services/FormsService?blob=mtom");

                //Enable BASIC HTTP authentication
                BasicHttpBinding b = (BasicHttpBinding)formsClient.Endpoint.Binding;
                b.MessageEncoding = WSMessageEncoding.Mtom;
                formsClient.ClientCredentials.UserName.UserName = "administrator";
                formsClient.ClientCredentials.UserName.Password = "password";
                b.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
                b.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
                b.MaxReceivedMessageSize = 2000000;
                b.MaxBufferSize = 2000000;
                b.ReaderQuotas.MaxArrayLength = 2000000;

                //Create a BLOB to store form data
                SignInteractiveForm.ServiceReference2.BLOB formData = new
SignInteractiveForm.ServiceReference2.BLOB();
                SignInteractiveForm.ServiceReference2.BLOB pdfForm = new
SignInteractiveForm.ServiceReference2.BLOB();

                //Specify a XML form data
                string path = "C:\\Adobe\Loan.xml";
                FileStream fs = new FileStream(path, FileMode.Open);

                //Get the length of the file stream
                int len = (int)fs.Length;
                byte[] ByteArray = new byte[len];

                fs.Read(ByteArray, 0, len);
                formData.MTOM = ByteArray;

                //Specify a XML form data
                string path2 = "C:\\Adobe\LoanSigXFA.pdf";
                FileStream fs2 = new FileStream(path2, FileMode.Open);

                //Get the length of the file stream
                int len2 = (int)fs2.Length;
                byte[] ByteArray2 = new byte[len2];

                fs2.Read(ByteArray2, 0, len2);
                pdfForm.MTOM = ByteArray2;

                PDFFormRenderSpec renderSpec = new PDFFormRenderSpec();
                renderSpec.generateServerAppearance = true;

                //Set out parameter values
                long pageCount = 1;
                String localValue = "en_US";
                FormsResult result = new FormsResult();

                //Render an interactive PDF form
                formsClient.renderPDFForm2(
                    pdfForm,
```

```
                    formData,
                    renderSpec,
                    null,
                    null,
                    out pageCount,
                    out localValue,
                    out result);

                //Write the data stream to the BLOB object
                SignInteractiveForm.ServiceReference2.BLOB outForm = result.outputContent;
                return outForm;
            }
            catch (Exception ee)
            {
                Console.WriteLine(ee.Message);
            }
            return null;
        }

        //Sign the form -- invoke the Signature service
        private static void SignForm(SignInteractiveForm.ServiceReference1.BLOB inDoc)
        {

            try
            {
                //Create a SignatureServiceClient object
                SignatureServiceClient signatureClient = new SignatureServiceClient();
                signatureClient.Endpoint.Address = new
System.ServiceModel.EndpointAddress("http://hiro-
xp:8080/soap/services/SignatureService?blob=mtom");

                //Enable BASIC HTTP authentication
                BasicHttpBinding b = (BasicHttpBinding)signatureClient.Endpoint.Binding;
                b.MessageEncoding = WSMessageEncoding.Mtom;
                signatureClient.ClientCredentials.UserName.UserName = "administrator";
                signatureClient.ClientCredentials.UserName.Password = "password";
                b.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
                b.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
                b.MaxReceivedMessageSize = 2000000;
                b.MaxBufferSize = 2000000;
                b.ReaderQuotas.MaxArrayLength = 2000000;

                //Specify the name of the signature field
              string fieldName = "form1[0].grantApplication[0].page1[0].SignatureField1[0]";

                //Create a Credential object
                Credential myCred = new Credential();
                myCred.alias = "secure";

                //Specify the reason to sign the document
                string reason = "The document was reviewed";

                //Specify the location of the signer
                string location = "New York HQ";

                //Specify contact information
                string contactInfo = "Tony Blue";
```

```
                //Create a PDFSignatureAppearanceOptions object
                //and show date information
            PDFSignatureAppearanceOptionSpec appear = new PDFSignatureAppearanceOptionSpec();
                appear.showDate = true;

                //Sign the PDF document
               SignInteractiveForm.ServiceReference1.BLOB signedDoc = signatureClient.sign(
                    inDoc,
                    fieldName,
                    myCred,
                    HashAlgorithm.SHA1,
                    reason,
                    location,
                    contactInfo,
                    appear,
                    true,
                    null,
                    null,
                    null);

                //Populate a byte array with BLOB data that represents the signed form
                byte[] outByteArray = signedDoc.MTOM;

                //Save the signed PDF document
                string fileName = "C:\\Adobe\LoanXFASigned.pdf";
                FileStream fs2 = new FileStream(fileName, FileMode.OpenOrCreate);

                //Create a BinaryWriter object
                BinaryWriter w = new BinaryWriter(fs2);
                w.Write(outByteArray);
                w.Close();
                fs2.Close();
            }

            catch (Exception ee)
            {
                Console.WriteLine(ee.Message);
            }
        }
    }
}
```

Services starting with the letter I produce invalid proxy files

The name of some AEM Forms generated proxy classes are incorrect when using Microsoft .Net 3.5 and WCF. This issue occurs when proxy classes are created for the IBMFilenetContentRepositoryConnector, IDPSchedulerService or any other service whose name starts with the letter I. For example, the name of the generated client in case of IBMFileNetContentRepositoryConnector is `BMFileNetContentRepositoryConnectorClient`. The letter I is missing in the generated proxy class.

Invoking AEM Forms using REST Requests

Processes created in Workbench can be configured so that you can invoke them through Representational State Transfer (REST) requests. REST requests are sent from HTML pages. That is, you can invoke a Forms process directly from a web page using a REST request. For example, you can open a new instance of a web page. Then you can invoke a Forms process and load a rendered PDF document with data that was sent in an HTTP POST request.

Two types of HTML clients exist. The first HTML client is an AJAX client that is written in JavaScript. The second client is an HTML form that contains a submit button. An HTML-based client application is not the only possible REST client. Any client application that supports HTTP requests can invoke a service using a REST invocation. For example, you can invoke a service by using a REST invocation from a PDF form. (See .)

When using REST requests, it is recommended that you do not invoke Forms services directly. Instead, invoke processes that were created in Workbench. When creating a process that is meant for REST invocation, use a programmatic start point. In this situation, the REST endpoint is added automatically. For information about creating processes in Workbench, see Using Workbench.

When you invoke a service using REST, you are prompted for a AEM forms user name and password. However, if you do not want to specify a user name and password, you can disable service security. (See "Disabling Service Security" on page 1083.)

To invoke a Forms service (a process becomes a service when the process is activated) using REST, configure a REST endpoint. (See "Managing Endpoints" in administration help.)

After a REST endpoint is configured, you can invoke a Forms service by using an HTTP GET method or a POST method.

```
action="http://hiro-xp:8080/rest/services/[ServiceName]/[OperationName]:[ServiceVersion]"
method="post" enctype="multipart/form-data"
```

The mandatory `ServiceName` value is the name of the Forms service to invoke. The optional `OperationName` value is the name of the service's operation. If this value is not specified, this name defaults to `invoke`, which is the operation name that starts the process. The optional `ServiceVersion` value is the version encoded in the X.Y format. If this value is not specified, the most current version is used. The `enctype` value can also be `application/x-www-form-urlencoded`.

Supported data types

The following data types are supported when invoking AEM Forms services using REST requests:

- Java primitive data types, such as Strings and integers
- `com.adobe.idp.Document` data type
- XML data types such as `org.w3c.Document` and `org.w3c.Element`
- Collection objects such as `java.util.List` and `java.util.Map`

  These data types are commonly accepted as input values to processes created in Workbench.

  If a Froms service is invoked with the HTTP POST method, the arguments are passed inside the HTTP request body. If the AEM Forms service's signature has a string input parameter, the request body can contain the text value of the input parameter. If the service's signature defines multiple string parameters, the request can follow the HTTP's `application/x-www-form-urlencoded` notation with the parameter's names used as the form's field names.

  If a Forms service returns a string parameter, the result is a textual representation of the output parameter. If a service returns multiple string parameters, the result is an XML document encoding the output parameters in the following format:

```
<result> <output-paramater1>output-parameter-value-as-string</output-paramater1> . . .
<output-paramaterN>output-parameter-value-as-string</output-paramaterN> </result>
```

Note: *The `output-paramater1` value represents the output parameter name.*

If a Forms service requires a `com.adobe.idp.Document` parameter, the service can only be invoked using the HTTP POST method. If the service requires one `com.adobe.idp.Document` parameter, the HTTP request body becomes the content of the input Document object.

If an AEM Forms service requires multiple input parameters, the HTTP request body must be a multipart MIME message as defined by RFC 1867. (RFC 1867 is a standard used by web browsers to upload files to websites.) Each input parameter must be sent as a separate part of the multipart message and encoded in the `multipart/form-data` format. The name of each part must match the parameter's name.

Lists and maps are also used as input values to AEM Forms processes created in Workbench. As a result, you can use these data types when using a REST request. Java arrays are not supported because they are not used as an input value to a AEM Forms process.

If an input parameter is a list, a REST client can send it by specifying the parameter multiple times (once for each item in the list). For example, if A is a list of documents, the input must be a multipart message consisting of multiple parts named A. In this case, each part named A becomes an item in the input list. If B is a list of strings, the input can be an `application/x-www-form-urlencoded` message consisting of multiple fields named B. In this case, each form field named B becomes an item in the input list.

If an input parameter is a map and it is the services only input parameter, then every part/field of the input message becomes a key/value record in the map. The name of each part/field becomes the record's key. The content of each part/field becomes the record's value.

If an input map is not the services only input parameter, then each key/value record that belongs to the map can be sent using a parameter named as a concatenation of the parameter name and the record's key. For example, an input map called `attributes` can be sent with a list of the following key/values pairs:

```
attributesColor=red

attributesShape=box

attributesWidth=5
```

This translates into a map of three records: `Color=red`, `Shape=box`, and `Width=5`.

The output parameters of the list and map types become part of the resultant XML message. The output list is represented in XML as a series of XML elements with one element for each item in the list. Every element is given the same name as the output list parameter. The value of each XML element is one of two things:

- A text representation of the item in the list (if the list consists of string types)
- A URL that points to the content of Document (if the list consists of `com.adobe.idp.Document` objects)

The following example is an XML message returned by a service that has a single output parameter named *list*, which is a list of integers.

```
<result>
    <list>12345</list>
    . . .
    <list>67890</list>
</result>
```

An output map parameter is represented in the resultant XML message as a series of XML elements with one element for each record in the map. Every element is given the same name as the map record's key. The value of each element is either a text representation of the map record's value (if the map consists of records with a string value) or a URL pointing to the Document's content (if the map consists of records with the `com.adobe.idp.Document` value). Below is an example of an XML message returned by a service that has a single output parameter named `map`. This parameter value is a map consisting of records that associate letters with `com.adobe.idp.Document` objects.

```
<result>
    <A>http://localhost:8080/DocumentManager/docm123/4567</A>
    . . .
    <Z>http://localhost:8080/DocumentManager/docm987/6543</Z>
</result>
```

Asynchronous invocations

Some AEM Forms services, such as human-centric long-lived processes, require a long time to complete. These services can be invoked asynchronously in a non-blocking manner. (See "Invoking Human-Centric Long-Lived Processes" on page 560.)

An AEM Forms service can be invoked asynchronously by substituting `services` with `async_invoke` in the invocation URL, as shown in the following example.

```
http://localhost:8080/rest/async_invoke/SomeService.
SomeOperation?integer_input_variable=123&string_input_variable=abc
```

This URL returns the identifier value (in "text/plain" format) of the job responsible for this invocation.

The status of the asynchronous invocation can be retrieved by using an invocation URL with `services` substituted with `async_status`. The URL must contain a `job_id` parameter specifying the identifier value of the job associated with this invocation. For example:

```
http://localhost:8080/rest/async_status/SomeService.SomeOperation?job_id=2345353443366564
```

This URL returns an integer value (in "text/plain" format) encoding the job status according to the Job Manager's specification (for example, 2 means running, 3 means completed, 4 means failed, and so on.) (See "Retrieving the Status of an AEM Forms Job" on page 1077.)

If the job is completed, the URL returns the same result as if the service was invoked synchronously.

Once the job is completed and the result is retrieved, the job can disposed of by using an invocation URL with `services` is substituted with `async_dispose`. The URL should also contain a `job_id` parameter specifying the identifier value of the job. For example:

```
http://localhost:8080/rest/async_dispose/SomeService.SomeOperation?job_id=2345353443366564
```

If the job is successfully disposed of, this URL returns an empty message.

Error reporting

If a synchronous or asynchronous invocation request cannot be completed due to an exception being thrown on the server, the exception is reported as part of the HTTP response message. If the invocation URL (or the `async_result` URL in the case of an asynchronous invocation) does not have an .xml suffix, the REST Provider returns the HTTP code `500 Internal Server Error` followed by an exception message.

If the invocation URL (or the `async_result` URL in the case of an asynchronous invocation) does have an .xml suffix, the REST Provider returns the HTTP code `200 OK` followed by an XML document describing the exception in the following format.

```
<exception>
      <exception_class_name>[
      <DSCError>
         <componentUID>component_UUD</componentUID>
        <errorCode>error_code</errorCode>
        <minorCode>minor_code</minorCode>
        <message>error_message</message>
      </DSCError>
]
      <message>exception_message</message>
    <stackTrace>exception_stack_trace</stackTrace>
      </exception_class_name>
    <exception>
      </exception>
</exception>
```

The `DSCError` element is optional and present only if the exception is an instance of `com.adobe.idp.dsc.DSCException`.

Security and authentication

To provide REST invocations with a secure transport, a AEM forms administrator can enable the HTTPS protocol on the J2EE application server hosting AEM Forms. This configuration is specific to the J2EE application server; it is not part of the forms server configuration.

Note: *As a Workbench developer that wants to expose your processes through a REST endpoint, keep in mind the XSS vulnerability issue. XSS vulnerabilities can be used to steal or manipulate cookies, modify presentation of content, and compromise confidential information. It is recommended that you extend the process logic with the additional input and output data validation rules if XSS vulnerability is an issue.*

AEM Forms services that support REST invocation

Although it is recommended that you invoke processes created using Workbench as opposed to services directly, there are some AEM Forms services that do support REST invocation. The reason why it is recommended that you invoke a process as opposed to a service directly is because it is more efficient to invoke a process. Consider the following scenario. Assume that you want to create a policy from a REST client. That is, you want the REST client to define values such as the policy name, the offline lease period.

To create a policy, you have to define complex data types such as a `PolicyEntry` object. A `PolicyEntry` object defines attributes such as permissions associated with the policy. (See "Creating Policies" on page 832.)

Instead of sending a REST request to create a policy (which would include defining complex data types such as a `PolicyEntry` object), create a process that creates a policy using Workbench. Define the process to accept primitive input variables such as a string value that defines the process name or an integer that defines the offline lease period.

This way, you do not have to create a REST invocation request that includes complex data types that required by the operation. The process defines the complex data types and all you do from the REST client is invoke the process and pass primitive data types. For information about invoking a process using REST, see .

The following lists specifies those AEM Forms services that support direct REST invocation.

- Distiller service
- Rights Management service

- GeneratePDF service
- Generate3dPDF service
- FormDataIntegration

REST invocation examples

The following REST invocation examples are provided:

-
-
-
-
-
-
-

  Each example demonstrates passing different data types to an AEM Forms process

Passing Boolean values to a process

The following HTML example passes two `Boolean` values to an AEM Forms process named `RestTest2`. The name of the invocation method is `invoke` and the version is 1.0. Notice that the HTML Post method is used.

```
<html>
<body>

<form name="input" action="http://localhost:9080/rest/services/RestTest2/invoke/1.0"
method="post">

Boolean 1: <input type="text" name="inBooleanList" value="true">
Boolean 2: <input type="text" name="inBooleanList" value="false">
<input type="submit" value="Submit">

</form>

</body>
</html>
```

Passing date values to a process

The following HTML example passes a date value to an AEM Forms process named `SOAPEchoService`. The name of the invocation method is `echoCalendar`. Notice that the HTML `Post` method is used.

```
<html>
<body>

<form name="input" action="http://localhost:9080/rest/services/SOAPEchoService/echoCalendar"
method="post">

Date: <input type="text" name="value-to-echo" value="2009-01-02T12:15:30Z">
<input type="submit" value="Submit">

</form>

</body>
</html>
```

Passing documents to a process

The following HTML example invokes an AEM Forms process named `MyApplication/EncryptDocument` that requires a PDF document. For information about this process, see "Invoking AEM Forms using MTOM" on page 529.

```
<html>
<body>

<form name="input"
action="http://localhost:9080/rest/services/MyApplication/EncryptDocument/invoke"
method="post"
         enctype="multipart/form-data">

File: <input type="file" name="value-to-echo">
<input type="submit" value="Submit"/>

</form>

</body>
</html>
```

Passing document and text values to a process

The following HTML example invokes an AEM Forms process named `RestTest3` that requires a document and two text values. Notice that the HTML Post method is used.

```
<html>
<body>

<form name="input" action="http://localhost:9080/rest/services/RestTest3" method="post"
         enctype="multipart/form-data">

Doc: <input type="file" name="inDoc">
String 1: <input type="text" name="inListOfStrings" value="hello">
String 2: <input type="text" name="inListOfStrings" value="privet">
<input type="submit" value="Submit"/>

</form>

</body>
</html>
```

Passing enumeration values to a process

The following HTML example invokes an AEM Forms process named `SOAPEchoService` that requires an enumeration value. Notice that the HTML Post method is used.

```
<html>
<body>

<form name="input" action="http://hiro-xp:8080/rest/services/SOAPEchoService/echoEnum"
method="post">

Color Enum Value: <input type="text" name="value-to-echo" value="green">
<input type="submit" value="Submit">

</form>

</body>
</html>
```

Invoking the MyApplication/EncryptDocument process using REST

You can invoke an AEM Forms short-lived process named *MyApplication/EncryptDocument* by using REST.

Note: *This process is not based on an existing AEM Forms process. To follow along with the code example, create a process named* `MyApplication/EncryptDocument` *using workbench. (See Using Workbench.)*

When this process is invoked, it performs the following actions:

1. Obtains the unsecured PDF document that is passed to the process. This action is based on the `SetValue` operation. The input parameter for this process is a `document` process variable named `inDoc`.
2. Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation. The password encrypted PDF document is returned in a process variable named `outDoc`.

   When this process is invoked using a REST request, the encrypted PDF document is displayed in the web browser. Before you view the PDF document, you specify the password (unless security is disabled). The following HTML code represents a REST invocation request to the `MyApplication/EncryptDocument` process.

   ```
   <html>
   <body>
   <form action="http://hiro-xp:8080/rest/services/MyApplication/EncryptDocument"
   method="post" enctype="multipart/form-data">
        <p>Chose a PDF file (.pdf) to send to the EncryptDocument process.</p>
        <p>file:
          <input type="file" name="inDoc" />
        </p>
        <p>
          <input type="submit"/>
        </p>
   </form>
   </body>
   ```

Invoking the MyApplication/EncryptDocument process from Acrobat

You can invoke a Forms process from Acrobat by using a REST request. For example, you can invoke the *MyApplication/EncryptDocument* process. To invoke a Forms process from Acrobat, place a submit button on a XDP file within Designer. (See Designer Help.)

Specify the URL to invoke the process within the button's *Submit to URL* field, as shown in the following illustration.

The complete URL to invoke the process is http://hiro-xp:8080/rest/services/MyApplication/EncryptDocument.

If the process requires a PDF document as an input value, ensure that you submit the form as PDF, as shown in the previous illustration. Also, to successfully invoke a process, the process must return a PDF document. Otherwise Acroabt cannot handle the return value and an error occurs. You do not have to specify the name of the input process variable. For example, the *MyApplication/EncryptDocument* process has an input variable named `inDoc`. You do not have to specify inDoc, as long as the form is submited as PDF.

You can also submit form data as XML to a Forms process, To submit XML data, ensure that the `Submit As` drop down specifies XML. Because the return value of the process must be a PDF document, the PDF document is displayed in Acrobat.

Invoking Human-Centric Long-Lived Processes

You can programmatically invoke human-centric long-lived processes that were created in Workbench using these client applications:

- A Java web-based client application that uses the Invocation API. (See "Invoking AEM Forms using the Java API" on page 490.)
- An ASP.NET application that uses web services. (See "Invoking AEM Forms using Web Services" on page 514.)
- A client application built with Flex that uses Remoting. (See "Invoking AEM Forms using Remoting" on page 444.)

The long-lived process that is invoked is named *FirstAppSolution/PreLoanProcess*. You can create this process by following the tutorial specified in Creating Your First AEM Forms Application.

A human-centric process involves a task that a user can respond to by using Workspace. For example, using Workbench, you can create a process that lets a bank manager approve or deny a loan application. The following illustration shows the process *FirstAppSolution/PreLoanProcess*.

The *FirstAppSolution/PreLoanProcess* process accepts an input parameter named *formData* whose data type is XML. The XML data is merged with a form design named *PreLoanForm.xdp*. The following illustration shows a form that represents a task assigned to a user to approve or deny a loan application. The user approves or denies the application by using Workspace. The Workspace user can approve the loan request by clicking the Approve button shown in the following illustration. Likewise, the user can deny the loan request by clicking the deny button.

A long-lived process is invoked asynchronously and cannot be invoked synchronously due to the following factors:

- A process can span a significant amount of time.
- A process can span organizational boundaries.
- A process needs external input in order for it to finish. For example, consider a situation where a form is sent to a manager, who is out of the office. In this situation, the process is not complete until the manager returns and fills out the form.

When a long-lived process is invoked, AEM Forms creates an invocation identifier value as part of creating a record. The record tracks the status of the long-lived process and is stored in the AEM Forms database. Using the invocation identifier value, you can track the status of the long-lived process. In addition, you can use the process invocation identifier value to perform Process Manager operations such as terminating a running process instance. (See "Terminating Process Instances" on page 1071.)

Note: *AEM Forms does not create an invocation identifier value or a record when a short-lived process is invoked.*

The `FirstAppSolution/PreLoanProcess` process is invoked when an applicant submits an application, which is represented as XML data. The name of the input process variable is `formData` and its data type is XML. For the purposes of this discussion, assume that the following XML data is used as input to the `FirstAppSolution/PreLoanProcess` process.

```
<?xml version="1.0" encoding="UTF-8"?>
<LoanApp>
<Name>Sam White</Name>
<LoanAmount>250000</LoanAmount>
<PhoneOrEmail>(555)555-5555</PhoneOrEmail>
<ApprovalStatus>PENDING APPROVAL</ApprovalStatus>
</LoanApp>
```

XML data passed to a process must match the fields located in the form used in the process. Otherwise, data is not displayed within the form. All applications that invoke the `FirstAppSolution/PreLoanProcess` process must pass this XML data source. The applications created in *Invoking Human-Centric Long-Lived Processes* dynamically create the XML data source from values that a user entered into a web client.

Using a client application, you can send the *FirstAppSolution/PreLoanProcess* process the required XML data. A long-lived process returns an invocation identifier value as its return value. The following illustration shows client applications invoking the*FirstAppSolution/PreLoanProcess* long-lived process. The client applications send XML data and get back a string value that represents the invocation identifier value.

See also

"Creating a Java web application that invokes a human-centric long-lived process" on page 561

"Creating an ASP.NET web application that invokes a human-centric long-lived process" on page 569

"Creating a client application built with Flex that invokes a human-centric long-lived process" on page 576

Creating a Java web application that invokes a human-centric long-lived process

You can create a web-based application that uses a Java servlet to invoke the `FirstAppSolution/PreLoanProcess` process. To invoke this process from a Java servlet, use the Invocation API within the Java servlet. (See "Invoking AEM Forms using the Java API" on page 490.)

The following illustration shows a web-based client application that posts name, phone (or email), and amount values. These values are sent to the Java servlet when the user clicks the Submit Application button.

The Java servlet performs the following tasks:

- Retrieves the values posted from the HTML page to the Java servlet.
- Dynamically creates an XML data source to pass to the *FirstAppSolution/PreLoanProcess* process. The name, phone (or email), and amount values are specified in the XML data source.
- Invokes the *FirstAppSolution/PreLoanProcess* process by using the AEM Forms Invocation API.
- Returns the invocation identifier value to the client web browser.

Summary of steps

To create a Java web-based application that invokes the `FirstAppSolution/PreLoanProcess` process, perform the following steps:

1 "Create a web project" on page 562.

2 "Create Java application logic for the servlet" on page 563.

3 "Create the web page for the web application" on page 567

4 "Package the web application to a WAR file" on page 568.

5 "Deploy the WAR file to the J2EE application server hosting AEM Forms" on page 568.

6 "Test your web application" on page 569.

Note: *Some of these steps depend on the J2EE application on which AEM Forms is deployed. For example, the method you use to deploy a WAR file depends on the J2EE application server that you are using. It is assumed that AEM Forms is deployed on JBoss®.*

Create a web project

The first step to create a web application is to create a web project. The Java IDE that this document is based on is Eclipse 3.3. Using the Eclipse IDE, create a web project and add the required JAR files to your project. Add an HTML page named *index.html* and a Java servlet to your project.

The following list specifies the JAR files to include in your web project:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- J2EE.jar

For the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Note: *The J2EE.jar file defines data types used by a Java servlet. You can obtain this JAR file from the J2EE application server on which AEM Forms is deployed.*

Create a web project

1 Start Eclipse and click **File** > **NewProject**.

2 In the **New Project** dialog box, select **Web** > **Dynamic Web Project**.

3 Type `InvokePreLoanProcess` for the name of your project and then click **Finish**.

Add required JAR files to your project

1 From the Project Explorer window, right-click the `InvokePreLoanProcess` project and select **Properties**.

2 Click **Java build path** and then click the **Libraries** tab.

3 Click the **Add External JARs** button and browse to the JAR files to include.

Add a Java servlet to your project

1 From the Project Explorer window, right-click the `InvokePreLoanProcess` project and select **New** > **Other**.

2 Expand the **Web** folder, select **Servlet**, and then click **Next**.

3 In the Create Servlet dialog box, type `SubmitXML`for the name of the servlet and then click **Finish**.

Add an HTML page to your project

1 From the Project Explorer window, right-click the `InvokePreLoanProcess` project and select **New** > **Other**.

2 Expand the **Web** folder, select **HTML**, and click **Next**.

3 In the New HTMLdialog box, type `index.html` for the filename and then click **Finish**.

Note: *For information about creating HTML content that invokes the SubmitXML Java servlet, see "Create the web page for the web application" on page 567.*

Create Java application logic for the servlet

Create Java application logic that invokes the `FirstAppSolution/PreLoanProcess` process from within the Java servlet. The following code shows the syntax of the `SubmitXML` Java Servlet:

```
public class SubmitXML extends HttpServlet implements Servlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp
    throws ServletException, IOException {
    doPost(req,resp);

    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp
    throws ServletException, IOException {
        //Add code here to invoke the FirstAppSolution/PreLoanProcess process
        }
```

Normally, you would not place client code within a Java servlet's `doGet` or `doPost` method. A better programming practice is to place this code within a separate class. Then instantiate the class from within the `doPost` method (or `doGet` method), and call the appropriate methods. However, for code brevity, code examples are kept to a minimum and are placed in the `doPost` method.

To invoke the `FirstAppSolution/PreLoanProcess` process using the Invocation API, perform the following tasks:

1. Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path. For information about the location of these files, see "Including AEM Forms Java library files" on page 491.
2. Retrieve the name, phone, and amount values that is submitted from the HTML page. Use these values to dynamically create an XML data source that is sent to the `FirstAppSolution/PreLoanProcess` process. You can use `org.w3c.dom` classes to create the XML data source (this application logic is shown in the following code example).
3. Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
4. Create a `ServiceClient` object by using its constructor and passing the `ServiceClientFactory` object. A `ServiceClient` object lets you invoke a service operation. It handles tasks such as locating, dispatching, and routing invocation requests.
5. Create a `java.util.HashMap` object by using its constructor.
6. Invoke the `java.util.HashMap` object's `put` method for each input parameter to pass to the long-lived process. Ensure that you specify the name of the process's input parameters. Because the `FirstAppSolution/PreLoanProcess` process requires one input parameter of type `XML` (named `formData`), you only have to invoke the `put` method once.

   ```
   //Get the XML to pass to the FirstAppSolution/PreLoanProcess process
   org.w3c.dom.Document inXML = GetDataSource(name,phone,amount);

   //Create a Map object to store the parameter value
   Map params = new HashMap();
   params.put("formData", inXML);
   ```

7. Create an `InvocationRequest` object by invoking the `ServiceClientFactory` object's `createInvocationRequest` method and passing the following values:
   - A string value that specifies the name of the long-lived process to invoke. To invoke the `FirstAppSolution/PreLoanProcess` process, specify `FirstAppSolution/PreLoanProcess`.
   - A string value that represents the process operation name. The name of the long-lived process operation is `invoke`.

- The `java.util.HashMap` object that contains the parameter values that the service operation requires.
- A Boolean value that specifies `false`, which creates an asynchronous request (this value is applicable to invoke a long-lived process).

Note: *A short-lived process can be invoked by passing the value true as the fourth parameter of the createInvocationRequest method. Passing the value true creates a synchronous request.*

8 Send the invocation request to AEM Forms by invoking the `ServiceClient` object's `invoke` method and passing the `InvocationRequest` object. The `invoke` method returns an `InvocationReponse` object.

9 A long-lived process returns a string value that represents an invocation identification value. Retrieve this value by invoking the `InvocationReponse` object's `getInvocationId` method.

```
//Send the invocation request to the long-lived process and
//get back an invocation response object
InvocationResponse lcResponse = myServiceClient.invoke(lcRequest);
String invocationId = lcResponse.getInvocationId();
```

10 Write the invocation identification value to the client web browser. You can use a `java.io.PrintWriter` instance to write this value to the client web browser.

Quick Start: Invoking a long-lived process using the Invocation API

The following Java code example represents the Java servlet that invokes the `FirstAppSolution/PreLoanProcess` process.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-livecycle-client.jar
    * 2. adobe-usermanager-client.jar
    *
    *  (Because this  quick start is implemented as a Java servlet, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs/common
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    * */
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.*;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
```

```
import javax.xml.transform.stream.StreamResult;

import com.adobe.idp.dsc.InvocationRequest;
import com.adobe.idp.dsc.InvocationResponse;
import com.adobe.idp.dsc.clientsdk.ServiceClient;
import org.w3c.dom.Element;

    public class SubmitXML extends javax.servlet.http.HttpServlet implements
javax.servlet.Servlet {
      static final long serialVersionUID = 1L;

       public SubmitXML() {
        super();
    }


    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        // TODO Auto-generated method stub
        doPost(request,response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_ENDPOINT,
"jnp://localhost:1099");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_EJB_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a ServiceClient object
            ServiceClient myServiceClient = myFactory.getServiceClient();

            //Get the values that are passed from the Loan HTML page
            String name = (String)request.getParameter("name");
            String phone = (String)request.getParameter("phone");
            String amount = (String)request.getParameter("amount");

            //Create XML to pass to the FirstAppSolution/PreLoanProcess process
            org.w3c.dom.Document inXML = GetDataSource(name,phone,amount);
```

```
        //Create a Map object to store the XML input parameter value
        Map params = new HashMap();
        params.put("formData", inXML);

        //Create an InvocationRequest object
        InvocationRequest lcRequest =  myFactory.createInvocationRequest(
            "FirstAppSolution/PreLoanProcess", //Specify the long-lived process name
                "invoke",       //Specify the operation name
                params,        //Specify input values
                false);        //Create an asynchronous request

        //Send the invocation request to the long-lived process and
        //get back an invocation response object
        InvocationResponse lcResponse = myServiceClient.invoke(lcRequest);
        String invocationId = lcResponse.getInvocationId();

        //Create a PrintWriter instance
        PrintWriter pp = response.getWriter();

        //Write the invocation identifier value back to the client web browser
        pp.println("The job status identifier value is: " +invocationId);

    }catch (Exception e) {
         System.out.println("The following exception occurred: "+e.getMessage());
  }
}


 //Create XML data to pass to the long-lived process
 private static org.w3c.dom.Document GetDataSource(String name, String phone, String amount)
 {
        org.w3c.dom.Document document = null;

        try
        {
            //Create DocumentBuilderFactory and DocumentBuilder objects
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();

            //Create a new Document object
            document = builder.newDocument();

            //Create MortgageApp - the root element in the XML
            Element root = (Element)document.createElement("LoanApp");
            document.appendChild(root);

            //Create an XML element for Name
            Element nameElement = (Element)document.createElement("Name");
            nameElement.appendChild(document.createTextNode(name));
            root.appendChild(nameElement);

            //Create an XML element for Phone
            Element phoneElement = (Element)document.createElement("PhoneOrEmail");
```

```
                phoneElement.appendChild(document.createTextNode(phone));
                root.appendChild(phoneElement);

                //Create an XML element for amount
                Element loanElement = (Element)document.createElement("LoanAmount");
                loanElement.appendChild(document.createTextNode(amount));
                root.appendChild(loanElement);

                //Create an XML element for ApprovalStatus
                Element approveElement = (Element)document.createElement("ApprovalStatus");
                approveElement.appendChild(document.createTextNode("PENDING APPROVAL"));
                root.appendChild(approveElement);

              }
         catch (Exception e) {
                  System.out.println("The following exception occurred: "+e.getMessage());
               }
        return document;
         }
        }
```

Create the web page for the web application

The *index.html* web page provides an entry point to the Java servlet that invokes the `FirstAppSolution/PreLoanProcess` process. This web page is a basic HTML form that contains an HTML form and a submit button. When the user clicks the submit button, form data is posted to the `SubmitXML` Java servlet.

The Java servlet captures the data that is posted from the HTML page by using the following Java code:

```
//Get the values that are passed from the Loan HTML page
String name = request.getParameter("name");
String phone = request.getParameter("phone");
String amount = request.getParameter("amount");
```

The following HTML code represents the index.html file that was created during setup of the development environment. (See "Create a web project" on page 562.)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<TABLE border="0">
    <TBODY>
        <TR>
            <TD width="172"><img src="financeCorpLogo.jpg" width="172" height="62"></TD>
            <TD width="314"><FONT size="+2"><strong>Java Loan Application
Page</strong></FONT></TD>
            <TD width="10"> </TD>
            <TD width="123"> </TD>
        </TR>

    </TBODY>
</TABLE>
    <FORM action="http://hiro-xp:8080/PreLoanProcess/SubmitXML" method="post">
```

```
        <TABLE border="0">
          <TBODY>
                <TR>
                      <TD width="114"><LABEL for="name">Name: </LABEL></TD>
                  <TD width="166"><INPUT type="text" name="name"></TD>
                  <TD width="267"><input type="submit" value="Submit Application"></TD>
                  </TR>
             <TR>
                    <TD width="114"> <LABEL for="phone">Phone/Email: </LABEL></TD>
               <TD width="166"><INPUT type="text" name="phone"></TD>
                    <TD width="267"></TD>
               </TR>

             <TR>
                    <TD width="114"><LABEL for="amount">Amount: </LABEL></TD>
               <TD width="166"><INPUT type="text" name="amount"></TD>
                  <TD width="267"></TD>
              </TR>
          </TBODY>
</TABLE>
      </FORM>
</body>
</html>
```

Package the web application to a WAR file

To deploy the Java servlet that invokes the `FirstAppSolution/PreLoanProcess` process, package your web application to a WAR file. Ensure that external JAR files that the component's business logic depends on, such as adobe-livecycle-client.jar and adobe-usermanager-client.jar, are also included in the WAR file.

The following illustration shows the Eclipse project's content, which is packaged to a WAR file.

Note: *In the previous illustration, the JPG file can be replaced by any JPG image file.*

Package a web application to a WAR file:

1. From the **Project Explorer** window, right-click the `InvokePreLoanProcess` project and select **Export** > **WAR file**.
2. In the **Web module** text box, type `InvokePreLoanProcess` for the name of the Java project.
3. In the **Destination** text box, type `PreLoanProcess.war`for thefilename, specify the location for your WAR file, and then click Finish.

Deploy the WAR file to the J2EE application server hosting AEM Forms

Deploy the WAR file to the J2EE application server on which AEM Forms is deployed. To deploy the WAR file to the J2EE application server, copy the WAR file from the export path to *[AEM Forms Install]*\Adobe\Adobe Experience Manager Forms\jboss\server\lc_turnkey\deploy.

Note: *if AEM Forms is not deployed on JBoss, then you must deploy the WAR file in compliance with the J2EE application server hosting AEM Forms.*

Test your web application

After you deploy the web application, you can test it by using a web browser. Assuming that you are using the same computer that is hosting AEM Forms, you can specify the following URL:

- http://localhost:8080/PreLoanProcess/index.html

  Enter values into the HTML form fields and click the Submit Application button. If problems occur, see the J2EE application server's log file.

  Note: *To confirm that the Java application invoked the process, start Workspace and accept the loan.*

Creating an ASP.NET web application that invokes a human-centric long-lived process

You can create an ASP.NET application that invokes the `FirstAppSolution/PreLoanProcess` process. To invoke this process from an ASP.NET application, use web services. (See "Invoking AEM Forms using Web Services" on page 514.)

The following illustration shows an ASP.NET client application obtaining data from an end user. The data is placed into an XML data source and sent to the `FirstAppSolution/PreLoanProcess` process when the user clicks the Submit Application button.

Notice after the process is invoked, an invocation identifier value is displayed. An invocation identifier value is created as part of a record that tracks the status of the long-lived process.

The ASP.NET application performs the following tasks:

- Retrieves the values that the user entered into the web page.
- Dynamically creates an XML data source that is passed to the *FirstAppSolution/PreLoanProcess* process. The three values are specified in the XML data source.
- Invokes the *FirstAppSolution/PreLoanProcess* process by using the web services.
- Returns the invocation identifier value and the status of the long-lived operation to the client web browser.

Summary of steps

To create an ASP.NET application that is able to invoke the FirstAppSolution/PreLoanProcess process, perform the following steps:

1 "Create an ASP.NET web application" on page 569.

2 "Create an ASP page that invokes FirstAppSolution/PreLoanProcess" on page 570.

3 "Run the ASP.NET application" on page 576.

Create an ASP.NET web application

Create a Microsoft .NET C# ASP.NET Web application. The following illustration shows the contents of the ASP.NET project named *InvokePreLoanProcess*.

Notice under Service References, there are two items. The first item is named *JobManager*. This reference enables the ASP.NET application to invoke the Job Manager service. This service returns information about the status of a long-lived process. For example, if the process is currently running, then this service returns a numeric value that specifies the process is currently running. The second reference is named*PreLoanProcess*. This service reference represents the reference to the *FirstAppSolution/PreLoanProcess* process. After you create a Service Reference, data types associated with the AEM Forms service are available for use within your .NET project.

Create a ASP.NET project:

1 Start Microsoft Visual Studio 2008.

2 From the **File** menu, select **New**, **Web Site**.

3 In the **Templates** list, select **ASP.NET Web Site**.

4 In the **Location** box, select a location for your project. Name your project *InvokePreLoanProcess*.

5 In the **Language** box, select Visual C#

6 Click OK.

Add service references:

1 In the Project menu, select **Add Service Reference**.

2 In the **Address** dialog box, specify the WSDL to the Job Manager service.

```
http://hiro-xp:8080/soap/services/JobManager?WSDL&lc_version=9.0.1
```

3 In the Namespace field, type `JobManager`.

4 Click **Go** and then click **OK**.

5 In the **Project** menu, select **Add Service Reference**.

6 In the **Address** dialog box, specify the WSDL to the FirstAppSolution/PreLoanProcess process.

```
http://hiro-xp:8080/soap/services/FirstAppSolution/PreLoanProcess?WSDL&lc_version=9.0.1
```

7 In the Namespace field, type `PreLoanProcess`.

8 Click **Go** and then click **OK**.

Note: *Replace `hiro-xp` with the IP address of the J2EE application server hosting AEM Forms. The `lc_version` option ensures that AEM Forms functionality, such as MTOM, is available. Without specifying the `lc_version` option, you cannot invoke AEM Forms using MTOM. (See "Invoking AEM Forms using MTOM" on page 529.)*

Create an ASP page that invokes FirstAppSolution/PreLoanProcess

Within the ASP.NET project, add a web form (an ASPX file) that is responsible for displaying an HTML page to the loan applicant. The web form is based on a class that is derived from `System.Web.UI.Page`. The C# application logic that invokes `FirstAppSolution/PreLoanProcess` is located in the `Button1_Click` method (this button represents the Submit Application button).

The following illustration shows the ASP.NET application

The following table lists the controls that are part of this ASP.NET application.

Control name	Description
TextBoxName	Specifies the customer's first and last name.
TextBoxPhone	Specifies the customer's phone or email address.
TextBoxAmount	Specifies the loan amount.
Button1	Represents the Submit Application button.
LabelJobID	A Label control that specifies the value of the invocation identifier value.
LabelStatus	A Label control that specifies the value of the job status. This value is retrieved by invoking the Job Manager service.

The application logic that is part of the ASP.NET application must dynamically create an XML data source to pass to the `FirstAppSolution/PreLoanProcess` process. The values that the applicant entered into the HTML page must be specified within the XML data source. These data values are merged into the form when the form is viewed in Workspace. The classes located in the `System.Xml` namespace are used to create the XML data source.

When invoking a process that requires XML data from an ASP.NET application, an XML data type is available for you to use. That is, you cannot pass a `System.Xml.XmlDocument` instance to the process. The fully qualified name of this XML instance to pass to the process is `InvokePreLoanProcess.PreLoanProcess.XML`. Convert the `System.Xml.XmlDocument` instance to `InvokePreLoanProcess.PreLoanProcess.XML`. You can perform this task by using the following code.

```
//Create the XML to pass to the FirstAppSolution/PreLoanProcess process
XmlDocument myXML = CreateXML(userName, phone, amount);

//Convert the XML to a InvokePreLoanProcess.PreLoanProcess.XML instance
StringWriter sw = new StringWriter();
XmlTextWriter xw = new XmlTextWriter(sw);
myXML.WriteTo(xw);

InvokePreLoanProcess.PreLoanProcess.XML inXML = new XML();
inXML.document = sw.ToString();
```

To create an ASP page that invokes the `FirstAppSolution/PreLoanProcess` process, perform the following tasks in the `Button1_Click` method:

1. Create a `FirstAppSolution_PreLoanProcessClient` object by using its default constructor.
2. Create a `FirstAppSolution_PreLoanProcessClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service and the encoding type:

   `http://hiro-xp:8080/soap/services/FirstAppSolution/PreLoanProcess?blob=mtom`

   You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, ensure that you specify`?blob=mtom`.

   Note: *Replace `hiro-xp` with the IP address of the J2EE application server hosting AEM Forms.*
3. Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `FirstAppSolution_PreLoanProcessClient.Endpoint.Binding` data member. Cast the return value to `BasicHttpBinding`.
4. Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` data member to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
5. Enable basic HTTP authentication by performing the following tasks:
   - Assign the AEM forms user name to the data member `FirstAppSolution_PreLoanProcessClient.ClientCredentials.UserName.UserName`.
   - Assign the corresponding password value to the data member `FirstAppSolution_PreLoanProcessClient.ClientCredentials.UserName.Password`.
   - Assign the constant value `HttpClientCredentialType.Basic` to the data member `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
   - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the data member `BasicHttpBindingSecurity.Security.Mode`.

   The following code example shows these tasks.

```
//Enable BASIC HTTP authentication
BasicHttpBinding b = (BasicHttpBinding)mortgageClient.Endpoint.Binding;
b.MessageEncoding = WSMessageEncoding.Mtom;
mortgageClient.ClientCredentials.UserName.UserName = "administrator";
mortgageClient.ClientCredentials.UserName.Password = "password";
b.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
b.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
b.MaxReceivedMessageSize = 2000000;
b.MaxBufferSize = 2000000;
b.ReaderQuotas.MaxArrayLength = 2000000;
```

6 Retrieve the name, phone, and amount values that the user entered into the web page. Use these values to dynamically create an XML data source that is sent to the `FirstAppSolution/PreLoanProcess` process. Create a `System.Xml.XmlDocument` that represents the XML data source to pass to the process (this application logic is shown in the following code example).

7 Convert the `System.Xml.XmlDocument` instance to `InvokePreLoanProcess.PreLoanProcess.XML` (this application logic is shown in the following code example).

8 Invoke the `FirstAppSolution/PreLoanProcess` process by invoking the `FirstAppSolution_PreLoanProcessClient` object's `invoke_Async` method. This method returns a string value that represents the invocation identifier value of the long-lived process.

9 Create a `JobManagerClient` by using is constructor. (Ensure that you have set a service reference to the Job Manager service.)

10 Repeat steps 1-5. Specify the following URL for step 2: `http://hiro-xp:8080/soap/services/JobManager?blob=mtom`.

11 Create a `JobId` object by using its constructor.

12 Set the `JobId` object's `id` data member with the return value of the `FirstAppSolution_PreLoanProcessClient` object's `invoke_Async` method.

13 Assign the `value` true to the `JobId` object's `persistent` data member.

14 Create a `JobStatus` object by invoking the `JobManagerService` object 's `getStatus` method and passing the `JobId` object.

15 Get the status value by retrieving the value of the `JobStatus` object's `statusCode` data member.

16 Assign the invocation identifier value to the `LabelJobID.Text` field.

17 Assign the status value to the `LabelStatus.Text` field.

Quick Start: Invoking a long-lived process using the web service API

The following C# code example invokes the `FirstAppSolution/PreLoanProcess` process.

```
???/**
    * Ensure that you create a .NET project that uses
    * MS Visual Studio 2008 and version 3.5 of the .NET
    * framework. This is required to invoke a
    * AEM Forms service using MTOM.


using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.ServiceModel;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.Xml;
using System.IO;

//A reference to FirstAppSolution/PreLoanProcess
using InvokePreLoanProcess.PreLoanProcess;

//A reference to JobManager service
using InvokePreLoanProcess.JobManager;


namespace InvokePreLoanProcess
{
       public partial class _Default : System.Web.UI.Page
       {
           //This method is called when the Submit Application button is
           //Clicked
           protected void Button1_Click(object sender, EventArgs e)
           {
               //Create a FirstAppSolution_PreLoanProcessClient object
               FirstAppSolution_PreLoanProcessClient mortgageClient = new
FirstAppSolution_PreLoanProcessClient();
               mortgageClient.Endpoint.Address = new
System.ServiceModel.EndpointAddress("http://hiro-
xp:8080/soap/services/FirstAppSolution/PreLoanProcess?blob=mtom");

               //Enable BASIC HTTP authentication
               BasicHttpBinding b = (BasicHttpBinding)mortgageClient.Endpoint.Binding;
               b.MessageEncoding = WSMessageEncoding.Mtom;
               mortgageClient.ClientCredentials.UserName.UserName = "administrator";
               mortgageClient.ClientCredentials.UserName.Password = "password";
               b.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
               b.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
               b.MaxReceivedMessageSize = 2000000;
               b.MaxBufferSize = 2000000;
               b.ReaderQuotas.MaxArrayLength = 2000000;

               //Retrieve values that user entered into the web page
```

```
                String userName = TextBoxName.Text;
                String phone = TextBoxPhone.Text;
                String amount = TextBoxAmount.Text;

                //Create the XML to pass to the FirstAppSolution/PreLoanProcess process
                XmlDocument myXML = CreateXML(userName, phone, amount);

                StringWriter sw = new StringWriter();
                XmlTextWriter xw = new XmlTextWriter(sw);
                myXML.WriteTo(xw);

                InvokePreLoanProcess.PreLoanProcess.XML inXML = new XML();
                inXML.document = sw.ToString();

                //INvoke the FirstAppSolution/PreLoanProcess process
                String invocationID =  mortgageClient.invoke_Async(inXML);

              //Create a JobManagerClient object to obtain the status of the long-lived operation
                JobManagerClient jobManager = new JobManagerClient();
                jobManager.Endpoint.Address = new
System.ServiceModel.EndpointAddress("http://hiro-xp:8080/soap/services/JobManager?blob=mtom");

                //Enable BASIC HTTP authentication
                BasicHttpBinding b1 = (BasicHttpBinding)jobManager.Endpoint.Binding;
                b1.MessageEncoding = WSMessageEncoding.Mtom;
                jobManager.ClientCredentials.UserName.UserName = "administrator";
                jobManager.ClientCredentials.UserName.Password = "password";
                b1.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
                b1.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
                b1.MaxReceivedMessageSize = 2000000;
                b1.MaxBufferSize = 2000000;
                b1.ReaderQuotas.MaxArrayLength = 2000000;


                //Create a JobID object that represents the status of the
                //long-lived operation
                JobId jobId = new JobId();
                jobId.id = invocationID;
                jobId.persistent = true;
                JobStatus jobStatus = jobManager.getStatus(jobId);
                System.Int16 val2 = jobStatus.statusCode;
                LabelJobID.Text = "The job status identifier value is " + invocationID;
                LabelStatus.Text = "The status of the long-lived operation is " +
getJobDescription(val2);

            }

            private static XmlDocument CreateXML(String name, String phone, String amount)
            {
                //This method dynamically creates a DDX document
                //to pass to the FirstAppSolution/PreLoanProcess process
                XmlDocument xmlDoc = new XmlDocument();

                //Create the root element and append it to the XML DOM
                System.Xml.XmlElement root = xmlDoc.CreateElement("LoanApp");
                xmlDoc.AppendChild(root);
```

```
    //Create the Name element
    XmlElement nameElement = xmlDoc.CreateElement("Name");
    nameElement.AppendChild(xmlDoc.CreateTextNode(name));
    root.AppendChild(nameElement);

    //Create the LoanAmount element
    XmlElement LoanAmount = xmlDoc.CreateElement("LoanAmount");
    LoanAmount.AppendChild(xmlDoc.CreateTextNode(amount));
    root.AppendChild(LoanAmount);

    //Create the PhoneOrEmail element
    XmlElement PhoneOrEmail = xmlDoc.CreateElement("PhoneOrEmail");
    PhoneOrEmail.AppendChild(xmlDoc.CreateTextNode(phone));
    root.AppendChild(PhoneOrEmail);

    //Create the ApprovalStatus element
    XmlElement ApprovalStatus = xmlDoc.CreateElement("ApprovalStatus");
    ApprovalStatus.AppendChild(xmlDoc.CreateTextNode("PENDING APPROVAL"));
    root.AppendChild(ApprovalStatus);

    //Return the XmlElement instance
    return xmlDoc;
}


//Returns the String value of the Job Manager status code
private String getJobDescription(int val)
{
    switch(val)
    {
        case 0:
            return "JOB_STATUS_UNKNOWN";

        case 1:
            return "JOB_STATUS_QUEUED";

        case 2:
            return "JOB_STATUS_RUNNING";

        case 3:
            return "JOB_STATUS_COMPLETED";

        case 4:
            return "JOB_STATUS_FAILED";

         case 5:
            return "JOB_STATUS_COMPLETED";

        case 6:
```

```
                    return "JOB_STATUS_SUSPENDED";

                case 7:
                    return "JOB_STATUS_COMPLETE_REQUESTED";

                case 8:
                    return "JOB_STATUS_TERMINATE_REQUESTED";

                 case 9:
                    return "JOB_STATUS_SUSPEND_REQUESTED";

                   case 10:
                    return "JOB_STATUS_RESUME_REQUESTED";
            }

            return "";
        }
    }
}
```

Note: *The values located in the getJobDescription user-defined method correspond to values returned by the Job Manager service. (See "Retrieving the Status of an AEM Forms Job" on page 1077.)*

Run the ASP.NET application

After you compile and deploy your ASP.NET application, you can execute it using a web browser. Assuming the name of the ASP.NET project is *InvokePreLoanProcess*, specify the following URL within a web browser:

*http://localhost:1629/InvokePreLoanProcess/*Default.aspx

where localhost is the name of the web server hosting the ASP.NET project and 1629 is the port number. When you compile and build your ASP.NET application, Microsoft Visual Studio, automatically deploys it.

Note: *To confirm that the ASP.NET application invoked the process, start Workspace and accept the loan.*

Creating a client application built with Flex that invokes a human-centric long-lived process

You can create a client application built with Flex to invoke the *FirstAppSolution/PreLoanProcess* process. This application uses Remoting to invoke the *FirstAppSolution/PreLoanProcess* process. (See "Invoking AEM Forms using Remoting" on page 444.)

The following illustration shows a client application built with Flex collecting data from an end user. The data is placed into an XML data source and sent to the process.

Notice after the process is invoked, an invocation identifier value is displayed. An invocation identifier value is created as part of a record that tracks the status of the long-lived process.

The client application built with Flex performs the following tasks:

- Retrieves the values that the user entered into the web page.
- Dynamically creates an XML data source that is passed to the *FirstAppSolution/PreLoanProcess* process. The three values are specified in the XML data source.
- Invokes the *FirstAppSolution/PreLoanProcess* process by using Remoting.
- Returns the invocation identifier value of the long-lived process.

Summary of steps

To create a client application built with Flex that is able to invoke the FirstAppSolution/PreLoanProcess process, perform the following steps:

1 Start a new Flex project.

2 Include the adobe-remoting-provider.swc file in your project's class path. (See "Including the AEM Forms Flex library file" on page 446.)

3 Create a `mx:RemoteObject` instance through either ActionScript or MXML. (See)

4 Set up a `ChannelSet` instance to communicate with AEM Forms, and associate it with the `mx:RemoteObject` instance. (See .)

5 Call the ChannelSet's `login` method or the service's `setCredentials` method to specify the user identifier value and password. (See "Using single sign-on" on page 454.)

6 Create the XML data source to pass to the `FirstAppSolution/PreLoanProcess` process by creating an XML instance. (This application logic is shown in the following code example.)

7 Create an object of type Object by using its constructor. Assign the XML to the object by specifying the name of the process's input parameter, as shown in the following code:

```
//Get the XML data to pass to the AEM Forms process
var xml:XML = createXML();
var params:Object = new Object();
params["formData"]=xml;
```

8 Invoke the `FirstAppSolution/PreLoanProcess` process by calling the `mx:RemoteObject` instance's `invoke_Async` method. Pass the `Object` that contains the input parameter. (See .)

9 Retrieve the invocation identification value that is returned from a long-lived process, as shown in the following code:

```
// Handles async call that invokes the long-lived process
private function resultHandler(event:ResultEvent):void
{
ji = event.result as JobId;
jobStatusDisplay.text = "Job Status ID: " + ji.jobId as String;
}
```

Invoking a long-lived process using Remoting

The following Flex code example invokes the `FirstAppSolution/PreLoanProcess` process.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*" backgroundColor="#FFFFFF"
     creationComplete="initializeChannelSet();">

<mx:Script>
         <![CDATA[

            import mx.controls.Alert;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import flash.net.navigateToURL;
            import mx.messaging.ChannelSet;
            import mx.messaging.channels.AMFChannel;
            import mx.collections.ArrayCollection;
            import mx.rpc.livecycle.JobId;
            import mx.rpc.livecycle.JobStatus;
            import mx.rpc.livecycle.DocumentReference;
            import mx.formatters.NumberFormatter;

            // Holds the job ID returned by LC.JobManager
            private var ji:JobId;

            private function initializeChannelSet():void
            {
            var cs:ChannelSet= new ChannelSet();
        cs.addChannel(new AMFChannel("remoting-amf", "http://hiro-
xp:8080/remoting/messagebroker/amf"));
        LC_MortgageApp.setCredentials("tblue", "password");
        LC_MortgageApp.channelSet = cs;
            }

           private function submitApplication():void
            {
            //Get the XML data to pass to the AEM Forms process
            var xml:XML = createXML();
            var params:Object = new Object();
            params["formData"]=xml;
            LC_MortgageApp.invoke_Async(params);
            }

            // Handles async call that invokes the long-lived process
            private function resultHandler(event:ResultEvent):void
            {
               ji = event.result as JobId;
               jobStatusDisplay.text = "Job Status ID: " + ji.jobId as String;
            }

            private function createXML():XML
            {
               //Calculate the Mortgage value to place in the XML data
               var propertyPrice:String = txtAmount.text ;
               var name:String = txtName.text ;
               var phone:String = txtPhone.text ;;

               var model:XML =
```

```
                <LoanApp>
                        <Name>{name}</Name>
                        <LoanAmount>{propertyPrice}</LoanAmount>
                        <PhoneOrEmail>{phone}</PhoneOrEmail>
                        <ApprovalStatus>PENDING APPROVAL</ApprovalStatus>
                </LoanApp>

            return model;
           }


        ]]>
     </mx:Script>

     <!-- Declare the RemoteObject and set its destination to the mortgage-app remoting endpoint
defined in AEM Forms. -->
     <mx:RemoteObject id="LC_MortgageApp" destination="FirstAppSolution/PreLoanProcess"
result="resultHandler(event);">
        <mx:method name="invoke_Async" result="resultHandler(event)"/>
    </mx:RemoteObject>


   <mx:Grid x="229" y="186">
       <mx:GridRow width="100%" height="100%">
           <mx:GridItem width="100%" height="100%">
               <mx:Image>
                   <mx:source>file:///D|/LiveCycle_9/FirstApp/financeCorpLogo.jpg</mx:source>
               </mx:Image>
           </mx:GridItem>
           <mx:GridItem width="100%" height="100%">
               <mx:Label text="Flex Loan Application Page" fontSize="20"/>
           </mx:GridItem>
           <mx:GridItem width="100%" height="100%">
           </mx:GridItem>
       </mx:GridRow>
       <mx:GridRow width="100%" height="100%">
           <mx:GridItem width="100%" height="100%">
               <mx:Label text="Name:" fontSize="12" fontWeight="bold"/>
           </mx:GridItem>
           <mx:GridItem width="100%" height="100%">
               <mx:TextInput id="txtName"/>
           </mx:GridItem>
           <mx:GridItem width="100%" height="100%">
               <mx:Button label="Submit Application" click="submitApplication()"/>
           </mx:GridItem>
       </mx:GridRow>
       <mx:GridRow width="100%" height="100%">
           <mx:GridItem width="100%" height="100%">
               <mx:Label text="Phone/Email:" fontSize="12" fontWeight="bold"/>
           </mx:GridItem>
           <mx:GridItem width="100%" height="100%">
               <mx:TextInput id="txtPhone"/>
           </mx:GridItem>
           <mx:GridItem width="100%" height="100%">
           </mx:GridItem>
       </mx:GridRow>
       <mx:GridRow width="100%" height="100%">
```

```
            <mx:GridItem width="100%" height="100%">
                <mx:Label text="Amount:" fontSize="12" fontWeight="bold"/>
            </mx:GridItem>
            <mx:GridItem width="100%" height="100%">
                <mx:TextInput id="txtAmount"/>
            </mx:GridItem>
            <mx:GridItem width="100%" height="100%">
            </mx:GridItem>
        </mx:GridRow>
        <mx:GridRow width="100%" height="100%">
            <mx:GridItem width="100%" height="100%">
            </mx:GridItem>
            <mx:GridItem width="100%" height="100%">
                <mx:Label text="Label" id="jobStatusDisplay" enabled="true" fontSize="12"
fontWeight="bold"/>
            </mx:GridItem>
            <mx:GridItem width="100%" height="100%">
            </mx:GridItem>
        </mx:GridRow>
    </mx:Grid>

</mx:Application>
```

Chapter 4: Performing Service Operations Using APIs

Performing Service Operations Using APIs

Before you start developing client applications by using the AEM Forms APIs, it is recommended that you first read Invoking AEM Forms, which describes the different ways in which to invoke services. (See "Service container" on page 443.)

After you become familiar with the different invocation methods, you can learn how to programmatically interact with each service. You can develop a client application in Adobe Flex® Builder™, in a Java development environment, or in an environment such as Microsoft Visual Studio .NET that permits you to use the exposed WSDL for consumption on a native SOAP stack.

Each topic includes introductory information (including a step summary section), code walkthroughs, and code examples. The summary of steps explain the required sub-tasks and each sub-task links to a section in the code walkthroughs. All topics have links to Quick Starts, which are full code examples that are designed to help you get started quickly by copying and pasting the code into your project. (See "Java API(SOAP) Quick Start (Code Examples)" on page 2.)

Rendering Forms

Rendering Forms

About the Forms service

The Forms service lets you create interactive data capture client applications that validate, process, transform, and deliver forms typically created in Designer. Form authors can develop a single form design that the Forms service renders in PDF, SWF, or HTML in various browser environments.

When an end-user requests a form, a client application sends the request to the Forms service, which returns the form in an appropriate format. As soon as the Forms service receives a request, it merges data with a form design and then delivers the form in the desired format. The Form service output is an interactive form, typically a PDF document. An interactive form enables users to fill in fields located on the form.

Depending upon the type of client application, you can write the form to a client web browser or save the form as a PDF file. A web-based application can write the form to web browser. A desktop application can save the form as a PDF file. To demonstrate how to write out to a web browser and to a PDF file, the quick starts located in the *Rendering Forms* section are organized in the following manner:

- The Java API strongly typed (SOAP mode) examples are a Java servlet.
- The web service (Java Base64) examples are a Java servlet.
- The web service (MTOM) examples are a console application (not all quick starts have a MTOM example).

  Note: *For information about creating a web application that uses java servlets to invoke the Forms service, see "Creating Web Applications that Renders Forms" on page 670.*

  You can pass a form design (an XDP file) or a PDF document to the Forms service using one of two ways:

- You can reference the form design using a URL value. This approach involves using a `URLSpec` object. The content root is passed to the Forms service using the `URLSpec` object's `setContentRootURI` method. The Form design name (`formQuery`) is passed as a separate parameter. The two values are concatenated to get the absolute reference to the form design. (Most of the quick starts located in the *Rendering Forms* section use this approach.)
- You can pass a `com.adobe.idp.Document` that contains the form design to the Forms service. Two new methods named `renderPDFForm2` and `renderHTMLForm2` accept a `com.adobe.idp.Document` object that contains a form design. (See "Passing Documents to the Forms Service" on page 591.)

You can accomplish these tasks using the Forms service:

- Render interactive PDF forms. (See "Rendering Interactive PDF Forms" on page 582.)
- Render forms at the client. (See "Rendering Forms at the Client" on page 596.)
- Render forms based on fragments. (See "Rendering Forms Based on Fragments" on page 600.)
- Render rights-enabled forms. (See "Rendering Rights-Enabled Forms" on page 605.)
- Render forms as HTML. (See "Rendering Forms as HTML" on page 609.)
- Rendering HTML Forms Using Custom CSS Files ("Rendering HTML Forms Using Custom CSS Files" on page 618.)
- Handle submitted forms. (See "Handling Submitted Forms" on page 630.)
- Creating PDF Documents with Submitted XML Data. (See "Creating PDF Documents with Submitted XML Data" on page 639.)
- Prepopulate forms. (See "Prepopulating Forms with Flowable Layouts" on page 644.)
- Passing Documents. (See "Passing Documents to the Forms Service" on page 591.)
- Calculate form data. (See "Calculating Form Data" on page 656.)
- Optimize an application. (See "Optimizing the Performance of the Forms Service" on page 666.)

The Adobe Developer web site contains the following article that discusses how to create a ASP.NET application that invokes the Forms service and renders forms. See Creating form rendering ASP.NET applications.

Rendering Interactive PDF Forms

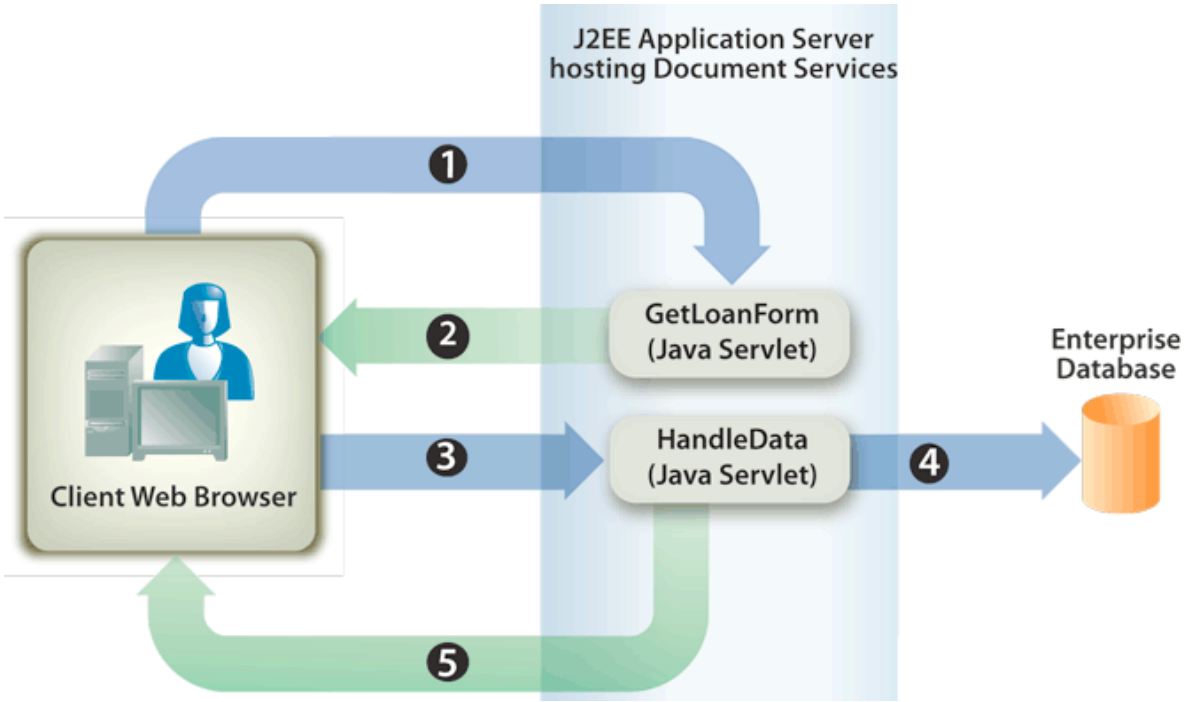
Rendering Interactive PDF Forms

The Forms service renders interactive PDF forms to client devices, typically web browsers, to collect information from users. After an interactive form is rendered, a user can enter data into form fields and click a submit button located on the form to send information back to the Forms service. Adobe Reader or Acrobat must be installed on the computer hosting the client web browser in order for an interactive PDF form to be visible.

Note: *Before you can render a form using the Forms service, create a form design. Typically, a form design is created in Designer and is saved as an XDP file. For information about creating a form design, see Forms Designer.*

Sample loan application

A sample loan application is introduced to demonstrate how the Forms service uses interactive forms to collect information from users. This application lets a user fill in a form with data required to secure a loan and then submits data to the Forms service. The following diagram shows the loan application's logic flow.

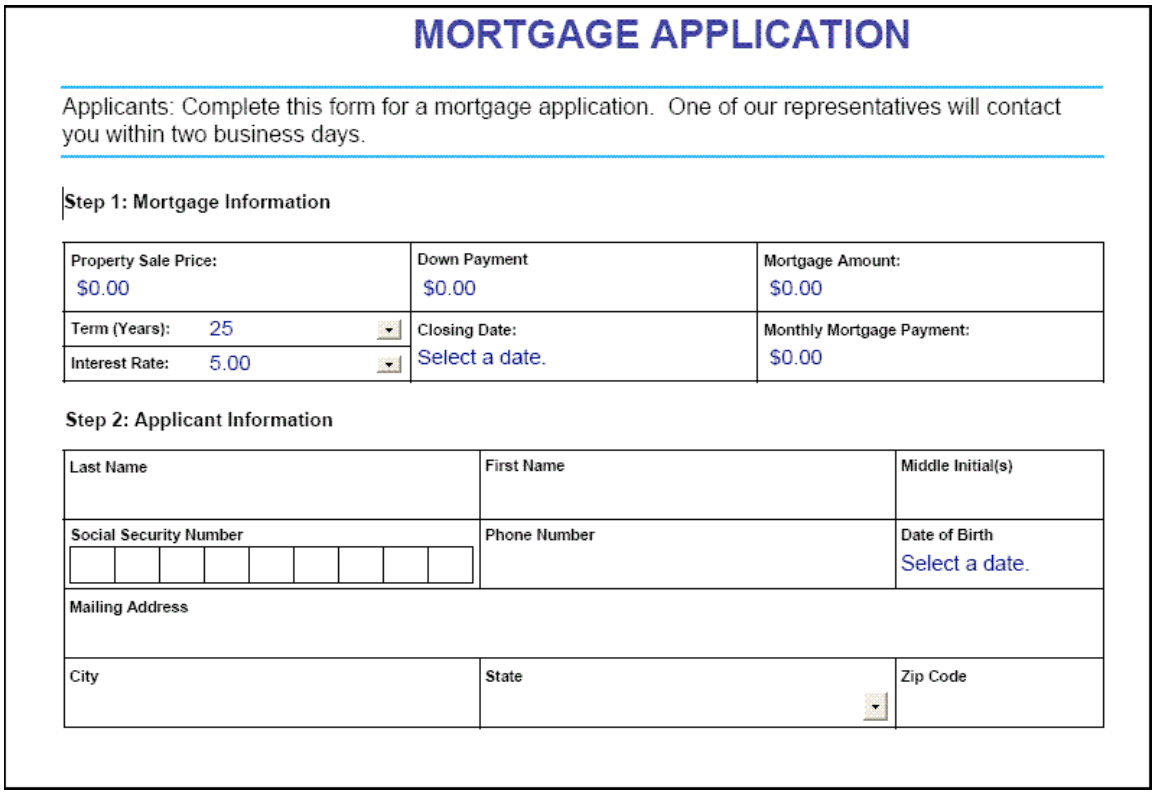


The following table describes the steps in this diagram.

Step	Description
1	The `GetLoanForm` Java Servlet is invoked from an HTML page.
2	The `GetLoanForm` Java Servlet uses the Forms service Client API to render the loan form to the client web browser. (See "Render an interactive PDF form using the Java API" on page 587.)
3	After the user fills the loan form and clicks the submit button, data is submitted to the `HandleData` Java Servlet. (See *"Loan form"*.)
4	The `HandleData` Java Servlet uses the Forms service Client API to process the form submission and retrieve form data. The data is then stored in an enterprise database. (See "Handling Submitted Forms" on page 630.)
5	A confirmation form is rendered back to the web browser. Data such as the user's first and last name is merged with the form before it is rendered. (See "Prepopulating Forms with Flowable Layouts" on page 644.)

Loan form

This interactive loan form is rendered by the sample loan application's `GetLoanForm` Java Servlet.



MORTGAGE APPLICATION

Applicants: Complete this form for a mortgage application. One of our representatives will contact you within two business days.

Step 1: Mortgage Information

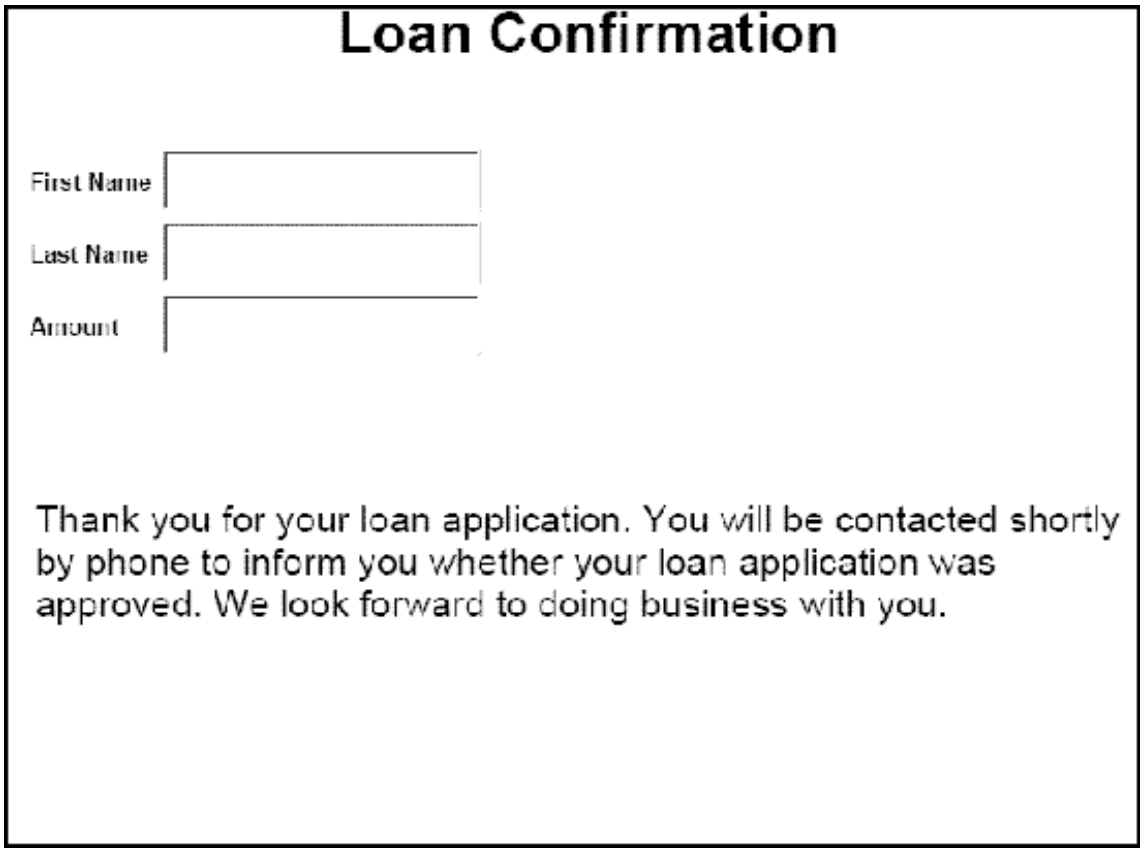
Property Sale Price: $0.00	Down Payment $0.00	Mortgage Amount: $0.00
Term (Years): 25	Closing Date: Select a date.	Monthly Mortgage Payment: $0.00
Interest Rate: 5.00		

Step 2: Applicant Information

Last Name	First Name	Middle Initial(s)
Social Security Number	Phone Number	Date of Birth Select a date.
Mailing Address		
City	State	Zip Code

Confirmation form

This form is rendered by the sample loan application's `HandleData` Java Servlet.



The `HandleData` Java Servlet prepopulates this form with the user's first and last name as well as the amount. After the form is prepopulated, it is sent to the client web browser. (See "Prepopulating Forms with Flowable Layouts" on page 644.)

Java Servlets

The sample loan application is an example of a Forms service application that exists as a Java Servlet. A Java Servlet is a Java program running on a J2EE application server, such as WebSphere, and contains Forms service Client API code.

The following code shows the syntax of a Java Servlet named GetLoanForm:

```
public class GetLoanForm extends HttpServlet implements Servlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp
    throws ServletException, IOException {

    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp
    throws ServletException, IOException {

        }
```

Normally, you would not place Forms service Client API code within a Java Servlet's `doGet` or `doPost` method. It is better programming practice to place this code within a separate class, instantiate the class from within the `doPost` method (or `doGet` method), and call the appropriate methods. However, for code brevity, the code examples in this section are kept to a minimum and code examples are placed in the `doPost` method.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To render an interactive PDF form, perform the following tasks:

1. Include project files.
2. Create a Forms Client API object.
3. Specify URI values.
4. Attach files to the form (Optional).
5. Render an interactive PDF form.
6. Write the form data stream to the client web browser.

Include project files

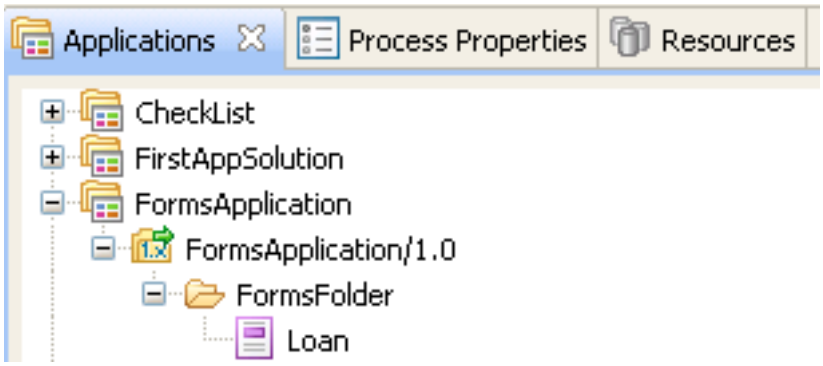
Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms Client API object. If you are using the Java API, create a `FormsServiceClient` object. If you are using the Forms web service API, create a `FormsService` object.

Specify URI values

You can specify URI values that are required by the Forms service to render a form. A form design that is saved as part of a Forms application can be referenced by using the content root URI value `repository:///`. For example, consider the following form design named *Loan.xdp* located within a Forms application named *FormsApplication*:



To access this form design, specify `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp` as the form name (the first parameter passed to the `renderPDFForm` method) and `repository:///` as the content root URI value.

Note: *For information about creating a Forms application using Workbench, see Workbench Help.*

The path to a resource located in a Forms application is:

```
Applications/Application-name/Application-version/Folder.../Filename
```

The following values show some examples of URI values:

- Applications/AppraisalReport/1.0/Forms/FullForm.xdp
- Applications/AnotherApp/1.1/Assets/picture.jpg
- Applications/SomeApp/2.0/Resources/Data/XSDs/MyData.xsd

When you render an interactive form, you can define URI values such as the target URL to where form data is posted. The target URL can be defined in one of the following ways:

- On the Submit button while designing the form design in Designer
- By using the Forms service Client API

If the target URL is defined within the form design, do not override it with the Forms service Client API. That is, setting the target URL using the Forms API resets the specified URL in the form design to the one specified using the API. If you wish to submit the PDF form to the target URL specified in the form design, then programmatically set the target URL to an empty string.

If you have a form that contains a submit button and a calculate button (with a corresponding script that runs at the server), you can programmatically define the URL to where the form is sent to execute the script. Use the submit button on the form design to specify the URL to where form data is posted. (See "Calculating Form Data" on page 656.)

Note: *Instead of specifying a URL value to reference a XDP file, you can also pass a* `com.adobe.idp.Document` *instance to the Forms service. The* `com.adobe.idp.Document` *instance contains a form design. (See "Passing Documents to the Forms Service" on page 591.)*

Attach files to the form

You can attach files to a form. When you render a PDF form with file attachments, users can retrieve the file attachments in Acrobat using the file attachment pane. You can attach different file types to a form, such as a text file, or to a binary file such as a JPG file.

Note: *Attaching file attachments to a form is optional.*

Render an interactive PDF form

To render a form, use a form design that was created in Designer and saved as an XDP or PDF file. As well, you can render a form that was created using Acrobat and saved as a PDF file. To render an interactive PDF form, invoke the `FormsServiceClient` object's `renderPDFForm` method or `renderPDFForm2` method.

The `renderPDFForm` uses a `URLSpec` object. The content root to the XDP file is passed to the Forms service using the `URLSpec` object's `setContentRootURI` method. The Form design name (`formQuery`) is passed as a separate parameter value. The two values are concatenated to get the absolute reference to the form design.

The `renderPDFForm2` method accepts a `com.adobe.idp.Document` instance that contains the XDP or PDF document to render.

Note: *The tagged PDF run-time option cannot be set if the input document is a PDF document. If the input file is an XDP file, the tagged PDF option can be set.*

Render an interactive PDF form using the Java API

Render an interactive PDF form by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Specify URI values

- Create a `URLSpec` object that stores URI values by using its constructor.
- Invoke the `URLSpec` object's `setApplicationWebRoot` method and pass a string value that represents the application's web root.
- Invoke the `URLSpec` object's `setContentRootURI` method and pass a string value that specifies the content root URI value. Ensure that the form design is located in the content root URI. If not, the Forms service throws an exception. To reference the repository, specify `repository:///`.
- Invoke the `URLSpec` object's `setTargetURL` method and pass a string value that specifies the target URL value to where form data is posted. If you define the target URL in the form design, you can pass an empty string. You can also specify the URL to where a form is sent in order to perform calculations.

4 Attach files to the form

- Create a `java.util.HashMap` object to store file attachments by using its constructor.
- Invoke the `java.util.HashMap` object's `put` method for each file to attach to the rendered form. Pass the following values to this method:
  - A string value that specifies the name of the file attachment, including the file name extension.
- A `com.adobe.idp.Document` object that contains the file attachment.

***Note:** Repeat this step for each file to attach to the form. This step is optional and you can pass `null` if you do not want to send file attachments.*

5 Render an interactive PDF form

Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- A `PDFFormRenderSpec` object that stores run-time options. This is an optional parameter and you can specify `null` if you do not want to specify run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

6 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.

- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

Render an interactive PDF form using the web service API

Render an interactive PDF form by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Specify URI values

- Create a `URLSpec` object that stores URI values by using its constructor.
- Invoke the `URLSpec` object's `setApplicationWebRoot` method and pass a string value that represents the application's web root.
- Invoke the `URLSpec` object's `setContentRootURI` method and pass a string value that specifies the content root URI value. Ensure that the form design is located in the content root URI. If not, the Forms service throws an exception. To reference the repository, specify `repository:///`.
- Invoke the `URLSpec` object's `setTargetURL` method and pass a string value that specifies the target URL value to where form data is posted. If you define the target URL in the form design, you can pass an empty string. You can also specify the URL to where a form is sent in order to perform calculations.

4 Attach files to the form

- Create a `java.util.HashMap` object to store file attachments by using its constructor.
- Invoke the `java.util.HashMap` object's `put` method for each file to attach to the rendered form. Pass the following values to this method:
  - A string value that specifies the name of the file attachment, including the file name extension
- A `BLOB` object that contains the file attachment

Note: *Repeat this step for each file to attach to the form.*

5 Render an interactive PDF form

Invoke the `FormsService` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.

- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`.
- A `PDFFormRenderSpec` object that stores run-time options. This is an optional parameter and you can specify `null` if you do not want to specify run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This is used to store the rendered PDF form.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. (This argument will store the number of pages in the form.)
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. (This argument will store the locale value.)
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

6 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

Write the form data stream to the client web browser

When the Forms service renders a form, it returns a form data stream that you must write to the client web browser. When written to the client web browser, the form is visible to the user.

More Help topics

"Creating Web Applications that Renders Forms" on page 670

"Quick Start (SOAP mode): Rendering an interactive PDF form using the Java API" on page 154

"Invoking AEM Forms using Base64 encoding" on page 525

"Prepopulating Forms with Flowable Layouts" on page 644

"Including AEM Forms Java library files" on page 491

"Calculating Form Data" on page 656

Passing Documents to the Forms Service

Passing Documents to the Forms Service

The AEM Forms service renders interactive PDF forms to client devices, typically web browsers, to collect information from users. An interactive PDF form is based on a form design that is typically saved as an XDP file and created in Designer. As of AEM Forms, you can pass a `com.adobe.idp.Document` object that contains the form design to the Forms service. The Forms service then renders the form design located in the `com.adobe.idp.Document` object.

An advantage of passing a `com.adobe.idp.Document` object to the Forms service is that other service operations return a `com.adobe.idp.Document` instance. That is, you can get a `com.adobe.idp.Document` instance from another service operation and render it. For example, assume that an XDP file is stored in a Content Services (deprecated) node named `/Company Home/Form Designs`, as shown in the following illustration.

You can programmatically retrieve Loan.xdp from Content Services (deprecated) (deprecated) and pass the XDP file to the Forms service within a `com.adobe.idp.Document` object.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To pass a document obtained from Content Services (deprecated) (deprecated) to the Forms service, perform the following tasks:

1. Include project files.
2. Create a Forms and a Document Management Client API object.
3. Retrieve the form design from Content Services (deprecated).
4. Render the interactive PDF form.
5. Perform an action with the form data stream.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create a Forms and a Document Management Client API object

Before you can programmatically perform a Forms service API operation, create a Forms Client API object. Also, because this workflow retrieves an XDP file from Content Services (deprecated), create a Document Management API object.

Retrieve the form design from Content Services (deprecated)

Retrieve the XDP file from Content Services (deprecated) by using the Java or web service API. The XDP file is returned within a `com.adobe.idp.Document` instance (or a `BLOB` instance if you are using web services). You can then pass the `com.adobe.idp.Document` instance to the Forms service.

Render an interactive PDF form

To render an interactive form, pass the `com.adobe.idp.Document` instance that was returned from Content Services (deprecated) to the Forms service.

Note: *You can pass a `com.adobe.idp.Document` that contains the form design to the Forms service. Two new methods named `renderPDFForm2` and `renderHTMLForm2` accept a `com.adobe.idp.Document` object that contains a form design.*

Perform an action with the form data stream

Depending on the type of client application, you can write the form to a client web browser or save the form as a PDF file. A web-based application typically writes the form to web browser. However, a desktop application typically saves the form as a PDF file.

See also

"Pass documents to the Forms Service using the Java API" on page 592

"Pass documents to the Forms Service using the web service API" on page 594

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

Retrieving Content from Content Services (deprecated)

Pass documents to the Forms Service using the Java API

Pass a document obtained from Content Services (deprecated) by using the Forms service and Content Services (deprecated) API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar and adobe-contentservices-client.jar, in your Java project's class path.

2 Create a Forms and a Document Management Client API object

- Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
- Create a `DocumentManagementServiceClientImpl` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the form design from Content Services (deprecated)

Invoke the `DocumentManagementServiceClientImpl` object's `retrieveContent` method and pass the following values:

- A string value that specifies the store where the content is added. The default store is `SpacesStore`. This value is a mandatory parameter.
- A string value that specifies the fully qualified path of the content to retrieve (for example, `/Company Home/Form Designs/Loan.xdp`). This value is a mandatory parameter.

- A string value that specifies the version. This value is an optional parameter, and you can pass an empty string. In this situation, the latest version is retrieved.

The `retrieveContent` method returns a `CRCResult` object that contains the XDP file. Obtain a `com.adobe.idp.Document` instance by invoking the `CRCResult` object's `getDocument` method.

4 Render an interactive PDF form

Invoke the `FormsServiceClient` object's `renderPDFForm2` method and pass the following values:

- A `com.adobe.idp.Document` object that contains the form design retrieved from Content Services (deprecated).
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- A `PDFFormRenderSpec` object that stores run-time options. This value is an optional parameter, and you can specify `null` if you do not want to specify run-time options.
- A `URLSpec` object that contains URI values. This value is an optional parameter, and you can specify `null`.
- A `java.util.HashMap` object that stores file attachments. This value is an optional parameter, and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Perform an action with the form data stream

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method. Pass the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Passing Documents to the Forms Service" on page 591

"Quick Start (SOAP mode): Passing documents to the Forms Service using the Java API" on page 204

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Pass documents to the Forms Service using the web service API

Pass a document obtained from Content Services (deprecated) by using the Forms service and Content Services (deprecated) API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Because this client application invokes two AEM Forms services, create two service references. Use the following WSDL definition for the service reference associated with the Forms service: `http://localhost:8080/soap/services/FormsService?WSDL&lc_version=9.0.1`.

Use the following WSDL definition for the service reference associated with the Document Management service: `http://localhost:8080/soap/services/DocumentManagementService?WSDL&lc_version=9.0.1`.

Because the `BLOB` data type is common to both service references, fully qualify the `BLOB` data type when using it. In the corresponding web service quick start, all `BLOB` instances are fully qualified.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Forms and a Document Management Client API object

- Create a `FormsServiceClient` object by using its default constructor.
- Create a `FormsServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/FormsService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `FormsServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `FormsServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `FormsServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

Note: *Repeat these steps for the `DocumentManagementServiceClient` service client.*

3 Retrieve the form design from Content Services (deprecated)

Retrieve content by invoking the `DocumentManagementServiceClient` object's `retrieveContent` method and passing the following values:

- A string value that specifies the store where the content is added. The default store is `SpacesStore`. This value is a mandatory parameter.
- A string value that specifies the fully qualified path of the content to retrieve (for example, `/Company Home/Form Designs/Loan.xdp`). This value is a mandatory parameter.
- A string value that specifies the version. This value is an optional parameter, and you can pass an empty string. In this situation, the latest version is retrieved.

- A string output parameter that stores the browse link value.
- A `BLOB` output parameter that stores the content. You can use this output parameter to retrieve the content.
- A `ServiceReference1.MyMapOf_xsd_string_To_xsd_anyType` output parameter that stores content attributes.
- A `CRCResult` output parameter. Instead of using this object, you can use the `BLOB` output parameter to obtain the content.

4 Render an interactive PDF form

Invoke the `FormsServiceClient` object's `renderPDFForm2` method and pass the following values:

- A `BLOB` object that contains the form design retrieved from Content Services (deprecated).
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass an empty `BLOB` object.
- A `PDFFormRenderSpec` object that stores run-time options. This value is an optional parameter, and you can specify `null` if you do not want to specify run-time options.
- A `URLSpec` object that contains URI values. This value is an optional parameter, and you can specify `null`.
- A `Map` object that stores file attachments. This value is an optional parameter, and you can specify `null` if you do not want to attach files to the form.
- A long output parameter that is used to store the page count.
- A string output parameter that is used to store the locale value.
- A `FormsResult` output parameter that is used to store the interactrive PDF form .

The `renderPDFForm2` method returns a `FormsResult` object that contains the interactive PDF form.

5 Perform an action with the form data stream

- Create a `BLOB` object that contains form data by getting the value of the `FormsResult` object's `outputContent` field.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the interactive PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object retrieved from the `FormsResult` object. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Passing Documents to the Forms Service" on page 591

Quick Start (MTOM): Passing documents to the Forms Service using the web service API

"Invoking AEM Forms using MTOM" on page 529

Rendering Forms at the Client

Rendering Forms at the Client

You can optimize the delivery of PDF content and improve the Forms service's ability to handle network load by using the client-side rendering capability of Acrobat or Adobe Reader. This process is known as rendering a form at the client. To render a form at the client, the client device (typically a web browser) must use Acrobat 7.0 or Adobe Reader 7.0 or later.

Changes to a form resulting from server-side script execution is not reflected in a form that is rendered at the client unless the root subform contains the `restoreState` attribute that is set to `auto`. For more information about this attribute, see Forms Designer.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To render a form at the client, perform the following tasks:

1. Include project files.
2. Create a Forms Client API object.
3. Set client rendering run-time options.
4. Render a form at the client.
5. Write the form to the client web browser.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms service client. If you are using the Java API, create a `FormsServiceClient` object. If you are using the Forms web service API, create a `FormsService` object.

Set client rendering run-time options

You must set the client rendering run-time option to render a form at the client by setting the `RenderAtClient` run-time option to `true`. This results in the form being delivered to the client device where it is rendered. If `RenderAtClient` is `auto` (the default value), the form design determines whether the form is rendered at the client. The form design must be a form design with a flowable layouyt.

An optional run-time option that you may set is the `SeedPDF` option. The `SeedPDF` option combines the PDF container (seed PDF document) with the form design and the XML data. Both the form design and the XML data are delivered to Acrobat or Adobe Reader, where the form is rendered. The `SeedPDF` option can be used when the client computer does not have fonts that are used in the form, such as when an end user is not licensed to use a font that the form owner is licensed to use.

You can use Designer to create a simple dynamic PDF file for use as a seed PDF file. The following steps are required to perform this task:

1 Determine whether you need to embed any fonts within the seed PDF file. The seed PDF file will need to contain additional fonts required by the form being rendered. When embedding fonts into the seed PDF file, ensure that you are not violating any font licensing agreements. In Designer, you can determine whether you can legally embed fonts. Upon saving, if there are fonts you cannot embed into the form, Designer displays a message listing the fonts you cannot embed. This message is not displayed in Designer for static PDF documents.

2 If you are creating the seed PDF file in Designer, it is recommended that, at a minimum, you add a text field that contains a message. The message should be directed at users of earlier versions of Adobe Reader stating that they need Acrobat 7.0 or later or Adobe Reader 7.0 or later to view the document.

3 Save the seed PDF file as a dynamic PDF file with the PDF file name extension.

Note: *You do not need to define the seed PDF run-time option to render a form on the client. If you do not specify a seed PDF, the Forms service creates a shell pdf which will not contain COS objects but will contain a PDF wrapper with the actual XDP content embedded inside. The steps in this section do not set the seed PDF run-time option. For information about COS objects, see the Adobe PDF Reference guide.*

Render a form at the client

To render a form at the client, you must ensure that the client rendering run-time options are included in your application logic to render a form.

Write the form data stream to the client web browser

The Forms service creates a form data stream that you must write to the client web browser. When written to the client web browser, the form is rendered by Acrobat 7.0 or Adobe Reader 7.0 or later, and is visible to the user.

See also

"Render a form at the client using the Java API" on page 597

"Render a form at the client using the web service API" on page 599

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Passing Documents to the Forms Service" on page 591

"Creating Web Applications that Renders Forms" on page 670

Render a form at the client using the Java API

Render a form at the client by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set client rendering run-time options

- Create a `PDFFormRenderSpec` object by using its constructor.
- Set the `RenderAtClient` run-time option by invoking the `PDFFormRenderSpec` object's `setRenderAtClient` method and passing the enum value `RenderAtClient.Yes`.

4 Render a form at the client

Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a AEM Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- A `PDFFormRenderSpec` object that stores run-time options required to render a form at the client.
- A `URLSpec` object that contains URI values that are required by the Forms service to render a form.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms at the Client" on page 596

"Quick Start (SOAP mode): Rendering a form at the client using the Java API" on page 156

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Render a form at the client using the web service API

Render a form at the client by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Set client rendering run-time options

- Create a `PDFFormRenderSpec` object by using its constructor.
- Set the `RenderAtClient` run-time option by invoking the `PDFFormRenderSpec` object's `setRenderAtClient` method and passing the string value `RenderAtClient.Yes`.

4 Render a form at the client

Invoke the `FormsService` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`. (See "Prepopulating Forms with Flowable Layouts" on page 644.)
- A `PDFFormRenderSpec` object that stores run-time options required to render a form at the client.
- A `URLSpec` object that contains URI values that are required by the Forms service. (See .)
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form. (See .)
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This parameter is used to store the rendered PDF form.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. (This argument will store the number of pages in the form).
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. (This argument will store the locale value).
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.

- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms at the Client" on page 596

Quick Start (Base64): Rendering a form at the client using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Rendering Forms Based on Fragments

Rendering Forms Based on Fragments

The Forms service can render forms that are based on fragments that you create using Designer. A *fragment* is a reusable part of a form and is saved as a separate XDP file that can be inserted into multiple form designs. For example, a fragment can include an address block or legal text.

Using fragments simplifies and speeds up the creation and maintenance of large numbers of forms. When creating a new form, you insert a reference to the required fragment and the fragment appears in the form. The fragment reference contains a subform that points to the physical XDP file. For information about creating form designs based on fragments, see Forms Designer

A fragment can include several subforms that are wrapped in a choice subform set. Choice subform sets control the display of subforms based on the flow of data from a data connection. You use conditional statements to determine which subform from within the set appears in the delivered form. For example, each subform in a set can include information for a particular geographic location and the subform that is displayed can be determined based on the location of the user.

A *script fragment* contains reusable JavaScript functions or values that are stored separately from any particular object, such as a date parser or a web service invocation. These fragments include a single script object that appears as a child of variables in the Hierarchy palette. Fragments cannot be created from scripts that are properties of other objects, such as event scripts like validate, calculate, or initialize.

Here are advantages of using fragments:

- **Content reuse**: You can use fragments to reuse content in multiple form designs. When you need to use some of the same content in multiple forms, it is faster and simpler to use a fragment than to copy or re-create the content. Using fragments also ensures that the frequently used parts of a form design have consistent content and appearance in all the referencing forms.
- **Global updates**: You can use fragments to make global changes to multiple forms only once, in one file. You can change the content, script objects, data bindings, layout, or styles in a fragment, and all XDP forms that reference the fragment will reflect the changes.
- For example, a common element across many forms might be an address block that includes a drop-down list object for the country. If you need to update the values for the drop-down list object, you must open many forms to make the changes. If you include the address block in a fragment, you only need to open one fragment file to make the changes.
- To update a fragment in a PDF form, you must resave the form in Designer.

- **Shared form creation**: You can use fragments to share the creation of forms among several resources. Form developers with expertise in scripting or other advanced features of Designer can develop and share fragments that take advantage of scripting and dynamic properties. Form designers can use those fragments to lay out form designs and to ensure that all parts of a form have a consistent appearance and functionality across multiple forms designed by multiple people.

Assembling a form design assembled using fragments

You can assemble a form design to pass to the Forms service based on multiple fragments. To assemble multiple fragments, use the Assembler service. To see an example of using the Assemble service to create a form design which is used by another Forms services (the Output service), see "Creating PDF Documents Using Fragments" on page 703. Instead of using the Output service, you can perform the same workflow using the Forms service.

When using the Assembler service, you are passing a form design that was assembled using fragments. The form design that was created does not reference other fragments. In contrast, this topic discusses passing a form design that references other fragments to the Forms service. However, the form design was not assembled by Assembler. It was created in Designer.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Note: *For information about creating a web-based application that renders forms based on fragments, see "Creating Web Applications that Renders Forms" on page 670.*

Summary of steps

To render a form based on fragments, perform the following tasks:

1. Include project files.
2. Create a Forms Client API object.
3. Specify URI values.
4. Render the form.
5. Write the form data stream to the client web browser.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms service client.

Specify URI values

To successfully render a form based on fragments, you must ensure that the Forms service can locate both the form and the fragments (the XDP files) that the form design references. For example, assume the form is named PO.xdp and this form uses two fragments named FooterUS.xdp and FooterCanada.xdp. In this situation, the Forms service must be able to locate all three XDP files.

You can organize a form and its fragments by placing the form in one location and the fragments in another location, or you can place all XDP files in the same location. For the purposes of this section, assume that all XDP files are located in the AEM Forms repository. For information about placing XDP files in the AEM Forms repository, see "Writing Resources" on page 1037.

When rendering a form based on fragments, you must reference only the form itself and not the fragments. For example, you must reference PO.xdp and not FooterUS.xdp or FooterCanada.xdp. Ensure that you place the fragments in a location where the Forms service can locate them.

Render the form

A form based on fragments can be rendered in the same manner as non-fragmented forms. That is, you can render the form as PDF, HTML, or form Guides (deprecated). The example in this section renders a form based on fragments as an interactive PDF form. (See "Rendering Interactive PDF Forms" on page 582.)

Write the form data stream to the client web browser

When the Forms service renders a form, it returns a form data stream that you must write to the client web browser. When written to the client web browser, the form is visible to the user.

See also

"Render forms based on fragments using the Java API" on page 602

"Render forms based on fragments using the web service API" on page 604

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Creating Web Applications that Renders Forms" on page 670

Render forms based on fragments using the Java API

Render a form based on fragments by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Specify URI values

- Create a `URLSpec` object that stores URI values by using its constructor.
- Invoke the `URLSpec` object's `setApplicationWebRoot` method and pass a string value that represents the application's web root.
- Invoke the `URLSpec` object's `setContentRootURI` method and pass a string value that specifies the content root URI value. Ensure that the form design and the fragments are located in the content root URI. If not, the Forms service throws an exception. To reference the repository, specify `repository://`.

- Invoke the `URLSpec` object's `setTargetURL` method and pass a string value that specifies the target URL value to where form data is posted. If you define the target URL in the form design, you can pass an empty string. You can also specify the URL to where a form is sent in order to perform calculations.

4 Render the form

Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- A `PDFFormRenderSpec` object that stores run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service to render a form based on fragments.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms Based on Fragments" on page 600

"Quick Start (SOAP mode): Rendering a form based on fragments using the Java API" on page 161

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Render forms based on fragments using the web service API

Render a form based on fragments using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Specify URI values

- Create a `URLSpec` object that store URI values by using its constructor.
- Invoke the `URLSpec` object's `setApplicationWebRoot` method and pass a string value that represents the application's web root.
- Invoke the `URLSpec` object's `setContentRootURI` method and pass a string value that specifies the content root URI value. Ensure that the form design is located in the content root URI. If not, the Forms service throws an exception. To reference the repository, specify `repository://`.
- Invoke the `URLSpec` object's `setTargetURL` method and pass a string value that specifies the target URL value to where form data is posted. If you define the target URL in the form design, you can pass an empty string. You can also specify the URL to where a form is sent in order to perform calculations.

4 Render the form

Invoke the `FormsService` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`.
- A `PDFFormRenderSpec` object that stores run-time options. Note that the tagged PDF option cannot be set if the input document is a PDF document. If the input file is an XDP file, the tagged PDF option can be set.
- A `URLSpec` object that contains URI values required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This parameter is used to store the rendered form.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. This argument will store the number of pages in the form.
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. This argument will store the locale value.
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms Based on Fragments" on page 600

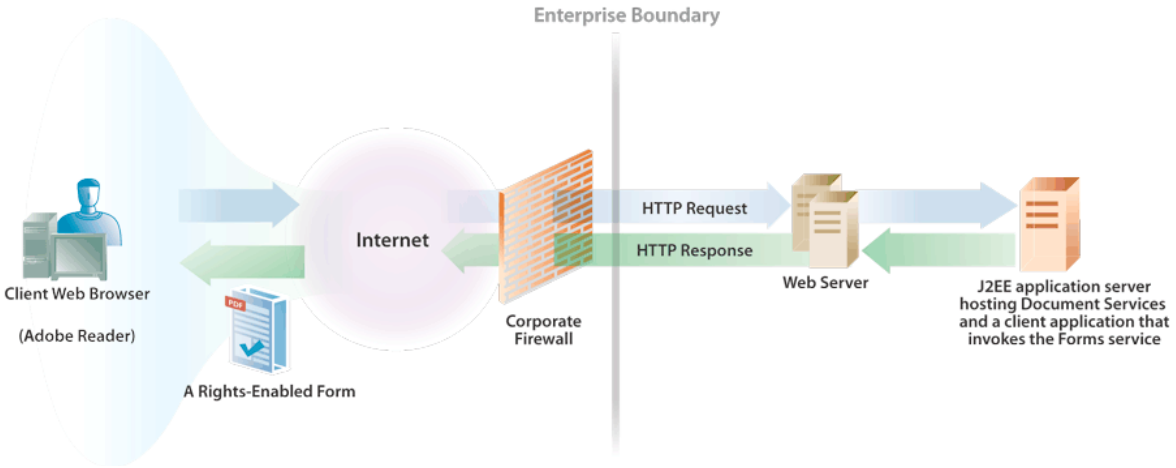
Quick Start (Base64): Rendering a form based on fragments using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Rendering Rights-Enabled Forms

Rendering Rights- Enabled Forms

The Forms service can render forms that have usage rights applied to them. Usage rights pertain to functionality that is available by default in Acrobat but not in Adobe Reader, such as the ability to add comments to a form or to fill in form fields and save the form. Forms that have usage rights applied to them are called rights-enabled forms. A user who opens a rights-enabled form in Adobe Reader can perform operations that are enabled for that form.



In order to apply usage rights to a form, the Acrobat Reader DC extensions service must be part of your AEM forms installation. Also, you must have a valid credential that enables you to apply usage rights to PDF documents. That is, you must properly configure the Acrobat Reader DC extensions service before you can render a rights-enabled form. (See "About the Acrobat Reader DC extensions Service" on page 750.)

Note: *To render a form that contains usage rights, you must use an XDP file as input, not a PDF file. If you use a PDF file as input, the form is still rendered; however, it will not be a rights-enabled form.*

Note: *You cannot prepopulate a form with XML data when you specify the following usage rights:* `enableComments`, `enableCommentsOnline`, `enableEmbeddedFiles`, *or* `enableDigitalSignatures`. *(See "Prepopulating Forms with Flowable Layouts" on page 644.)*

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To render a rights-enabled form, perform the following tasks:

1. Include project files.
2. Create a Forms Client API object.
3. Set usage rights run-time options.
4. Render a rights-enabled form.
5. Write the rights-enabled form to the client web browser.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms service client.

Set usage rights run-time options

You must set usage rights run-time options to render a rights-enabled form. You must also specify the alias of the credential that is used to apply usage rights to a form. After you specify the alias value, you specify each usage right to apply to the form.

Render a rights-enabled form

To render a rights-enabled form, you use the same application logic as rendering a form without usage rights. The only difference is that you must ensure that the usage rights run-time options are included in your application logic.

Note: *When rendering a rights-enabled form using the Forms web service API, you cannot attach files to the form.*

Write the form data stream to the client web browser

When the Forms service renders a rights-enabled form, it returns a form data stream that you must write to the client web browser. Once written to the client web browser, the form is visible to the user. A user viewing the rights-enabled form in Adobe Reader is able to perform operations that are enabled for that form.

See also

"Render rights-enabled forms using the Java API" on page 607

"Render rights-enabled forms using the web service API" on page 608

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Creating Web Applications that Renders Forms" on page 670

Render rights-enabled forms using the Java API

Render a rights-enabled form by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set usage rights run-time options

- Create a `ReaderExtensionSpec` object by using its constructor.
- Specify the alias of the credential by invoking the `ReaderExtensionSpec` object's `setReCredentialAlias` method and specify a string value that represents the alias value.
- Set each usage right by invoking the corresponding method that belongs to the `ReaderExtensionSpec` object. However, you can only set a usage right if the credential that you reference allows you to do so. That is, you cannot set a usage right if the credential does not allow you to set it. For example. to set the usage right that enables a user to fill in form fields and save the form, invoke the `ReaderExtensionSpec` object's `setReFillIn` method and pass `true`.

Note: *It is not necessary to invoke the `ReaderExtensionSpec` object's `setReCredentialPassword` method. This method is not used by the Forms service.*

4 Render a rights-enabled form

Invoke the `FormsServiceClient` object's `renderPDFFormWithUsageRights` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- A `PDFFormRenderSpec` object that stores run-time options.
- A `ReaderExtensionSpec` object that stores usage rights run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.

The `renderPDFFormWithUsageRights` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Rights-Enabled Forms" on page 605

"Quick Start (SOAP mode): Rendering a rights-enabled form using the Java API" on page 164

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Render rights-enabled forms using the web service API

Render a rights-enabled form by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Set usage rights run-time options

- Create a `ReaderExtensionSpec` object by using its constructor.
- Specify the alias of the credential by invoking the `ReaderExtensionSpec` object's `setReCredentialAlias` method and specify a string value that represents the alias value.
- Set each usage right by invoking the corresponding method that belongs to the `ReaderExtensionSpec` object. However, you can only set a usage right if the credential that you reference allows you to do so. That is, you cannot set a usage right if the credential does not allow you to set it. To set the usage right that enables a user to fill in form fields and save the form, invoke the `ReaderExtensionSpec` object's `setReFillIn` method and pass `true`.

4 Render a rights-enabled form

Invoke the `FormsService` object's `renderPDFFormWithUsageRights` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data with the form, you must pass a `BLOB` object that is based on an empty XML data source. You cannot pass a `BLOB` object that is null; otherwise, an exception is thrown.
- A `PDFFormRenderSpec` object that stores run-time options.
- A `ReaderExtensionSpec` object that stores usage rights run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.

The `renderPDFFormWithUsageRights` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Rights-Enabled Forms" on page 605

Quick Start (Base64): Rendering a rights-enabled form using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Rendering Forms as HTML

The Forms service renders forms as HTML in response to an HTTP request from a web browser. A benefit of rendering a form as HTML is that the computer on which the client web browser is located does not require Adobe Reader, Acrobat, or Flash Player (for form Guides (deprecated)).

To render a form as HTML, the form design must be saved as an XDP file. A form design that is saved as a PDF file cannot be rendered as HTML. When developing a form design in Designer that will be rendered as HTML, consider the following criteria:

- Do not use an object's border properties to draw lines, boxes, or grids on your form. Some browsers may not line up borders exactly as they appear in a preview. Objects may appear layered or may push other objects off their expected position.
- You can use lines, rectangles, and circles to define the background.

- Draw text slightly larger than what seems to be required to accommodate the text. Some web browsers do not display the text legibly.

Note: *When rendering a form that contains TIFF images using the* `FormServiceClient` *object's* `(Deprecated) renderHTMLForm` *and* `renderHTMLForm2` *methods, the TIFF images are not visible in the rendered HTML form that is displayed in Internet Explorer or Mozilla Firefox browsers. These browsers do not provide native support for TIFF images.*

HTML pages

When a form design is rendered as an HTML form, each second-level subform is rendered as an HTML page (panel). You can view a subform's hierarchy in Designer. Child subforms that belong to the root subform (the default name of a root subform is form1) are the panel subforms. The following example shows a form design's subforms.

```
form1
    Master Pages
    PanelSubform1
        NestedDynamicSubform
            TextEdit1
    PanelSubform2
        TextEdit1
    PanelSubform3
        TextEdit1
    PanelSubform4
        TextEdit1
```

When form designs are rendered as HTML forms, the panels are not constrained to any particular page size. If you have dynamic subforms, they should be nested within the panel subform. Dynamic subforms are able to expand to an infinite number of HTML pages.

When a form is rendered as an HTML form, page sizes (required for paginating forms rendered as PDF) have no meaning. Because a form with a flowable layout can expand to an infinite number of HTML pages, it is important to avoid footers on the master page. A footer beneath the content area on a master page can overwrite HTML content that flows past a page boundary.

You must explicitly move from panel to panel using the `xfa.host.pageUp` and `xfa.host.pageDown` methods. You change pages by sending a form to the Forms service and having the Forms service render the form back to the client device, typically a web browser.

Note: *The process of sending a form to the Forms service and then having the Forms service render the form back to the client device is referred to as round tripping data to the server.*

Note: *If you want to customize the look of the HTML Digital Signature button on an HTML form, you must change the following properties in the fscdigsig.css file (within the adobe-forms-ds.ear > adobe-forms-ds.war file):*

.fsc-ds-ssb: This style sheet is applicable in case of a blank sign field.

.fsc-ds-ssv: This style sheet is applicable in case of a Valid sign field.

.fsc-ds-ssc: This style sheet is applicable in case of a Valid sign field but data has changed.

.fsc-ds-ssi: This style sheet is applicable in case of a invalid sign field.

.fsc-ds-popup-bg: This style sheet property is not being used.

.fsc-ds-popup-btn: This style sheet property is not being used.

Running scripts

A form author specifies whether a script executes on the server or the client. The Forms service creates a distributed, event processing environment for execution of form intelligence that can be distributed between the client and the server by using the `runAt` attribute. For information about this attribute or creating scripts within form designs, see Forms Designer

The Forms service can execute scripts while the form is being rendered. As a result, you can prepopulate a form with data by connecting to a database or to web services that may not be available on the client. You can also set a button's `Click` event to run on the server so that the client will round trip data to the server. This allows the client to run scripts that may require server resources, such as an enterprise database, while a user is interacting with a form. For HTML forms, formcalc scripts can be executed on server only. As a result, you must mark these scripts to run at `server` or `both`.

You can design forms that move between pages (panels) by calling `xfa.host.pageUp` and `xfa.host.pageDown` methods. This script is placed in a button's `Click` event and the `runAt` attribute is set to `Both`. The reason you choose `Both` is so that Adobe Reader or Acrobat (for forms that are rendered as PDF) can change pages without going to the server and HTML forms can change pages by round tripping data to the server. That is, a form is sent to the Forms service, and a form is rendered back as HTML with the new page displayed.

It is recommended that you do not give script variables and form fields the same names such as item. Some web browsers, such as Internet Explorer, may not initialize a variable with the same name as a form field that results in a script error occurring. It is good practice to give form fields and script variables different names.

When rendering HTML forms that contain both page navigation functionality and form scripts (for example, assume that a script retrieves field data from a database each time the form is rendered), ensure that the form script is located in the form:calculate event instead in of the form:readyevent.

Form scripts that are located in the form:ready event are executed only once during the initial rendering of the form and are not executed for subsequent page retrievals. In contrast, the form:calculate event is executed for each page navigation where the form is rendered.

Note: *On a multipage form, changes made by JavaScript to a page are not retained if you move to a different page.*

You can invoke custom scripts before submitting a form. This feature works on all available browsers. However, it can be used only when users render the HTML form that has its `Output Type` property set to `Form Body`. It will not work when the `Output Type` is `Full HTML`. Refer to Configuring forms in administration help for steps to configure this feature.

You must first define a callback function that is called before submitting the form, where the name of the function is `_user_onsubmit`. It is assumed that the function will not throw any exception, or if it does, the exception will be ignored. It is recommended to place the JavaScript function in the head section of the html; however, you can declare it anywhere before the end of the script tags that include `xfasubset.js`.

When formserver renders an XDP that contains a drop-down list, in addition to creating the drop-down list, it also creates two hidden text fields. These text fields store the data of the drop-down list (one stores the display name of the options and other stores the value for the options). Therefore, every time a user submits the form, the entire data of the drop down list is submitted. Assuming that you don't want to submit that much data everytime, you can write a custom script to disable that. For example: The name of the drop down list is `drpOrderedByStateProv` and it is wrapped under subform header. The name of the HTML input element will be `header[0].drpOrderedByStateProv[0]`. The name of the hidden fields that store and submit the data of the dropdown have the following names:

```
header[0].drpOrderedByStateProv_DISPLAYITEMS_[0]
header[0].drpOrderedByStateProv_VALUEITEMS_[0]
```

You can disable these input elements in the following way if you don't want to post the data.

```
var __CUSTOM_SCRIPTS_VERSION = 1; //enabling the feature
   function _user_onsubmit() {
  var elems = document.getElementsByName("header[0].drpOrderedByStateProv_DISPLAYITEMS_[0]");
   elems[0].disabled = true;
   elems = document.getElementsByName("header[0].drpOrderedByStateProv_VALUEITEMS_[0]");
   elems[0].disabled = true;
   }
```

XFA subsets

When creating form designs to render as HTML, you must restrict your scripting to the XFA subset for scripts in javascript language.

Scripts that run on the client or run on both the client and the server must be written within the XFA subset. Scripts that run on the server can use the full XFA scripting model and also use FormCalc. For information about using JavaScript, see Forms Designer.

When running scripts on the client, only the current panel being displayed can use script; for example, you cannot script against fields that are located in panel A when panel B is displayed. When running scripts on the server, all panels can be accessed.

You must also be careful when using Scripting Object Model (SOM) expressions within scripts that run on the client. Only a simplified subset of SOM expressions are supported by scripts that run on the client.

Event timing

The XFA subset defines the XFA events that are mapped to HTML events. There is a slight difference in behavior on the timing of calculate and validate events. In a web browser, a full calculate event is executed when you exit a field. Calculate events are not automatically executed when you make a change to a field value. You can force a calculate event by calling the `xfa.form.execCalculate` method.

In a web browser, validate events are only executed when exiting a field or submitting a form. You can force a validate event by using the `xfa.form.execValidate` method.

Forms displayed in a web browser (as opposed to Adobe Reader or Acrobat) conform to the XFA null test (errors or warnings) for mandatory fields.

- If the null test produces an error and you exit a field without specifying a value, a message box is displayed and you are repositioned to the field after clicking OK.
- If a null test produces a warning and you exit a field without specifying a value, you are prompted to click either OK or Cancel, giving you the option of proceeding without specifying a value or returning to the field to enter a value.

For more information about a null test, see Forms Designer.

Form buttons

Clicking a submit button sends form data to the Forms service and represents the end of form processing. The `preSubmit` event can be set to run on the client or server. The `preSubmit` event runs prior to the form submission if it is configured to run on the client. Otherwise, the `preSubmit` event runs on the server during the form submission. For more information about the `preSubmit` event, see Forms Designer.

If a button has no client-side script associated with it, data is submitted to the server, calculations are performed on the server, and the HTML form is regenerated. If a button contains a client-side script, data is not sent to the server and the client-side script is executed in the web browser.

HTML 4.0 web browser

A web browser that only supports HTML 4.0 cannot support the XFA subset client-side scripting model. When creating a form design to work in both HTML 4.0 and MSDHTML or CSS2HTML, a script that is marked to run at the client will actually run on the server. For example, assume a user clicks a button that is located on a form displayed in an HTML 4.0 web browser. In this situation, the form data is sent to the server where the client-side script is executed.

It is recommended that you place your form logic in calculate events, which run at the server in HTML 4.0 and on the client for MSDHTML or CSS2HTML.

Maintaining presentation changes

As you move between HTML pages (panels), only the state of the data is maintained. Settings such as background color or mandatory field settings are not maintained (if different than the initial settings). To maintain the presentation state, you must create fields (usually hidden) that represent the presentation state of fields. If you add a script to a field's `Calculate` event that changes the presentation based on hidden field values, you are able to preserve the presentation state as you move back and forth between HTML pages (panels).

The following script maintains the `fillColor` of a field based on the value of `hiddenField`. Assume this script is located in a field's `Calculate` event.

```
If (hiddenField.rawValue == 1)
    this.fillColor = "255,0,0"
else
    this.fillColor = "0,255,0"
```

Note: *Static objects are not displayed in a rendered HTML form when nested inside a table cell. For example, a circle and rectangle nested inside a table cell are not displayed within a render HTML form. However these same static objects are properly displayed when located outside of the table.*

Digitally signing HTML forms

You cannot sign an HTML form that contains a digital signature field if the form is rendered as one of the following HTML transformations:

- AHTML
- HTML4
- StaticHTML
- NoScriptXHTML

For information about digitally signing a document, see "Digitally Signing and Certifying Documents" on page 879.

Rendering an accessibility guidelines-compliant XHTML form

You can render a full HTML form that is compliant with accessibility guidelines. That is, the form is rendered within full HTML tags as opposed to the HTML form being rendered within body tags (not a complete HTML page).

Validating form data

It is recommended that you limit your use of validation rules for form fields when rendering the form as an HTML form. Some validation rules may not be supported for HTML forms. For example, when a validation pattern of MM-DD-YYYY is applied to a `Date/Time` field that is located in a form design that is rendered as an HTML form, it does not work properly, even if the date is typed in properly. However, this validation pattern works properly for forms rendered as PDF.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To render an HTML form, perform the following steps:

1. Include project files.
2. Create a Forms Client API object.
3. Set HTML run-time options.
4. Render an HTML form.
5. Write the form data stream to the client web browser.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically import data into a PDF formClient API, you must create a Form Data Integration service client. When creating a service client, you define connection settings that are required to invoke a service.

Set HTML run-time options

You set HTML run-time options when rendering an HTML form. For example, you can add a toolbar to an HTML form to enable users to select file attachments located on the client computer or to retrieve file attachments that are rendered with the HTML form. By default, an HTML toolbar is disabled. To add a toolbar to an HTML form, you must programmatically set run-time options. By default, an HTML toolbar consists of the following buttons:

- `Home`: Provides a link to the application's web root.
- `Upload`: Provides a user interface to select files to attach to the current form.
- `Download`: Provides a user interface to display the attached files.

When a HTML toolbar appears on a HTML form, a user can select a maximum of ten files to submit along with form data. Once the files are submitted, the Forms service can retrieve the files.

When rendering a form as HTML, you can specify a user-agent value. A user-agent value provides browser and system information. This is an optional value, and you can pass an empty string value. The Rendering an HTML form using the Java API quick start shows how to obtain a user agent value and use it to render a form as HTML.

HTTP URLs to where form data is posted may be specified by setting the target URL using the Forms Service Client API or may be specified in the Submit button contained in the XDP form design. If the target URL is specified in the form design, then do not set a value using the Forms Service Client API.

Note: *Rendering an HTML form with a toolbar is optional.*

Note: *If you render an AHTML form, it is recommended that you do not add a toolbar to the form.*

Render an HTML form

To render an HTML form, you must specify a form design created in Designer and saved as an XDP file. You must also select an HTML transformation type. For example, you can specify the HTML transformation type that renders a dynamic HTML for Internet Explorer 5.0 or later.

Rendering an HTML form also requires values, such as URI values that are required to render other form types.

Write the form data stream to the client web browser

When the Forms service renders an HTML form, it returns a form data stream that you must write to the client web browser. When written to the client web browser, the HTML form is visible to the user.

See also

"Render a form as HTML using the Java API" on page 615

"Render a form as HTML using the web service API" on page 616

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Rendering HTML Forms with Custom Toolbars" on page 623

"Creating Web Applications that Renders Forms" on page 670

Render a form as HTML using the Java API

Render an HTML form by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set HTML run-time options

- Create an `HTMLRenderSpec` object by using its constructor.
- To render an HTML form with a toolbar, invoke the `HTMLRenderSpec` object's `setHTMLToolbar` method and pass an `HTMLToolbar` enum value. For example, to display a vertical HTML toolbar, pass `HTMLToolbar.Vertical`.
- To set the locale value for the HTML form, invoke the `HTMLRenderSpec` object's `setLocale` method and pass a string value that specifies the locale value. (This is an optional setting.)
- To render the HTML form within full HTML tags, invoke the `HTMLRenderSpec` object's `setOutputType` method and pass `OutputType.FullHTMLTags`. (This is an optional setting.)

Note: *Forms are not successfully rendered in HTML when the `StandAlone` option is `true` and the `ApplicationWebRoot` references a server other than the J2EE application server hosting AEM Forms (the `ApplicationWebRoot` value is specified using the `URLSpec` object that is passed to the `FormsServiceClient` object's `(Deprecated) renderHTMLForm` method). When the `ApplicationWebRoot` is another server from the one hosting AEM Forms, the value of the web root URI in the administration console needs to be set as the Form's web application URI value. This can be done by logging in to the administration console, clicking Services > Forms, and setting the Web Root URI as http://server-name:port/FormServer. Then, save your settings.*

4 Render an HTML form

Invoke the `FormsServiceClient` object's `(Deprecated) renderHTMLForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `TransformTo` enum value that specifies the HTML preference type. For example, to render an HTML form that is compatible with dynamic HTML for Internet Explorer 5.0 or later, specify `TransformTo.MSDHTML`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- The `HTMLRenderSpec` object that stores HTML run-time options.
- A string value that specifies the `HTTP_USER_AGENT` header value; for example, `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`.
- A `URLSpec` object that stores URI values required to render an HTML form.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `(Deprecated) renderHTMLForm` method returns a `FormsResult` object that contains a form data stream that can be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms as HTML" on page 609

"Quick Start (SOAP mode): Rendering an HTML form using the Java API" on page 167

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Render a form as HTML using the web service API

Render an HTML form by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.

- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Set HTML run-time options

- Create an `HTMLRenderSpec` object by using its constructor.
- To render an HTML form with a toolbar, invoke the `HTMLRenderSpec` object's `setHTMLToolbar` method and pass an `HTMLToolbar` enum value. For example, to display a vertical HTML toolbar, pass `HTMLToolbar.Vertical`.
- To set the locale value for the HTML form, invoke the `HTMLRenderSpec` object's `setLocale` method and pass a string value that specifies the locale value. For more information, see AEM Forms API Reference.
- To render the HTML form within full HTML tags, invoke the `HTMLRenderSpec` object's `setOutputType` method and pass `OutputType.FullHTMLTags`.

Note: *Forms are not successfully rendered in HTML when the `StandAlone` option is `true` and the `ApplicationWebRoot` references a server other than the J2EE application server hosting AEM Forms (the `ApplicationWebRoot` value is specified using the `URLSpec` object that is passed to the `FormsServiceClient` object's `(Deprecated) renderHTMLForm` method). When the `ApplicationWebRoot` is another server from the one hosting AEM Forms, the value of the web root URI in the administration console needs to be set as the Form's web application URI value. This can be done by logging in to the administration console, clicking Services > Forms, and setting the Web Root URI as http://server-name:port/FormServer. Then, save your settings.*

4 Render an HTML form

Invoke the `FormsService` object's `(Deprecated) renderHTMLForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `TransformTo` enum value that specifies the HTML preference type. For example, to render an HTML form that is compatible with dynamic HTML for Internet Explorer 5.0 or later, specify `TransformTo.MSDHTML`.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`. (See "Prepopulating Forms with Flowable Layouts" on page 644.)
- The `HTMLRenderSpec` object that stores HTML run-time options.
- A string value that specifies the `HTTP_USER_AGENT` header value; for example, `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`. You can pass an empty string if you do not want to set this value.
- A `URLSpec` object that stores URI values required to render an HTML form. (See .)
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form. (See .)
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This parameter value stores the rendered form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This parameter will store the output XML data.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. This argument will store the number of pages in the form.

- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. This argument will store the locale value.
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. This argument will store the HTML rendering value that is used.
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `(Deprecated) renderHTMLForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms as HTML" on page 609

Quick Start (Base64): Rendering an HTML form using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Rendering HTML Forms Using Custom CSS Files

Rendering HTML Forms Using Custom CSS Files

The Forms service renders HTML forms in response to an HTTP request from a web browser. When rendering an HTML form, the Forms service can reference a custom CSS file. You can create a custom CSS file to meet your business requirements and reference that CSS file when using the Forms service to render HTML forms.

The Forms service silently parses the custom CSS file. That is, the Forms service does not report errors that may be encountered if the custom CSS file does not comply with CSS standards. In this situation, the Forms service ignores the style and continues with the remaining styles located in the CSS file.

The following list specifies styles that are supported in a custom CSS file:

- **Class level selector-style pairs**: If present in a custom CSS file, selectors used in the HTML form as class styles are used. Unused class styles are ignored.
- **Identifier level selector-style pairs**: All identifier styles are used if they are used in the HTML form.

- **Element level selector-style pairs**: All element styles are used if they are used in the HTML form.
- **Style Priority**: Style priority (like important) is supported and can be used in a custom CSS file.
- **Media Type**: One or more selector-style pairs can be wrapped in @media style to define the media type. The Forms service does not check whether the specified media type is supported. The media type specified in the custom CSS file is merged in the HTML form.

You can retrieve a sample CSS file by using the FormsIVS application. Upload the form, select it in the Test Form Design page, and click GenerateCSS. You are not required to set the HTML transformation type before clicking the button. Next select save. You can edit this CSS file to meet your business requirements.

Note: *Before rendering an HTML form that uses a custom CSS file, it is important that you have a solid understanding of rendering HTML forms. (See "Rendering Forms as HTML" on page 609.)*

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To render an HTML form that uses a CSS file, perform the following tasks:

1. Include project files.
2. Create a Forms Java API object.
3. Reference the CSS file.
4. Render an HTML form.
5. Write the form data stream to the client web browser.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Java API object

Before you can programmatically perform an operation supported by the Forms service, you must create a Forms client object.

Reference the CSS file

To render an HTML form that uses a custom CSS file, ensure that you reference an existing CSS file.

Render an HTML form

To render an HTML form, you must specify a form design created in Designer and saved as an XDP file. You must also select an HTML transformation type. For example, you can specify the HTML transformation type that renders a dynamic HTML for Internet Explorer 5.0 or later.

Rendering an HTML form also requires values, such as URI values needed to render other form types.

Write the form data stream to the client web browser

When the Forms service renders an HTML form, it returns a form data stream that you must write to the client web browser to make the HTML form visible to the user.

See also

"Render an HTML form that uses a CSS file using the Java API" on page 620

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Rendering Forms as HTML" on page 609

"Creating Web Applications that Renders Forms" on page 670

Render an HTML form that uses a CSS file using the Java API

Render an HTML form that uses a custom CSS file by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Java API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference the CSS file

- Create an `HTMLRenderSpec` object by using its constructor.
- To render the HTML form that uses a custom CSS file, invoke the `HTMLRenderSpec` object's `setCustomCSSURI` method and pass a string value that specifies the location and name of the CSS file.

4 Render an HTML form

Invoke the `FormsServiceClient` object's `(Deprecated) (Deprecated) renderHTMLForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `TransformTo` enum value that specifies the HTML preference type. For example, to render an HTML form that is compatible with dynamic HTML for Internet Explorer 5.0 or later, specify `TransformTo.MSDHTML`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- The `HTMLRenderSpec` object that stores HTML run-time options.
- A string value that specifies the `HTTP_USER_AGENT` header value, such as `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`.
- A `URLSpec` object that stores URI values required to render an HTML form.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter, and you can specify `null` if you do not want to attach files to the form.

The `(Deprecated) renderHTMLForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.

- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.h\ttp.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering HTML Forms Using Custom CSS Files" on page 618

"Quick Start (SOAP mode): Rendering an HTML form that uses a CSS file using the Java API" on page 169

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Render an HTML form that uses a CSS file using the web service API

Render an HTML form that uses a custom CSS file by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes in your class path.

2 Create a Forms Java API object

Create a `FormsService` object and set authentication values.

3 Reference the CSS file

- Create an `HTMLRenderSpec` object by using its constructor.
- To render the HTML form that uses a custom CSS file, invoke the `HTMLRenderSpec` object's `setCustomCSSURI` method and pass a string value that specifies the location and name of the CSS file.

4 Render an HTML form

Invoke the `FormsService` object's `(Deprecated) renderHTMLForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `TransformTo` enum value that specifies the HTML preference type. For example, to render an HTML form that is compatible with dynamic HTML for Internet Explorer 5.0 or later, specify `TransformTo.MSDHTML`.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`. (See "Prepopulating Forms with Flowable Layouts" on page 644.)
- The `HTMLRenderSpec` object that stores HTML run-time options.

- A string value that specifies the `HTTP_USER_AGENT` header value, such as `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`. You can pass an empty string if you do not want to set this value.
- A `URLSpec` object that stores URI values required to render an HTML form. (See .)
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter, and you can specify `null` if you do not want to attach files to the form. (See .)
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the `(Deprecated) renderHTMLForm` method. This parameter value stores the rendered form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the `(Deprecated) renderHTMLForm` method. This parameter stores the output XML data.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the `(Deprecated) renderHTMLForm` method. This argument stores the number of pages in the form.
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the `(Deprecated) renderHTMLForm` method. This argument stores the locale value.
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the `(Deprecated) renderHTMLForm` method. This argument stores the HTML rendering value that is used.
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `(Deprecated) renderHTMLForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering HTML Forms Using Custom CSS Files" on page 618

Quick Start (Base64): Rendering an HTML form that uses a CSS file using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Rendering HTML Forms with Custom Toolbars

Rendering HTML Forms with Custom Toolbars

The Forms service lets you customize a toolbar that is rendered with an HTML form. A toolbar can be customized to alter its appearance by overriding default CSS styles and to add dynamic behavior by overriding Java scripts. A toolbar is customized by using an XML file named fscmenu.xml. By default, the Forms service retrieves this file from an internally specified URI location.

Note: *This URI location is located in the adobe-forms-core.jar file, which is located in the adobe-forms-dsc.jar file. The adobe-forms-dsc.jar file is located in C:\Adobe\Adobe_Experience_Manager_forms\ folder (C:\ is the installation directory). You can use a file extraction tool, such as Win RAR, to open the adobe.*

You can copy the fscmenu.xml from this location, modify it to meet your requirements, and then place it in a custom URI location. Next, using the Forms Service API, set run-time options that result in the Forms service using your fscmenu.xml file from the specified location. These actions result in the Forms service rendering an HTML form that has a custom toolbar.

In addition to the fscmenu.xml file, you also need to obtain the following files:

- fscmenu.js
- fscattachments.js
- fscmenu.css
- fscmenu-v.css
- fscmenu-ie.css
- fscdialog.css

fscJS is the Java script that is associated with each node. It is necessary to supply one for the `div#fscmenu` node and optionally for `ul#fscmenuItem` nodes. The JS files implement core toolbar functionality and the default files work.

fscCSS is a style sheet that is associated with a particular node. The styles in the CSS files specify the toolbar appearance. *fscVCSS* is a style sheet for a vertical toolbar, which is displayed on the left of the rendered HTML form. *fscIECSS* is a style sheet used for HTML forms that are rendered in Internet Explorer.

Ensure that all the above files are referenced in the fscmenu.xml file. That is, in the fscmenu.xml file, specify URI locations to point to these files so that the Forms service can locate them. By default, these files are available at URI locations starting with internal keywords `FSWebRoot` or `ApplicationWebRoot`.

To customize the toolbar, replace the keywords by using the external keyword `FSToolBarURI`. This keyword represents the URI that is passed to the Forms service at run time (this approach is shown later in this section).

You can also specify the absolute locations of these JS and CSS files, such as http://www.mycompany.com/scripts/misc/fscmenu.js. In this situation, you do not need to use the `FSToolBarURI` keyword.

Note: *It is not recommended that you mix the ways in which these files are referenced. That is, all URIs should be referenced by using either the* `FSToolBarURI` *keyword or an absolute location.*

You can obtain the JS and CSS files by opening the adobe-forms-<appserver>.ear file. Within this file, open the adobe-forms-res.war. All of these files are located in the WAR file. The adobe-forms-<appserver>.ear file is located in the AEM forms installation folder (C:\ is the installation directory). You can open the adobe-forms-<appserver>.ear using a file extraction tool such as WinRAR.

The following XML syntax shows an example fscmenu.xml file.

```
<div id="fscmenu" fscJS="FSToolBarURI/scripts/fscmenu.js" fscCSS="FSToolBarURI/fscmenu.css"
fscVCSS="FSToolBarURI/fscmenu-v.css" fscIECSS="FSToolBarURI/fscmenu-ie.css">
        <ul class="fscmenuItem" id="Home">
            <li>
                <a href="#" fscTarget="_top" tabindex="1">Home</a>
            </li>
        </ul>
        <ul class="fscmenuItem" id="Upload" fscJS="FSToolBarURI/scripts/fscattachments.js"
fscCSS="FSToolBarURI/fscdialog.css">
            <li>
                <a tabindex="2">Upload Attachments</a>
                <ul class="fscmenuPopup" id="fscUploadAttachments">
                    <li>
                        <a href="javascript:doUploadDialog();" tabindex="3">Add ...</a>
                    </li>
                    <li>
                        <a href="javascript:doDeleteDialog();" tabindex="4">Delete ...</a>
                    </li>
                </ul>
            </li>
        </ul>
        <ul class="fscmenuItem" id="Download">
            <li>
                <a tabindex="100">Download Attachments</a>
                <ul class="fscmenuPopup">
                    <li>
                        <a tabindex="101">None available</a>
                    </li>
                </ul>
            </li>
        </ul>
    </div>
```

Note: *The bold text represents the URIs to the CSS and JS files that must be referenced.*

The following items describe how you can customize a toolbar:

- Change the values of `fscJS`, `fscCSS`, `fscVCSS`, `fscIECSS` attributes (in the fscmenu.xml file) to reflect the custom locations of the referenced files by using one of the methods that are described in this section (for example, `fscJS="FSToolBarURI/scripts/fscmenu.js"`).
- All the CSS and JS files must be specified. If none of the files are modified, provide the default one at the custom location. You can obtain the default files by opening various files as described in this section.
- Providing an absolute reference (for example, http://www.example.com/scripts/custom-vertical-fscmenu.css) for any file is allowed.
- The JS and CSS files that the `div#fscmenu` node requires are essential for toolbar functionality. Individual `ul#fscmenuItem` nodes may or may not have supporting JS or CSS files.

Changing the local value

As part of customizing a toolbar, you can change the locale value of the toolbar. That is, you can display it in another language. The following illustration shows a custom toolbar that is displayed in French.

Note: *It is not possible to create a custom toolbar in more than one language. Toolbars cannot use different XML files based on the locale settings.*

To change the locale value of a toolbar, ensure that the fscmenu.xml file contains the language you want to display. The following XML syntax shows the fscmenu.xml file that is used to display a French toolbar.

```
<div id="fscmenu" fscJS="FSToolBarURI/scripts/fscmenu.js" fscCSS="FSToolBarURI/fscmenu.css"
fscVCSS="FSToolBarURI/fscmenu-v.css" fscIECSS="FSToolBarURI/fscmenu-ie.css">
        <ul class="fscmenuItem" id="Home">
            <li>
                <a href="#" fscTarget="_top" tabindex="1">Accueil</a>
            </li>
        </ul>
        <ul class="fscmenuItem" id="Upload" fscJS="FSToolBarURI/scripts/fscattachments.js"
fscCSS="FSToolBarURI/fscdialog.css">
            <li>
                <a tabindex="2">Télécharger les pièces jointes</a>
                <ul class="fscmenuPopup" id="fscUploadAttachments">
                    <li>
                        <a href="javascript:doUploadDialog();" tabindex="3">Ajouter...</a>
                    </li>
                    <li>
                        <a href="javascript:doDeleteDialog();" tabindex="4">Supprimer...</a>
                    </li>
                </ul>
            </li>
        </ul>
        <ul class="fscmenuItem" id="Download">
            <li>
                <a tabindex="100">Télécharger les pièces jointes</a>
                <ul class="fscmenuPopup">
                    <li>
                        <a tabindex="101">Aucune disponible</a>
                    </li>
                </ul>
            </li>
        </ul>
    </div>
```

Note: *The Quick Starts that are associated with this section use this XML file to display a French custom toolbar, as shown in the previous illustration.*

Also, specify a valid locale value by invoking the `HTMLRenderSpec` object's `setLocale` method and passing a string value that specifies the locale value. For example, pass `fr_FR` to specify French. The Forms service is bundled with localized toolbars.

Note: *Before you render an HTML form that uses a custom toolbar, you must know how HTML forms are rendered. (See "Rendering Forms as HTML" on page 609.)*

For more information about the Forms service, seeServices Reference for AEM Forms.

Summary of steps

To render an HTML form that contains a custom toolbar, perform these tasks:

1 Include project files.

2 Create a Forms Java API object.

3 Reference a custom fscmenu XML file.

4 Render an HTML form.

5 Write the form data stream to the client web browser.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create a Forms Java API object

Before you can programmatically perform an operation that the Forms service supports, you must create a Forms client object.

Reference a custom fscmenu XML file

To render an HTML form that contains a custom toolbar, reference a fscmenu XML file that describes the toolbar. (This section provides two examples of a fscmenu XML file.) Also, ensure that the fscmenu.xml file specifies the locations of all referenced files correctly. As mentioned earlier in this section, ensure that all files are referenced by either the `FSToolBarURI` keyword or their absolute locations.

Render an HTML form

To render an HTML form, specify a form design that was created in Designer and saved as an XDP file. Also select an HTML transformation type. For example, you can specify the HTML transformation type that renders a dynamic HTML for Internet Explorer 5.0 or later.

Rendering an HTML form also requires values, such as URI values for rendering other form types.

Write the form data stream to the client web browser

When the Forms service renders an HTML form, it returns a form data stream that you must write to the client web browser to make the HTML form visible to users.

See also

"Render an HTML Form with a custom toolbar using the Java API" on page 626

"Rendering an HTML Form with a custom toolbar using the web service API" on page 628

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Rendering Forms as HTML" on page 609

"Creating Web Applications that Renders Forms" on page 670

Render an HTML Form with a custom toolbar using the Java API

Render an HTML Form that contains a custom toolbar by using the Forms Service API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Java API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference a custom fscmenu XML file

- Create an `HTMLRenderSpec` object by using its constructor.
- To render an HTML form with a toolbar, invoke the `HTMLRenderSpec` object's `setHTMLToolbar` method and pass an `HTMLToolbar` enum value. For example, to display a vertical HTML toolbar, pass `HTMLToolbar.Vertical`.
- Specify the location of the fscmenu XML file by invoking the `HTMLRenderSpec` object's `setToolbarURI` method and passing a string value that specifies the URI location of the XML file.
- If applicable, set the locale value by invoking the `HTMLRenderSpec` object's `setLocale` method and passing a string value that specifies the locale value. The default value is English.

Note: *The Quick Starts that are associated with this section sets this value to* `fr_FR`.

4 Render an HTML form

Invoke the `FormsServiceClient` object's `renderHTMLForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `TransformTo` enum value that specifies the HTML preference type. For example, to render an HTML form that is compatible with dynamic HTML for Internet Explorer 5.0 or later, specify `TransformTo.MSDHTML`.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- The `HTMLRenderSpec` object that stores HTML run-time options.
- A string value that specifies the `HTTP_USER_AGENT` header value, such as `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`.
- A `URLSpec` object that stores URI values that are required to render an HTML form.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter, and you can specify `null` if you do not want to attach files to the form.

The `renderHTMLForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object that is used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.

- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering HTML Forms with Custom Toolbars" on page 623

"Quick Start (SOAP mode): Rendering an HTML Form with a custom toolbar using the Java API" on page 172

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Rendering an HTML Form with a custom toolbar using the web service API

Render an HTML form that contains a custom toolbar by using the Forms Service API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes in your class path.

2 Create a Forms Java API object

Create a `FormsService` object and set authentication values.

3 Reference a custom fscmenu XML file

- Create an `HTMLRenderSpec` object by using its constructor.
- To render an HTML form with a toolbar, invoke the `HTMLRenderSpec` object's `setHTMLToolbar` method and pass an `HTMLToolbar` enum value. For example, to display a vertical HTML toolbar, pass `HTMLToolbar.Vertical`.
- Specify the location of the fscmenu XML file by invoking the `HTMLRenderSpec` object's `setToolbarURI` method and passing a string value that specifies the URI location of the XML file.
- If applicable, set the locale value by invoking the `HTMLRenderSpec` object's `setLocale` method and passing a string value that specifies the locale value. The default value is English.

Note: *The Quick Starts that are associated with this section sets this value to* `fr_FR`.

4 Render an HTML form

Invoke the `FormsService` object's `renderHTMLForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension. If you reference a form design that is part of a Forms application, ensure that you specify the complete path, such as `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.
- A `TransformTo` enum value that specifies the HTML preference type. For example, to render an HTML form that is compatible with dynamic HTML for Internet Explorer 5.0 or later, specify `TransformTo.MSDHTML`.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`.
- The `HTMLRenderSpec` object that stores HTML run-time options.
- A string value that specifies the `HTTP_USER_AGENT` header value, such as `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`. You can pass an empty string if you do not want to set this value.

- A `URLSpec` object that stores URI values that are required to render an HTML form.
- A `java.util.HashMap` object that stores file attachments. This parameter is optional, and you can specify `null` if you do not intend to attach files to the form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the `renderHTMLForm` method. This parameter value stores the rendered form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the `renderHTMLForm` method. This parameter stores the output XML data.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the `renderHTMLForm` method. This argument stores the number of pages in the form.
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the `renderHTMLForm` method. This argument stores the locale value.
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the `renderHTMLForm` method. This argument stores the HTML rendering value that is used.
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderHTMLForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object that is used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering HTML Forms with Custom Toolbars" on page 623

Quick Start (Base64): Rendering an HTML Form with a custom toolbar using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

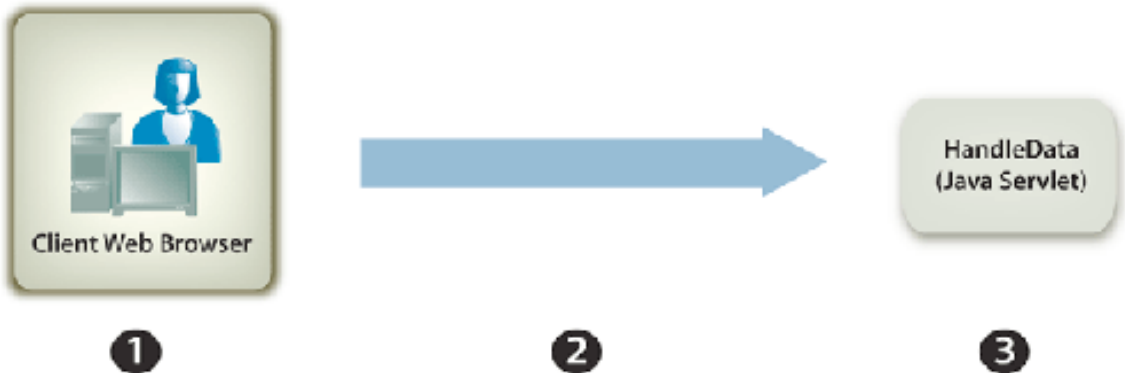
Handling Submitted Forms

Handling Submitted Forms

Web-based applications that enable a user to fill in interactive forms require the data to be submitted back to the server. Using the Forms service, you can retrieve the data that the user entered into an interactive form. After you retrieve the data, you can process the data to meet your business requirements. For example, you can store the data in a database, send the data to another application, send the data to another service, merge the data in a form design, display the data in a web browser, and so on.

Form data is submitted to the Forms service as either XML or PDF data, which is an option that is set in Designer. A form that is submitted as XML enables you to extract individual field data values. That is, you can extract the value of each form field that the user entered into the form. A form that is submitted as PDF data is binary data, not XML data. You can save the form as a PDF file, or send the form to another service. If you want to extract data from a form submitted as XML and then use the form data to create a PDF document, invoke another AEM Forms operation. (See "Creating PDF Documents with Submitted XML Data" on page 639)

The following diagram shows data being submitted to a Java Servlet named `HandleData` from an interactive form displayed in a web browser.



The following table explains the steps in the diagram.

Step	Description
1	A user fills in an interactive form and clicks the form's Submit button.
2	Data is submitted to the `HandleData` Java Servlet as XML data.
3	The `HandleData` Java Servlet contains application logic to retrieve the data.

Handling submitted XML data

When form data is submitted as XML, you can retrieve XML data that represents the submitted data. All form fields appear as nodes in an XML schema. The node values correspond to the values that the user filled in. Consider a loan form where each field in the form appears as a node within the XML data. The value of each node corresponds to the value that a user fills in. Assume a user fills the loan form with data shown in the following form.

Fin@nce corp.

MORTGAGE APPLICATION

Applicants: Complete this form for a mortgage application. One of our representatives will contact you within two business days

Step 1: Mortgage Information

Property Sale Price:	Down Payment:	Mortgage Amount:
$300,000.00	$5,000.00	$295,000.00
Term (Years): 25 Interest Rate: 5.00	Closing Date:	Monthly Mortgage Payment:

Step 2: Applicant Information

Last Name:	First Name:	Middle Initial(s):
Johnson	Jerry	D
Social Security Number: 555-66-5555	Phone Number: (555) 555-0000	Date of Birth: 2/5/1973
Mailing Address: JJohnson@NoMailServer.com		
City: New York	State: New York	Zip Code: 00501

The following illustration shows corresponding XML data that is retrieved by using the Forms service Client API.

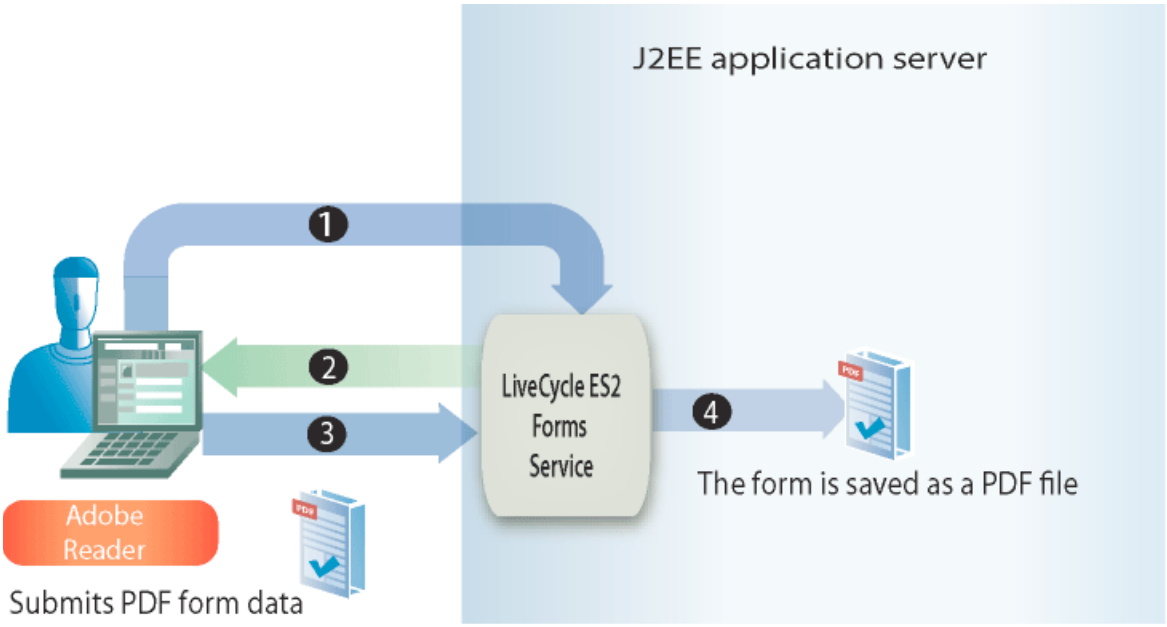
```
<?xml version="1.0" encoding="UTF-8" ?>
- <data>
  - <Layer>
      <btnSubmit />
      <approval>0</approval>
      <closeDate>2007-01-26</closeDate>
      <lastName>Johnson</lastName>
      <firstName>Jerry</firstName>
      <mailingAddress>JJohnson@NoMailServer.com</mailingAddress>
      <city>New York</city>
      <zipCode>00501</zipCode>
      <state>NY</state>
      <dateBirth>26/8/1973</dateBirth>
      <middleInitials>D</middleInitials>
      <socialSecurityNumber>5656565656</socialSecurityNumber>
      <phoneNumber>5555550000.00000000</phoneNumber>
    </Layer>
  - <Mortgage>
      <mortgageAmount>295000.00000000</mortgageAmount>
      <monthlyMortgagePayment>1724.54000000</monthlyMortgagePayment>
      <purchasePrice>300000.00000000</purchasePrice>
      <downPayment>5000.00000000</downPayment>
      <term>25</term>
      <interestRate>5.00</interestRate>
    </Mortgage>
</data>
```

The fields in the loan form. These values can be retrieved using Java XML classes.

Note: *The form design must be configured correctly in Designer for data to be submitted as XML data. To properly configure the form design to submit XML data, ensure that the Submit button that is located on the form design is set to submit XML data. For information about setting the Submit button to submit XML data, see AEM Forms Designer.*

Handling submitted PDF data

Consider a web application that invokes the Forms service. After the Forms service renders an interactive PDF form to a client web browser, the user fills in the form and submits it back as PDF data. When the Forms service receives the PDF data, it can send the PDF data to another service or save it as a PDF file. The following diagram shows the application's logic flow.



The following table describes the steps in this diagram.

Step	Description
1	A web page contains a link that accesses a Java Servlet that invokes the Forms service.
2	The Forms service renders an interactive PDF form to the client web browser.
3	The user fills in an interactive form and clicks a submit button. The form is submitted back to the Forms service as PDF data. This option is set in Designer.
4	The Forms service saves the PDF data as a PDF file.

Handling submitted Guide (deprecated) data

The Flash Player security specifies that a Flash application can only submit data to the URL location (domain) from which it was served. When using the Forms service to render a guide, this is not the case. The URL location from where a Guide (deprecated) is served and the URL location to where the form data is posted is different (that is typical Forms service functionality). For Flash Player to permit data to be posted to a different URL, a crossdomain.xml file must be available. Otherwise, form data cannot be posted to a different URL.

Handling submitted URL UTF-16 data

If form data is submitted as URL UTF-16 data, the client computer requires Adobe Reader or Acrobat 8.1 or later. Also if the form design contains a submit button that has URL-encoded Data (HTTP Post) and the data encoding option is UTF-16, the form design must be modified in a text editor such as Notepad. You can set the encoding option to either `UTF-16LE` or `UTF-16BE` for the submit button. Designer does not provide this functionality.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To handle submitted forms, perform the following tasks:

1. Include project files.
2. Create a Forms Client API object.
3. Retrieve form data.
4. Determine if the form submission contains file attachments.
5. Process the submitted data.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms service client. If you are using the Java API, create a `FormsServiceClient` object. If you are using the Forms web service API, create a `FormsService` object.

Retrieve form data

To retrieve submitted form data, you invoke the `FormsServiceClient` object's `processFormSubmission` method. When invoking this method, you have to specify the submitted form's content type. When data is submitted from a client web browser to the Forms service, it can be submitted as either XML or PDF data. To retrieve the data that is entered into form fields, the data can be submitted as XML data.

You can also retrieve form fields from a form submitted as PDF data by setting the following run-time options:

- Pass the following value to the `processFormSubmission` method as the content type parameter: `CONTENT_TYPE=application/pdf`.
- Set the `RenderOptionsSpec` object's `PDFToXDP` value to `true`
- Set the `RenderOptionsSpec` object's `ExportDataFormat` value to `XMLData`

You specify the content type of the submitted form when you invoke the `processFormSubmission` method. The following list specifies applicable content type values:

- **text/xml**: Represents the content type to use when a PDF form submits form data as XML.
- **application/x-www-form-urlencoded**: Represents the content type to use when an HTML form submits data as XML.
- **application/pdf**: Represents the content type to use when a PDF form submits data as PDF.

Note: *You will notice that there are three corresponding quick starts associated with the Handling Submitted Forms section. The Handling PDF forms submitted as PDF using the Java API quick start demonstrates how to handle submitted PDF data. The content type specified in this quick start is* `application/pdf`*. The Handling PDF forms submitted as XML using the Java API quick start demonstrates how to handle submitted XML data that is submitted from a PDF form. The content type specified in this quick start is* `text/xml`*. Likewise, the Handling HTML forms submitted as XML using the Java API quick start demonstrates how to handle submitted XML data that is submitted from an HTML form. The content type specified in this quick start is application/x-www-form-urlencoded.*

You retrieve form data that was posted to the Forms service and determine its processing state. That is, when data is submitted to the Forms service, it does not necessarily mean that the Forms service is finished processing the data and the data is ready to be processed. For example, data can be submitted to the Forms service so that a calculation can be performed. When the calculation is complete, the form is rendered back to the user with the calculation results displayed. Before you process submitted data, it is recommended that you determine whether the Forms service has finished processing the data.

The Forms service returns the following values to indicate whether it has finished processing the data:

- **0 (Submit):** Submitted data is ready to be processed.
- **1 (Calculate):** The Forms service performed a calculation operation on the data and the results must be rendered back to the user.
- **2 (Validate):** The Forms service validated form data and the results must be rendered back to the user.
- **3 (Next):** The current page has changed with results that must be written to the client application.
- **4 (Previous**): The current page has changed with results that must be written to the client application.

Note: *Calculations and validations must be rendered back to the user. (See "Calculating Form Data" on page 656.)*

Determine if the form submission contains file attachments

Forms submitted to the Forms service can contain file attachments. For example, using Acrobat's built-in attachment pane, a user can select file attachments to submit along with the form. As well, a user can also select file attachments using an HTML toolbar that is rendered with an HTML file.

After you determine if a form contains file attachments, you can process the data. For example, you can save the file attachment to the local file system.

Note: *The form must be submitted as PDF data in order to retrieve file attachments. If the form is submitted as XML data, file attachments are not submitted.*

Process the submitted data

Depending on the content type of the submitted data, you can extract individual form field values from the submitted XML data or save the submitted PDF data as a PDF file (or send it to another service). To extract individual form fields, convert submitted XML data to an XML data source and then retrieve XML data source values by using `org.w3c.dom` classes.

See also

"Handle submitted forms using the Java API" on page 636

"Handle submitted PDF data using the web service API" on page 637

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Passing Documents to the Forms Service" on page 591

"Creating Web Applications that Renders Forms" on page 670

Handle submitted forms using the Java API

Handle a submitted form by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve form data

- To retrieve form data that was posted to a Java Servlet, create a `com.adobe.idp.Document` object by using its constructor and invoking the `javax.servlet.http.HttpServletResponse` object's `getInputStream` method from within the constructor.
- Create a `RenderOptionsSpec` object by using its constructor. Set the locale value by invoking the `RenderOptionsSpec` object's `setLocale` method and passing a string value that specifies the locale value.

Note: *You can instruct the Forms service to create XDP or XML data from submitted PDF content by invoking the `RenderOptionsSpec` object's `setPDF2XDP` method and passing `true` and also calling `setXMLData` and passing `true`. You can then invoke the `FormsResult` object's `getOutputXML` method to retrieve the XML data that corresponds to the XDP/XML data. (The `FormsResult` object is returned by the `processFormSubmission` method, which is explained in the next sub-step.)*

- Invoke the `FormsServiceClient` object's `processFormSubmission` method and pass the following values:
  - The `com.adobe.idp.Document` object that contains the form data.
  - A string value that specifies environment variables including all relevant HTTP headers. Specify the content type to handle. To handle XML data, specify the following string value for this parameter: `CONTENT_TYPE=text/xml`. To handle PDF data, specify the following string value for this parameter: `CONTENT_TYPE=application/pdf`.
  - A string value that specifies the `HTTP_USER_AGENT` header value, for example, `.Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`. This parameter value is optional.
  - A `RenderOptionsSpec` object that stores run-time options.

  The `processFormSubmission` method returns a `FormsResult` object containing the results of the form submission.
- Determine whether the Forms service is finished processing the form data by invoking the `FormsResult` object's `getAction` method. If this method returns the value `0`, the data is ready to be processed.

4 Determine if the form submission contains file attachments

- Invoke the `FormsResult` object's `getAttachments` method. This method returns a `java.util.List` object that contains files that were submitted with the form.
- Iterate through the `java.util.List` object to determine if there are file attachments. If there are file attachments, each element is a `com.adobe.idp.Document` instance. You can save the file attachments by invoking the `com.adobe.idp.Document` object's `copyToFile` method and passing a `java.io.File` object.

Note: *This step is only applicable if the form is submitted as PDF.*

5 Process the submitted data

- If the data content type is `application/vnd.adobe.xdp+xml` or `text/xml`, create application logic to retrieve XML data values.
  - Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object's `getOutputContent` method.
  - Create a `java.io.InputStream` object by invoking the `java.io.DataInputStream` constructor and passing the `com.adobe.idp.Document` object.
  - Create an `org.w3c.dom.DocumentBuilderFactory` object by calling the static `org.w3c.dom.DocumentBuilderFactory` object's `newInstance` method.
  - Create an `org.w3c.dom.DocumentBuilder` object by invoking the `org.w3c.dom.DocumentBuilderFactory` object's `newDocumentBuilder` method.
  - Create an `org.w3c.dom.Document` object by invoking the `org.w3c.dom.DocumentBuilder` object's `parse` method and passing the `java.io.InputStream` object.
  - Retrieve the value of each node within the XML document. One way to accomplish this task is to create a custom method that accepts two parameters: the `org.w3c.dom.Document` object and the name of the node whose value you want to retrieve. This method returns a string value representing the value of the node. In the code example that follows this process, this custom method is called `getNodeText`. The body of this method is shown.
- If the data content type is `application/pdf`, create application logic to save the submitted PDF data as a PDF file.
  - Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object's `getOutputContent` method.
  - Create a `java.io.File` object by using its public constructor. Be sure to specify PDF as the file name extension.
  - Populate the PDF file by invoking the `com.adobe.idp.Document` object's `copyToFile` method and passing the `java.io.File` object.

See also

"Handling Submitted Forms" on page 630

"Quick Start (SOAP mode): Handling PDF forms submitted as XML using the Java API" on page 175

"Quick Start (SOAP mode): Handling HTML forms submitted as XML using the Java API" on page 182

"Quick Start (SOAP mode): Handling PDF forms submitted as PDF using the Java API" on page 179

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Handle submitted PDF data using the web service API

Handle a submitted form by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Retrieve form data

- To retrieve form data that was posted to a Java Servlet, create a `BLOB` object by using its constructor.
- Create a `java.io.InputStream` object by invoking the `javax.servlet.http.HttpServletResponse` object's `getInputStream` method.
- Create a `java.io.ByteArrayOutputStream` object by using its constructor and passing the length of the `java.io.InputStream` object.
- Copy the contents of the `java.io.InputStream` object into the `java.io.ByteArrayOutputStream` object.
- Create a byte array by invoking the `java.io.ByteArrayOutputStream` object's `toByteArray` method.
- Populate the `BLOB` object by invoking its `setBinaryData` method and passing the byte array as an argument.
- Create a `RenderOptionsSpec` object by using its constructor. Set the locale value by invoking the `RenderOptionsSpec` object's `setLocale` method and passing a string value that specifies the locale value.
- Invoke the `FormsService` object's `processFormSubmission` method and pass the following values:
  - The `BLOB` object that contains the form data.
  - A string value that specifies environment variables including all relevant HTTP headers. Specify the content type to handle. To handle XML data, specify the following string value for this parameter: `CONTENT_TYPE=text/xml`. To handle PDF data, specify the following string value for this parameter: `CONTENT_TYPE=application/pdf`.
  - A string value that specifies the `HTTP_USER_AGENT` header value; for example, `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`.
  - A `RenderOptionsSpec` object that stores run-time options.
  - An empty `BLOBHolder` object that is populated by the method.
  - An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method.
  - An empty `BLOBHolder` object that is populated by the method.
  - An empty `BLOBHolder` object that is populated by the method.
  - An empty `javax.xml.rpc.holders.ShortHolder` object that is populated by the method.
  - An empty `MyArrayOf_xsd_anyTypeHolder` object that is populated by the method. This parameter is used to store file attachments that are submitted along with the form.
  - An empty `FormsResultHolder` object that is populated by the method with the form that is submitted.

  The `processFormSubmission` method populates the `FormsResultHolder` parameter with the results of the form submission.
- Determine whether the Forms service is finished processing the form data by invoking the `FormsResult` object's `getAction` method. If this method returns the value `0`, the form data is ready to be processed. You can get a `FormsResult` object by getting the value of the `FormsResultHolder` object's `value` data member.

4 Determine if the form submission contains file attachments

Get the value of the `MyArrayOf_xsd_anyTypeHolder` object's `value` data member (the `MyArrayOf_xsd_anyTypeHolder` object was passed to the `processFormSubmission` method). This data member returns an array of `Objects`. Each element within the `Object` array is an `Object` that corresponds to the files that were submitted along with the form. You can get each element within the array and cast it to a `BLOB` object.

5 Process the submitted data

- If the data content type is `application/vnd.adobe.xdp+xml` or `text/xml`, create application logic to retrieve XML data values.
  - Create a `BLOB` object by invoking the `FormsResult` object's `getOutputContent` method.
  - Create a byte array by invoking the `BLOB` object's `getBinaryData` method.
  - Create a `java.io.InputStream` object by invoking the `java.io.ByteArrayInputStream` constructor and passing the byte array.
  - Create an `org.w3c.dom.DocumentBuilderFactory` object by calling the static `org.w3c.dom.DocumentBuilderFactory` object's `newInstance` method.
  - Create an `org.w3c.dom.DocumentBuilder` object by invoking the `org.w3c.dom.DocumentBuilderFactory` object's `newDocumentBuilder` method.
  - Create an `org.w3c.dom.Document` object by invoking the `org.w3c.dom.DocumentBuilder` object's `parse` method and passing the `java.io.InputStream` object.
  - Retrieve the value of each node within the XML document. One way to accomplish this task is to create a custom method that accepts two parameters: the `org.w3c.dom.Document` object and the name of the node whose value you want to retrieve. This method returns a string value representing the value of the node. In the code example that follows this process, this custom method is called `getNodeText`. The body of this method is shown.
- If the data content type is `application/pdf`, create application logic to save the submitted PDF data as a PDF file.
  - Create a `BLOB` object by invoking the `FormsResult` object's `getOutputContent` method.
  - Create a byte array by invoking the `BLOB` object's `getBinaryData` method.
  - Create a `java.io.File` object by using its public constructor. Be sure to specify PDF as the file name extension.
  - Create a `java.io.FileOutputStream` object by using its constructor and passing the `java.io.File` object.
  - Populate the PDF file by invoking the `java.io.FileOutputStream` object's `write` method and passing the byte array.

See also

"Handling Submitted Forms" on page 630

Quick Start (Base64): Handling submitted forms using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Creating PDF Documents with Submitted XML Data

Creating PDF Documents with Submitted XML Data

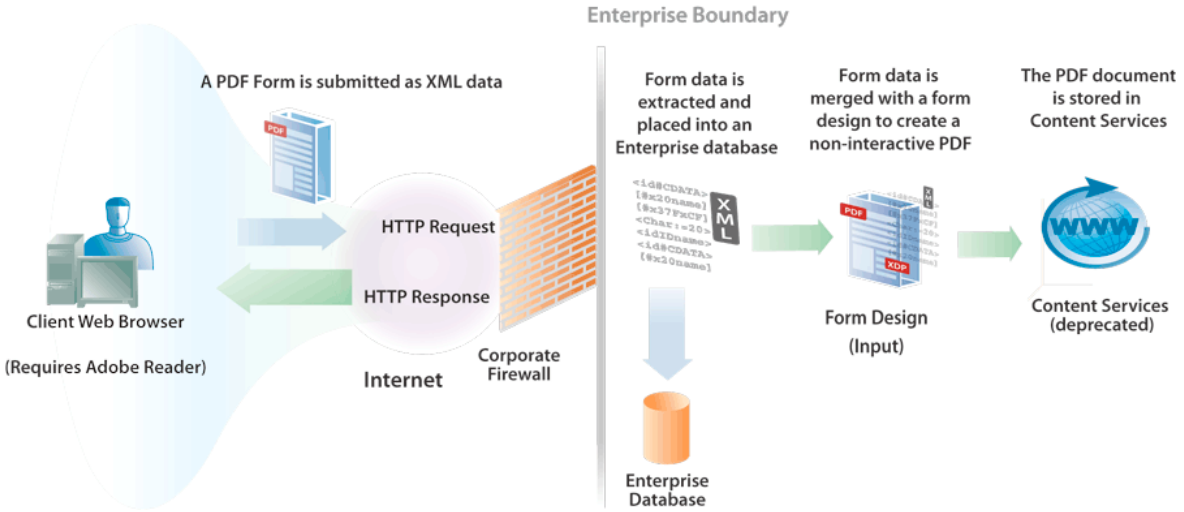
Web-based applications that enable users to fill interactive forms require the data to be submitted back to the server. Using the Forms service, you can retrieve the form data that the user entered into an interactive form. Then you can pass the form data to another AEM Forms service operation and create a PDF document using the data.

Note: *Before you read this content, it is recommended that you have a solid understanding of handling submitted forms. Concepts such as the relationship between a form design and submitted XML data are covered in "Handling Submitted Forms" on page 630.*

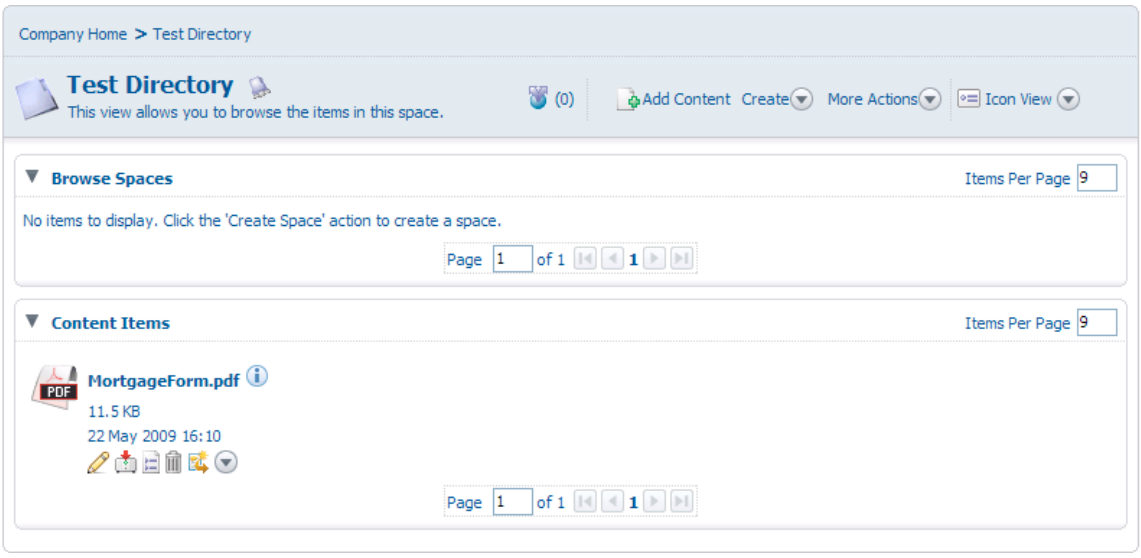
Consider the following workflow that involves three AEM Forms services:

- A user submits XML data to the Forms service from a web-based application.
- The Forms service is used to process the submitted form and extract form fields. Form data can be processed. For example, the data can be submitted to an enterprise database.
- Form data is sent to the Output service to create a non-interactive PDF document.
- The non-interactive PDF document is stored in Content Services (deprecated).

The following diagram provides a visual representation of this workflow.



After the user submits the form from the client web browser, the non-interactive PDF document is stored in Content Services (deprecated). The following illustration shows a PDF document stored in Content Services (deprecated).



Summary of steps

To create a non-interactive PDF document with submitted XML data and store in the PDF document in Content Services (deprecated), perform the following tasks:

1 Include project files.

2 Create Forms, Output, and Document Management objects.

3 Retrieve form data by using the Forms service.

4 Create a non-interactive PDF document by using the Output service.

5 Store the PDF form in Content Services (deprecated) by using the Document Management service.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create Forms, Output, and Document Management objects

Before you can programmatically perform a Forms service API operation, create a Forms Client API object. Likewise, because this workflow invokes the Output and Document Management services, create both an Output Client API object and a Document Management Client API object.

Retrieve form data using the Forms service

Retrieve form data that was submitted to the Forms service. You can process submitted data to meet your business requirements. For example, you can store form data in an enterprise database. However, to create a non-interactive PDF document, the form data is passed to the Output service.

Create a non-interactive PDF document using the Output service.

Use the Output service to create a non-interactive PDF document that is based on a form design and XML form data. In the workflow, the form data is retrieved from the Forms service.

Store the PDF form in Content Services (deprecated) using the Document Management service

Use the Document Management service API to store a PDF document in Content Services (deprecated).

See also

"Create a PDF Document with submitted XML data using the Java API" on page 642

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Creating PDF Documents" on page 681

Adding Content to Content Services (deprecated)

Create a PDF Document with submitted XML data using the Java API

Create a PDF document with submitted XML data by using the Forms, Output, and Document Management API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, adobe-output-client.jar, and adobe-contentservices-client.jar in your Java project's class path.

2 Create Forms, Output, and Document Management objects

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
- Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.
- Create a `DocumentManagementServiceClientImpl` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve form data using the Forms service

- Invoke the `FormsServiceClient` object's `processFormSubmission` method and pass the following values:
  - The `com.adobe.idp.Document` object that contains the form data.
  - A string value that specifies environment variables, including all relevant HTTP headers. Specify the content type to handle by specifying one or more values for the `CONTENT_TYPE` environment variable. For example, to handle XML data, specify the following string value for this parameter: `CONTENT_TYPE=text/xml`.
  - A string value that specifies the `HTTP_USER_AGENT` header value, such as `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)`.
  - A `RenderOptionsSpec` object that stores run-time options.

  The `processFormSubmission` method returns a `FormsResult` object containing the results of the form submission.
- Determine whether the Forms service is finished processing the form data by invoking the `FormsResult` object's `getAction` method. If this method returns the value `0`, the data is ready to be processed.
- Retrieve form data by creating a `com.adobe.idp.Document` object by invoking the `FormsResult` object's `getOutputContent` method. (This object contains form data that can be sent to the Output service.)
- Create a `java.io.InputStream` object by invoking the `java.io.DataInputStream` constructor and passing the `com.adobe.idp.Document` object.
- Create an `org.w3c.dom.DocumentBuilderFactory` object by calling the static `org.w3c.dom.DocumentBuilderFactory` object's `newInstance` method.
- Create an `org.w3c.dom.DocumentBuilder` object by invoking the `org.w3c.dom.DocumentBuilderFactory` object's `newDocumentBuilder` method.
- Create an `org.w3c.dom.Document` object by invoking the `org.w3c.dom.DocumentBuilder` object's `parse` method and passing the `java.io.InputStream` object.
- Retrieve the value of each node within the XML document. One way to accomplish this task is to create a custom method that accepts two parameters: the `org.w3c.dom.Document` object and the name of the node whose value you want to retrieve. This method returns a string value representing the value of the node. In the code example that follows this process, this custom method is called `getNodeText`. The body of this method is shown.

4 Create a non-interactive PDF document using the Output service.

Create a PDF document by invoking the `OutputClient` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enum value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the form design. Ensure that the form design is compatible with the form data retrieved from the Forms service.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contains the XML data source that contains data to merge with the form design. Ensure that this object was returned by the `FormsResult` object's `getOutputContent` method.
- The `generatePDFOutput` method returns an `OutputResult` object that contains the results of the operation.
- Retrieve the non-interactive PDF document by invoking the `OutputResult` object's `getGeneratedDoc` method. This method returns a `com.adobe.idp.Document` instance that represents the non-interactive PDF document.

5 Store the PDF form in Content Services (deprecated)using the Document Management service

Add the content by invoking the `DocumentManagementServiceClientImpl` object's `storeContent` method and passing the following values:

- A string value that specifies the store where the content is added. The default store is `SpacesStore`. This value is a mandatory parameter.
- A string value that specifies the fully qualified path of the space where the content is added (for example, `/Company Home/Test Directory`). This value is a mandatory parameter.
- The node name that represents the new content (for example, `MortgageForm.pdf`). This value is a mandatory parameter.
- A string value that specifies the node type. To add new content, such as a PDF file, specify `{http://www.alfresco.org/model/content/1.0}content`. This value is a mandatory parameter.
- A `com.adobe.idp.Document` object that represents the content. This value is a mandatory parameter.
- A string value that specifies the encoding value (for example, `UTF-8`). This value is a mandatory parameter.
- An `UpdateVersionType` enumeration value that specifies how to handle version information (for example, `UpdateVersionType.INCREMENT_MAJOR_VERSION` to increment the content version.) This value is a mandatory parameter.
- A `java.util.List` instance that specifies aspects related to the content. This value is an optional parameter and you can specify `null`.
- A `java.util.Map` object that stores content attributes.

The `storeContent` method returns a `CRCResult` object that describes the content. Using a `CRCResult` object, you can, for example, obtain the content's unique identifier value. To perform this task, invoke the `CRCResult` object's `getNodeUuid` method.

See also

"Creating PDF Documents with Submitted XML Data" on page 639

"Quick Start (SOAP mode): Creating PDF Documents with submitted XML data using the Java API" on page 185

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Prepopulating Forms with Flowable Layouts

Prepopulating Forms with Flowable Layouts

Prepopulating forms displays data to users within a rendered form. For example, assume a user logs in to a website with a user name and password. If authentication is successful, the client application queries a database for user information. The data is merged into the form and then the form is rendered to the user. As a result, the user is able to view personalized data within the form.

Prepopulating a form has the following advantages:

- Enables the user to view custom data in a form.
- Reduces the amount of typing the user does to fill in a form.
- Ensures data integrity by having control over where data is placed.

The following two XML data sources can prepopulate a form:

- An XDP data source, which is XML that conforms to XFA syntax (or XFDF data to prepopulate a form created using Acrobat).
- An arbitrary XML data source that contains name/value pairs matching the form's field names (the examples in this section use an arbitrary XML data source).

An XML element must exist for every form field that you want to prepopulate. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed, as long as all XML elements are specified.

When you prepopulate a form that already contains data, you must specify the data that is already displayed within the XML data source. Assume that a form containing 10 fields has data in four fields. Next, assume that you want to prepopulate the remaining six fields. In this situation, you must specify 10 XML elements in the XML data source that is used to prepopulate the form. If you specify only six elements, the original four fields are empty.

For example, you can prepopulate a form such as the sample confirmation form. (See "Confirmation form" in "Rendering Interactive PDF Forms" on page 582.)

To prepopulate the sample confirmation form, you have to create an XML data source that contains three XML elements that match the three fields in the form. This form contains the following three fields: `FirstName`, `LastName`, and `Amount`. The first step is to create an XML data source that contains XML elements that match the fields located in the form design. The next step is to assign data values to the XML elements, as shown in the following XML code.

```
<Untitled>
    <FirstName>Jerry</FirstName>
    <LastName>Johnson</LastName>
    <Amount>250000</Amount>
</Untitled>
```

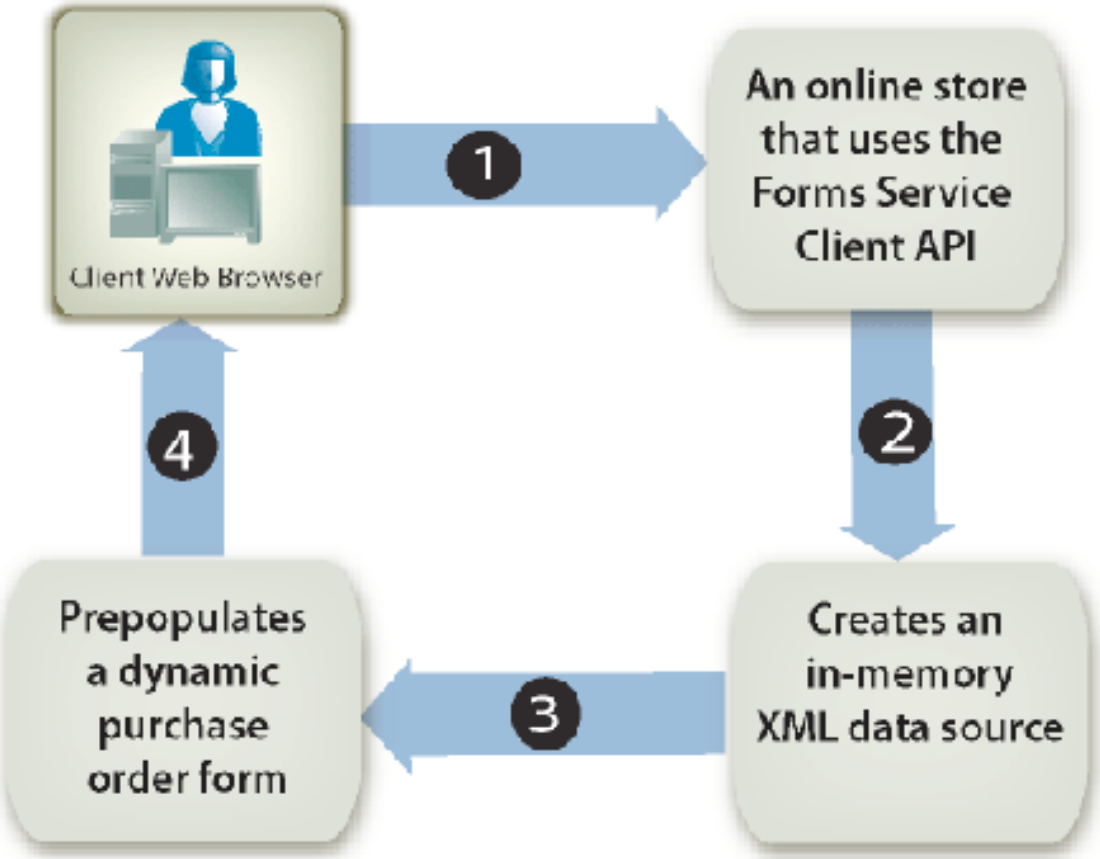
After you prepopulate the confirmation form with this XML data source and then render the form, the data values that you assigned to the XML elements are displayed, as shown in the following diagram.

Prepopulating forms with flowable layouts

Forms with flowable layouts are useful to display an undetermined amount of data to users. Because the layout of the form adjusts automatically to the amount of data that is merged, you do not need to predetermine a fixed layout or number of pages for the form as you need to do with a form with a fixed layout.

A form is typically populated with data that is obtained during run-time. As a result, you can prepopulate a form by creating an in-memory XML data source and placing the data directly into the in-memory XML data source.

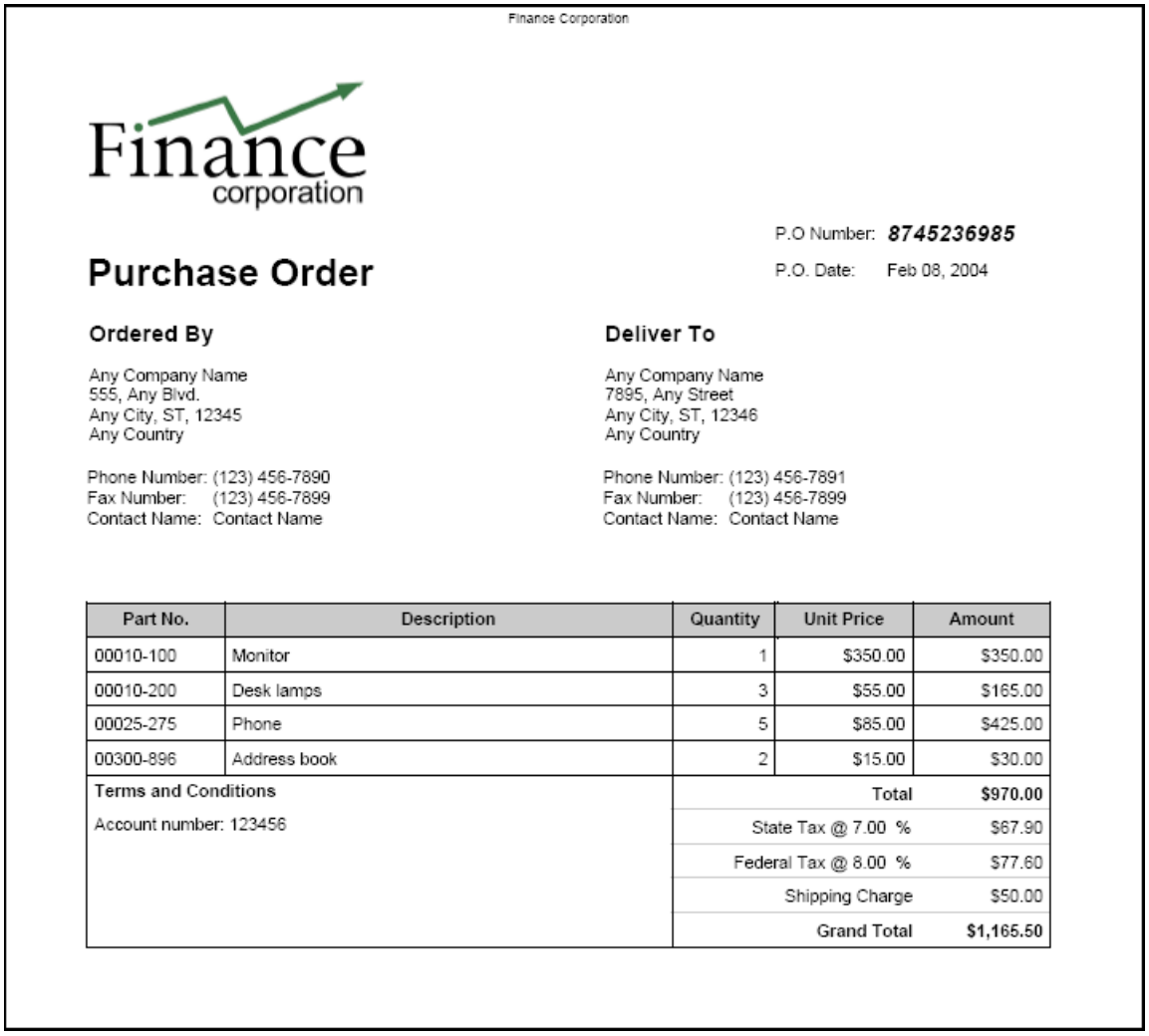
Consider a web-based application, such as an online store. After an online shopper finishes purchasing items, all purchased items are placed into an in-memory XML data source that is used to prepopulate a form. The following diagram shows this process, which is explained in the table following the diagram.



The following table describes the steps in this diagram.

Step	Description
1	A user purchases items from a web-based online store.
2	After the user finishes purchasing items and clicks the Submit button, an in-memory XML data source is created. Purchased items and user information are placed into the in-memory XML data source.
3	The XML data source is used to prepopulate a purchase order form (an example of this form is shown following this table).
4	The purchase order form is rendered to the client web browser.

The following diagram shows an example of a purchase order form. The information in the table can adjust to the number of records in the XML data.



Note: *A form can be prepopulated with data from other sources such as an enterprise database or external applications.*

Form design considerations

Forms with flowable layouts are based on form designs that are created in Designer. A form design specifies a set of layout, presentation, and data capture rules, including calculating values based on user input. The rules are applied when data is entered into a form. Fields that are added to a form are subforms that are within the form design. For example, in the purchase order form shown in the previous diagram, each line is a subform. For information about creating a form design that contains subforms, see Creating a purchase order form that has a flowable layout.

Understanding data subgroups

An XML data source is used to prepopulate forms with fixed layouts and flowable layouts. However, the difference is that an XML data source that prepopulates a form with a flowable layout contains repeating XML elements that are used to prepopulate subforms that are repeated within the form. These repeating XML elements are called data subgroups.

An XML data source that is used to prepopulate the purchase order form shown in the previous diagram contains four repeating data subgroups. Each data subgroup corresponds to a purchased item. The purchased items are a monitor, a desk lamp, a phone, and an address book.

The following XML data source is used to prepopulate the purchase order form.

```
<header>
    <!-- XML elements used to prepopulate non-repeating fields such as address
    <!and city
    <txtPONum>8745236985</txtPONum>
    <dtmDate>2004-02-08</dtmDate>
    <txtOrderedByCompanyName>Any Company Name</txtOrderedByCompanyName>
    <txtOrderedByAddress>555, Any Blvd.</txtOrderedByAddress>
    <txtOrderedByCity>Any City</txtOrderedByCity>
    <txtOrderedByStateProv>ST</txtOrderedByStateProv>
    <txtOrderedByZipCode>12345</txtOrderedByZipCode>
    <txtOrderedByCountry>Any Country</txtOrderedByCountry>
    <txtOrderedByPhone>(123) 456-7890</txtOrderedByPhone>
    <txtOrderedByFax>(123) 456-7899</txtOrderedByFax>
    <txtOrderedByContactName>Contact Name</txtOrderedByContactName>
    <txtDeliverToCompanyName>Any Company Name</txtDeliverToCompanyName>
    <txtDeliverToAddress>7895, Any Street</txtDeliverToAddress>
    <txtDeliverToCity>Any City</txtDeliverToCity>
    <txtDeliverToStateProv>ST</txtDeliverToStateProv>
    <txtDeliverToZipCode>12346</txtDeliverToZipCode>
    <txtDeliverToCountry>Any Country</txtDeliverToCountry>
    <txtDeliverToPhone>(123) 456-7891</txtDeliverToPhone>
    <txtDeliverToFax>(123) 456-7899</txtDeliverToFax>
    <txtDeliverToContactName>Contact Name</txtDeliverToContactName>
</header>
<detail>
    <!-- A data subgroup that contains information about the monitor>
    <txtPartNum>00010-100</txtPartNum>
    <txtDescription>Monitor</txtDescription>
    <numQty>1</numQty>
    <numUnitPrice>350.00</numUnitPrice>
</detail>
<detail>
```

```
        <!-- A data subgroup that contains information about the desk lamp>
        <txtPartNum>00010-200</txtPartNum>
        <txtDescription>Desk lamps</txtDescription>
        <numQty>3</numQty>
        <numUnitPrice>55.00</numUnitPrice>
    </detail>
    <detail>
        <!-- A data subgroup that contains information about the Phone>
            <txtPartNum>00025-275</txtPartNum>
            <txtDescription>Phone</txtDescription>
            <numQty>5</numQty>
            <numUnitPrice>85.00</numUnitPrice>
    </detail>
    <detail>
        <!-- A data subgroup that contains information about the address book>
        <txtPartNum>00300-896</txtPartNum>
        <txtDescription>Address book</txtDescription>
        <numQty>2</numQty>
        <numUnitPrice>15.00</numUnitPrice>
    </detail>
```

Notice that each data subgroup contains four XML elements that correspond to this information:

- Items part number
- Items description
- Quantity of items
- Unit price

The name of a data subgroup's parent XML element must match the name of the subform that is located in the form design. For example, in the previous diagram, notice that the name of the data subgroup's parent XML element is `detail`. This corresponds to the name of the subform that is located in the form design on which the purchase order form is based. If the name of the data subgroup's parent XML element and the subform do not match, a server-side form is not prepopulated.

Each data subgroup must contain XML elements that match the field names in the subform. The `detail` subform located in the form design contains the following fields:

- txtPartNum
- txtDescription
- numQty
- numUnitPrice

Note: *If you attempt to prepopulate a form with a data source that contains repeating XML elements and you set the* `RenderAtClient` *option to* `No`*, only the first data record is merged into the form. To ensure that all data records are merged into the form, set the* `RenderAtClient` *to* `Yes`*. For information about the* `RenderAtClient` *option, see* *"Rendering Forms at the Client" on page 596.*

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To prepopulate a form with a flowable layout, perform the following tasks:

1 Include project files.

2 Create an in-memory XML data source.

3 Convert the XML data source.

4 Render a prepopulated form.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create an in-memory XML data source

You can use `org.w3c.dom` classes to create an in-memory XML data source to prepopulate a form with a flowable layout. You must place data into an XML data source that conforms to the form. For information about the relationship between a form with a flowable layout and the XML data source, see "Understanding data subgroups" on page 648.

Convert the XML data source

An in-memory XML data source that is created by using `org.w3c.dom` classes can be converted to a `com.adobe.idp.Document` object before it can be used to prepopulate a form. An in-memory XML data source can be converted by using Java XML transform classes.

Note: *If you are using the Forms service's WSDL to prepopulate a form, you must convert a* `org.w3c.dom.Document` *object into a* `BLOB` *object.*

Render a prepopulated form

You render a prepopulated form just like other form. The only difference is that you use the `com.adobe.idp.Document` object that contains the XML data source to prepopulate the form.

See also

"Prepopulating forms using the Java API" on page 650

"Prepopulating forms using the web service API" on page 653

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Creating Web Applications that Renders Forms" on page 670

Prepopulating forms using the Java API

To prepopulate a form with a flowable layout by using the Forms API (Java), perform the following steps:

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path. For information about the location of these files, see "Including AEM Forms Java library files" on page 491.

2 Create an in-memory XML data source

- Create a Java `DocumentBuilderFactory` object by calling the `DocumentBuilderFactory` class' `newInstance` method.
- Create a Java `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.
- Call the `DocumentBuilder` object's `newDocument` method to instantiate a `org.w3c.dom.Document` object.
- Create the XML data source's root element by invoking the `org.w3c.dom.Document` object's `createElement` method. This creates an `Element` object that represents the root element. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the root element to the document by calling the `Document` object's `appendChild` method, and pass the root element object as an argument. The following lines of code shows this application logic:

```
Element root = (Element)document.createElement("transaction");
document.appendChild(root);
```

- Create the XML data source's header element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the header element to the root element by calling the `root` object's `appendChild` method, and pass the header element object as an argument. The XML elements that are appended to the header element correspond to the static portion of the form. The following lines of code show this application logic:

```
Element header = (Element)document.createElement("header");
root.appendChild(header);
```

- Create a child element that belongs to the header element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling its `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that appears as the child element's value. Finally, append the child element to the header element by calling the header element's `appendChild` method, and pass the child element object as an argument. The following lines of code show this application logic:

```
Element poNum= (Element)document.createElement("txtPONum");
poNum.appendChild(document.createTextNode("8745236985"));
header.appendChild(LastName);
```

- Add all remaining elements to the header element by repeating the last sub-step for each field appearing in the static portion of the form (in the XML data source diagram, these fields are shown in section A. (See "Understanding data subgroups" on page 648.)
- Create the XML data source's detail element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the detail element to the root element by calling the `root` object's `appendChild` method, and pass the detail element object as an argument. The XML elements that are appended to the detail element correspond to the dynamic portion of the form. The following lines of code show this application logic:

```
Element detail = (Element)document.createElement("detail");
root.appendChild(detail);
```

- Create a child element that belongs to the detail element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling its `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that appears as the child element's value. Finally, append the child element to the detail element by calling the detail element's `appendChild` method, and pass the child element object as an argument. The following lines of code show this application logic:

```
Element txtPartNum = (Element)document.createElement("txtPartNum");
txtPartNum.appendChild(document.createTextNode("00010-100"));
detail.appendChild(txtPartNum);
```

- Repeat the last sub-step for all XML elements to append to the detail element. To properly create the XML data source used to populate the purchase order form, you must append the following XML elements to the detail element: `txtDescription`, `numQty`, and `numUnitPrice`.
- Repeat the last two sub-steps for all data items used to prepopulate the form.

3 Convert the XML data source

- Create a `javax.xml.transform.Transformer` object by invoking the `javax.xml.transform.Transformer` object's static `newInstance` method.
- Create a `Transformer` object by invoking the `TransformerFactory` object's `newTransformer` method.
- Create a `ByteArrayOutputStream` object by using its constructor.
- Create a `javax.xml.transform.dom.DOMSource` object by using its constructor and passing the `org.w3c.dom.Document` object that was created in step 1.
- Create a `javax.xml.transform.dom.DOMSource` object by using its constructor and passing the `ByteArrayOutputStream` object.
- Populate the Java `ByteArrayOutputStream` object by invoking the `javax.xml.transform.Transformer` object's `transform` method and passing the `javax.xml.transform.dom.DOMSource` and the `javax.xml.transform.stream.StreamResult` objects.
- Create a byte array and allocate the size of the `ByteArrayOutputStream` object to the byte array.
- Populate the byte array by invoking the `ByteArrayOutputStream` object's `toByteArray` method.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the byte array.

4 Render a prepopulated form

Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension.
- A `com.adobe.idp.Document` object that contains data to merge with the form. Ensure that you use the `com.adobe.idp.Document` object created in steps one and two.
- A `PDFFormRenderSpec` object that stores run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

- Create a `javax.servlet.ServletOutputStream` object used to send a form data stream to the client web browser.
- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.

- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Prepopulating forms with flowable layouts" on page 645

"Quick Start (SOAP mode): Prepopulating Forms with Flowable Layouts using the Java API" on page 190

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Prepopulating forms using the web service API

To prepopulate a form with a flowable layout by using the Forms API (web service), perform the following steps:

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL. (See "Creating Java proxy classes using Apache Axis" on page 523.)
- Include the Java proxy classes into your class path.

2 Create an in-memory XML data source

- Create a Java `DocumentBuilderFactory` object by calling the `DocumentBuilderFactory` class' `newInstance` method.
- Create a Java `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.
- Call the `DocumentBuilder` object's `newDocument` method to instantiate a `org.w3c.dom.Document` object.
- Create the XML data source's root element by invoking the `org.w3c.dom.Document` object's `createElement` method. This creates an `Element` object that represents the root element. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the root element to the document by calling the `Document` object's `appendChild` method, and pass the root element object as an argument. The following lines of code show this application logic:

```
Element root = (Element)document.createElement("transaction");
document.appendChild(root);
```

- Create the XML data source's header element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the header element to the root element by calling the `root` object's `appendChild` method, and pass the header element object as an argument. The XML elements that are appended to the header element correspond to the static portion of the form. The following lines of code show this application logic:

```
Element header = (Element)document.createElement("header");
root.appendChild(header);
```

- Create a child element that belongs to the header element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling its `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that appears as the child element's value. Finally, append the child element to the header element by calling the header element's `appendChild` method, and pass the child element object as an argument. The following lines of code shows this application logic:

```
Element poNum= (Element)document.createElement("txtPONum");
poNum.appendChild(document.createTextNode("8745236985"));
header.appendChild(LastName);
```

- Add all remaining elements to the header element by repeating the last sub-step for each field appearing in the static portion of the form (in the XML data source diagram, these fields are shown in section A. (See "Understanding data subgroups" on page 648.)
- Create the XML data source's detail element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the detail element to the root element by calling the `root` object's `appendChild` method, and pass the detail element object as an argument. The XML elements that are appended to the detail element correspond to the dynamic portion of the form. The following lines of code shows this application logic:

```
Element detail = (Element)document.createElement("detail");
root.appendChild(detail);
```

- Create a child element that belongs to the detail element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling its `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that appears as the child element's value. Finally, append the child element to the detail element by calling the detail element's `appendChild` method, and pass the child element object as an argument. The following lines of code shows this application logic:

```
Element txtPartNum = (Element)document.createElement("txtPartNum");
txtPartNum.appendChild(document.createTextNode("00010-100"));
detail.appendChild(txtPartNum);
```

- Repeat the last sub-step for all XML elements to append to the detail element. To properly create the XML data source used to populate the purchase order form, you must append the following XML elements to the detail element: `txtDescription`, `numQty`, and `numUnitPrice`.
- Repeat the last two sub-steps for all data items used to prepopulate the form.

3 Convert the XML data source

- Create a `javax.xml.transform.Transformer` object by invoking the `javax.xml.transform.Transformer` object's static `newInstance` method.
- Create a `Transformer` object by invoking the `TransformerFactory` object's `newTransformer` method.
- Create a `ByteArrayOutputStream` object by using its constructor.
- Create a `javax.xml.transform.dom.DOMSource` object by using its constructor and passing the `org.w3c.dom.Document` object that was created in step 1.
- Create a `javax.xml.transform.dom.DOMSource` object by using its constructor and passing the `ByteArrayOutputStream` object.
- Populate the Java `ByteArrayOutputStream` object by invoking the `javax.xml.transform.Transformer` object's `transform` method and passing the `javax.xml.transform.dom.DOMSource` and the `javax.xml.transform.stream.StreamResult` objects.
- Create a byte array and allocate the size of the `ByteArrayOutputStream` object to the byte array.
- Populate the byte array by invoking the `ByteArrayOutputStream` object's `toByteArray` method.
- Create a `BLOB` object by using its constructor and invoke its `setBinaryData` method and pass the byte array.

4 Render a prepopulated form

Invoke the `FormsService` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension.
- A `BLOB` object that contains data to merge with the form. Ensure that you use the `BLOB` object that was created in steps one and two.
- A `PDFFormRenderSpecc` object that stores run-time options. For more information, see AEM Forms API Reference.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This is used to store the rendered PDF form.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. (This argument will store the number of pages in the form).
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. (This argument will store the locale value).
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

Note: *The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.*

See also

"Prepopulating forms with flowable layouts" on page 645

Quick Start (Base64): Prepopulating Forms with Flowable Layouts using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Calculating Form Data

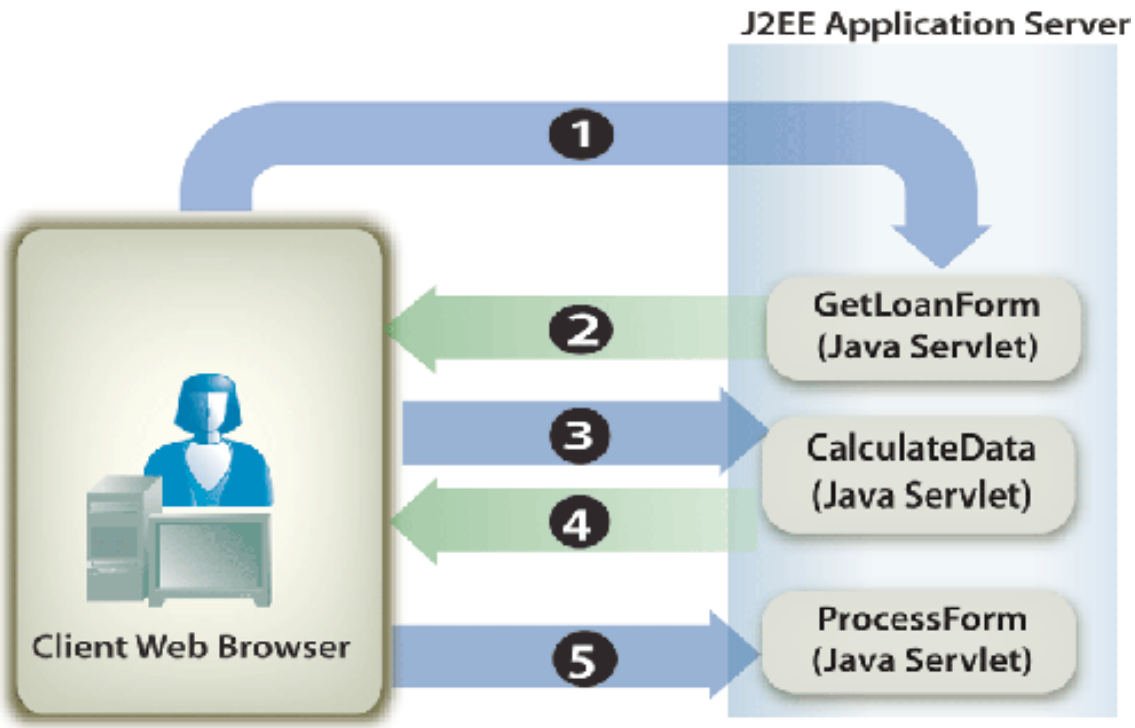
The Forms service can calculate the values that a user enters into a form and display the results. To calculate form data, you must perform two tasks. First, you create a form design script that calculates form data. A form design supports three types of scripts. One script type runs on the client, another runs on the server, and the third type runs on both the server and the client. The script type discussed in this topic runs on the server. Server-side calculations are supported for HTML, PDF, and form Guide (deprecated) transformations.

As part of the form design process, you can make use of calculations and scripts to provide a richer user experience. Calculations and scripts can be added to most form fields and objects. You must create a form design script to perform calculation operations on data that a user enters into an interactive form.

The user enters values into the form and clicks the Calculate button to view the results. The following process describes an example application that enables a user to calculate data:

- The user accesses an HTML page named StartLoan.html that acts as the web application's start page. This page invokes a Java Servlet named `GetLoanForm`.
- The `GetLoanForm` servlet renders a loan form. This form contains a script, interactive fields, a calculate button, and a submit button.
- The user enters values into the form's fields and clicks the Calculate button. The form is sent to the `CalculateData` Java Servlet where the script is executed. The form is sent back to the user with the calculation results displayed in the form.
- The user continues entering and calculating values until a satisfactory result is displayed. When satisfied, the user clicks the Submit button to process the form. The form is sent to another Java Servlet named `ProcessForm` that is responsible for retrieving submitted data. (See "Handling Submitted Forms" on page 630.)

The following diagram shows the application's logic flow.

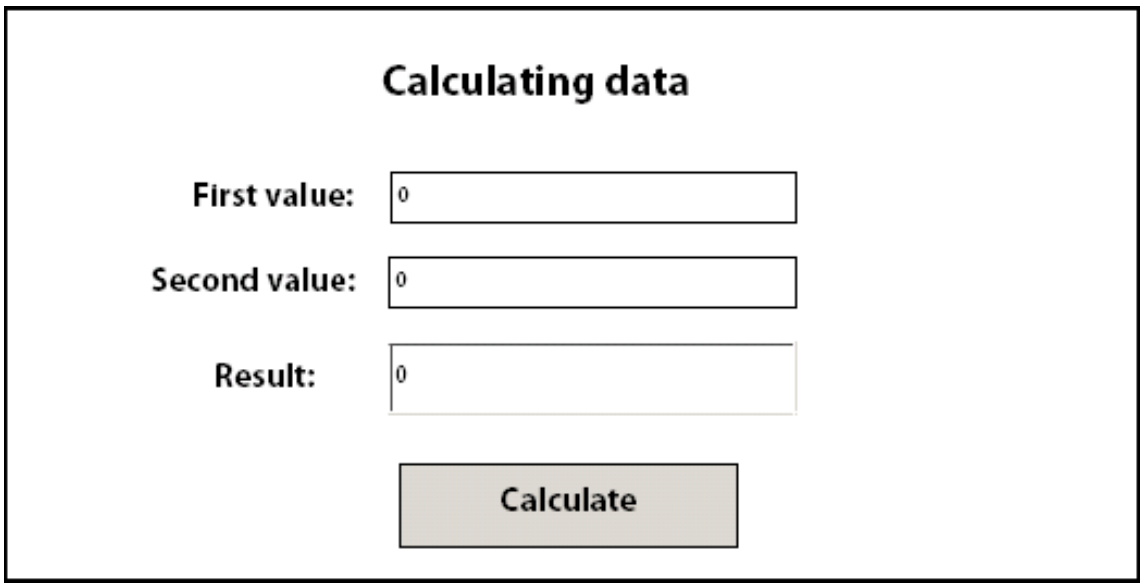


The following table describes the steps in this diagram.

Step	Description
1	The `GetLoanForm` Java Servlet is invoked from the HTML start page.
2	The `GetLoanForm` Java Servlet uses the Forms service Client API to render the loan form to the client web browser. The difference between rendering a form that contains a script configured to run on the server and rendering a form that does not contain a script is that you must specify the target location used to execute the script. If a target location is not specified, a script that is configured to run on the server is not executed. For example, consider the application introduced in this section. The `CalculateData` Java Servlet is the target location where the script is executed.
3	The user enters data into interactive fields and clicks the Calculate button. The form is sent to the `CalculateData` Java Servlet, where the script is executed.
4	The form is rendered back to the web browser with the calculation results displayed in the form.
5	The user clicks the Submit button when the values are satisfactory. The form is sent to another Java Servlet named `ProcessForm`.

Typically, a form that is submitted as PDF content contains scripts that are executed on the client. However, server-side calculations can also be executed. A Submit button cannot be used to calculate scripts. In this situation, calculations are not executed because the Forms service considers the interaction to be complete.

To illustrate the usage of a form design script, this section examines a simple interactive form that contains a script that is configured to run on the server. The following diagram shows a form design containing a script that adds values that a user enters into the first two fields and displays the result in the third field.



***A.** A field named NumericField1 **B.** A field named NumericField2 **C.** A field named NumericField3*

The syntax of the script located in this form design is as follows:

```
NumericField3 = NumericField2 + NumericField1
```

In this form design, the Calculate button is a command button, and the script is located in this button's `Click` event. When a user enters values into the first two fields (NumericField1 and NumericField2) and clicks the Calculate button, the form is sent to the Forms service, where the script is executed. The Forms service renders the form back to the client device with the results of the calculation displayed in the NumericField3 field.

***Note:** For information about creating a form design script, see Forms Designer.*

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To calculate form data, perform the following tasks:

1 Include project files.

2 Create a Forms Client API object.

3 Retrieve a form containing a calculation script.

4 Write the form data stream back to the client web browser

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms service client. If you are using the Java API, create a `FormsServiceClient` object. If you are using the Forms web service API, create a `FormsServiceService` object.

Retrieve a form containing a calculation script

You use the Forms service Client API to create application logic that handles a form that contains a script configured to run on the server. The process is similar to handling a submitted form. (See "Handling Submitted Forms" on page 630.)

Verify that the processing state associated with the submitted form is `1(Calculate)`, which means that the Forms service is performing a calculation operation on the form data and the results must be written back to the user. In this situation, a script configured to run on the server is automatically executed.

Write the form data stream back to the client web browser

After you verify the processing state associated with a submitted form is `1`, you must write the results back to the client web browser. When the form is displayed, the calculated value will appear in the appropriate field(s).

See also

"Calculate form data using the Java API" on page 659

"Calculate form data using the web service API" on page 660

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Rendering Forms as HTML" on page 609

"Creating Web Applications that Renders Forms" on page 670

Calculate form data using the Java API

Calculate form data by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve a form containing a calculation script

- To retrieve form data that contains a calculation script, create a `com.adobe.idp.Document` object by using its constructor and invoking the `javax.servlet.http.HttpServletResponse` object's `getInputStream` method from within the constructor.
- Invoke the `FormsServiceClient` object's `processFormSubmission` method and pass the following values:
  - The `com.adobe.idp.Document` object that contains the form data.
  - A string value that specifies environment variables including all relevant HTTP headers. You must specify the content type to handle by specifying one or more values for the `CONTENT_TYPE` environment variable. For example, to handle XML and PDF data, specify the following string value for this parameter: `CONTENT_TYPE=application/xml&CONTENT_TYPE=application/pdf`
  - A string value that specifies the `HTTP_USER_AGENT` header value; for example, `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322).`
  - A `RenderOptionsSpec` object that stores run-time options.

  The `processFormSubmission` method returns a `FormsResult` object containing the results of the form submission.
- Verify that the processing state associated with a submitted form is `1` by invoking the `FormsResult` object's `getAction` method. If this method returns the value `1`, the calculation was performed and the data can be written back to the client web browser.

4 Write the form data stream back to the client web browser

- Create a `javax.servlet.ServletOutputStream` object used to send a form data stream to the client web browser.
- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Calculating Form Data" on page 656

"Quick Start (SOAP mode): Handling a form containing a calculation script using the Java API" on page 197

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Calculate form data using the web service API

Calculate form data by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Retrieve a form containing a calculation script

- To retrieve form data that was posted to a Java Servlet, create a `BLOB` object by using its constructor.
- Create a `java.io.InputStream` object by using the `javax.servlet.http.HttpServletResponse` object's `getInputStream` method.
- Create a `java.io.ByteArrayOutputStream` object by using its constructor and passing the length of the `java.io.InputStream` object.
- Copy the contents of the `java.io.InputStream` object into the `java.io.ByteArrayOutputStream` object.
- Create a byte array by invoking the `java.io.ByteArrayOutputStream` object's `toByteArray` method.
- Populate the `BLOB` object by invoking its `setBinaryData` method and passing the byte array as an argument.
- Create a `RenderOptionsSpec` object by using its constructor. Set the locale value by invoking the `RenderOptionsSpec` object's `setLocale` method and passing a string value that specifies the locale value.
- Invoke the `FormsServiceClient` object's `processFormSubmission` method and pass the following values:
  - The `BLOB` object that contains the form data.
  - A string value that specifies environment variables included all relevant HTTP headers. For example, you can specify the following string value: `HTTP_REFERER=referrer&HTTP_CONNECTION=keep-alive&CONTENT_TYPE=application/xml`
  - A string value that specifies the `HTTP_USER_AGENT` header value; for example, `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322).`
  - A `RenderOptionsSpec` object that stores run-time options. For more information, .
  - An empty `BLOBHolder` object that is populated by the method.
  - An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method.
  - An empty `BLOBHolder` object that is populated by the method.
  - An empty `BLOBHolder` object that is populated by the method.
  - An empty `javax.xml.rpc.holders.ShortHolder` object that is populated by the method.
  - An empty `MyArrayOf_xsd_anyTypeHolder` object that is populated by the method. This parameter is used to store file attachments that are submitted along with the form.
  - An empty `FormsResultHolder` object that is populated by the method with the form that is submitted.

The `processFormSubmission` method populates the `FormsResultHolder` parameter with the results of the form submission. The `processFormSubmission` method returns a `FormsResult` object containing the results of the form submission.

- Verify that the processing state associated with a submitted form is `1` by invoking the `FormsResult` object's `getAction` method. If this method returns the value `1`, the calculation was performed and the data can be written back to the client web browser.

4 Write the form data stream back to the client web browser

- Create a `javax.servlet.ServletOutputStream` object used to send a form data stream to the client web browser.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Calculating Form Data" on page 656

Quick Start (Base64): Handling a form containing a calculation script using web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Rendering Forms By Value

Rendering Forms by Value

Typically, a form design that is created in Designer is passed by reference to the Forms service. Form designs can be large and, as a result, it is more efficient to pass them by reference to avoid having to marshal form design bytes by value. The Forms service can also cache the form design so that when cached, it does not have to continually read the form design.

If a form design contains a UUID attribute, then it is cached. The UUID value is unique for all form designs and is used to uniquely identify a form. When rendering a form by value, the form should only be cached when it is used repeatedly. However, if the form is not used repeatedly and has to be unique, you can avoid caching the form using caching options that are set using the AEM Forms API.

The Forms service can also resolve the location of linked content within the form design. For example, linked images that are referenced from within the form design are relative URLs. Linked content is always assumed to be relative to the form design location. Therefore, resolving linked content is a matter of determining its location by applying the relative path to the absolute form design location.

Instead of passing a form design by reference, you can pass a form design by value. Passing a form design by value is efficient when a form design is dynamically created; that is, when a client application generates the XML that creates a form design during run-time. In this situation a form design is not stored in a physical repository because it is stored in memory. When dynamically creating a form design at run-time and passing it by value, you can cache the form and improve performance of the Forms service.

Limitations of passing a form by value

The following limitations apply when a form design is passed by value:

- No relative linked content can be within the form design. All images and fragments must be embedded inside the form design or referred to absolutely.
- Server-side calculations cannot be performed after the form is rendered. If the form is submitted back to the Forms service, the data is extracted and returned without any server-side calculations.
- Because HTML can only use linked images at run time, it is not possible to generate HTML with embedded images. This is because the Forms service supports embedded images with HTML by retrieving the images from a referenced form design. Because a form design that is passed by value does not have a referenced location, embedded images cannot be extracted when the HTML page is displayed. Therefore, image references must be absolute paths to be rendered in HTML.

Note: *Although you can render different types of forms by value (for example, HTML forms or forms that contain usage rights), this section discusses rendering interactive PDF forms.*

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To render a form by value, perform the following steps:

1. Include project files.
2. Create a Forms Client API object.
3. Reference the form design.
4. Render a form by value.
5. Write the form data stream to the client web browser.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically import data into a PDF form Client API, you must create a Data Integration service client. When creating a service client, you define connection settings that are required to invoke a service.

Reference the form design

When rendering a form by value, you have to create a `com.adobe.idp.Document` object that contains the form design to render. You can reference an existing XDP file or you can dynamically create an form design at run-time and populate a `com.adobe.idp.Document` with that data.

Note: *This section and the corresponding quick start references an existing XDP file.*

Render a form by value

To render a form by value, pass a `com.adobe.idp.Document` instance that contains the form design to the render method's `inDataDoc` parameter (can be any of the `FormsServiceClient` object's render methods such as `renderPDFForm`, `(Deprecated) renderHTMLForm`, and so on). This parameter value is normally reserved for data that is merged with the form. Likewise, pass an empty string value to the `formQuery` parameter. Normally this parameter requires a string value that specifies the name of the form design.

Note: *If you want to display data within the form, the data must be specified within the* `xfa:datasets` *element. For information about XFA architecture, go to http://partners.adobe.com/public/developer/xml/index_arch.html.*

Write the form data stream to the client web browser

When the Forms service renders a form by value, it returns a form data stream that you must write to the client web browser. When written to the client web browser, the form is visible to the user.

See also

"Render a form by value using the Java API" on page 663

"Render a form by value using the web service API" on page 664

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Passing Documents to the Forms Service" on page 591

"Creating Web Applications that Renders Forms" on page 670

Render a form by value using the Java API

Render a form by value using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference the form design

- Create a `java.io.FileInputStream` object that represents the form design to render by using its constructor and passing a string value that specifies the location of the XDP file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Render a form by value

Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- An empty string value. (Normally this parameter requires a string value that specifies the name of the form design.)
- A `com.adobe.idp.Document` object that contains the form design. Normally this parameter value is reserved for data that is merged with the form.
- A `PDFFormRenderSpec` object that stores run-time options. This is an optional parameter and you can specify `null` if you do not want to specify run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that can be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and allocate the size of the `InputStream` object. Invoke the `InputStream` object's `available` method to obtain the size of the `InputStream` object.
- Populate the byte array with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms By Value" on page 661

"Quick Start (SOAP mode): Rendering by value using the Java API" on page 202

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Render a form by value using the web service API

Render a form by value by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Reference the form design

- Create a `java.io.FileInputStream` object by using its constructor. Pass a string value that specifies the location of the XDP file.
- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is encrypted with a password.
- Create a byte array that stores the content of the `java.io.FileInputStream` object. You can determine the size of the byte array by getting the `java.io.FileInputStream` object's size using its `available` method.

- Populate the byte array with stream data by invoking the `java.io.FileInputStream` object's `read` method and passing the byte array.
- Populate the `BLOB` object by invoking its `setBinaryData` method and passing the byte array.

4 Render a form by value

Invoke the `FormsService` object's `renderPDFForm` method and pass the following values:

- An empty string value. (Normally this parameter requires a string value that specifies the name of the form design.)
- A `BLOB` object that contains the form design. Normally this parameter value is reserved for data that is merged with the form.
- A `PDFFormRenderSpec` object that stores run-time options. This is an optional parameter and you can specify `null` if you do not want to specify run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This is used to store the rendered PDF form.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. (This argument stores the number of pages in the form.)
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. (This argument stores the locale value.)
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Get the content type of the `BLOB` object by invoking its `getContentType` method.
- Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `BLOB` object.
- Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Rendering Forms By Value" on page 661

Quick Start (Base64): Rendering by value using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Optimizing the Performance of the Forms Service

Optimizing the Performance of the Forms Service

When rendering a form, you can set run-time options that will optimize the performance of the Forms service. Another task that you can perform to improve the performance of the Forms service is to store XDP files in the repository. However, this section does not describe how to perform this task. (See "Invoking a service using a Java client library" on page 511.)

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To optimize the performance of the Forms service while rendering a form, perform the following tasks:

1. Include project files.
2. Create a Forms Client API object.
3. Set performance run-time options.
4. Render the form.
5. Write the form data stream to the client web browser.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Forms Client API object

Before you can programmatically perform a Forms service Client API operation, you must create a Forms service client. If you are using the Java API, create a `FormsServiceClient` object. If you are using the Forms web service API, create a `FormsService` object.

Set performance run-time options

You can set the following performance run-time options to improve the performance of the Forms service:

- **Form caching**: You can cache a form that is rendered as PDF in the server cache. Each form is cached after it is generated for the first time. On a subsequent render, if the cached form is newer than the form design's timestamp, the form is retrieved from the cache. By caching forms, you improve the performance of the Forms service because it does not have to retrieve the form design from a repository.
- Form Guides (deprecated) may take longer to render than other transformation types. It is recommended that you cache form Guides (deprecated) in order to improve performance.
- **Standalone option**: If you do not require the Forms service to perform server-side calculations, you can set the Standalone option to `true`, which results in forms being rendered without state information. State information is necessary if you want to render an interactive form to an end user who then enters information into the form and submits the form back to the Forms service. The Forms service then performs a calculation operation and renders the form back to the user with the results displayed in the form. If a form without state information is submitted back to the Forms service, only the XML data is available and server-side calculations are not performed.

- **Linearized PDF**: A linearized PDF file is organized to enable efficient incremental access in a network environment. The PDF file is valid PDF in all respects, and is compatible with all existing viewers and other PDF applications. That is, a linearized PDF can be viewed while it is still being downloaded.
- This option does not improve performance when a PDF form is rendered on the client.
- **GuideRSL option**: Enables form Guide (deprecated) generation using run-time shared libraries. This means the first request will download a smaller SWF file, plus larger shared-libraries that are stored in the browser cache. For more information, see RSL in the Flex documentation.
- You can also improve the performance of the Forms service by rendering a form on the client. (See "Rendering Forms at the Client" on page 596.)

Render the form

To render the form after setting performance options, you use the same application logic as rendering a form without performance options.

Write the form data stream to the client web browser

After the Forms service renders a form, it returns a form data stream that you must write to the client web browser. When written to the client web browser, the form is visible to the user.

See also

"Optimize the performance using the Java API" on page 667

"Optimize the performance using the web service API" on page 668

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Forms Service API Quick Starts" on page 153

"Rendering Interactive PDF Forms" on page 582

"Rendering Forms as HTML" on page 609

"Creating Web Applications that Renders Forms" on page 670

Optimize the performance using the Java API

Render a form with optimized performance by using the Forms API (Java):

1 Include project files

Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path.

2 Create a Forms Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set performance run-time options

- Create a `PDFFormRenderSpec` object by using its constructor.
- Set the form cache option by invoking the `PDFFormRenderSpec` object's `setCacheEnabled` method and passing `true`.

- Set the linearized option by invoking the `PDFFormRenderSpec` object's `setLinearizedPDF` method and passing `true`.

4 Render the form

Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension.
- A `com.adobe.idp.Document` object that contains data to merge with the form. If you do not want to merge data, pass an empty `com.adobe.idp.Document` object.
- A `PDFFormRenderSpec` object that stores run-time options to improve performance.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `javax.servlet.ServletOutputStream` object used to send a form data stream to the client web browser.
- Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.
- Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array and populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.
- Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Optimizing the Performance of the Forms Service" on page 666

"Quick Start (SOAP mode): Optimizing performance using the Java API" on page 199

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Optimize the performance using the web service API

Render a form with optimized performance by using the Forms API (web service):

1 Include project files

- Create Java proxy classes that consume the Forms service WSDL.
- Include the Java proxy classes into your class path.

2 Create a Forms Client API object

Create a `FormsService` object and set authentication values.

3 Set performance run-time options

- Create a `PDFFormRenderSpec` object by using its constructor.
- Set the form cache option by invoking the `PDFFormRenderSpec` object's `setCacheEnabled` method and passing true.
- Set the standalone option by invoking the `PDFFormRenderSpec` object's `setStandAlone` method and passing true.
- Set the linearized option by invoking the `PDFFormRenderSpec` object's `setLinearizedPDF` method and passing true.

4 Render the form

Invoke the `FormsService` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension.
- A `BLOB` object that contains data to merge with the form. If you do not want to merge data, pass `null`.
- A `PDFFormRenderSpecc` object that stores run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.
- An empty `com.adobe.idp.services.holders.BLOBHolder` object that is populated by the method. This is used to store the rendered PDF form.
- An empty `javax.xml.rpc.holders.LongHolder` object that is populated by the method. (This argument will store the number of pages in the form).
- An empty `javax.xml.rpc.holders.StringHolder` object that is populated by the method. (This argument will store the locale value).
- An empty `com.adobe.idp.services.holders.FormsResultHolder` object that will contain the results of this operation.

The `renderPDFForm` method populates the `com.adobe.idp.services.holders.FormsResultHolder` object that is passed as the last argument value with a form data stream that must be written to the client web browser.

5 Write the form data stream to the client web browser

- Create a `FormResult` object by getting the value of the `com.adobe.idp.services.holders.FormsResultHolder` object's `value` data member.
- Create a `javax.servlet.ServletOutputStream` object used to send a form data stream to the client web browser.
- Create a `BLOB` object that contains form data by invoking the `FormsResult` object's `getOutputContent` method.
- Create a byte array and populate it by invoking the `BLOB` object's `getBinaryData` method. This task assigns the content of the `FormsResult` object to the byte array.
- Invoke the `javax.servlet.http.HttpServletResponse` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

See also

"Optimizing the Performance of the Forms Service" on page 666

Quick Start (Base64): Optimizing performance using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Creating Web Applications that Renders Forms

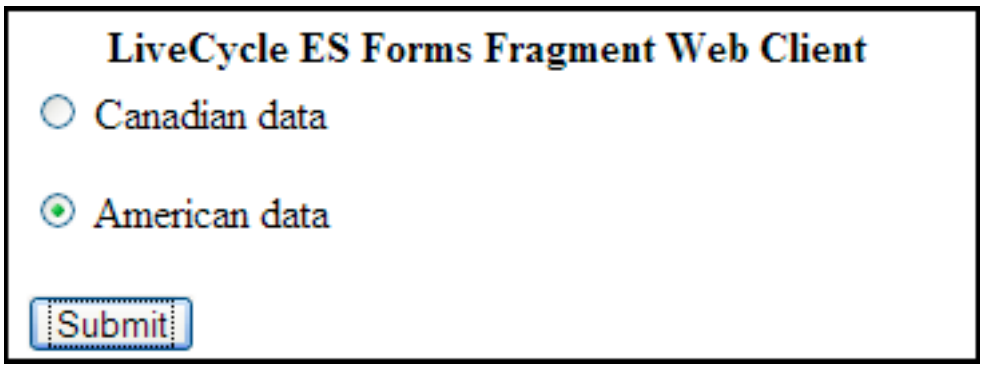
Creating Web Applications that Renders Forms

You can create a web-based application that uses Java servlets to invoke the Forms service and render forms. An advantage of using a Java™ servlet is that you can write the return value of the process to a client web browser. That is, a Java servlet can be used as the link between the Forms service that returns a form and a client web browser.

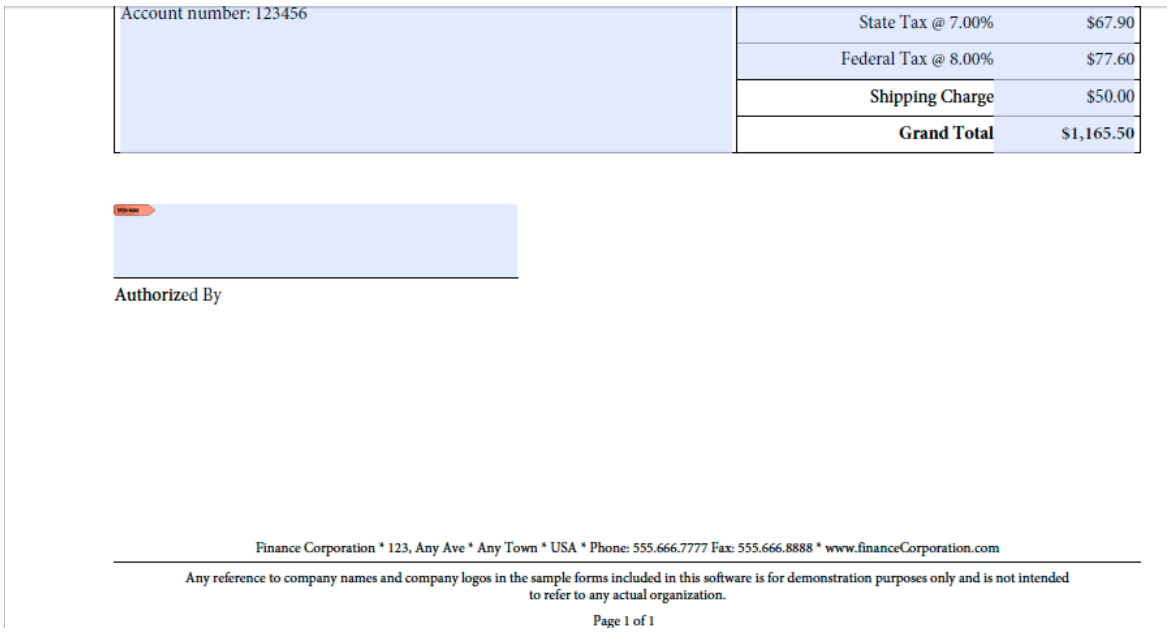
Note: *This section describes how to create a web-based application that uses a Java servlet that invokes the Forms service and renders forms-based on fragments. (See "Rendering Forms Based on Fragments" on page 600.)*

Using a Java servlet, you can write a form to a client web browser so that a customer can view and enter data into the form. After populating the form with data, the web user clicks a submit button located on the form to send information back to the Java servlet, where the data can be retrieved and processed. For example, the data can be sent to another process.

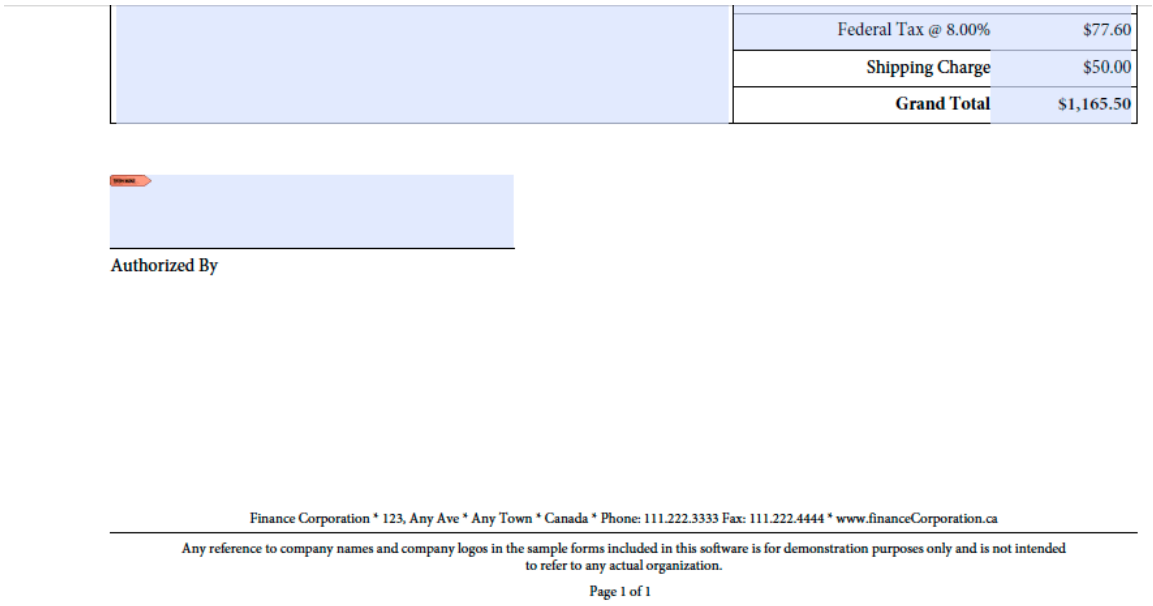
This section discusses how to create a web-based application that enables the user to select either American-based form data or Canadian-based form data, as shown in the following illustration.



The form that is rendered is a form that is based on fragments. That is, if the user selects American data, then the returned form uses fragments based on American data. For example, the footer of the form contains an American address, as shown in the following illustration.



Likewise, if the user selects Canadian data, then the returned form contains a Canadian address, as shown in the following illustration.



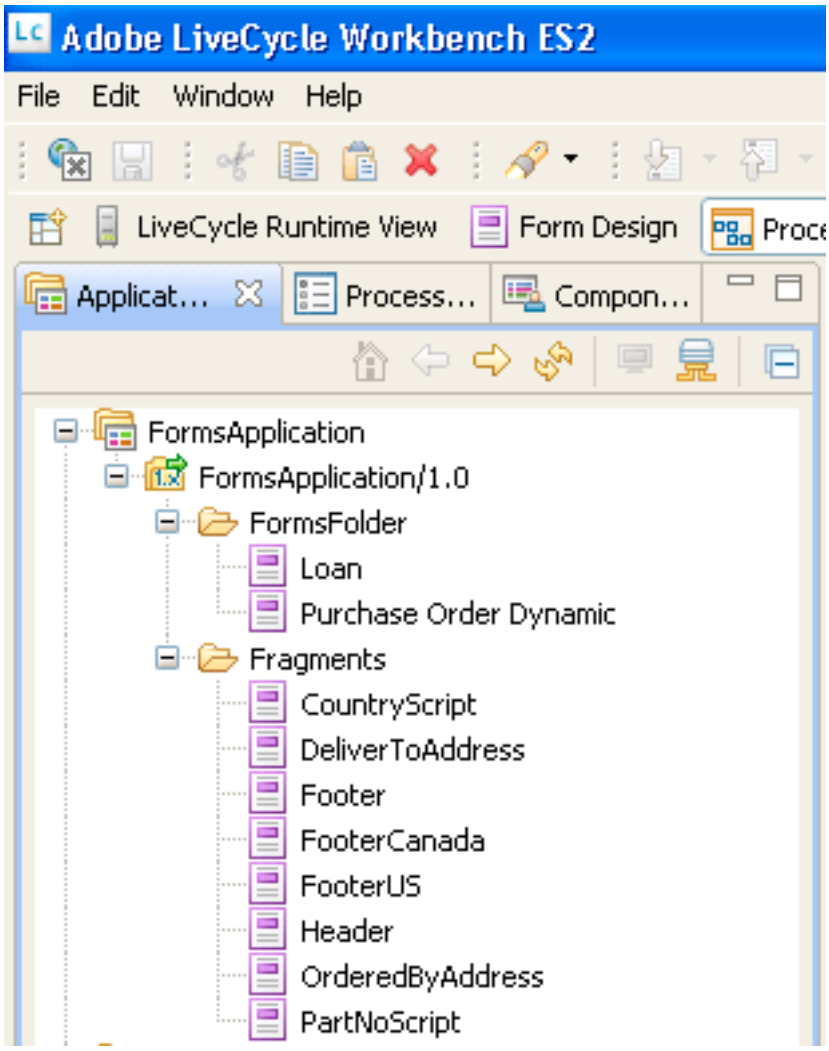
Note: *For information about creating form designs based on fragments, see Forms Designer.*

Sample Files

This section uses sample files that can be located in the following location:

<Forms Designer install directory>/Samples/Forms/Purchase Order/Form Fragments

where *<install directory>* is the installation path. For the purposes of the client application, the Purchase Order Dynamic.xdp file was copied from this installation location and deployed to a Forms application named *Applications/FormsApplication*. The Purchase Order Dynamic.xdp file is placed in a folder named FormsFolder. Likewise, the fragments are placed in folder named Fragments, as shown in the following illustration.



To access the Purchase Order Dynamic.xdp form design, specify `Applications/FormsApplication/1.0/FormsFolder/Purchase Order Dynamic.xdp` as the form name (the first parameter passed to the `renderPDFForm` method) and `repository:///` as the content root URI value.

The XML data files used by the web application were moved from the Data folder to `C:\Adobe` (the file system that belongs to the J2EE application server hosting AEM Forms). The file names are Purchase Order *Canada.xml* and Purchase Order *US.xml*.

Note: *For information about creating a Forms application using Workbench, see workbench Help.*

Summary of steps

To create a web-based applications that renders forms based on fragments, perform the following steps:

1 Create a new web project.

2 Create Java application logic that represents the Java servlet.

3 Create the web page for the web application.

4 Package the web application to a WAR file.

5 Deploy the WAR file to the J2EE application server.

6 Test your web application.

Note: *Some of these steps depend on the J2EE application on which AEM Forms is deployed. For example, the method you use to deploy a WAR file depends on the J2EE application server that you are using. This section assumes that AEM Forms is deployed on JBoss®.*

Creating a web project

The first step to create a web application that contains a Java servlet that can invoke the Forms service is to create a new web project. The Java IDE that this document is based on is Eclipse 3.3. Using the Eclipse IDE, create a web project and add the required JAR files to your project. Finally, add an HTML page named *index.html* and a Java servlet to your project.

The following list specifies the JAR files that you must add to your web project:

- adobe-forms-client.jar
- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar

For the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

To create a web project:

1 Start Eclipse and click **File** > **NewProject**.

2 In the **New Project** dialog box, select **Web** > **Dynamic Web Project**.

3 Type `FragmentsWebApplication` for the name of your project and then click **Finish**.

To add required JAR files to your project:

1 From the Project Explorer window, right-click the `FragmentsWebApplication` project and select **Properties**.

2 Click **Java build path** and then click the **Libraries** tab.

3 Click the **Add External JARs** button and browse to the JAR files to include.

To add a Java servlet to your project:

1 From the Project Explorer window, right-click the `FragmentsWebApplication` project and select **New** > **Other**.

2 Expand the **Web** folder, select **Servlet**, and then click **Next**.

3 In the Create Servlet dialog box, type `RenderFormFragment`for the name of the servlet and then click **Finish**.

To add an HTML page to your project:

1 From the Project Explorer window, right-click the `FragmentsWebApplication` project and select **New** > **Other**.

2 Expand the **Web** folder, select **HTML**, and click **Next**.

3 In the New HTMLdialog box, type `index.html` for the file name and then click **Finish**.

Note: *For information about creating the HTML page that invokes the* `RenderFormFragment` *Java servlet, see* *"Creating the web page" on page 678.*

Creating Java application logic for the servlet

You create Java application logic that invokes the Forms service from within the Java servlet. The following code shows the syntax of the `RenderFormFragment` Java Servlet:

```
public class RenderFormFragment extends HttpServlet implements Servlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp
    throws ServletException, IOException {
    doPost(req,resp);

    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp
    throws ServletException, IOException {
        //Add code here to invoke the Forms service
        }
```

Normally, you would not place client code within a Java servlet's `doGet` or `doPost` method. A better programming practice is to place this code within a separate class, instantiate the class from within the `doPost` method (or `doGet` method), and call the appropriate methods. However, for code brevity, the code examples in this section are kept to a minimum and code examples are placed in the `doPost` method.

To render a form based on fragments using the Forms service API, perform the following tasks:

1 Include client JAR files, such as adobe-forms-client.jar, in your Java project's class path. For information about the location of these files, see "Including AEM Forms Java library files" on page 491.

2 Retrieve the value of the radio button that is submitted from the HTML form and specifies whether to use American or Canadian data. If American is submitted, create a `com.adobe.idp.Document` that stores data located in the *Purchase Order US.xml*. Likewise, if Canadian, then create a `com.adobe.idp.Document` that stores data located in the *Purchase Order Canada.xml* file.

3 Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)

4 Create an `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

5 Create a `URLSpec` object that stores URI values by using its constructor.

6 Invoke the `URLSpec` object's `setApplicationWebRoot` method and pass a string value that represents the application's web root.

7 Invoke the `URLSpec` object's `setContentRootURI` method and pass a string value that specifies the content root URI value. Ensure that the form design and the fragments are located in the content root URI. If not, the Forms service throws an exception. To reference the AEM Forms repository, specify `repository://`.

8 Invoke the `URLSpec` object's `setTargetURL` method and pass a string value that specifies the target URL value to where form data is posted. If you define the target URL in the form design, you can pass an empty string. You can also specify the URL to where a form is sent in order to perform calculations.

9 Invoke the `FormsServiceClient` object's `renderPDFForm` method and pass the following values:

- A string value that specifies the form design name, including the file name extension.
- A `com.adobe.idp.Document` object that contains data to merge with the form (created in step 2).
- A `PDFFormRenderSpec` object that stores run-time options. For more information, see AEM Forms API Reference.
- A `URLSpec` object that contains URI values that are required by the Forms service to render a form based on fragments.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm` method returns a `FormsResult` object that contains a form data stream that must be written to the client web browser.

10 Create a `com.adobe.idp.Document` object by invoking the `FormsResult` object 's `getOutputContent` method.

11 Get the content type of the `com.adobe.idp.Document` object by invoking its `getContentType` method.

12 Set the `javax.servlet.http.HttpServletResponse` object's content type by invoking its `setContentType` method and passing the content type of the `com.adobe.idp.Document` object.

13 Create a `javax.servlet.ServletOutputStream` object used to write the form data stream to the client web browser by invoking the `javax.servlet.http.HttpServletResponse` object's `getOutputStream` method.

14 Create a `java.io.InputStream` object by invoking the `com.adobe.idp.Document` object's `getInputStream` method.

15 Create a byte array populate it with the form data stream by invoking the `InputStream` object's `read` method and passing the byte array as an argument.

16 Invoke the `javax.servlet.ServletOutputStream` object's `write` method to send the form data stream to the client web browser. Pass the byte array to the `write` method.

The following code example represents the Java servlet that invokes the Forms service and renders a form based on fragments.

```
/*
    * This Java Quick Start uses the following JAR files
    * 1. adobe-forms-client.jar
    * 2. adobe-livecycle-client.jar
    * 3. adobe-usermanager-client.jar
    *
    *  (Because Forms quick starts are implemented as Java servlets, it is
    *  not necessary to include J2EE specific JAR files - the Java project
    *  that contains this quick start is exported as a WAR file which
    *  is deployed to the J2EE application server)
    *
    *  These JAR files are located in the following path:
    * <install directory>/sdk/client-libs
    *
    * For complete details about the location of these JAR files,
    * see "Including AEM Forms library files" in Programming with AEM forms
    */
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.adobe.livecycle.formsservice.client.*;
import java.util.*;
import java.io.InputStream;
import java.net.URL;

import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;

public class RenderFormFragment extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
            doPost(req,resp);

    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {



        try{
            //Set connection properties required to invoke AEM Forms
            Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_SOAP_ENDPOINT,
"http://[server]:[port]");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PROTOCOL,ServiceClien
tFactoryProperties.DSC_SOAP_PROTOCOL);
            connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_USERNAME,
"administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PASSWORD,
"password");

            //Get the value of selected radio button
            String radioValue = req.getParameter("radio");

            //Create an Document object to store form data
            Document oInputData = null;

            //The value of the radio button determines the form data to use
            //which determines which fragments used in the form
            if (radioValue.compareTo("AMERICAN") == 0){
                FileInputStream myData = new FileInputStream("C:\\Adobe\Purchase Order
US.xml");
                oInputData = new Document(myData);
```

```
            }
            else if (radioValue.compareTo("CANADIAN") == 0){
                FileInputStream myData = new FileInputStream("C:\\Adobe\Purchase Order
Canada.xml");
                oInputData = new Document(myData);
            }

            //Create a ServiceClientFactory object
            ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

            //Create a FormsServiceClient object
            FormsServiceClient formsClient = new FormsServiceClient(myFactory);

            //Set the parameter values for the renderPDFForm method
            String formName = "Applications/FormsApplication/1.0/FormsFolder/Purchase Order
Dynamic.xdp";

            //Cache the PDF form
            PDFFormRenderSpec pdfFormRenderSpec = new PDFFormRenderSpec();
            pdfFormRenderSpec.setCacheEnabled(new Boolean(true));

            //Specify URI values that are required to render a form
            //design based on fragments
            URLSpec uriValues = new URLSpec();
            uriValues.setApplicationWebRoot("http://[server]:[port]/RenderFormFragment");
            uriValues.setContentRootURI("repository:///");
            uriValues.setTargetURL("http://[server]:[port]/FormsServiceClientApp/HandleData");

            //Invoke the renderPDFForm method and write the
            //results to a client web browser
            FormsResult formOut = formsClient.renderPDFForm(
                        formName,               //formQuery
                        oInputData, //inDataDoc
                        pdfFormRenderSpec,      //PDFFormRenderSpec
                        uriValues,      //urlSpec
                        null                //attachments
                        );

            //Create a Document object that stores form data
            Document myData = formOut.getOutputContent();

            //Get the content type of the response and
            //set the HttpServletResponse object's content type
            String contentType = myData.getContentType();
            resp.setContentType(contentType);
```

```
            //Create a ServletOutputStream object
            ServletOutputStream oOutput = resp.getOutputStream();

            //Create an InputStream object
            InputStream inputStream = myData.getInputStream();

            //Write the data stream to the web browser
            byte[] data = new byte[4096];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(data)) > 0)
            {
                oOutput.write(data, 0, bytesRead);
            }

        }catch (Exception e) {
             System.out.println("The following exception occurred: "+e.getMessage());
      }
    }
}
```

Creating the web page

The index.html web page provides an entry point to the Java servlet and invokes the Forms service. This web page is a basic HTML form that contains two radio buttons and a submit button. The name of the radio buttons is radio. When the user clicks the submit button, form data is posted to the `RenderFormFragment` Java servlet.

The Java servlet captures the data that is posted from the HTML page by using the following Java code:

```
            Document oInputData = null;

            //Get the value of selected radio button
            String radioValue = req.getParameter("radio");

            //The value of the radio button determines the form data to use
            //which determines which fragments used in the form
            if (radioValue.compareTo("AMERICAN") == 0){
                FileInputStream myData = new FileInputStream("C:\\Adobe\Purchase Order US.xml");
                oInputData = new Document(myData);
            }
            else if (radioValue.compareTo("CANADIAN") == 0){
                FileInputStream myData = new FileInputStream("C:\\Adobe\Purchase Order
Canada.xml");
                oInputData = new Document(myData);
            }
```

The following HTML code is located in the index.html file that was created during setup of the development environment. (See “Creating a web project” on page 673.)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Untitled Document</title>
</head>

<body>
<form name="myform"
action="http://[server]:[port]/FragmentsWebApplication/RenderFormFragment" method="post">
     <table>
     <tr>
       <th width="344" scope="col">Forms Fragment Web Client</th>
     </tr>
     <tr>
       <td>
         <label>
         <input type="radio" name="radio" id="radio_Data" value="CANADIAN" />
         Canadian data<br />
         </label>
         <p>
           <label>
           <input type="radio" name="radio" id="radio_Data" value="AMERICAN" checked/>
           American data</label>
         </p>
       </td>
     </tr>
     <tr>
     <td>
       <label>
         <input type="submit" name="button_Submit" id="button_Submit" value="Submit" />
           </label>
           </td>
        </tr>
       </table>
     </form>
</body>
</html>
```

Packaging the web application

To deploy the Java servlet that invokes the Forms service, package your web application to a WAR file. Ensure that external JAR files that the component's business logic depends on, such as adobe-livecycle-client.jar and adobe-forms-client.jar, are also included in the WAR file.

To package a web application to a WAR file:

1. From the **Project Explorer** window, right-click the `FragmentsWebApplication` project and select **Export** > **WAR file**.
2. In the **Web module** text box, type `FragmentsWebApplication` for the name of the Java project.
3. In the **Destination** text box, type `FragmentsWebApplication.war`for thefile name, specify the location for your WAR file, and then click Finish.

Deploying the WAR file to the J2EE application server

You can deploy the WAR file to the J2EE application server on which AEM Forms is deployed. After the WAR file is deployed, you can access the HTML web page by using a web browser.

To deploy the WAR file to the J2EE application server:

- Copy the WAR file from the export path to *[Forms Install]*\Adobe\Adobe Experience Manager Forms\jboss\server\all\deploy.

Testing your web application

After you deploy the web application, you can test it by using a web browser. Assuming that you are using the same computer that is hosting AEM Forms, you can specify the following URL:

- http://localhost:8080/FragmentsWebApplication/index.html

  Select a radio button and click the Submit button. A Form that is based on fragments will appear in the web browser. If problems occur, see the J2EE application server's log file.

Creating Document Output Streams

About the Output Service

The Output service lets you output documents as PDF (including PDF/A documents), PostScript, Printer Control Language (PCL), and the following label formats:

- Zebra - ZPL
- Intermec - IPL
- Datamax - DPL
- TecToshiba - TPCL

Using the Output service, you can merge XML form data with a form design and output the document to a network printer or file.

There are two ways in which you can pass a form design (an XDP file) to the Output service. You can either pass a `com.adobe.idp.Document` instance that contains a form design to the Output service. Or you can pass a URI value that specifies the location of the form design. Both of these ways are discussed in *Programming with AEM forms*.

Note: *The Output service does not support Acroform PDF documents that contain application object specific scripts. Acroform PDF documents that contain application object specific scripts are not rendered.*

The following sections show how to pass a form design to the Output service using a URI value:

- "Creating PDF Documents" on page 681
- "Creating PDF/A Documents" on page 690

The following sections show how to pass a form design within a `com.adobe.idp.Document` instance:

- "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696
- "Creating PDF Documents Using Fragments" on page 703

One consideration when deciding which technique to use is if you are getting the form design from another AEM Forms service, then pass it within a `com.adobe.idp.Document` instance. Both the *Passing Documents to the Output Service* and *Creating PDF Documents using Fragments* sections show how to get a form design from another AEM Forms service. The first section retrieves the form design from Content Services (deprecated). The second section retrieves the form design from the Assembler service.

If you are getting the form design from a fixed location, such as the file system, then you can use either technique. That is, you can specify the URI value to a XDP file or use a `com.adobe.idp.Document` instance.

To pass a URI value that specifies the location of the form design when creating a PDF document, use the `generatePDFOutput` method. Likewise, to pass a `com.adobe.idp.Document` instance to the Output service when creating a PDF document, use the `generatePDFOutput2` method.

When sending an output stream to a network printer, you can also use either technique. To send an output stream to a printer by passing a `com.adobe.idp.Document` instance that contains a form design, use the `sendToPrinter2` method. To send an output stream to a printer by passing a URI value, use the `sendToPrinter` method. The *Sending Print Streams to Printers* section uses the `sendToPrinter` method.

You can accomplish these tasks by using the Output service:

- "Creating PDF Documents" on page 681
- "Creating PDF/A Documents" on page 690
- "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696
- "Creating PDF Documents Using Fragments" on page 703
- "Printing to Files" on page 708
- "Sending Print Streams to Printers" on page 713
- "Creating Multiple Output Files" on page 718
- "Creating Search Rules" on page 726
- "Flattening PDF Documents" on page 732

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Creating PDF Documents

You can use the Output service to create a PDF document that is based on a form design and XML form data that you provide. The PDF document that is created by the Output service is not an interactive PDF document; a user cannot enter or modify form data.

If you want to create a PDF document meant for long-term storage, it is recommended that you create a PDF/A document. (See "Creating PDF/A Documents" on page 690.)

To create an interactive PDF form that lets a user enter data, use the Forms service. (See "Rendering Interactive PDF Forms" on page 582.)

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To create a PDF document, perform the following steps:

1. Include project files.
2. Create an Output Client object.
3. Reference an XML data source.

4 Set PDF run-time options.

5 Set rendering run-time options.

6 Generate a PDF document.

7 Retrieve the results of the operation.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you will need to replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

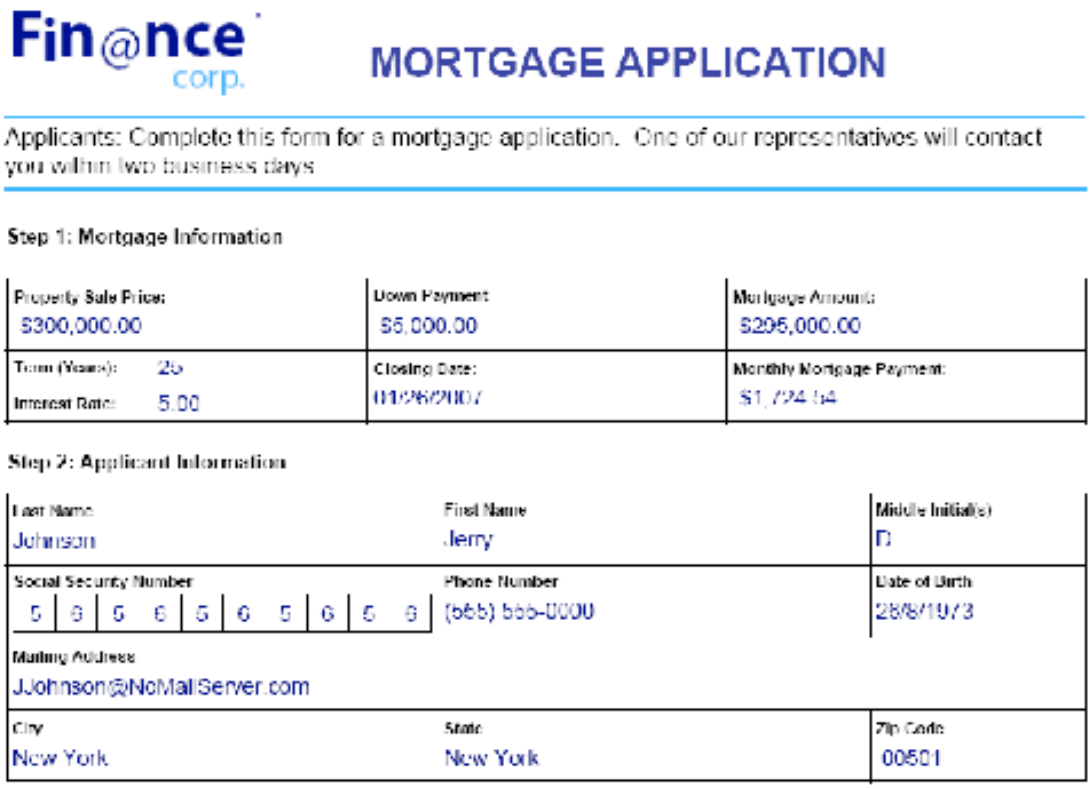
Create an Output Client object

Before you can programmatically perform an Output service operation, you must create an Output service client object. If you are using the Java API, create an `OutputClient` object. If you are using the Output web service API, create an `OutputServiceService` object.

Reference an XML data source

To merge data with the form design, you must reference an XML data source that contains data. An XML element must exist for every form field that you plan to populate with data. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed if all XML elements are specified.

Consider the following example loan application form.



Fin@nce corp.

MORTGAGE APPLICATION

Applicants: Complete this form for a mortgage application. One of our representatives will contact you within two business days

Step 1: Mortgage Information

Property Sale Price:	Down Payment:	Mortgage Amount:
$300,000.00	$5,000.00	$295,000.00
Term (Years): 25 Interest Rate: 5.00	Closing Date:	Monthly Mortgage Payment:

Step 2: Applicant Information

Last Name:	First Name:	Middle Initial(s):
Johnson	Jerry	D
Social Security Number: 5 6 5 6 5 6 5 6 5 6	Phone Number: (555) 555-0000	Date of Birth: 28/8/1973
Mailing Address: JJohnson@NoMailServer.com		
City: New York	State: New York	Zip Code: 00501

To merge data into this form design, you must create an XML data source that corresponds to the form. The following XML represents an XDP XML data source that corresponds to the example mortgage application form.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
- <xfa:data>
- <data>
    - <Layer>
        <closeDate>1/26/2007</closeDate>
        <lastName>Johnson</lastName>
        <firstName>Jerry</firstName>
        <mailingAddress>JJohnson@NoMailServer.com</mailingAddress>
        <city>New York</city>
        <zipCode>00501</zipCode>
        <state>NY</state>
        <dateBirth>26/08/1973</dateBirth>
        <middleInitials>D</middleInitials>
        <socialSecurityNumber>(555) 555-5555</socialSecurityNumber>
        <phoneNumber>5555550000</phoneNumber>
    </Layer>
    - <Mortgage>
        <mortgageAmount>295000.00</mortgageAmount>
        <monthlyMortgagePayment>1724.54</monthlyMortgagePayment>
        <purchasePrice>300000</purchasePrice>
        <downPayment>5000</downPayment>
        <term>25</term>
        <interestRate>5.00</interestRate>
    </Mortgage>
</data>
</xfa:data>
</xfa:datasets>
```

Set PDF run-time options

Set the file URI option when creating a PDF document. This option specifies the name and location of the PDF file that the Output service generates.

Note: *Instead of setting the file URI run-time option, you can programmatically retrieve the PDF document from the complex data type that is returned by the Output service. However, by setting the file URI run-time option, you do not need to create application logic that programmatically retrieves the PDF document.*

Set rendering run-time options

You can set rendering run-time options when creating a PDF document. Although these options are not required (unlike PDF run-time options that are required), you can perform tasks such as improving the performance of the Output service. For example, you can cache the form design that the Output service uses in order to improve its performance.

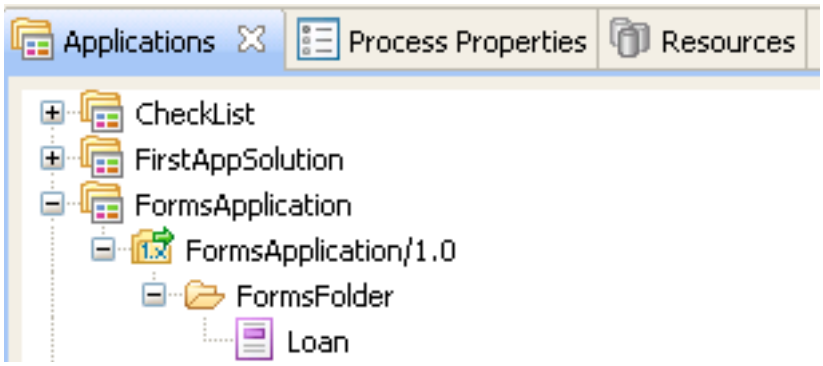
If you use a tagged Acrobat form as input, you cannot use the Output service Java or web service API to turn off the tagged setting. If you attempt to programmatically set this option to `false`, the result PDF document is still tagged.

Note: *If you do not specify rendering run-time options, then default values are used. For information about rendering run-time options, see the* `RenderOptionsSpec` *class reference. (See AEM Forms API Reference).*

Generate a PDF document

After you reference a valid XML data source that contains form data and you set run-time options, you can invoke the Output service, which results in it generating a PDF document.

When generating a PDF document, you specify URI values that are required by the Output service to create a PDF document. A form design can be stored in locations such as the server file system or as part of an AEM Forms application. A form design (or other resources such as an image file) that exists as part of a Forms application can be referenced by using the content root URI value `repository:///`. For example, consider the following form design named *Loan.xdp* located within a Forms application named *Applications/FormsApplication*:



To access the Loan.xdp file shown in the previous illustration, specify `repository:///Applications/FormsApplication/1.0/FormsFolder/` as the third parameter passed to the `OutputClient` object's `generatePDFOutput` method. Specify the form name (*Loan.xdp*) as the second parameter passed to the `OutputClient` object's `generatePDFOutput` method.

If the XDP file contains images (or other resources such as fragments), place the resources in the same application folder as the XDP file. AEM Forms uses the content root URI as the base path to resolve references to images. For example, if the Loan.xdp file contains an image, ensure that you place the image in `Applications/FormsApplication/1.0/FormsFolder/`.

Note: *You can reference a Forms application URI when invoking the `OutputClient` object's `generatePDFOutput` or `generatePrintedOutput` methods.*

Note: *To see a complete quick start that creates a PDF document by referencing a XDP located in a Forms application, see "Quick Start (SOAP mode): Creating a PDF document based on an application XDP file using the Java API" on page 242.*

Retrieve the results of the operation

After the Output service performs an operation, it returns various data items such as status XML data that specifies whether the operation was successful.

See also

"Create a PDF document using the Java API" on page 686

"Create a PDF document using the web service API" on page 687

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Create a PDF document using the Java API

Create a PDF document by using the Output API (Java):

1 Include project files.

Include client JAR files, such as adobe-output-client.jar, in your Java project's class path.

2 Create an Output Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an XML data source.

- Create a `java.io.FileInputStream` object that represents the XML data source that is used to populate the PDF document by using its constructor and passing a string value that specifies the location of the XML file.
- Create a `com.adobe.idp.Document` object by using its constructor. Pass the `java.io.FileInputStream` object.

4 Set PDF run-time options.

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Set the File URI option by invoking the `PDFOutputOptionsSpec` object's `setFileURI` method. Pass a string value that specifies the location of the PDF file that the Output service generates. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.

5 Set rendering run-time options.

- Create a `RenderOptionsSpec` object by using its constructor.
- Cache the form design to improve the performance of the Output service by invoking the `RenderOptionsSpec` object's `setCacheEnabled` and passing `true`.

Note: *You cannot set the version of the PDF document by using the `RenderOptionsSpec` object's `setPdfVersion` method if the input document is an Acrobat form (a form created in Acrobat) or an XFA document that is signed or certified. The output PDF document retains the original PDF version. Likewise, you cannot set the tagged Adobe PDF option by invoking the `RenderOptionsSpec` object's `setTaggedPDF` method if the input document is an Acrobat form or a signed or certified XFA document.*

Note: *You cannot set the linearized PDF option by using the `RenderOptionsSpec` object's `setLinearizedPDF` method if the input PDF document is certified or digitally signed. (See "Digitally Signing PDF Documents" on page 892.)*

6 Generate a PDF document.

Create a PDF document by invoking the `OutputClient` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contains the XML data source that contains data to merge with the form design.

The `generatePDFOutput` method returns an `OutputResult` object that contains the results of the operation.

Important: *When generating a PDF document by invoking the `generatePDFOutput` method, be aware that you cannot merge data with an XFA PDF form that is signed or certified. (See "Digitally Signing and Certifying Documents" on page 879.)*

Note: *The `OutputResult` object's `getRecordLevelMetaDataList` method returns `null`.*

Note: *You can also create a PDF document by invoking the `OutputClient` object's `generatePDFOutput2` method. (See "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696.)*

7 .

- Retrieve a `com.adobe.idp.Document` object that represents the status of the `generatePDFOutput` operation by invoking the `OutputResult` object's `getStatusDoc` method. This method returns status XML data that specifies whether the operation was successful.
- Create a `java.io.File` object that contains the results of the operation. Ensure that the file name extension is .xml.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `getStatusDoc` method).

Although the Output service writes the PDF document to the location specified by the argument that is passed to the `PDFOutputOptionsSpec` object's `setFileURI` method, you can programmatically retrieve the PDF/A document by invoking the `OutputResult` object's `getGeneratedDoc` method.

See also

"Summary of steps" on page 681

"Quick Start (SOAP mode): Creating a PDF document using the Java API" on page 240

"Quick Start (SOAP mode): Creating a PDF document using the Java API" on page 248

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Create a PDF document using the web service API

Create a PDF document by using the Output API (web service):

1 .

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 .

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 .

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store XML data that will be merged with the PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the XML file that contains form data.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 .

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Set the File URI option by assigning a string value that specifies the location of the PDF file that the Output service generates to the `PDFOutputOptionsSpec` object's `fileURI` data member. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.

5 .

- Create a `RenderOptionsSpec` object by using its constructor.
- Cache the form design to improve the performance of the Output service by assigning the value `true` to the `RenderOptionsSpec` object's `cacheEnabled` data member.

Note: *You cannot set the version of the PDF document by using the `RenderOptionsSpec` object's `setPdfVersion` method if the input document is an Acrobat form (a form created in Acrobat) or an XFA document that is signed or certified. The output PDF document retains the original PDF version. Likewise, you cannot set the tagged Adobe PDF option by invoking the `RenderOptionsSpec` object's `setTaggedPDF` method if the input document is an Acrobat form or a signed or certified XFA document.*

Note: *You cannot set the linearized PDF option by using the `RenderOptionsSpec` object's `linearizedPDF` member if the input PDF document is certified or digitally signed. (See "Digitally Signing PDF Documents" on page 892.)*

6 .

Create a PDF document by invoking the `OutputServiceService` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `BLOB` object that contains the XML data source that contains data to merge with the form design.
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with generated metadata that describes the document. (This parameter value is required only for web service invocation).
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with result data. (This parameter value is required only for web service invocation).
- An `OutputResult` object that contains the results of the operation. (This parameter value is required only for web service invocation).

Important: *When generating a PDF document by invoking the `generatePDFOutput` method, be aware that you cannot merge data with an XFA PDF form that is signed or certified. (See "Digitally Signing and Certifying Documents" on page 879.)*

Note: *You can also create a PDF document by invoking the `OutputClient` object's `generatePDFOutput2` method. (See "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696.)*

7 .

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents an XML file location that contains result data. Ensure that the file name extension is .xml.
- Create a byte array that stores the data content of the `BLOB` object that was populated with result data by the `OutputServiceService` object's `generatePDFOutput` method (the eighth parameter). Populate the byte array by getting the value of the `BLOB` object's `MTOMfield`.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to the XML file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 681

Quick Start (MTOM): Creating a PDF document using the web service API

Quick Start (SwaRef): Creating a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Note: *The `OutputServiceService` object's `generateOutput` method is deprecated.*

Creating PDF/A Documents

You can use the Output service to create a PDF/A document. Because PDF/A is an archival format for long-term preservation of the document's content, all fonts are embedded and the file is uncompressed. As a result, a PDF/A document is typically larger than a standard PDF document. In addition, a PDF/A document does not contain audio and video content. Like other Output service tasks, you provide both a form design and data to merge with a form design to create a PDF/A document.

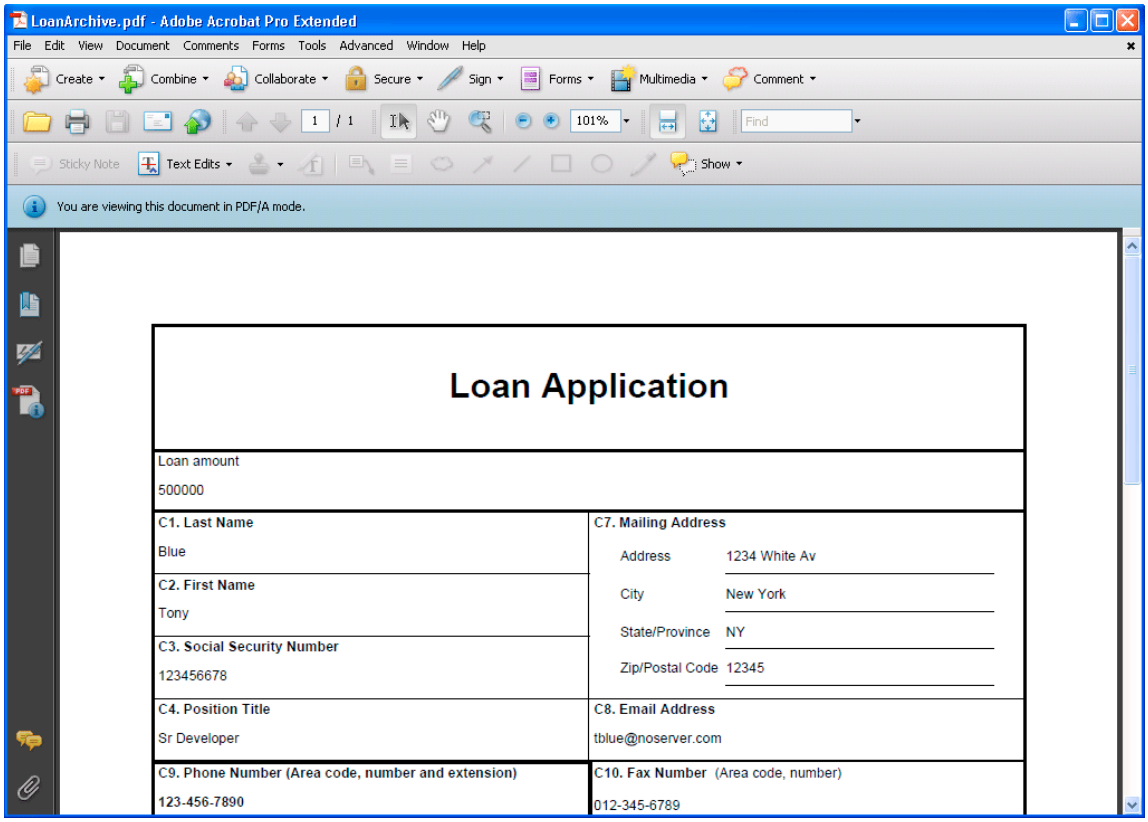
The PDF/A-1 specification consists of two levels of conformance, namely a and b. The major difference between the two is regarding the logical structure (accessibility) support, which is not required for conformance level b. Regardless of the conformance level, PDF/A-1 dictates that all fonts are embedded in the generated PDF/A document.

Although PDF/A is the standard for archiving PDF documents, it is not mandatory that PDF/A be used for archiving if a standard PDF document meets your company's needs. The purpose of the PDF/A standard is to establish a PDF file that can be stored for a long period of time as well as meet document preservation requirements. For example, a URL cannot be embedded in a PDF/A because over time the URL may become invalid.

Your organization must assess its own needs, the length of time you intend to keep the document, file size considerations, and determine your own archiving strategy. You can programmatically determine if a PDF document is PDF/A compliant by using the DocConverter service. (See "Programmatically Determining PDF/A Compliancy" on page 994.)

A PDF/A document must use the font that is specified in the form design and fonts cannot be substituted. As a result, if a font that is located within a PDF document is not available on the host operating system (OS), then an exception occurs.

When a PDF/A document is opened in Acrobat, a message is displayed that confirms that the document is a PDF/A document, as shown in the following illustration.



Note: *The AIIM web site has a PDF/A FAQ section that you can access at http://www.aiim.org/documents/standards/19005-1_FAQ.pdf.*

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To create a PDF/A document, perform the following steps:

1. Include project files.
2. Create an Output Client object.
3. Reference an XML data source.
4. Set PDF/A run-time options.
5. Set rendering run-time options.
6. Generate a PDF/A document.
7. Retrieve the results of the operation.

Include project files

Include necessary files in your development project. If you are creating a custom application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you will need to replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

Create an Output Client object

Before you can programmatically perform an Output service operation, you must create an Output service client object. If you are using the Java API, create an `OutputClient` object. If you are using the Output web service API, create an `OutputServiceService` object.

Reference an XML data source

To merge data with the form design, you must reference an XML data source that contains data. An XML element must exist for every form field that you want to populate with data. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed if all XML elements are specified.

Set PDF/A run-time options

You can set the File URI option when creating a PDF/A document. The URI is relative to the J2EE application server hosting AEM Forms. That is, if you set C:\Adobe, the file is written to the folder on the server, not the client computer. The URI specifies the name and location of the PDF/A file that the Output service generates.

Set rendering run-time options

You can set rendering run-time options when creating PDF/A documents. Two PDF/A related options that you can set are the `PDFAConformance` and `PDFARevisionNumber` values. The `PDFAConformance` value refers to how a PDF document adheres to requirements that specify how long-term electronic documents are preserved. Valid values for this option are `A` and `B`. For information about level a and b conformance, see the PDF/A-1 ISO specification that is titled *ISO 19005-1 Document management*.

The `PDFARevisionNumber` value refers to the revision number of a PDF/A document. For information about the revision number of a PDF/A document, see the PDF/A-1 ISO specification that is titled *ISO 19005-1 Document management*.

Note: *You cannot set the tagged Adobe PDF option to `false` when creating a PDF/A 1A document. PDF/A 1A will always be a tagged PDF document. Also, you cannot set the tagged Adobe PDF option to `true` when creating a PDF/A 1B document. PDF/A 1B will always be an untagged PDF document.*

Generate a PDF/A document

After you reference a valid XML data source that contains form data and you set run-time options, you can invoke the Output service, causing it to generate a PDF/A document.

Retrieve the results of the operation

After the Output service performs an operation, it returns various data items such as XML data that specifies whether the operation was successful.

See also

"Create a PDF/A document using the Java API" on page 692

"Create a PDF/A document using the web service API" on page 694

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Create a PDF/A document using the Java API

Create a PDF/A document by using the Output API (Java):

1 Include project files.

   Include client JAR files, such as adobe-output-client.jar, in your Java project's class path.

2 Create an Output Client object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an XML data source.

   - Create a `java.io.FileInputStream` object that represents the XML data source that is used to populate the PDF/A document by using its constructor and passing a string value that specifies the location of the XML file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set PDF/A run-time options.

   - Create a `PDFOutputOptionsSpec` object by using its constructor.

- Set the File URI option by invoking the `PDFOutputOptionsSpec` object's `setFileURI` method. Pass a string value that specifies the location of the PDF file that the Output service generates. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.

5 Set rendering run-time options.

- Create a `RenderOptionsSpec` object by using its constructor.
- Set the `PDFAConformance` value by invoking the `RenderOptionsSpec` object's `setPDFAConformance` method and passing a `PDFAConformance` enum value that specifies the conformance level. For example, to specify conformance level A, pass `PDFAConformance.A`.
- Set the `PDFARevisionNumber` value by invoking the `RenderOptionsSpec` object's `setPDFARevisionNumber` method and passing `PDFARevisionNumber.Revision_1`.

***Note:** The PDF version of a PDF/A document is 1.4 regardless of which value you specify for the `RenderOptionsSpec` object's `setPdfVersion`method.*

6 Generate a PDF/A document.

Create a PDF/A document by invoking the `OutputClient` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enumeration value. To generate a PDF/A document, specify `TransformationFormat.PDFA`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contain the XML data source that contains data to merge with the form design.

The `generatePDFOutput` method returns an `OutputResult` object that contains the results of the operation.

***Note:** The `OutputResult` object's `getRecordLevelMetaDataList` method returns `null`.*

***Note:** You can also create a PDF /A document by invoking the `OutputClient` object's `generatePDFOutput2` method. (See "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696.)*

7 Retrieve the results of the operation.

- Create a `com.adobe.idp.Document` object that represents the status of the `generatePDFOutput` method by invoking the `OutputResult` object's `getStatusDoc` method.
- Create a `java.io.File` object that will contain the results of the operation. Ensure that the file name extension is .xml.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `getStatusDoc` method).

***Note:** Although the Output service writes the PDF/A document to the location specified by the argument that is passed to the `PDFOutputOptionsSpec` object's `setFileURI` method, you can programmatically retrieve the PDF/A document by invoking the `OutputResult` object's `getGeneratedDoc` method.*

See also

"Summary of steps" on page 691

Quick Start (SOAP mode): Creating a PDF/A document using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Create a PDF/A document using the web service API

Create a PDF/A document by using the Output API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create an Output Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an XML data source.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store data that will be merged with the PDF/A document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to encrypt and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the byte array contents.

4 Set PDF/A run-time options.

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Set the File URI option by assigning a string value that specifies the location of the PDF file that the Output service generates to the `PDFOutputOptionsSpec` object's `fileURI` data member. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer

5 Set rendering run-time options.

- Create a `RenderOptionsSpec` object by using its constructor.
- Set the `PDFAConformance` value by assigning a `PDFAConformance` enum value to the `RenderOptionsSpec` object's `PDFAConformance` data member. For example, to specify conformance level A, assign `PDFAConformance.A` to this data member.
- Set the `PDFARevisionNumber` value by assigning a `PDFARevisionNumber` enum value to the `RenderOptionsSpec` object's `PDFARevisionNumber` data member. Assign `PDFARevisionNumber.Revision_1` to this data member.

Note: *The PDF version of a PDF/A document is 1.4 regardless of which value you specify.*

6 Generate a PDF/A document.

Create a PDF document by invoking the `OutputServiceService` object's `generatePDFOutput` method and passing the following values:

- A TransformationFormat enumeration value. To generate a PDF document, specify `TransformationFormat.PDFA`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `BLOB` object that contains the XML data source that contains data to merge with the form design.
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with generated metadata that describes the document. (This parameter value is required for web service invocation only.)
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with result data. (This parameter value is required for web service invocation only.)
- An `OutputResult` object that contains the results of the operation. (This parameter value is required for web service invocation only.)

Note: *You can also create a PDF /A document by invoking the `OutputClient` object's `generatePDFOutput2` method. (See "Passing Documents located in Content Services (deprecated) to the Output Service" on page 696.)*

7 Retrieve the results of the operation.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents an XML file location that contains result data. Ensure that the file name extension is .xml.
- Create a byte array that stores the data content of the `BLOB` object that was populated with result data by the `OutputServiceService` object's `generatePDFOutput` method (the eighth parameter). Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

- Write the contents of the byte array to the XML file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 691

Quick Start (MTOM): Creating a PDF/A document using the web service API

Quick Start (SwaRef): Creating a PDF/A document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Passing Documents located in Content Services (deprecated) to the Output Service

The Output service renders a non-interactive PDF form that is based on a form design that is typically saved as an XDP file and created in Designer. You can pass a `com.adobe.idp.Document` object that contains the form design to the Output service. The Output service then renders the form design located in the `com.adobe.idp.Document` object.

An advantage of passing a `com.adobe.idp.Document` object to the Output service is that other AEM Forms service operations return a `com.adobe.idp.Document` instance. That is, you can get a `com.adobe.idp.Document` instance from another service operation and render it. For example, assume that an XDP file is stored in a Content Services (deprecated) node named `/Company Home/Form Designs`, as shown in the following illustration.

You can programmatically retrieve Loan.xdp from Content Services (deprecated) and pass the XDP file to the Output service within a `com.adobe.idp.Document` object.

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To pass a document obtained from Content Services (deprecated) to the Output service, perform the following tasks:

1. Include project files.
2. Create a Output and a Document Management Client API object.
3. Retrieve the form design from Content Services (deprecated).
4. Render the non-interactive PDF form.
5. Perform an action with the data stream.

Include project files

Include the necessary files to your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create an Output and a Document Management Client API object

Before you can programmatically perform a Output service API operation, create a Output Client API object. Also, because this workflow retrieves an XDP file from Content Services (deprecated), create a Document Management API object.

Retrieve the form design from Content Services (deprecated)

Retrieve the XDP file from Content Services (deprecated) by using the Java or web service API. The XDP file is returned within a `com.adobe.idp.Document` instance (or a `BLOB` instance if you are using web services). You can then pass the `com.adobe.idp.Document` instance to the Output service.

Render the non-interactive PDF form

To render a non-interactive form, pass the `com.adobe.idp.Document` instance that was returned from Content Services (deprecated) to the Output service.

Note: *Two new methods named `generatePDFOutput2` and `generatePrintedOutput2` accept a `com.adobe.idp.Document` object that contains a form design. You can also pass a `com.adobe.idp.Document` that contains the form design to the Output service when sending a print stream to a network printer.*

Perform an action with the form data stream

You can save the non-interactive form as a PDF file. The form can be viewed in Adobe Reader or Acrobat.

See also

"Pass documents to the Output Service using the Java API" on page 697

"Pass documents to the Output Service using the web service API" on page 699

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

"Creating PDF Documents Using Fragments" on page 703

Pass documents to the Output Service using the Java API

Pass a document retrieved from Content Services (deprecated) by using the Output service and Content Services (deprecated) API (Java):

1 Include project files.

   Include client JAR files, such as adobe-output-client.jar and adobe-contentservices-client.jar, in your Java project's class path.

2 Create an Output and a Document Management Client API object.

   - Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
   - Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.
   - Create a `DocumentManagementServiceClientImpl` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the form design from Content Services (deprecated).

   Invoke the `DocumentManagementServiceClientImpl` object's `retrieveContent` method and pass the following values:

   - A string value that specifies the store where the content is added. The default store is `SpacesStore`. This value is a mandatory parameter.
   - A string value that specifies the fully qualified path of the content to retrieve (for example, `/Company Home/Form Designs/Loan.xdp`). This value is a mandatory parameter.

- A string value that specifies the version. This value is an optional parameter, and you can pass an empty string. In this situation, the latest version is retrieved.

The `retrieveContent` method returns a `CRCResult` object that contains the XDP file. Retrieve a `com.adobe.idp.Document` instance by invoking the `CRCResult` object's `getDocument` method.

4 Render the non-interactive PDF form.

Invoke the `OutputClient` object's `generatePDFOutput2` method and pass the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the content root where the additional resources such as images are located.
- A `com.adobe.idp.Document` object that represents the form design (use the instance returned by the `CRCResult` object's `getDocument` method).
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contains the XML data source that contains data to merge with the form design.

The `generatePDFOutput2` method returns an `OutputResult` object that contains the results of the operation.

5 Perform an action with the form data stream.

- Retrieve a `com.adobe.idp.Document` object that represents the non-interactive form by invoking the `OutputResult` object's `getGeneratedDoc` method.
- Create a `java.io.File` object that contains the results of the operation. Ensure that the file name extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `getGeneratedDoc` method).

See also

"Summary of steps" on page 696

"Quick Start (SOAP mode): Passing documents to the Output Service using the Java API" on page 252

Quick Start (SOAP mode): Passing documents to the Output Service using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Pass documents to the Output Service using the web service API

Pass a document retrieved from Content Services (deprecated) by using the Output service and Content Services (deprecated) API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Because this client application invokes two AEM Forms services, create two service references. Use the following WSDL definition for the service reference associated with the Output service: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Use the following WSDL definition for the service reference associated with the Document Management service: `http://localhost:8080/soap/services/DocumentManagementService?WSDL&lc_version=9.0.1`.

Because the `BLOB` data type is common to both service references, fully qualify the `BLOB` data type when using it. In the corresponding web service quick start, all `BLOB` instances are fully qualified.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Output and a Document Management Client API object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

Note: *Repeat these steps for the `DocumentManagementServiceClient` service client.*

3 Retrieve the form design from Content Services (deprecated).

Retrieve content by invoking the `DocumentManagementServiceClient` object's `retrieveContent` method and passing the following values:

- A string value that specifies the store where the content is added. The default store is `SpacesStore`. This value is a mandatory parameter.
- A string value that specifies the fully qualified path of the content to retrieve (for example, `/Company Home/Form Designs/Loan.xdp`). This value is a mandatory parameter.
- A string value that specifies the version. This value is an optional parameter, and you can pass an empty string. In this situation, the latest version is retrieved.

- A string output parameter that stores the browse link value.
- A `BLOB` output parameter that stores the content. You can use this output parameter to retrieve the content.
- A `ServiceReference1.MyMapOf_xsd_string_To_xsd_anyType` output parameter that stores content attributes.
- A `CRCResult` output parameter. Instead of using this object, you can use the `BLOB` output parameter to retrieve the content.

4 Render the non-interactive PDF form.

Invoke the `OutputServiceClient` object's `generatePDFOutput2` method and pass the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the content root where the additional resources such as images are located.
- A `BLOB` object that represents the form design (use the `BLOB` instance returned by Content Services (deprecated)).
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `BLOB` object that contains the XML data source that contains data to merge with the form design.
- An output `BLOB` object that is populated by the `generatePDFOutput2` method. The `generatePDFOutput2` method populates this object with generated metadata that describes the document. (This parameter value is required only for web service invocation).
- An output `OutputResult` object that contains the results of the operation. (This parameter value is required only for web service invocation).

The `generatePDFOutput2` method returns a `BLOB` object that contains the non-interactive PDF form.

5 Perform an action with the form data stream.

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the interactive PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object retrieved from the `generatePDFOutput2` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 696

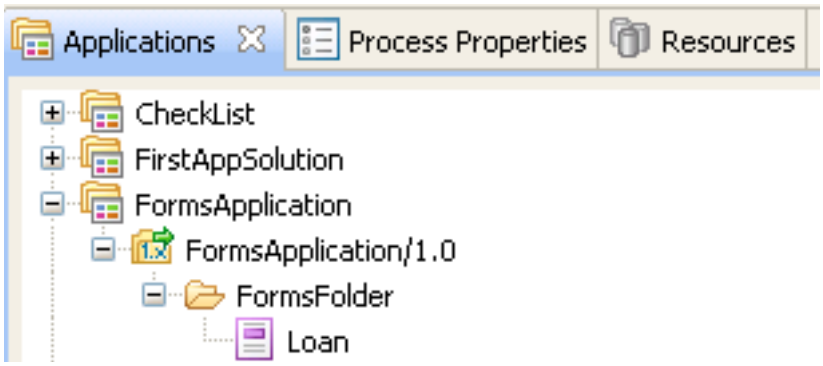
"Invoking AEM Forms using MTOM" on page 529

Quick Start (MTOM): Passing documents to the Output Service using the web service API

Passing Documents located in the Repository to the Output Service

The Output service renders a non-interactive PDF form that is based on a form design that is typically saved as an XDP file and created in Designer. You can pass a `com.adobe.idp.Document` object that contains the form design to the Output service. The Output service then renders the form design located in the `com.adobe.idp.Document` object.

An advantage of passing a `com.adobe.idp.Document` object to the Output service is that other AEM Forms service operations return a `com.adobe.idp.Document` instance. That is, you can get a `com.adobe.idp.Document` instance from another service operation and render it. For example, assume that an XDP file is stored in the AEM Forms repository, as shown in the following illustration.



The *FormsFolder* folder is a user-defined location in the AEM Forms repository (this location is an example and does not exist by default). In this example, a form design named Loan.xdp is located in this folder. In addition to the form design, other form collateral such as images can be stored in this location. The path to a resource located in the AEM Forms repository is:

```
Applications/Application-name/Application-version/Folder.../Filename
```

You can programmatically retrieve Loan.xdp from the AEM Forms repository and pass it to the Output service within a `com.adobe.idp.Document` object.

You can create a PDF based on an XDP file located in the repository using one of two ways. You can pass the XDP loction by reference or you can programmatically retrieve the XDP from the repository and pass it to the Output service within an XDP file.

"Quick Start (SOAP mode): Creating a PDF document based on an application XDP file using the Java API" on page 242 (shows how to pass the location of the XDP file by reference).

"Quick Start (SOAP mode): Passing a document located in the Repository to the Output service using the Java API" on page 245 (shows how to programmatically retrieve the XDP file from the AEM Forms Repository and pass it to the Output service within a `com.adobe.idp.Document` instance). (This section discusses how to perform this task)

Note: *For more information about the Forms service, see Services Reference for AEM Forms.*

Summary of steps

To pass a document obtained from the AEM Forms repository to the Output service, perform the following tasks:

1. Include project files.
2. Create a Output and a Document Management Client API object.
3. Retrieve the form design from the AEM Forms repository.
4. Render the non-interactive PDF form.
5. Perform an action with the data stream.

Include project files

Include the necessary files to your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create an Output and a Document Management Client API object

Before you can programmatically perform a Output service API operation, create a Output Client API object. Also, because this workflow retrieves an XDP file from Content Services (deprecated), create a Document Management API object.

Retrieve the form design from the AEM Forms Repository

Retrieve the XDP file from the AEM Forms Repository by using the Repository API. (See "Reading Resources" on page 1043.)

The XDP file is returned within a `com.adobe.idp.Document` instance (or a `BLOB` instance if you are using web services). You can then pass the `com.adobe.idp.Document` instance o the Output service.

Render the non-interactive PDF form

To render a non-interactive form, pass the `com.adobe.idp.Document` instance that was returned using the AEM Forms Repository API.

Note: *Two new methods named `generatePDFOutput2` and `generatePrintedOutput2` accept a `com.adobe.idp.Document` object that contains a form design. You can also pass a `com.adobe.idp.Document` that contains the form design to the Output service when sending a print stream to a network printer.*

Perform an action with the form data stream

You can save the non-interactive form as a PDF file. The form can be viewed in Adobe Reader or Acrobat.

See also

"Pass documents located in the Repository to the Output Service using the Java API" on page 702

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Retrieving Content from Content Services (deprecated)

ResourceRepositoryClient

Pass documents located in the Repository to the Output Service using the Java API

Pass a document retrieved from the Repository by using the Output service and Repository API (Java):

1 Include project files.

   Include client JAR files, such as adobe-output-client.jar and adobe-repository-client.jar, in your Java project's class path.

2 Create an Output and a Document Management Client API object.

   - Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
   - Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.
   - Create a `DocumentManagementServiceClientImpl` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the form design from the AEM Forms Repository.

Invoke the `ResourceRepositoryClient` object's `readResourceContent` method and pass a string value that specifies the URI location to the XDP file. For example, `/Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`. This value is a mandatory. This method returns a `com.adobe.idp.Document` instance that represents the XDP file.

4 Render the non-interactive PDF form.

Invoke the `OutputClient` object's `generatePDFOutput2` method and pass the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the content root where the additional resources such as images are located. For example, `repository:///Applications/FormsApplication/1.0/FormsFolder/`.
- A `com.adobe.idp.Document` object that represents the form design (use the instance returned by the `ResourceRepositoryClient` object's `readResourceContent` method).
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contains the XML data source that contains data to merge with the form design.

The `generatePDFOutput2` method returns an `OutputResult` object that contains the results of the operation.

5 Perform an action with the form data stream.

- Retrieve a `com.adobe.idp.Document` object that represents the non-interactive form by invoking the `OutputResult` object's `getGeneratedDoc` method.
- Create a `java.io.File` object that contains the results of the operation. Ensure that the file name extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `getGeneratedDoc` method).

See also

"Summary of steps" on page 701

"Quick Start (SOAP mode): Passing a document located in the Repository to the Output service using the Java API" on page 245

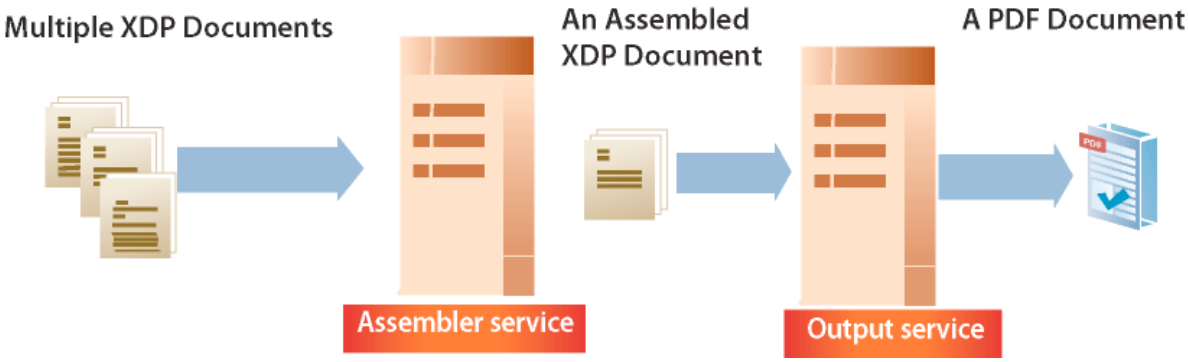
"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Creating PDF Documents Using Fragments

You can use the Output and Assembler services to create an output stream, such as a PDF document, that is based on fragments. The Assembler service assembles an XDP document that is based on fragments located in multiple XDP files. The assembled XDP document is passed to the Output service, which creates a PDF document. Although this workflow shows a PDF document being generated, the Output service can generate other output types, such as ZPL, for this workflow. A PDF document is used for discussion purposes only.

The following illustration shows this workflow.



Before reading *Creating PDF Documents using Fragments*, it is recommended that you become familiar with using the Assembler service to assemble multiple XDP documents. (See "Assembling Multiple XDP Fragments" on page 983.)

Note: *You can also pass a form design assembled by the Assembler service to the Forms service instead of the Output service. The primary difference between the Output service and Forms service is that the Forms service generates interactive PDF documents and the Output service produces non-interactive PDF documents. Also the Forms service cannot generate printer-based output streams like ZPL.*

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To create a PDF document based on fragments, perform the following steps:

1. Include project files.
2. Create an Output and Assembler Client object.
3. Use the Assembler service to generate the form design.
4. Use the Output service to generate the PDF document.
5. Save the PDF document as a PDF file.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create an Output and Assembler Client object

Before you can programmatically perform an Output service API operation, create an Output Client API object. Also, because this workflow invokes the Assembler service to create the form design, create an Assembler Client API object.

Use the Assembler service to generate the form design

Use the Assembler service to generate the form design using fragments. The Assembler service returns a `com.adobe.idp.Document` instance that contains the form design.

Use the Output service to generate the PDF document

You can use the Output service to generate a PDF document using the form design that the Assembler service created. Pass the `com.adobe.idp.Document` instance that the Assembler service returned to the Output service.

Save the PDF document as a PDF file

After the Output service generates a PDF document, you can save it as a PDF file.

See also

"Create a PDF document based on fragments using the Java API" on page 705

"Create a PDF document based on fragments using the web service API" on page 706

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

"Assembling Multiple XDP Fragments" on page 983

"Creating PDF Documents" on page 681

Create a PDF document based on fragments using the Java API

Create a PDF document based on fragments by using the Output Service API and Assembler Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-output-client.jar, in your Java project's class path.

2 Create an Output and Assembler Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.
- Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Use the Assembler service to generate the form design.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to use.
- A `java.util.Map` object that contains the input XDP files.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including the default font and the job log level.

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the assembled XDP document. To retrieve the assembled XDP document, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This method returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the assembled XDP document.

4 Use the Output service to generate the PDF document.

Invoke the `OutputClient` object's `generatePDFOutput2` method and pass the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`
- A string value that specifies the content root where the additional resources, such as images, are located
- A `com.adobe.idp.Document` object that represents the form design (use the instance returned by the Assembler service)

- A `PDFOutputOptionsSpec` object that contains PDF run-time options
- A `RenderOptionsSpec` object that contains rendering run-time options
- The `com.adobe.idp.Document` object that contains the XML data source that contains data to merge with the form design

The `generatePDFOutput2` method returns an `OutputResult` object that contains the results of the operation

5 Save the PDF document as a PDF file.

- Retrieve a `com.adobe.idp.Document` object that represents the PDF document by invoking the `OutputResult` object's `getGeneratedDoc` method.
- Create a `java.io.File` object that contains the results of the operation. Ensure that the filename extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file. (Ensure that you use the `com.adobe.idp.Document` object that the `getGeneratedDoc` method returned.).

See also

"Summary of steps" on page 704

"Quick Start (SOAP mode): Creating a PDF document based on fragments using the Java API" on page 255

Quick Start (SOAP mode): Creating a PDF document based on fragments using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Create a PDF document based on fragments using the web service API

Create a PDF document based on fragments by using the Output Service API and Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Use the following WSDL definition for the service reference associated with the Output service:

```
http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1.
```

Use the following WSDL definition for the service reference associated with the Assembler service:

```
http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1.
```

Because the `BLOB` data type is common to both service references, fully qualify the `BLOB` data type when using it. In the corresponding web service quick start, all `BLOB` instances are fully qualified.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Output and Assembler Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the `OutputServiceClient.ClientCredentials.UserName.UserName` field.
  - Assign the corresponding password value to the `OutputServiceClient.ClientCredentials.UserName.Password` field.
  - Assign the constant value `HttpClientCredentialType.Basic` to the `BasicHttpBindingSecurity.Transport.ClientCredentialType` field.
- Assign the `BasicHttpSecurityMode.TransportCredentialOnly` constant value to the `BasicHttpBindingSecurity.Security.Mode` field.

Note: *Repeat these steps for the `AssemblerServiceClient` object.*

3 Use the Assembler service to generate the form design.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the DDX document
- The `MyMapOf_xsd_string_To_xsd_anyType` object that contains the required files
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeDDX` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that occurred. To obtain the newly created XDP document, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the resultant PDF documents.
- Iterate through the `Map` object to retrieve the assembled form design. Cast that array member's `value` to a `BLOB`. Pass this `BLOB` instance to the Output service.

4 Use the Output service to generate the PDF document.

Invoke the `OutputServiceClient` object's `generatePDFOutput2` method and pass the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the content root where the additional resources, such as images, are located.
- A `BLOB` object that represents the form design (use the `BLOB` instance returned by the Assembler service).
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `BLOB` object that contains the XML data source that contains data to merge with the form design.
- An output `BLOB` object that the `generatePDFOutput2` method populates. The `generatePDFOutput2` method populates this object with generated metadata that describes the document. (This parameter value is required only for web service invocation).
- An output `OutputResult` object that contains the results of the operation. (This parameter value is required only for web service invocation).

The `generatePDFOutput2` method returns a `BLOB` object that contains the non-interactive PDF form.

5 Save the PDF document as a PDF file.

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the interactive PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object retrieved from the `generatePDFOutput2` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 704

Quick Start (MTOM): Creating a PDF document based on fragments using the web service API

"Invoking AEM Forms using MTOM" on page 529

Printing to Files

You can use the Output service to print streams such as PostScript, Printer Control Language (PCL), or the following label formats to a file:

- Zebra - ZPL
- Intermec - IPL
- Datamax - DPL
- TecToshiba - TPCL

Using the Output service, you can merge XML data with a form design and print the form to a file. The following illustration shows the Output service creating laser and label files.

Note: *For information about sending print streams to printers, see "Sending Print Streams to Printers" on page 713.*

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To print to a file, perform the following steps:

1 Include project files.

2 Create an Output Client object.

3 Reference an XML data source.

4 Set print run-time options required to print to a file.

5 Print the print stream to a file.

6 Retrieve the results of the operation.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you will need to replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed. (See "Including AEM Forms Java library files" on page 491.)

Create an Output Client object

Before you can programmatically perform an Output service operation, you must create an Output service client object. If you are using the Java API, create an `OutputClient` object. If you are using the Output web service API, create an `OutputServiceService` object.

Reference an XML data source

To print a document that contains data, you must reference an XML data source that contains XML elements for every form field that you want to populate with data. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed if all XML elements are specified.

Set print run-time options required to print to a file

To print to a file, you must set the File URI run-time option by specifying the location and the name of the file to which the Output service prints. For example, to instruct the Output service to print a PostScript file named *MortgageForm.ps* to C:\Adobe, specify C:\Adobe\MortgageForm.ps.

Note: *There are optional run-time options that you can define. For information about all the options that you can set, see the* `PrintedOutputOptionsSpec` *class reference in AEM Forms API Reference.*

Print the print stream to a file

After you reference a valid XML data source that contains form data and you set print run-time options, you can invoke the Output service, which causes it to print a file.

Retrieve the results of the operation

After the Output service performs an operation, it returns various data items, such as XML data, that specifies whether the operation was successful.

See also

"Print to files using the Java API" on page 710

"Print to files using the web service API" on page 711

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Print to files using the Java API

Print to a file using the Output API (Java):

1 Include project files.

Include client JAR files, such as the adobe-output-client.jar, in your Java project's class path.

2 Create an Output Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an XML data source.

- Create a `java.io.FileInputStream` object that represents the XML data source that is used to populate the document by using its constructor and passing a string value that specifies the location of the XML file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set print run-time options required to print to a file.

- Create a `PrintedOutputOptionsSpec` object by using its constructor.
- Specify the file by invoking the PrintedOutputOptionsSpec object's `setFileURI` method and passing a string value that represents the name and location of the file. For example, if you want the Output service to print to a PostScript file named *MortgageForm.ps* located in C:\Adobe, specify C:\\Adobe\MortgageForm.ps.
- Specify the number of copies to print by invoking the `PrintedOutputOptionsSpec` object's `setCopies` method and passing an integer value that represents the number of copies.

5 Print the print stream to a file.

Print to a file by invoking the `OutputClient` object's `generatePrintedOutput` method and passing the following values:

- A `PrintFormat` enumeration value that specifies the print stream format to create. For example, to create a PostScript print stream, pass `PrintFormat.PostScript`.
- A string value that specifies the name of the form design.
- A string value that specifies the location of related collateral files such as image files.
- A string value that specifies the location of the XDC file to use (you can pass `null` if you specified the XDC file to use by using the `PrintedOutputOptionsSpec` object).
- The `PrintedOutputOptionsSpec` object that contains run-time options required to print to a file.
- The `com.adobe.idp.Document` object that contain the XML data source that contains form data.

The `generatePrintedOutput` method returns an `OutputResult` object that contains the results of the operation.

Note: *The `OutputResult` object's `getRecordLevelMetaDataList` method returns `null`.*

6 Retrieve the results of the operation.

- Create a `com.adobe.idp.Document` object that represents the status of the `generatePrintedOutput` method by invoking the `OutputResult` object's `getStatusDoc` method (the `OutputResult` object was returned by the `generatePrintedOutput` method).
- Create a `java.io.File` object that will contain the results of the operation. Ensure that the file extension is XML.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `getStatusDoc` method).

See also

"Summary of steps" on page 708

Quick Start (SOAP mode): Printing to a file using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Print to files using the web service API

Print to a file using the Output API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Output Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an XML data source.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store form data.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that specifies the location of the XML file that contains form data.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.

4 Set print run-time options required to print to a file.

- Create a `PrintedOutputOptionsSpec` object by using its constructor.
- Specify the file by assigning a string value that represents the location and name of the file to the `PrintedOutputOptionsSpec` object's `fileURI` data member. For example, if you want the Output service to print to a PostScript file named *MortgageForm.ps* located in C:\Adobe, specify C:\\Adobe\MortgageForm.ps.
- Specify the number of copies to print by assigning an integer value that represents the number of copies to the `PrintedOutputOptionsSpec` object's `copies` data members.

5 Print the print stream to a file.

Print to a file by invoking the `OutputServiceService` object's `generatePrintedOutput` method and passing the following values:

- A `PrintFormat` enumeration value that specifies the print stream format to create. For example, to create a PostScript print stream, pass `PrintFormat.PostScript`.
- A string value that specifies the name of the form design.
- A string value that specifies the location of related collateral files such as image files.
- A string value that specifies the location of the XDC file to use (you can pass `null` if you specified the XDC file to use by using the `PrintedOutputOptionsSpec` object).
- The `PrintedOutputOptionsSpec` object that contains print run-time options required to print to a file.
- The `BLOB` object that contains the XML data source that contains form data.
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with generated metadata that describes the document. (This parameter value is required for web service invocation only.)
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with result data. (This parameter value is required for web service invocation only.)
- An `OutputResult` object that contains the results of the operation. (This parameter value is required for web service invocation only.)

6 Retrieve the results of the operation.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents an XML file location that contains result data. Ensure that the file extension is XML.
- Create a byte array that stores the data content of the `BLOB` object that was populated with result data by the `OutputServiceService` object's `generatePDFOutput` method (the eighth parameter). Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to the XML file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 708

Quick Start (MTOM): Printing to a file using the web service API

Quick Start (SwaRef): Printing to a file using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Sending Print Streams to Printers

You can use the Output service to send print streams such as PostScript, Printer Control Language (PCL), or the following label formats to network printers:

- Zebra - ZPL
- Intermec - IPL
- Datamax - DPL
- TecToshiba - TPCL

Using the Output service, you can merge XML data with a form design and output the form as a print stream. For example, you can create a PostScript print stream and send it to a network printer. The following illustration shows the Output service sending print streams to network printers.

Note: *To demonstrate how to send a print stream to a network printer, this section sends a PostScript print stream to a network printer by using the SharedPrinter printer protocol.*

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To send a print stream to a network printer, perform the following steps:

1. Include project files.
2. Create an Output Client object.
3. Reference an XML data source.
4. Set print run-time options
5. Retrieve a document to print.
6. Send the document to a network printer.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you will need to replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

Create an Output Client object

Before you can programmatically perform an Output service operation, create an Output service client object. If you are using the Java API, create an `OutputClient` object. If you are using the Output web service API, create an `OutputServiceClient` object.

Reference an XML data source

To print a document that contains data, you must reference an XML data source that contains XML elements for every form field that you want to populate with data. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed if all XML elements are specified.

Set print run-time options

You can set the run-time options when sending a print stream to a printer, including the following options:

- **Copies**: Specifies the number of copies to send to the printer. The default value is 1.
- **Staple**: An XCI option is set when a stapler is used. This option can be specified in the configuration model by the staple element and is used for PS and PCL printers only.
- **OutputJog**: An XCI option is set when output pages should be jogged (physically shifted in the output tray). This option is for PS and PCL printers only.
- **OutputBin**: XCI value that is used to enable the print driver to select the appropriate output bin.

Note: *For information about all run-time options that you can set, see the* `PrintedOutputOptionsSpec` *class reference.*

Retrieve a document to print

Retrieve a print stream to send to a printer. For example, you can retrieve a PostScript file and send it to a printer.

You can choose to send a PDF file if your printer supports PDF. However, an issue with sending a PDF document to a printer is that each printer manufacturer has a different implementation of the PDF interpreter. That is, some print manufacturers use Adobe PDF interpretation, but it depends on the printer. Other printers have their own PDF interpreter. As a result, printing results may vary.

Another limitation of sending a PDF document to a printer is that it just prints; it cannot access duplex, paper tray selection, and stapling, except through settings on the printer.

To retrieve a document to print, you use the `generatePrintedOutput` method. The following table specifies content types that are set for a given print stream when using the `generatePrintedOutput` method.

Print format	Description
DPL	Creates a dpl203.xdc by default or custom xdc output stream.
DPL300DPI	Creates a DPL 300 DPI output stream.
DPL406DPI	Creates a DPL 400 DPI output stream.
DPL600DPI	Creates a DPL 600 DPI output stream.
GenericColorPCL	Creates a Generic Color PCL (5c) output stream.
GenericPSLevel3	Creates a Generic PostScript Level 3 output stream.
IPL	Creates a Custom IPL output stream.
IPL300DPI	Creates a IPL 300 DPI output stream.
IPL400DPI	Creates a IPL 400 DPI output stream.

Print format	Description
PCL	Creates a Generic Monochrome PCL (5e) output stream.
PostScript	Creates a Generic PostScript Level 2 output stream.
TPCL	Creates a Custom TPCL output stream.
TPCL305DPI	Creates a TPCL 305 DPI output stream.
TPCL600DPI	Creates a TPCL 600 DPI output stream.
ZPL	Creates a ZPL 203 DPI output stream.
ZPL300DPI	Creates a ZPL 300 DPI output stream.

Note: *You can also send a print stream to a printer by using the* `generatePrintedOutput2` *method. However the quick starts associated with the Sending Print Streams to Printers section use the* `generatePrintedOutput` *method.*

Send the print stream to a network printer

After you retrieve a document to print, you can invoke the Output service, which causes it to send a print stream to a network printer. For the Output service to successfully locate the printer, you have to specify both the print server and the printer name. In addition, you must also specify the printing protocol.

Note: *If PDFG is installed on the forms server and the server runs on Windows Server 2008, you cannot use the SharedPrinter property. In this situation, use a different printer protocol.*

Note: *If you are using a network printer and the access mechanism is SharedPrinter, you need to specify the complete network path of printer.Send a print stream to a network printer using the Java API*

Send a print stream to a network printer by using the Output API (Java):

1. Include project files.

   Include client JAR files, such as the adobe-output-client.jar, in your Java project's class path.

2. Create an Output Client object
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Reference an XML data source
   - Create a `java.io.FileInputStream` object that represents the XML data source that is used to populate the document by using its constructor and passing a string value that specifies the location of the XML file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Set print run-time options

   Create a `PrintedOutputOptionsSpec` object that represents print run-time options. For example, you can specify the number of copies to print by invoking the `PrintedOutputOptionsSpec` object's `setCopies` method.

   Note: *You cannot set the pagination value by using the* `PrintedOutputOptionsSpec` *object's* `setPagination` *method if you are generating a ZPL print stream. Likewise, you cannot set the following options for a ZPL print stream: OutputJog, PageOffset, and Staple. The* `setPagination` *method is not valid for PostScript generation. It is valid only for PCL generation.*

5 Retrieve a document to print

- Retrieve a document to print by invoking the `OutputClient` object's `generatePrintedOutput` method and passing the following values:
  - A `PrintFormat` enumeration value that specifies the print stream. For example, to create a PostScript print stream, pass `PrintFormat.PostScript`.
  - A string value that specifies the name of the form design.
  - A string value that specifies the location of related collateral files, such as image files.
  - A string value that specifies the location of the XDC file to use.
  - The `PrintedOutputOptionsSpec` object that contains run-time options that are required to print to a file.
  - The `com.adobe.idp.Document` object that represents the XML data source that contains form data to merge with the form design.

  This method returns an `OutputResult` object that contains the results of the operation.
- Create a `com.adobe.idp.Document` object to send to the printer by invoking the `OutputResult` object 's `getGeneratedDoc` method. This method returns a `com.adobe.idp.Document` object.

6 Send the print stream to a network printer

Send the print stream to a network printer by invoking the `OutputClient` object's `sendToPrinter` method and passing the following values:

- A `com.adobe.idp.Document` object that represents the print stream to send to the printer.
- A `PrinterProtocol` enumeration value that specifies the printer protocol to use. For example, to specify the SharedPrinter protocol, pass `PrinterProtocol.SharedPrinter`.
- A string value that specifies the name of the print server. For example, assuming the name of the print server is PrintSever1, pass`\\\PrintSever1`.
- A string value that specifies the name of the printer. For example, assuming the name of the printer is Printer1, pass `\\\PrintSever1\Printer1`.

Note: *The `sendToPrinter` method was added to the AEM Forms API in version 8.2.1.*

Send a print stream to a printer using the web service API

Send a print stream to a network printer by using the Output API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Output Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.

- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an XML data source.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store form data.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that specifies the location of the XML file that contains form data.
- Create a byte array that stores the content of the `System.IO.FileStream` object. Determine the byte array length by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Set print run-time options.

Create a `PrintedOutputOptionsSpec` object by using its constructor. For example, you can specify the number of copies to print by assigning an integer value that represents the number of copies to the `PrintedOutputOptionsSpec` object's `copies` data member.

Note: *You cannot set the pagination value by using the `PrintedOutputOptionsSpec` object's `pagination` data member if you are generating a ZPL print stream. Likewise, you cannot set the following options for a ZPL print stream: OutputJog, PageOffset and Staple. The`pagination` data member is not valid for PostScript generation. It is valid only for PCL generation.*

5 Retrieve a document to print.

- Retrieve a document to print by invoking the `OutputServiceService` object's `generatePrintedOutput` method and passing the following values:
  - A `PrintFormat` enumeration value that specifies the print stream. For example, to create a PostScript print stream, pass `PrintFormat.PostScript`.
  - A string value that specifies the name of the form design.
  - A string value that specifies the location of related collateral files, such as image files.
  - A string value that specifies the location of the XDC file to use.
  - The `PrintedOutputOptionsSpec` object that contains print run-time options that are used when sending a print stream to a network printer.
  - The `BLOB` object that contains the XML data source that contains form data.

- A `BLOB` object that is populated by the `generatePrintedOutput` method. The `generatePrintedOutput` method populates this object with generated metadata that describes the document. (This parameter value is required for web service invocation only.)
- A `BLOB` object that is populated by the `generatePrintedOutput` method. The `generatePrintedOutput` method populates this object with result data. (This parameter value is required for web service invocation only.)
- An `OutputResult` object that contains the results of the operation. (This parameter value is required for web service invocation only.)

- Create a `BLOB` object to send to the printer by getting the value of the `OutputResult` object 's `generatedDoc` method. This method returns a `BLOB` object that contains PostScript data returned by the `generatePrintedOutput` method.

6 Send the print stream to a network printer.

Send the print stream to a network printer by invoking the `OutputClient` object's `sendToPrinter` method and passing the following values:

- A `BLOB` object that represents the print stream to send to the printer.
- A `PrinterProtocol` enumeration value that specifies the printer protocol to use. For example, to specify the SharedPrinter protocol, pass `PrinterProtocol.SharedPrinter`.
- A `bool` value that specifies whether to user the previous parameter value. Pass the value `true`. (This parameter value is required for web service invocation only.)
- A string value that specifies the name of the print server. For example, assuming that the name of the print server is PrintSever1, pass `\\\PrintSever1`.
- A string value that specifies the name of the printer. For example, assuming that the name of the printer is Printer1, pass `\\\PrintSever1\Printer1`.

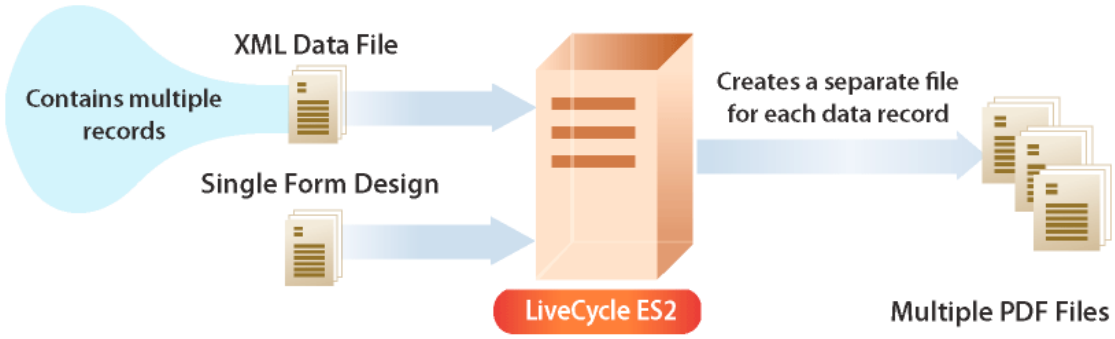
Note: *The `sendToPrinter` method was added to the AEM Forms API in version 8.2.1.*

Creating Multiple Output Files

The Output service can create separate documents for each record within an XML data source or a single file that contains all records (this functionality is the default). For example, assume that ten records are located within an XML data source and you instruct the Output service to create separate PDF documents (or other types of output) for each record by using the Output Service API. As a result, the Output service generates ten PDF documents. (Instead of creating documents, you can send multiple print streams to a printer.)

The following illustration also shows the Output service processing an XML data file that contains multiple records. However, assume that you instruct the Output service to create a single PDF document that contains all data records. In this situation, the Output service generates one document that contains all of the records.

The following illustration shows the Output service processing an XML data file that contains multiple records. Assume that you instruct the Output service to create a separate PDF document for each data record. In this situation, the Output service generates a separate PDF document for each data record.



The following XML data shows an example of a data file that contains three data records.

```
<?xml version="1.0" encoding="UTF-8"?>
<batch>
<LoanRecord>
    <mortgageAmount>500000</mortgageAmount>
    <lastName>Blue</lastName>
    <firstName>Tony</firstName>
    <SSN>555666777</SSN>
    <PositionTitle>Product Manager</PositionTitle>
    <Address>555 No Where Dr</Address>
    <City>New York</City>
    <StateProv>New York</StateProv>
    <ZipCode>51256</ZipCode>
    <Email>TBlue@NoMailServer.com</Email>
    <PhoneNum>555-7418</PhoneNum>
    <FaxNum>555-9981</FaxNum>
    <Description>Buy a home</Description>
</LoanRecord>
<LoanRecord>
    <mortgageAmount>300000</mortgageAmount>
    <lastName>White</lastName>
    <firstName>Sam</firstName>
    <SSN>555666222</SSN>
    <PositionTitle>Program Manager</PositionTitle>
    <Address>557 No Where Dr</Address>
    <City>New York</City>
    <StateProv>New York</StateProv>
    <ZipCode>51256</ZipCode>
    <Email>SWhite@NoMailServer.com</Email>
```

```
    <PhoneNum>555-7445</PhoneNum>
    <FaxNum>555-9986</FaxNum>
    <Description>Buy a home</Description>
</LoanRecord>
<LoanRecord>
    <mortgageAmount>700000</mortgageAmount>
    <lastName>Green</lastName>
    <firstName>Steve</firstName>
    <SSN>55566688</SSN>
    <PositionTitle>Project Manager</PositionTitle>
    <Address>445 No Where Dr</Address>
    <City>New York</City>
    <StateProv>New York</StateProv>
    <ZipCode>51256</ZipCode>
    <Email>SGreeb@NoMailServer.com</Email>
    <PhoneNum>555-2211</PhoneNum>
    <FaxNum>555-2221</FaxNum>
    <Description>Buy a home</Description>
</LoanRecord>
</batch>
```

Notice that the XML element that starts and ends each data record is `LoanRecord`. This XML element is referenced by the application logic that generates multiple files.

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To create multiple PDF files based on an XML data source, perform the following steps:

1. Include project files.
2. Create an Output Client object.
3. Reference an XML data source.
4. Set PDF run-time options.
5. Set rendering run-time options.
6. Generate multiple PDF files.
7. Retrieve the results of the operation.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you will need to replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

Create an Output Client object

Before you can programmatically perform an Output service operation, you must create an Output service client object. If you are using the Java API, create an `OutputClient` object. If you are using the Output web service API, create an `OutputServiceService` object.

Reference an XML data source

Reference an XML data source that contains multiple records. An XML element must be used to separate the data records. For example, in the example XML data source that is shown earlier in this section, the XML element that separates data records is named `LoanRecord`.

An XML element must exist for every form field that you want to populate with data. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed if all XML elements are specified.

Set PDF run-time options

You must set the following run-time options for the Output service to successfully create multiple files based on an XML data source:

- **Many Files**: Specifies whether the Output service creates a single document or multiple documents. You can specify true or false. To create a separate document for each data record in the XML data source, specify true.
- **File URI**: Specifies the location of the files that the Output service generates. For example, assume that you specify C:\\Adobe\forms\Loan.pdf. In this situation, the Output service creates a file named Loan.pdf and places the file in the C:\\Adobe\forms folder. When there are multiple files, the file names are Loan0001.pdf, Loan0002.pdf, Loan0003.pdf and so on. If you specify a file location, the files are placed on the server, not the client computer.
- **Record Name**: Specifies the XML element name in the data source that separates the data records. For example, in the example XML data source that is shown earlier in this section, the XML element that separates data records is called `LoanRecord`. (Instead of setting the Record Name run-time option, you can set the Record Level by assigning it a numeric value that indicates the element level that contains data records. However, you can set only the Record Name or the Record Level. You cannot set both values.)

Set rendering run-time options

You can set rendering run-time options while creating multiple files. Although these options are not required (unlike output run-time options, which are required), you can perform tasks such as improving the performance of the Output service. For example, you can cache the form design that the Output service uses in order to improve performance.

When the Output service processes batch records, it reads data that contains multiple records in an incremental manner. That is, the Output service reads the data into memory and releases the data as the batch of records is processed. The Output service loads data in an incremental manner when either one of two run-time options are set. If you set the Record Name run-time option, the Output service reads data in an incremental manner. Likewise, if you set the Record Level run-time option to 2 or greater, the Output service reads data in an incremental manner.

You can control whether the Output service performs incremental loading by using the `PDFOutputOptionsSpec` or the `PrintedOutputOptionSpec` object's `setLazyLoading` method. You can pass the value `false` to this method which turns off incremental loading.

Generate multiple PDF files

After you reference a valid XML data source that contains multiple data records and set run-time options, you can invoke the Output service, which causes it to generate multiple files. When generating multiple records, the `OutputResult` object's `getGeneratedDoc` method returns `null`.

Retrieve the results of the operation

After the Output service performs an operation, it returns XML data that specifies whether the operation was successful. The following XML is returned by the Output service. In this situation, the Output service generated 42 documents.

```
<?xml version="1.0" encoding="UTF-8"?>
<printResult>
<status>0</status>
<requestId>4ad85f9e2</requestId>
<context/>
<messages>
<message>Printed all 42 records successfully.</message>
</messages>
<printSpec>
<input>
<validated>true</validated>
<dataFile recordIdField="" recordLevel="0" recordName="LoanRecord"/>
<sniffRules lookAhead="300"/>
<formDesign>Loan.xdp</formDesign>
<contentRoot>C:\Adobe</contentRoot>
<metadata-spec record="false"/>
</input>
<output>
<format>PDF</format>
<fileURI>C:\Adobe\forms\Loan.pdf</fileURI>
<optionString>cacheenabled=true&amp;padebug=false&amp;linearpdf=false&amp;pdfarevisionnumber
=1&amp;pdfaconformance=A&amp;taggedpdf=false&amp;TransactionTimeOut=180</optionString>
<waitForResponse>true</waitForResponse>
<outputStream>multiple</outputStream>
</output>
</printSpec>
</printResult>
```

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Create multiple PDF files using the Java API

Create multiple PDF files by using the Output API (Java):

1 Include project files"

Include client JAR files, such as adobe-output-client.jar, in your Java project's class path. .

2 Create an Output Client object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an XML data source

- Create a `java.io.FileInputStream` object that represents the XML data source that contains multiple records by using its constructor and passing a string value that specifies the location of the XML file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set PDF run-time options

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Set the Many Files option by invoking the `PDFOutputOptionsSpec` object's `setGenerateManyFiles` method. For example, pass the value `true` to instruct the Output service to create a separate PDF file for each record in the XML data source. (If you pass `false`, the Output service generates a single PDF document that contains all records).
- Set the File URI option by invoking the `PDFOutputOptionsSpec` object's `setFileUri` method and passing a string value that specifies the location of the files that the Output service generates. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.
- Set the Record Name option by invoking the `OutputOptionsSpec` object's `setRecordName` method and passing a string value that specifies the XML element name in the data source that separates the data records. (For example, consider the XML data source shown earlier in this section. The name of the XML element that separates data records is LoanRecord).

5 Set rendering run-time options

- Create a `RenderOptionsSpec` object by using its constructor.
- Cache the form design to improve the performance of the Output service by invoking the `RenderOptionsSpec` object's `setCacheEnabled` and passing a `Boolean` value of `true`.

6 Generate multiple PDF files

Generate multiple PDF files by invoking the `OutputClient` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enum value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contains the XML data source that contains data to merge with the form design.

The `generatePDFOutput` method returns an `OutputResult` object that contains the results of the operation.

7 Retrieve the results of the operation

- Create a `java.io.File` object that represents an XML file that will contain the results of the `generatePDFOutput` method. Ensure that the file name extension is .xml.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `applyUsageRights` method).

See also

"Summary of steps" on page 720

"Quick Start (SOAP mode): Creating multiple PDF files using the Java API" on page 264

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Create multiple PDF files using the web service API

Create multiple PDF files by using the Output API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create an Output Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an XML data source.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store form data that contains multiple records.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the XML file that contains multiple records.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.

- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Set PDF run-time options.

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Set the Many Files option by assigning a Boolean value to the `OutputOptionsSpec` object's `generateManyFiles` data member. For example, assign the value `true` to this data member to instruct the Output service to create a separate PDF file for each record in the XML data source. (If you assign `false` to this data member, then the Output service generates a single PDF that contains all records).
- Set the file URI option by assigning a string value that specifies the location of the file(s) that the Output service generates to the `OutputOptionsSpec` object's `fileURI` data member. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.
- Set the record name option by assigning a string value that specifies the XML element name in the data source that separates the data records to the `OutputOptionsSpec` object's `recordName` data member.
- Set the copies option by assigning an integer value that specifies the number of copies that the Output service generates to the `OutputOptionsSpec` object's `copies` data member.

5 Set rendering run-time options.

- Create a `RenderOptionsSpec` object by using its constructor.
- Cache the form design to improve the performance of the Output service by assigning the value `true` to the `RenderOptionsSpec` object's `cacheEnabled` data member.

6 Generate multiple PDF files.

Create multiple PDF files by invoking the `OutputServiceService` object's `generatePDFOutput` method and passing the following values:

- A TransformationFormat enum value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `BLOB` object that contains the XML data source that contains data to merge with the form design.
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with generated metadata that describes the document.
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with result data.
- An `OutputResult` object that contains the results of the operation.

7 Retrieve the results of the operation

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents an XML file location that contains result data. Ensure that the file name extension is .xml.
- Create a byte array that stores the data content of the `BLOB` object that was populated with result data by the `OutputServiceService` object's `generatePDFOutput` method (the eighth parameter). Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

- Write the contents of the byte array to the XML file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 720

Quick Start (MTOM): Creating multiple PDF files using the web service API

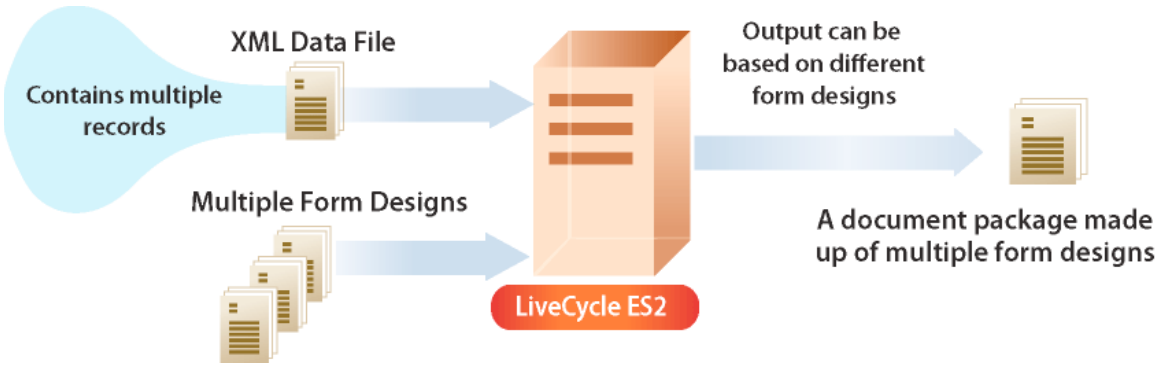
"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Creating Search Rules

You can create search rules that result in the Output service examining input data and using different form designs based on the data content to generate output. For example, if the text *mortgage* is located within the input data, then the Output service can use a form design named Mortgage.xdp. Likewise, if the text *automobile* is located in the input data, then the Output service can use a form design that is saved as AutomobileLoan.xdp. Although the Output service can generate different output types, this section assumes that the Output service generates a PDF file. The following diagram shows the Output service generating a PDF file by processing an XML data file and using one of many form designs.

In addition, the Output service is able to generate document packages, where multiple records are provided in the data set and each record is matched to a form design and a single document is generated made up of multiple form designs.



Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To instruct the Output service to use search rules while generating a document, perform the following steps:

1. Include project files.
2. Create an Output Client object.
3. Reference an XML data source.
4. Define search rules.
5. Set PDF run-time options.
6. Set rendering run-time options.

7 Generate a PDF document.

8 Retrieve the results of the operation.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, then you will need to replace adobe-utilities.jar and jbossall-client.jar with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

Create an Output Client object

Before you can programmatically perform an Output service operation, you must create an Output service client object.

Reference an XML data source

An XML element must exist for every form field that you want to populate with data. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name. It is not necessary to match the order in which the XML elements are displayed, as long as all XML elements are specified.

Define search rules

To define search rules, you define one or more text patterns that the Output services searches for in the input data. For each text pattern that you define, you specify a corresponding form design that is used if the text pattern is located. If a text pattern is located, then the Output service uses the corresponding form design to generate the output. An example of a text pattern is *mortgage*.

Note: *If text patterns are not located, then the default form is used. Make sure that all form designs that you use are located in the content root.*

Set PDF run-time options

Set the following PDF run-time options in order for the Output service to successfully create a PDF document based on multiple form designs:

- **File URI**: Specifies the name and location of the PDF file that the Output service generates.
- **Rules**: Specifies rules that you defined.
- **LookAHead**: Specifies the number of bytes to use from the beginning of the input data file to scan for the defined text patterns. The default is 500 bytes.

Set rendering run-time options

You can set rendering run-time options while creating PDF files. Although these options are not required (unlike PDF run-time options), you can perform tasks such as improving the performance of the Output service. For example, you can cache the form design that the Output service uses in order to improve performance.

Generate a PDF document

After you reference a valid XML data source and set run-time options, you can invoke the Output service resulting in it generating a PDF document. If the Output service locates a specified text pattern in the input data, then it uses the corresponding form design. If a text pattern is not used, then the Output service uses the default form design.

Retrieve the results of the operation

After the Output service performs an operation, it returns XML data that specifies whether the operation was successful.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Create search rules using the Java API

Create search rules by using the Output API (Java):

1 Include project files.

   Include client JAR files, such as adobe-output-client.jar, in your Java project's class path.

2 Create an Output Client object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an XML data source.

   - Create a `java.io.FileInputStream` object that represents the XML data source that is used to populate the PDF document by using its constructor and passing a string value that specifies the location of the XML file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Define search rules.

   - Create a `Rule` object by using its constructor.
   - Define a text pattern by invoking the `Rule` object's `setPattern` method and passing a string value that specifies a text pattern.
   - Define the corresponding form design by invoking the `Rule` object's `setForm` method . Pass a string value that specifies the name of the form design.

   Note: *For each text pattern that you want to define, repeat the previous three sub-steps.*

   - Create a `java.util.List` object by using an `java.util.ArrayList` constructor.
   - For each `Rule` object that you created, invoke the `java.util.List` object's `add` method and pass the `Rule` object.

5 Set PDF run-time options.

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Specify the name and location of the PDF file that the Output service generates by invoking the `PDFOutputOptionsSpec` object's `setFileURI` method. Pass a string value that specifies the location of the PDF file. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.
- Set the rules that you defined by invoking the `PDFOutputOptionsSpec` object's `setRules` method. Pass the `java.util.List` object that contains the `Rule` objects.
- Set the number of bytes to scan for the defined text patterns by invoking the `PDFOutputOptionsSpec` object's `setLookAhead` method. Pass an integer value that represents the numbers of bytes.

6 Set rendering run-time options.

- Create a `RenderOptionsSpec` object by using its constructor.
- Cache the form design in order to improve the performance of the Output service by invoking the `RenderOptionsSpec` object's `setCacheEnabled` and passing `true`.

7 Generate a PDF document.

Generate a PDF document that is based on multiple form designs by invoking the `OutputClient` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the default form design. That is, the form design that is used if a text pattern is not located.
- A string value that specifies the content root where the form designs are located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `com.adobe.idp.Document` object that contains the form data that is searched by the Output service for the defined text patterns.

The `generatePDFOutput` method returns an `OutputResult` object that contains the results of the operation.

8 Retrieve the results of the operation.

- Create a `com.adobe.idp.Document` object that represents the status of the `generatePDFOutput` method by invoking the `OutputResult` object's `getStatusDoc` method.
- Create a `java.io.File` object that will contain the results of the operation. Ensure that the file extension is .xml.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `getStatusDoc` method).

See also

"Summary of steps" on page 726

"Quick Start (SOAP mode): Creating search rules using the Java API" on page 266

Quick Start (SOAP mode): Creating search rules using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Create search rules using the web service API

Create search rules by using the Output API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Output Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an XML data source.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store data that will be merged with the PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to encrypt and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Define search rules.

- Create a `Rule` object by using its constructor.
- Define a text pattern by assigning a string value that specifies a text pattern to the `Rule` object's `pattern` data member.

- Define the corresponding form design by assigning a string value that specifies the form design to the `Rule` object's `form` data member.

Note: *For each text pattern that you want to define, repeat the previous three sub-steps.*

- Create a `MyArrayOf_xsd_anyType` object that stores the rules.
- Assign each `Rule` object to an element of the `MyArrayOf_xsd_anyType` array. Invoke the `MyArrayOf_xsd_anyType` object's `Add` method for each `Rule` object.

5 Set PDF run-time options

- Create a `PDFOutputOptionsSpec` object by using its constructor.
- Set the file URI option by assigning a string value that specifies the location of the PDF file that the Output service generates to the `PDFOutputOptionsSpec` object's `fileURI` data member. The File URI option is relative to the J2EE application server hosting AEM Forms, not the client computer.
- Set the copies option by assigning an integer value that specifies the number of copies that the Output service generates to the `PDFOutputOptionsSpec` object's `copies` data member.
- Set the rules that you defined by assigning the `MyArrayOf_xsd_anyType` object that stores the rules to the `PDFOutputOptionsSpec` object's `rules` data member.
- Set the number of bytes to scan for the defined text patterns by assigning an integer value that represents the numbers of bytes to scan to the `PDFOutputOptionsSpec` object's `lookAhead` data method.

6 Set rendering run-time options

- Create a `RenderOptionsSpec` object by using its constructor.
- Cache the form design in order to improve the performance of the Output service by assigning the value `true` to the `RenderOptionsSpec` object's `cacheEnabled` data member.

Note: *You cannot set the version of the PDF document by using the `RenderOptionsSpec` object's `pdfVersion` member if the input document is an Acrobat form. The output PDF document retains the PDF version of the Acrobat form. Likewise, you cannot set the tagged PDF option by using the `RenderOptionsSpec` object's `taggedPDF` method if the input document is an Acrobat form.*

Note: *You cannot set the linearized PDF option by using the `RenderOptionsSpec` object's `linearizedPDF` member if the input PDF document is certified or digitally signed. For information, see "Digitally Signing PDF Documents" on page 892.*

7 Generate a PDF document

Create a PDF document by invoking the `OutputServiceService` object's `generatePDFOutput` method and passing the following values:

- A `TransformationFormat` enumeration value. To generate a PDF document, specify `TransformationFormat.PDF`.
- A string value that specifies the name of the form design.
- A string value that specifies the content root where the form design is located.
- A `PDFOutputOptionsSpec` object that contains PDF run-time options.
- A `RenderOptionsSpec` object that contains rendering run-time options.
- The `BLOB` object that contains the XML data source that contains data to merge with the form design.
- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with generated metadata that describes the document. (This parameter value is required only for web service invocation).

- A `BLOB` object that is populated by the `generatePDFOutput` method. The `generatePDFOutput` method populates this object with result data. (This parameter value is required only for web service invocation).
- An `OutputResult` object that contains the results of the operation. (This parameter value is required only for web service invocation).

Important: *When generating a PDF document by invoking the `generatePDFOutput` method, be aware that you cannot merge data with an XFA PDF form that is signed, certified, or contains usage rights. For information about usage rights, see "Applying Usage Rights to PDF Documents" on page 751.*

8 Retrieve the results of the operation

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents an XML file location that contains result data. Ensure that the file extension is XML.
- Create a byte array that stores the data content of the `BLOB` object that was populated with result data by the `OutputServiceService` object's `generatePDFOutput` method (the eighth parameter). Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to the XML file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 726

Quick Start (MTOM): Creating search rules using the web service API

Quick Start (SwaRef): Creating search rules using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Flattening PDF Documents

You can use the Output service to transform an interactive PDF document to a non-interactive PDF. An interactive PDF document lets users enter or modify data that is in the PDF document fields. The process of transforming an interactive PDF document to a non-interactive PDF document is called *flattening*. When a PDF document is flattened, a user cannot modify the data in the document fields. One reason to flatten a PDF document is to ensure that data cannot be modified.

You can flatten the following types of PDF documents:

- Interactive XFA PDF documents
- Acrobat Forms

Attempting to flatten a PDF that is a non-interactive PDF document causes an exception.

Note: *For more information about the Output service, see Services Reference for AEM Forms.*

Summary of steps

To flatten an interactive PDF document to a non-interactive PDF document, perform the following steps:

1 Include project files.

2 Create an Output Client object.

3 Retrieve an interactive PDF document.

4 Transform the PDF document.

5 Save the non-interactive PDF document as a PDF file.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are by using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-output-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you will need to replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create an Output Client object

Before you can programmatically perform an Output service operation, you must create an Output service client object. If you are using the Java API, create an `OutputClient` object. If you are using the Output web service API, create an `OutputServiceService` object.

Retrieve an interactive PDF document

Retrieve an interactive PDF document that you want to transform to a non-interactive PDF document. Attempting to transform a non-interactive PDF document, causes an exception.

Transform the PDF document

After you retrieve an interactive PDF document, you can transform it to a non-interactive PDF document. The Output service returns a non-interactive PDF document.

Save the non-interactive PDF document as a PDF file

You can save the non-interactive PDF document as a PDF file.

See also

"Flatten a PDF document using the Java API" on page 734

"Flatten a PDF document using the web service API" on page 735

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Flatten a PDF document using the Java API

Flatten an interactive PDF document to a non-interactive PDF document by using the Output API (Java):

1 Include project files.

Include client JAR files, such as adobe-output-client.jar, in your Java project's class path.

2 Create an Output Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `OutputClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve an interactive PDF document.

- Create a `java.io.FileInputStream` object that represents the interactive PDF document to transform by using its constructor and passing a string value that specifies the location of the interactive PDF file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Transform the PDF document.

Transform the interactive PDF document to a non-interactive PDF document by invoking the `OutputServiceService` object's `transformPDF` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the interactive PDF document.
- A `TransformationFormat` enum value. To generate a non-interactive PDF document, specify `TransformationFormat.PDF`.
- A `PDFARevisionNumber` enum value that specifies the revision number. Because this parameter is meant for a PDF/A document, you can specify `null`.
- A string value that represents the amendment number and year, separated by a colon. Because this parameter is meant for a PDF/A document, you can specify `null`.
- A `PDFAConformance` enum value that represents the PDF/A conformance level. Because this parameter is meant for a PDF/A document, you can specify `null`.

The `transformPDF` method returns a `com.adobe.idp.Document` object that contains a non-interactive PDF document.

5 Save the non-interactive PDF document as a PDF file.

- Create a `java.io.File` object and ensure that the file name extension is .pdf.
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `transformPDF` method).

See also

"Summary of steps" on page 732

"Quick Start (SOAP mode): Transforming a PDF document using the Java API" on page 269

Quick Start (SOAP mode): Transforming a PDF document using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Flatten a PDF document using the web service API

Flatten an interactive PDF document to a non-interactive PDF document by using the Output API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/OutputService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Output Client object.

- Create an `OutputServiceClient` object by using its default constructor.
- Create an `OutputServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/OutputService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `OutputServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `OutputServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `OutputServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve an interactive PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the interactive PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the interactive PDF document.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Transform the PDF document.

Transform the interactive PDF document to a non-interactive PDF document by invoking the `OutputClient` object's `transformPDF` method and passing the following values:

- A `BLOB` object that contains the interactive PDF document.
- A `TransformationFormat` enumeration value. To generate a non-interactive PDF document, specify `TransformationFormat.PDF`.

- A `PDFARevisionNumber` enum value that specifies the revision number.
- A Boolean value that specifies whether the `PDFARevisionNumber` enum value is used. Because this parameter is meant for a PDF/A document, you can specify `false`.
- A string value that represents the amendment number and year, separated by a colon. Because this parameter is meant for a PDF/A document, you can specify `null`.
- A `PDFAConformance` enum value that represents the PDF/A conformance level.
- Boolean value that specifies whether the `PDFAConformance` enum value is used. Because this parameter is meant for a PDF/A document, you can specify `false`.

The `transformPDF` method returns a `BLOB` object that contains a non-interactive PDF document.

5 Save the non-interactive PDF document as a PDF file.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the non-interactive PDF document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `transformPDF` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 732

Quick Start (MTOM): Transforming a PDF document using the web service API

Quick Start (SwaRef): Transforming a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Importing and Exporting Data

About the Form Data Integration Service

The Form Data Integration service can import data into a PDF form and export data from a PDF form. The import and export operations support two types of PDF forms:

- An Acrobat form (created in Acrobat) is a PDF document that contains form fields.
- An Adobe XML form (created in Designer) is a PDF document that conforms to the XML Adobe XML Forms Architecture (XFA).

Form data can exist in one of the following formats depending on the type of PDF form:

- An XFDF file, which is an XML version of the Acrobat form data format.
- An XDP file, which is an XML file that contains form field definitions. It may also contain form field data and an embedded PDF file. An XDP file generated by Designer can only be used if it carries an embedded base-64-encoded PDF document.

You can accomplish these tasks using the Form Data Integration service:

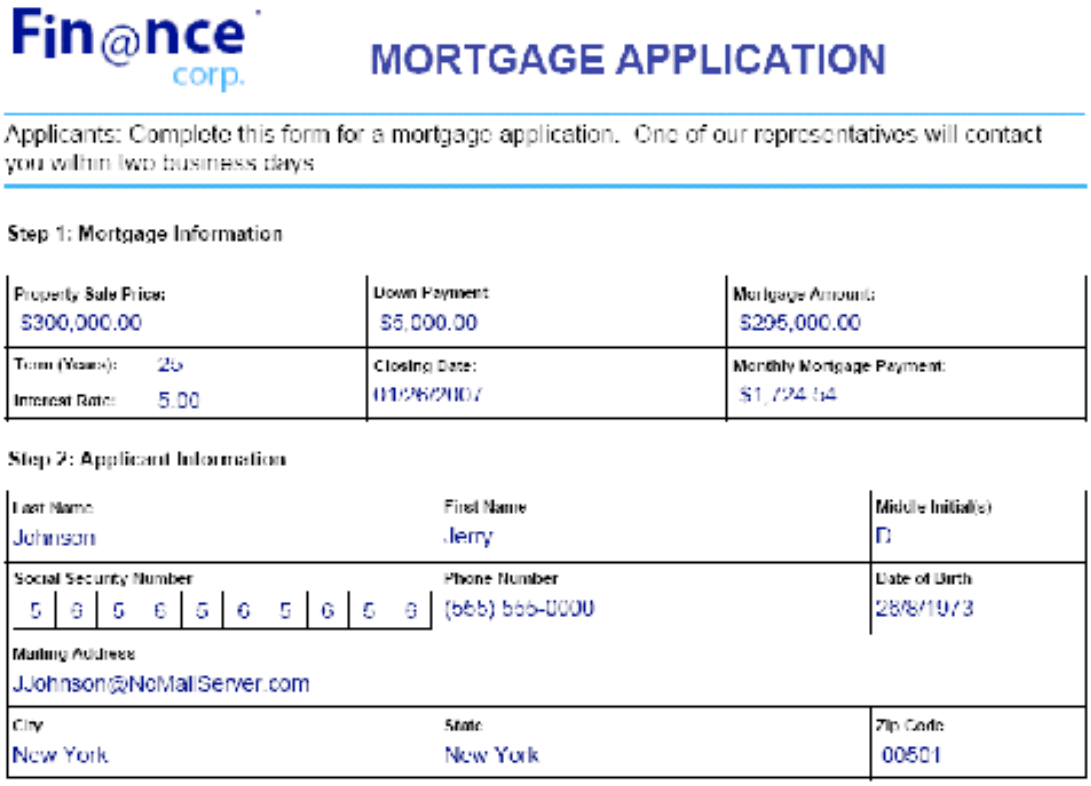
- Import data into PDF forms. For information, see "Importing Form Data" on page 737.
- Export data from PDF forms. For information, see "Exporting Form Data" on page 742.

Note: *For more information about the Form Data Integration service, see Services Reference for AEM Forms.*

Importing Form Data

You can import form data into interactive PDF forms by using the Form Data Integration service. An interactive PDF form is a PDF document that contains one or more fields for collecting information from a user or for displaying custom information. The Form Data Integration service does not support form calculations, validation, or scripting.

To import data into a form created in Designer, you must reference a valid XDP XML data source. Consider the following example mortgage application form.



In order to import data values into this form, you must have a valid XDP XML data source that corresponds to the form. You cannot use an arbitrary XML data source to import data into a form using the Form Data Integration service. The difference between an arbitrary XML data source and an XDP XML data source is that an XDP data source conforms to the XML Forms Architecture (XFA). The following XML represents an XDP XML data source that corresponds to the example mortgage application form.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
- <xfa:data>
- <data>
    - <Layer>
        <closeDate>1/26/2007</closeDate>
        <lastName>Johnson</lastName>
        <firstName>Jerry</firstName>
        <mailingAddress>JJohnson@NoMailServer.com</mailingAddress>
        <city>New York</city>
        <zipCode>00501</zipCode>
        <state>NY</state>
        <dateBirth>26/08/1973</dateBirth>
        <middleInitials>D</middleInitials>
        <socialSecurityNumber>(555) 555-5555</socialSecurityNumber>
        <phoneNumber>5555550000</phoneNumber>
    </Layer>
    - <Mortgage>
        <mortgageAmount>295000.00</mortgageAmount>
        <monthlyMortgagePayment>1724.54</monthlyMortgagePayment>
        <purchasePrice>300000</purchasePrice>
        <downPayment>5000</downPayment>
        <term>25</term>
        <interestRate>5.00</interestRate>
    </Mortgage>
</data>
</xfa:data>
</xfa:datasets>
```

Note: *For more information about the Form Data Integration service, see Services Reference for AEM Forms.*

Summary of steps

To import form data into a PDF form, perform the following steps:

1. Include project files.
2. Create a Form Data Integration service client.
3. Reference a PDF form.
4. Reference an XML data source.
5. Import data into the PDF form.
6. Save the PDF form as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-formdataintegration-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)

- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Form Data Integration service client

Before you can programmatically import data into a PDF form Client API, you must create a Data Integration service client. When creating a service client, you define connection settings that are required to invoke a service. For information, see "Setting connection properties" on page 500.

Reference a PDF form

To import data into a PDF form, you must reference either an XML form created in Designer or an Acrobat form created in Acrobat.

Reference an XML data source

In order to import form data, you must reference a valid data source. To import data into an XFA XML form created in Designer, you must use an XDP XML data source. If you reference an Acrobat form, then you must use an XFDF data source. For each field that you want to import data into, a value must be specified. If an element located in the XML data source does not correspond to a field in the form, then the element is ignored.

Import data into the PDF form

After you reference a PDF form and a valid XML data source, you can import the data into the PDF form.

Save the PDF form as a PDF file

After you import data into a form, you can save the form as a PDF file. Once saved as a PDF file, a user can open the form in Adobe Reader or Acrobat and see the form with the imported data.

See also

"Import form data using the Java API" on page 739

"Import form data using the web service API" on page 740

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Form Data Integration Service Java API Quick Start(SOAP)" on page 207

"Exporting Form Data" on page 742

Import form data using the Java API

Import form data by using the Form Data Integration API (Java):

1 Include project files.

   Include client JAR files, such as adobe-formdataintegration-client.jar, in your Java project's class path.

2 Create a Form Data Integration service client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `FormDataIntegrationClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference a PDF form.

- Create a `java.io.FileInputStream` object by using its constructor. Pass a string value that specifies the location of the PDF form.
- Create a `com.adobe.idp.Document` object that stores the PDF form by using the `com.adobe.idp.Document` constructor. Pass the `java.io.FileInputStream` object that contains the PDF form to the constructor.

4 Reference an XML data source.

- Create a `java.io.FileInputStream` object by using its constructor and pass a string value that specifies the location of the XML file that contains data to import into the form.
- Create a `com.adobe.idp.Document` object that stores form data by using the `com.adobe.idp.Document` constructor. Pass the `java.io.FileInputStream` object that contains form data to the constructor.

5 Import data into the PDF form.

Import data into PDF form by invoking the `FormDataIntegrationClient` object's `importData` method and passing the following values:

- The `com.adobe.idp.Document` object that stores the PDF form.
- The `com.adobe.idp.Document` object that stores form data.

The `importData` method returns a `com.adobe.idp.Document` object that stores a PDF form that contains the data located in the XML data source.

6 Save the PDF form as a PDF file.

- Create a `java.io.File` object and ensure that the file extension is ".PDF".
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `importData` method).

See also

"Summary of steps" on page 738

"Quick Start (SOAP mode): Importing form data using the Java API" on page 208

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Import form data using the web service API

Import form data by using the Form Data Integration API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/FormDataIntegration?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Form Data Integration service client.

- Create a `FormDataIntegrationClient` object by using its default constructor.

- Create a `FormDataIntegrationClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/FormDataIntegration?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `FormDataIntegrationClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `FormDataIntegrationClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `FormDataIntegrationClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference a PDF form.

- Create a `BLOB` object by using its constructor. This `BLOB` object is used to store the PDF form.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that specifies the location of the PDF form and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reference an XML data source.

- Create a `BLOB` object by using its constructor. This `BLOB` object is used to store the data that is imported into the form.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that specifies the location of the XML file that contains data to import and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

5 Import data into the PDF form.

Import data into the PDF form by invoking the `FormDataIntegrationClient` object's `importData` method and passing the following values:

- The `BLOB` object that stores the PDF form.

- The `BLOB` object that stores form data.

The `importData` method returns a `BLOB` object that stores a PDF form that contains the data located in the XML data source.

6 Save the PDF form as a PDF file.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF file.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `importData` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 738

Quick Start (MTOM): Importing form data using the web service API

"Invoking AEM Forms using MTOM" on page 529

Exporting Form Data

You can export form data from an interactive PDF form by using the Form Data Integration service. The format of the data that is exported depends on the form type. If the form type is an Acrobat form created in Acrobat then the exported data is XFDF. If the form type is an XML form that was created in Designer, then the exported data is XDP.

Note: *For more information about the Form Data Integration service, see Services Reference for AEM Forms.*

Summary of steps

To export form data from a PDF form, perform the following steps:

1 Include project files

2 Create a Form Data Integration service client.

3 Reference a PDF form.

4 Export data from the PDF form.

5 Save the exported data as an XML file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-formdataintegration-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

Create a Form Data Integration service client

Before you can programmatically import data into a PDF formClient API, you must create a Data Integration service client. When creating a service client, you define connection settings that are required to invoke a service. For information, "Setting connection properties" on page 500.

Reference a PDF form

To export data from a PDF form, you must reference PDF form that was created in Designer or Acrobat and that contains form data. If you attempt to export data from an empty PDF form, you will get an empty XML schema.

Export data from the PDF form

After you reference a PDF form that contains form data, you can export the data from the form. The data is exported within an XML schema that is based on the form.

Save the form data as an XML file

After you export form data, you can save the data as an XML file. Once saved as an XML file, you can open the XML file within an XML viewer to view the form data.

See also

"Export form data using the Java API" on page 743

"Export form data using the web service API" on page 744

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Form Data Integration Service Java API Quick Start(SOAP)" on page 207

"Importing Form Data" on page 737

Export form data using the Java API

Export form data by using the Form Data Integration API (Java):

1 Include project files.

   Include client JAR files, such as adobe-formdataintegration-client.jar, in your Java project's class path.

2 Create a Form Data Integration service client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `FormDataIntegrationClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference a PDF form.

   - Create a `java.io.FileInputStream` object by using its constructor and pass a string value that specifies the location of the PDF form that contains data to export.
   - Create a `com.adobe.idp.Document` object that stores the PDF form by using the `com.adobe.idp.Document` constructor. Pass the `java.io.FileInputStream` object that contains the PDF form to the constructor.

4 Export data from the PDF form.

   Export form data by invoking the `FormDataIntegrationClient` object's `exportData` method and pass the `com.adobe.idp.Document` object that stores the PDF form. This method returns a `com.adobe.idp.Document` object that stores form data as an XML schema.

5 Save the PDF form as a PDF file.

- Create a `java.io.File` object and ensure that the file extension is XML.
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `exportData` method).

See also

"Summary of steps" on page 742

"Quick Start (SOAP mode): Exporting form data using the Java API" on page 210

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Export form data using the web service API

Export form data by using the Form Data Integration API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/FormDataIntegration?WSDL&lc_version=9.0.1`.

- Replace `localhost` with the IP address of the server hosting AEM Forms.

2 Create a Form Data Integration service client.

- Create a `FormDataIntegrationClient` object by using its default constructor.
- Create a `FormDataIntegrationClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/FormDataIntegration?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `FormDataIntegrationClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `FormDataIntegrationClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `FormDataIntegrationClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference a PDF form.

- Create a `BLOB` object by using its constructor. This `BLOB` object is used to store the PDF form from which data is exported.

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that specifies the location of the PDF form and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Export data from the PDF form.

Import data into PDF form by invoking the `FormDataIntegrationClient` object's `exportData` method and pass the `BLOB` object that stores the PDF form. This method returns a `BLOB` object that stores form data as an XML schema.

5 Save the PDF form as a PDF file.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the location of the XML file.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `exportData` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a XML file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 742

Quick Start (MTOM): Exporting form data using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Working with barcoded forms

About the barcoded forms Service

The barcoded forms service automates the capture of data from fill-and-print forms and integrates captured information into an organization's core IT systems.

Using the barcoded forms service, you can add one-dimensional and two-dimensional barcodes to interactive PDF forms. You can then publish the barcoded forms to a website or distribute them by email or CD. When a user fills a barcoded form by using Adobe Reader, Acrobat Professional, or Acrobat Standard, the barcode is updated automatically to encode the user-supplied form data. The user can submit the form electronically, or print it to paper and submit it by mail, fax, or hand. You can later extract the user-supplied data as part of an automated workflow, routing the data among approval processes and business systems.

For more information about the barcoded forms service, see Services Reference for AEM Forms.

Decoding Barcoded Form Data

You can use the barcoded forms service API to decode data from a PDF form or an image that contains a barcode. Decoding form data means extracting data that is located in the barcode. Before data can be decoded from a PDF form (or image), a user has to populate the form with data.

Note: *For more information about the barcoded forms service, see Services Reference for AEM Forms.*

Summary of steps

To decode data from a PDF form, perform the following steps:

1. Include project files.
2. Create a barcoded formsClient API object.
3. Get a PDF form that contains barcoded data.
4. Decode the data from PDF form.
5. Convert the data to an XML data source.
6. Process the decoded data.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-barcodedforms-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)
- xercesImpl.jar (located in <install directory>/Adobe/Adobe_Experience_Manager_forms/sdk/client-libs\thirdparty)

If AEM Forms is deployed on a supported J2EE application server that is not JBOSS, then you will need to replace adobe-utilities.jar and jbossall-client.jar with JAR files that are specific to the J2EE application server on which AEM Forms is deployed. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create a barcoded forms Client API object

Before you can programmatically perform a barcoded forms service operation, you must create a Barcoded Forms service client. If you are using the Java API, create a `BarcodedFormsServiceClient` object. If you are using the barcoded forms web service API, create a `BarcodedFormsServiceService` object.

Get a PDF form that contains barcoded data

You must obtain a PDF form that contains a barcode that has been populated with user data.

Decode the data from the PDF form

After you obtain a PDF form (or image) that contains a barcode, you can decode data. The Barcoded Forms service supports the following types of barcodes:

- PDF417 barcodes.
- Data matrix barcodes.
- QR code barcodes.
- Codabar barcodes.
- Code 128 barcodes.
- Code 39 barcodes.
- EAN-13 barcodes.
- EAN-8 barcodes.

Character set input as hex in the decode API implies that the content of the barcode is encoded as a hex string. For example, if UTF-8 is specified as the Character encoding in the form and Hex is specified in the decode operation, the content of the barcode is encoded as a Hex string in the `<xb:content>` element in the decoded output. You can convert this Hex value to get the original content by creating application logic in your client application.

Convert the data to an XML data source

After you decode form data, you can convert it to XDP or XFDF data. For example, assume that you want to import the data into another form. To import the data into an XFA form, then you have to convert the data to XDP data. For information, see "Importing Form Data" on page 737.

Process the decoded data

You can process the converted data to meet your business requirements. For example, after you decode and convert the data, you can save it to a file, store it in an enterprise database, populate another form, and so on. This section discusses how to save the converted data as an XML file.

Note: *The barcoded forms service fails to decode barcode data when the line delimiter and field delimiter parameters have the same value*

See also

"Decode barcoded form data using the Java API" on page 747

"Decode barcoded form data using the web service API" on page 749

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Decode barcoded form data using the Java API

Decode form data by using the barcoded forms API(Java):

1

Include client JAR files in your Java project's class path.

2

Create a `BarcodedFormsServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3

- Create a `java.io.FileInputStream` object that represents the PDF form that contains barcoded data by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4

Decode the form data by invoking the `BarcodedFormsServiceClient` object's `decode` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the PDF form.
- A `java.lang.Boolean` object that specifies whether to decode a PDF417 barcode.
- A `java.lang.Boolean` object that specifies whether to decode a data matrix barcode.
- A `java.lang.Boolean` object that specifies whether to decode a QR code barcode.
- A `java.lang.Boolean` object that specifies whether to decode a codabar barcode.
- A `java.lang.Boolean` object that specifies whether to decode a code 128 barcode.
- A `java.lang.Boolean` object that specifies whether to decode a code 39 barcode.
- A `java.lang.Boolean` object that specifies whether to decode an EAN-13 barcode.
- A `java.lang.Boolean` object that specifies whether to decode an EAN-8 barcode.
- A `com.adobe.livecycle.barcodedforms.CharSet` enumeration value that specifies the character set encoding value used in the barcode.

The `decode` method returns an `org.w3c.dom.Document` object that contains decoded form data.

5

Convert the decoded data into either XDP or XFDF data by invoking the `BarcodedFormsServiceClient` object's `extractToXML` method and passing the following values:

- The `org.w3c.dom.Document` object that contains decoded data (ensure that you use the `decode` method's return value).
- A `com.adobe.livecycle.barcodedforms.Delimiter` enumeration value that specifies the line delimiter. It is recommended that you specify `Delimiter.Carriage_Return`.
- A `com.adobe.livecycle.barcodedforms.Delimiter` enumeration value that specifies the field delimiter. For example, specify `Delimiter.Tab`.
- A `com.adobe.livecycle.barcodedforms.XMLFormat` enumeration value that specifies whether to convert the barcode data into XDP or XFDF XML data. For example, specify `XMLFormat.XDP` to convert the data to XDP data.

Note: *Do not specify the same values for the line delimiter and field delimiter parameters.*

The `extractToXML` method returns a `java.util.List` object where each element is an `org.w3c.dom.Document` object. There is a separate element for each barcode that is located on the form. That is, if there are four barcodes on the form, then there are four elements in the returned `java.util.List` object.

6

- Iterate through the `java.util.List` object to get each `org.w3c.dom.Document` object that is located in the list.
- For each element in the list, convert the `org.w3c.dom.Document` object to a `com.adobe.idp.Document` object. (The application logic that converts a `org.w3c.dom.Document` object into a `com.adobe.idp.Document` object is shown in the Decoding barcoded form data using the Java API example).

- Save the XML data as an XML file by invoking the `com.adobe.idp.Document` object's `copyToFile`, and passing a File object that represents the XML file.

See also

"Quick Start (SOAP mode): Decoding barcoded form data using the Java API" on page 64

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Decode barcoded form data using the web service API

Decode form data by using the barcoded forms API(web service):

1

- Create a Microsoft .NET client assembly that consumes the barcoded forms service WSDL. For information, see "Invoking AEM Forms using Base64 encoding" on page 525.
- Reference the Microsoft .NET client assembly. For information, see "Referencing the .NET client assembly" in "Invoking AEM Forms using Base64 encoding" on page 525.

2

Using the Microsoft .NET client assembly that consumes the barcoded forms service WSDL, create an `BarcodedFormsServiceService` object by invoking its default constructor.

3

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that contains a barcode.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.

4

Decode the form data by invoking the `BarcodedFormsServiceService` object's `decode` method and passing the following values:

- The `BLOB` object that contains the PDF form.
- A `Boolean` object that specifies whether to decode a PDF417 barcode.
- A `Boolean` object that specifies whether to decode a data matrix barcode.
- A `Boolean` object that specifies whether to decode a QR code barcode.
- A `Boolean` object that specifies whether to decode a codabar barcode.
- A `Boolean` object that specifies whether to decode a code 128 barcode.
- A `Bolean` object that specifies whether to decode a code 39 barcode.
- A `Boolean` object that specifies whether to decode an EAN-13 barcode.
- A `Boolean` object that specifies whether to decode an EAN-8 barcode.

- A `CharSet` enumeration value that specifies the character set encoding value used in the barcode.

The `decode` method returns a string value that contains decoded form data.

5

Convert the decoded data into either XDP or XFDF data by invoking the `BarcodedFormsServiceService` object's `extractToXML` method and passing the following values:

- A string value that contains decoded data (ensure that you use the `decode` method's return value).
- A `Delimiter` enumeration value that specifies the line delimiter. It is recommended that you specify `Delimiter.Carriage_Return`.
- A `Delimiter` enumeration value that specifies the field delimiter. For example, specify `Delimiter.Tab`.
- A `XMLFormat` enumeration value that specifies whether to convert the barcode data into XDP or XFDF XML data. For example, specify `XMLFormat.XDP` to convert the data to XDP data.

Note: *Do not specify the same values for the line delimiter and field delimiter parameters.*

The `extractToXML` method returns an `Object` array where each element is an `BLOB` instance. There is a separate element for each barcode that is located on the form. That is, if there are four barcodes on the form, then there are four elements in the returned `Object` array.

6

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the secured PDF document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `encryptPDFUsingPassword` method. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

Quick Start (Base64): Decoding barcoded form data using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Assigning Usage Rights

About the Acrobat Reader DC extensions Service

The Acrobat Reader DC extensions service enables your organization to easily share interactive PDF documents by extending the functionality of Adobe Reader. The Acrobat Reader DC extensions service fully supports any PDF document, up to and including PDF 1.7. It works with Adobe Reader 7.0 and later. The service adds usage rights to a PDF document, activating features that are not usually available when a PDF document is opened using Adobe Reader. Third party users do not require additional software or plug-ins to work with the rights-enabled documents.

You can accomplish these tasks using the Acrobat Reader DC extensions service:

- Apply usage rights to PDF documents. For information, see "Applying Usage Rights to PDF Documents" on page 751.

- Remove usage rights from PDF documents. For information, see "Removing Usage Rights from PDF Documents" on page 755.
- Retrieve credential details. For information, see "Retrieving Credential Information" on page 758.

Note: *For more information about the Acrobat Reader DC extensions service, see Services Reference for AEM Forms.*

Applying Usage Rights to PDF Documents

You can apply usage rights to PDF documents using the Acrobat Reader DC extensions Java Client API and web service. Usage rights pertain to functionality that is available by default in Acrobat but not in Adobe Reader, such as the ability to add comments to a form or to fill in form fields and save the form. PDF documents that have usage rights applied to them are called rights-enabled documents. A user who opens a rights-enabled document in Adobe Reader can perform operations that are enabled for that specific document.

Note: *When applying usage rights to PDF documents using the* `applyUsageRights` *method, which is part of the Java API, you can set the* `isModeFinal` *parameter of the* `ReaderExtensionsOptionSpec` *object to* `false`*. This results in the forms processed counter not being updated and an improvement in performance. If you are not concerned about updating the forms processed counter, it is recommended that you set the* `isModeFinal` *parameter to* `false`*.*

Note: *For more information about the Acrobat Reader DC extensions service, see Services Reference for AEM Forms.*

Summary of steps

To apply usage rights to a PDF document, perform the following steps:

1. Include project files.
2. Create a Acrobat Reader DC extensions Client object.
3. Retrieve a PDF document.
4. Specify usage rights to apply.
5. Apply usage rights to the PDF document.
6. Save the rights-enabled PDF document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Acrobat Reader DC extensions Client object

To programmatically perform a Acrobat Reader DC extensionsservice operation, you must create a Acrobat Reader DC extensions service client object. If you are using the Acrobat Reader DC extensions Java API, create a `ReaderExtensionsServiceClient` object. If you are using the Acrobat Reader DC extensions web service API, create a `ReaderExtensionsServiceService` object.

Retrieve a PDF document

You must retrieve a PDF document in order to apply usage rights. Rights-enabled PDF documents contain a usage rights dictionary. When Adobe Reader opens a document containing such a dictionary, it enables the usage rights specified in the dictionary for that document only. If the document does not contain a usage rights dictionary, the Acrobat Reader DC extensions service creates one. If it already contains a dictionary, the Acrobat Reader DC extensions service overwrites existing usage rights with the ones you specify. The dictionary specifies which usage rights are enabled. When a user opens the document in Adobe Reader, only the usage rights specified in the dictionary are permitted.

Specify usage rights to apply

The usage rights that you can set are determined by a credential that you purchase from Adobe Systems Incorporated. Credentials typically provide permission to set a group of related usage rights, such as those pertaining to interactive forms. Each credential provides the right to create a certain number of rights-enabled PDF documents. An evaluation credential gives the right to create an unlimited number of draft documents.

Note: *If you attempt to assign a usage right that is not permitted by your credential, you will cause an exception.*

Apply usage rights to the PDF document

To apply usage rights to a PDF document, you reference the alias of the credential that you are using to apply usage rights (a credential is typically installed during the installation of AEM Forms). Also you must specify the PDF document to which usage rights is applied. For information abut configuring a credential, see the installing and deploying guide for your application server.

Save the rights-enabled PDF document

After the Acrobat Reader DC extensions service applies usage rights to a PDF document, you can save the rights-enabled PDF document as a PDF file.

See also

"Apply usage rights using the Java API" on page 752

"Apply usage rights using the web service API" on page 753

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Acrobat Reader DC extensions Service Java API Quick Start(SOAP)" on page 297

Apply usage rights using the Java API

Apply usage rights to a PDF document by using the Acrobat Reader DC Extensions API (Java):

1 Include project files

Include client JAR files, such as adobe-reader-extensions-client.jar, in your Java project's class path.

2 Create a Acrobat Reader DC extensions Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ReaderExtensionsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve a PDF document.

- Create a `java.io.FileInputStream` object that represents the PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Specify usage rights to apply.

- Create a `UsageRights` object that represents usage rights by using its constructor.
- For each usage right to apply, invoke a corresponding method that belongs to the `UsageRights` object. For example, to add the `enableFormFillIn` usage right, invoke the `UsageRights` object's `enableFormFillIn` method and pass `true`. (Repeat this step for each usage right to apply).

5 Apply usage rights to the PDF document.

- Create a `ReaderExtensionsOptionSpec` object by using its constructor. This object contains run-time options that are required by the Acrobat Reader DC extensions service. When invoking this constructor, you must specify the following values:
  - The `UsageRights` object that contains the usage rights to apply to the document.
  - A string value that specifies a message that a user sees when the rights-enabled PDF document is opened in Adobe Reader 7.x. This message is not displayed in Adobe Reader 8.0.
- Apply usage rights to the PDF document by invoking the `ReaderExtensionsServiceClient` object's `applyUsageRights` method and passing the following values:
  - The `com.adobe.idp.Document` object that contains the PDF document to which usage rights is applied.
  - A string value that specifies the alias of the credential that enables you to apply usage rights.
  - A string value that specifies the corresponding password value. (Currently this parameter is ignored. You can pass `null`.)
- The `ReaderExtensionsOptionSpec` object that contains run-time options.

The `applyUsageRights` method returns a `com.adobe.idp.Document` object that contains the rights-enabled PDF document.

6 Save the rights-enabled PDF document.

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file (ensure that you use the `com.adobe.idp.Document` object that was returned by the `applyUsageRights` method).

See also

"Applying Usage Rights to PDF Documents" on page 751

"Quick Start (SOAP mode):Applying usage rights using the Java API" on page 297

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Apply usage rights using the web service API

Apply usage rights to a PDF document by using the Acrobat Reader DC Extensions API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/ReaderExtensionsService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Acrobat Reader DC extensions Client object.

- Create a `ReaderExtensionsServiceClient` object by using its default constructor.
- Create a `ReaderExtensionsServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/ReaderExtensionsService?blob=mtom`. Ensure you specify `?blob=mtom`.)

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `ReaderExtensionsServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `ReaderExtensionsServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `ReaderExtensionsServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document to which a usage rights is applied.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Specify usage rights to apply.

- Create a `UsageRights` object that represents usage rights by using its constructor.
- For each usage right to apply, assign the value `true` to the corresponding data member that belongs to the `UsageRights` object. For example, to add the `enableFormFillIn` usage right, assign `true` to the `UsageRights` object's `enableFormFillIn` data member. (Repeat this step for each usage right to apply).

5 Apply usage rights to the PDF document.

- Create a `ReaderExtensionsOptionSpec` object by using its constructor. This object contains run-time options that are required by the Acrobat Reader DC extensions service.
- Assign the `UsageRights` object to the `ReaderExtensionsOptionSpec` object's `usageRights` data member.
- Assign a string value that specifies the message that a user sees when the rights-enabled PDF document is opened in Adobe Reader to the `ReaderExtensionsOptionSpec` object's `message` data member.
- Apply usage rights to the PDF document by invoking the `ReaderExtensionsServiceClient` object's `applyUsageRights` method and passing the following values:
  - The `BLOB` object that contains the PDF document to which usage rights is applied.
  - A string value that specifies the alias of the credential that enables you to apply usage rights.
  - A string value that specifies the corresponding password value. (Currently this parameter is ignored. You can pass `null`.)
- The `ReaderExtensionsOptionSpec` object that contains run-time options.

The `applyUsageRights` method returns a `BLOB` object that contains the rights-enabled PDF document.

6 Save the rights-enabled PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the rights-enabled PDF document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `applyUsageRights` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Applying Usage Rights to PDF Documents" on page 751

Quick Start (MTOM): Applying usage rights using the web service API

Quick Start (SwaRef): Applying usage rights using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Removing Usage Rights from PDF Documents

You can remove usage rights from a rights-enabled document. Removing usage-rights from a rights-enabled PDF document is also necessary in order to perform other AEM Forms operations on it. For example, you must digitally sign (or certify) a PDF document before you set usage rights. Therefore if you want to perform operations on a rights-enabled document, you must remove usage rights from the PDF document, perform the other operations, such as digitally signing the document, and then re-apply usage rights to the document.

Note: *For more information about the Acrobat Reader DC extensions service, see Services Reference for AEM Forms.*

Summary of steps

To remove usage rights from a rights-enabled PDF document, perform the following steps:

1 Include project files.

2 Create a Acrobat Reader DC extensions Client object.

3 Retrieve a rights-enabled PDF document.

4 Remove usage rights from the PDF document.

5 Save the PDF document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Acrobat Reader DC extensions Client object

Before you can programmatically perform a Acrobat Reader DC extensions service operation, you must create a Acrobat Reader DC extensions service client object. If you are using the Java API, create a `ReaderExtensionsServiceClient` object. If you are using the Acrobat Reader DC extensions web service API, create a `ReaderExtensionsServiceService` object.

Retrieve a rights-enabled PDF document

Retrieve a rights-enabled PDF document in order to remove usage rights.

Remove usage rights from the PDF document

After you retrieve a rights-enabled PDF document, you can remove usage rights. After you remove usage rights, the PDF document will not have any additional functionality while viewed within Adobe Reader.

Save the PDF document

You can save the PDF document that no longer contains usage-rights as a PDF file. Once saved as a PDF file, the PDF document can be viewed in Adobe Reader or Acrobat.

See also

"Remove usage rights using the Java API" on page 756

"Remove usage rights using the web service API" on page 757

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Acrobat Reader DC extensions Service Java API Quick Start(SOAP)" on page 297

"Applying Usage Rights to PDF Documents" on page 751

Remove usage rights using the Java API

Remove usage rights from a rights-enabled PDF document by using the Acrobat Reader DC extensions API (Java):

1 Include project files.

   Include client JAR files, such as adobe-reader-extensions-client.jar, in your Java project's class path.

2 Create a Acrobat Reader DC extensions Client object.

   Create a `ReaderExtensionsServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Retrieve a PDF document.

   - Create a `java.io.FileInputStream` object that represent the rights-enabled PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Remove usage rights from the PDF document.

   Remove usage rights from the PDF document by invoking the `ReaderExtensionsServiceClient` object's `removeUsageRights` method and passing the `com.adobe.idp.Document` object that contains the rights-enabled PDF document. This method returns a `com.adobe.idp.Document` object that contains a PDF document that does not have usage rights.

5 Apply usage rights to the PDF document.

- Create a `java.io.File` object and ensure that the file extension is .PDF.
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `removeUsageRights` method).

See also

"Removing Usage Rights from PDF Documents" on page 755

Quick Start (SOAP mode): Removing usage rights from a PDF document using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Remove usage rights using the web service API

Remove usage rights from a rights-enabled PDF document by using the Acrobat Reader DC extensions API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/ReaderExtensionsService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Acrobat Reader DC extensions Client object.

- Create a `ReaderExtensionsServiceClient` object by using its default constructor.
- Create a `ReaderExtensionsServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/ReaderExtensionsService?blob=mtom`. Ensure you specify `?blob=mtom`.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `ReaderExtensionsServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `ReaderExtensionsServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `ReaderExtensionsServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the rights-enabled PDF document from which usage rights are removed.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Remove usage rights from the PDF document.

Remove usage rights from the PDF document by invoking the `ReaderExtensionsServiceClient` object's `removeUsageRights` method and passing the `BLOB` object that contains the rights-enabled PDF document. This method returns a `BLOB` object that contains a PDF document that does not have usage rights.

5 Apply usage rights to the PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the PDF file location.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `removeUsageRights` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

See also

"Removing Usage Rights from PDF Documents" on page 755

Quick Start (MTOM): Removing usage rights from a PDF document using the web service API

Quick Start (SwaRef): Removing usage rights from a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Retrieving Credential Information

You can retrieve information about the credential that was used to apply usage rights to a rights-enabled PDF document. By retrieving information about a credential, you can obtain information such as the date after which the certificate is no longer valid.

Note: *For more information about the Acrobat Reader DC extensions service, see Services Reference for AEM Forms.*

Summary of steps

To retrieve information about the credential that was used to apply usage rights to a PDF document, perform the following steps:

1 Include project files.

2 Create a Acrobat Reader DC extensions Client object.

3 Retrieve a rights-enabled PDF document.

4 Retrieve information about the credential.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Acrobat Reader DC extensions Client object

Before you can programmatically perform a Acrobat Reader DC extensions service operation, you must create a Acrobat Reader DC extensions service client object. If you are using the Java API, create a `ReaderExtensionsServiceClient` object. If you are using the Acrobat Reader DC extensions web service API, create a `ReaderExtensionsServiceService` object.

Retrieve a rights-enabled PDF document

You must retrieve a rights-enabled PDF document in order to retrieve information about the credential. You can also retrieve information about a credential by specifying its alias; however, if you want to retrieve information about a credential that was used to apply usage rights to a specific rights-enabled PDF document, then you must retrieve the document.

Retrieve information about the credential

After you retrieve a rights-enabled PDF document, you can obtain information about the credential that was used to apply usage rights to it. You can obtain the following information about the credential:

- The message that is displayed within Adobe Reader when the rights-enabled PDF document is opened.
- The date after which the credential is no longer valid.
- The date before which the credential is not valid.
- The usage rights that were set for this rights-enabled PDF document.
- The number of times that the credential has been used.

See also

"Remove usage rights using the Java API" on page 756

"Remove usage rights using the web service API" on page 757

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Acrobat Reader DC extensions Service Java API Quick Start(SOAP)" on page 297

Retrieve credential information using the Java API

Retrieve credential information by using the Acrobat Reader DC extensions API (Java):

1 Include project files.

   Include client JAR files, such as adobe-reader-extensions-client.jar, in your Java project's class path.

2 Create a Acrobat Reader DC extensions Client object.

   Create a `ReaderExtensionsServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Retrieve a PDF document.

   - Create a `java.io.FileInputStream` object that represent the rights-enabled PDF document by using its constructor and passing a string value that specifies the location of the rights-enabled PDF document.

- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Remove usage rights from the PDF document.

- Retrieve information about the credential used to apply usage-rights to the PDF document by invoking the `ReaderExtensionsServiceClient` object's `getDocumentUsageRights` method and passing the `com.adobe.idp.Document` object that contains the rights-enabled PDF document. This method return a `GetUsageRightsResult` object that contains credential information.
- Retrieve the date after which the credential is no longer valid by invoking the `GetUsageRightsResult` object's `getNotAfter` method. This method returns a `java.util.Date` object that represents the date after which the credential is no longer valid.
- Retrieve the message that is displayed in Adobe Reader when the rights-enabled PDF document is opened by invoking the `GetUsageRightsResult` object's `getMessage` method. This method returns a string value that represents the message.

See also

"Retrieving Credential Information" on page 758

Quick Start (SOAP mode): Retrieving credential information using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve credential information using the web service API

Retrieve credential information using the Acrobat Reader DC extensions API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/ReaderExtensionsService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Acrobat Reader DC extensions Client object.

- Create an `ReaderExtensionsServiceClient` object by using its default constructor.
- Create a `ReaderExtensionsServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/ReaderExtensionsService?blob=mtom`. Ensure you specify `?blob=mtom`.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `ReaderExtensionsServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `ReaderExtensionsServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `ReaderExtensionsServiceClient.ClientCredentials.UserName.Password`.

- Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a rights-enabled PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the rights-enabled PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Remove usage rights from the PDF document.

- Retrieve information about the credential used to apply usage-rights to the PDF document by invoking the `ReaderExtensionsServiceClient` object's `getDocumentUsageRights` method and passing the `com.adobe.idp.Document` object that contains the rights-enabled PDF document. This method returns a `GetUsageRightsResult` object that contains credential information.
- Retrieve the date after which the credential is no longer valid by getting the value of the `GetUsageRightsResult` object's `notAfter` data member. The data type of this data member is `System.DateTime`.
- Retrieve the message that is displayed when the rights-enabled PDF document is opened in Adobe Reader by getting the value of the `GetUsageRightsResult` object's `message` data member. The data type of this data member is a string.
- Retrieve the number of times that the credential is used by getting the value of the `GetUsageRightsResult` object's `useCount` data member. The data type of this data member is an integer.

See also

"Retrieving Credential Information" on page 758

Quick Start (MTOM): Retrieving credential information using the web service API

Quick Start (SwaRef): Retrieving credential information using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Preparing AEM Forms for Backup

About the Backup and Restore Service

The Backup and Restore service lets you put AEM Forms into *backup mode*, which enables hot backups to be performed. The Backup and Restore service does not actually perform a backup of AEM Forms or restore your system. Instead, it puts your server in a state for consistent and reliable backups while allowing your server to continue to run. You are responsible for the actions to back up the Global Document Storage (GDS) and the database connected to the forms server. The GDS is a directory used to store files used within a long-lived process.

Backup mode is a state that the server enters so that files in the GDS are not being purged while a backup procedure is taking place. Instead, subdirectories are created under the GDS directory to maintain a record of files to be purged after save backup mode ends. A file is intended to survive system restarts and can span days, or even years. These files are a critical part of the overall state of the forms server and may include PDF files, policies, or form templates. If any of these files are lost or become corrupted, the processes on the forms server may become unstable and data could be lost.

You can choose to perform snapshot backups, where you would usually enter backup mode for a period and then leave backup mode after you complete your backup activities. Leaving backup mode is required so that files can be purged from the GDS to ensure that it does not grow unnecessarily large. You can either leave backup mode explicitly or wait for the time to expire on a backup mode session.

You can also leave your server in perpetual backup mode, which is typical for backup strategies for rolling backups or continuous system coverage. Rolling backup mode indicates that the system is always in backup mode, with a new backup mode session initiated as soon as the previous session is released. When in continuous backup mode, a file is purged after two backup mode session and is no longer referenced.

You can use the Backup and Restore service to add to existing applications or new applications that you create to perform backups of the GDS or database connected to the forms server.

Important: *As with any other aspect of your AEM Forms implementation, your backup and recovery strategy should be developed and tested in a development or staging environment before being used in production to ensure that the entire solution is working as expected with no data loss.*

You can perform these tasks using the Backup and Restore service:

- Enter backup mode.
- Leave backup mode.

Note: *For more information about what to consider when performing backups for AEM Forms, see administration help.*

Note: *For more information about the Backup and Restore service, see Services Reference for AEM Forms.*

Entering Backup Mode on the forms server

You enter backup mode to allow for hot backups of a forms server. When you enter backup mode, you specify the following information based on your organization's backup procedures:

- A unique label to identify the backup mode session that may be useful for your backup processes.
- The time for the backup procedure to complete.
- A flag to indicate whether to be in continuous backup mode, which is useful only if you are performing rolling backups.

Before you write applications to enter into backup mode, it is recommended that you understand the backup procedures that will be used after you put the forms server in backup mode. For more information about what to consider when performing backups for AEM Forms, see administration help.

Note: *For more information about the Backup and Restore service, see Services Reference for AEM Forms.*

Summary of steps

To create an application that enters backup mode, perform the following steps:

1. Include project files.
2. Create an BackupService client object.
3. Determine a unique label, the amount of time to perform the backup, and whether to be in continuous backup mode.

4 Enter backup mode.

5 (Optional) Retrieve information about the backup mode session on the server.

6 Perform the backup of the GDS (Global Data Store) and database.

Include project files

Include necessary files in your development project. These files are important to include in your project for compiling your code properly and using the Backup and Restore Service API.

For information about the location of these files, see "Including AEM Forms Java library files" on page 491.

Create a BackupService Client API object

To programmatically leave backup mode, you create a BackupService client object to use the Backup and Restore Service API.

Decide upon a unique label, determine the amount of time to perform the backup, and decide whether to be in continuous backup mode

Before you enter backup mode, you should decide upon a unique label, determine the amount of time that you want to allocate to perform the backup, and decide whether you want the forms server to stay in backup mode. These considerations are important to integrate with the backup procedures established by your organization. (See administration help.)

Enter backup mode

Enter backup mode with the parameters that are consistent with the backup procedures at your organization.

Retrieve information about the backup mode session on the server

After you enter backup mode, you can retrieve information about the session. This information can be used to integrate with your backup procedures

Perform the backup of the GDS and database

After you successfully enter backup mode, you can perform a backup of the Global Document Storage (GDS) and the database that the forms server is connected to. This step is specific to your organization, since you can perform this step manually or you can run other tools to perform the backup procedure.

Enter backup mode using the Java API

Enter backup mode by using the Backup and Restore Service API:

1 Include project files

Include necessary client JAR files, such as adobe-backup-restore-client-sdk.jar, in your Java project's class path. To create the Java client application, the following JAR files must be added to your project's class path:

- adobe-backup-restore-client-sdk.jar
- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

2 Create a BackupService Client API object

You use a `ServiceClientFactory` object and the BackupService client API object together.

- Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
- Create an `BackupService` object by using its constructor and passing the `ServiceClientFactory` object.

3 Decide upon a unique label, determine the amount of time to perform the backup, and decide whether to be in continuous backup mode

Decide upon a unique label, determine the amount of time that you want to allocate to perform the backup, and decide whether you want the forms server to stay in continuous backup mode.

4 Enter backup mode

Enter backup mode by invoking the `enterBackupMode` method with the following parameters:

- A `String` value that specifies a unique human-readable label that identifies the backup mode session. It is recommended that you do not use spaces or characters that cannot be encoded into XML format.
- An `int` value that specifies the number of minutes to stay in backup mode. You can specify a value from `1` to `10080` (the number of minutes in one week). This value is ignored when using continuous backup mode.
- A `Boolean` value that specifies whether to be in continuous backup mode. A value of `True` specifies to be in continuous backup mode. When in continuous backup mode, the value you specify for the number of minutes to stay in backup mode is ignored.

  Continuous backup mode means that a new backup mode session is started after the current one is completed. A value of `False` means that continuous backup mode is not used and, after leaving backup mode, the purging of files from the GDS resumes.

5 Retrieve information about the backup mode session on the server

Retrieve information using the `BackupModeEntryResult` object that is returned after invoking the `enterBackupMode` method. The information that you can retrieve after you enter backup mode may be useful for integrating with your backup procedures. For example, the label, backup ID, and start time may be useful as input for filenames for your backup procedure.

6 Perform the backup of the GDS and database

Backup the Global Document Storage (GDS) and the database which your forms server is connected to. The actions to perform the backup are not part of the AEM Forms SDK and may even include manual steps specific to the backup procedures in your organization.

Enter backup mode using the web service API

Enter backup mode by using the web service provided by Backup and Restore Service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Backup and Restore Service API WSDL.
- Reference the Microsoft .NET client assembly.

2 Create a BackupService Client API object

Using the Microsoft .NET client assembly, create a `BackupServiceService` object by invoking its default constructor and specify the credentials using the `Credentials` method.

3 Decide upon a unique label, determine the amount of time to perform the backup, and decide whether to be in continuous backup mode

Decide upon a unique label, determine the amount of time that you want to allocate to perform the backup, and decide whether you want the forms server to stay in continuous backup mode.

4 Enter backup mode

To enter backup mode, invoke the enterBackupMode method and pass the following values:

- A `String` value that specifies a unique human-readable label that identifies the backup mode session. It is recommended that you do not use spaces or characters that cannot be encoded into XML format.
- A `Uint32` value that specifies the number of minutes to stay in backup mode. You can specify a value from `1` to `10080` (number of minutes in one week). This value is ignored when using continuous backup mode.
- A `Boolean` value that specifies whether to be in continuous backup mode. A value of `True` specifies to be in continuous backup mode. When in continuous backup mode, the value you specify for the number of minutes to stay in backup mode is ignored. Continuous backup mode means that a new backup mode session is started after the current one is completed.

  A value of `False` means that continuous backup mode is not used and, after leaving backup mode, the purging of files from the GDS resumes.

5 Retrieve information about the backup mode session on the server

Retrieve information about the backup mode session after invoking the enterBackupMode method from the BackupModeEntryResult that is returned to verify that it was successful. The information that you can retrieve after you enter backup mode may be useful for integrating with your backup procedures. For example, the label, backup ID, and start time may be useful as input for filenames for your backup procedure.

6 Perform the backup of the GDS and database

Backup the Global Document Storage (GDS) and the database which your forms server is connected to. The actions to perform the backup are not part of the AEM Forms SDK and may even include manual steps specific to the backup procedures in your organization.

Leaving Backup Mode on the forms server

You leave backup mode so that the forms server resumes purging of files from the GDS (Global Document Storage) on the forms server.

Before you write applications to enter into leave mode, it is recommended that you understand the backup procedures that are used with AEM Forms. For more information about what to consider when performing backups for AEM Forms, see administration help.

Note: *For more information about the Backup and Restore service, see Services Reference for AEM Forms.*

Summary of steps

To leave backup mode, perform the following steps:

1 Include project files.

2 Create a BackupService client object.

3 Leave backup mode.

4 (Optional) Retrieve information about the backup mode session that was running on the forms server.

Include project files

Include all necessary files in your development project. These files are important for compiling your code properly and using the Backup and Restore Service API.

For information about the location of these files, see "Including AEM Forms Java library files" on page 491.

Create a BackupService Client API object

To programmatically leave backup mode, you create a BackupService client object to use the Backup and Restore Service API.

Leave backup mode

Leave backup mode to resume normal purging of files from the Global Document Storage (GDS). Before you leave backup mode, you should verify that your backup procedures have been completed.

Retrieve information about the backup mode session that ended

After you leave backup mode, you can retrieve information about the session. This information can be used to integrate with your backup procedures.

Leave backup mode using the Java API

Leave backup mode by using the Backup and Restore Service API (Java):

1 Include project files

Include necessary client JAR files, such as adobe-backup-restore-client-sdk.jar, in your Java project's class path. To create Java client application, the following JAR files must be added to your project's class path:

- adobe-backup-restore-client-sdk.jar
- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

2 Create a BackupService Client API object

You use a `ServiceClientFactory` object and the BackupService client API object together.

- Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
- Create a `BackupService` object by using its constructor and passing the `ServiceClientFactory` object as parameter.

3 Enter backup mode

Leave backup mode by invoking the `leaveBackupMode` method.

4 Retrieve information about the backup mode session on the server

Retrieve information about the operation using the `BackupModeResult` object that is returned. The information that you can retrieve after you enter backup mode may be useful for integrating with your backup procedures. For example, the label, backup ID, and start time may be useful as input for filenames for your backup procedure.

Leave backup mode using the web service API

Leave backup mode by using the Backup and Restore Service API (web service):

1 Include project files

To use web services, you must make sure that you include the proxy files. Follow these steps to configure your project to use the Backup and Restore Service API as a web service.

- Create a Microsoft .NET client assembly that consumes the Backup and Restore Service API WSDL.
- Reference the Microsoft .NET client assembly.

2 Create a BackupService Client API object

Using the Microsoft .NET client assembly, create a `BackupServiceService` object by invoking its default constructor.

3 Enter backup mode

Leave backup mode by invoking the `leaveBackupMode` web service operation.

4 Retrieve information about the backup mode session on the server

Retrieve the backup mode identifier after the operation to verify that it was successful. The information that you can retrieve after you leave backup mode may be useful for integrating with your backup procedures.

More Help topics

"Quick Start (SOAP mode): Entering backup mode using the Java API" on page 59

"Quick Start (SOAP mode): Leaving backup mode using the Java API" on page 62

Converting Postscript to PDF Documents

About the Distiller Service

The Distiller® service converts PostScript®, Encapsulated PostScript (EPS), and PRN files to compact, reliable, and more secure PDF files over a network. The Distiller service is frequently used to convert large volumes of print documents to electronic documents, such as invoices and statements. Converting documents to PDF also allows enterprises to send their customers a paper version and an electronic version of a document.

Note: *For more information about the Distiller service, see Services Reference for AEM Forms.*

Converting PostScript to PDF documents

This topic describes how you can use the Distiller Service API (Java and web service) to programmatically convert PostScript (PS), Encapsulated PostScript (EPS), and PRN files to PDF documents.

Note: *For more information about the Distiller service, see Services Reference for AEM Forms.*

Note: *To convert PostScript files to PDF documents, one of the following needs to be installed on the server hosting AEM Forms: Acrobat 9 or Microsoft Visual C++ 2005 redistributable package.*

Summary of steps

To convert any of the supported types to a PDF document, perform the following steps:

1 Include project files.

2 Create a Distiller service client.

3 Retrieve the file to convert.

4 Invoke the PDF creation operation.

5 Save the PDF document.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure you include the proxy files.

Create a Distiller service client

Before you can programmatically perform a Distiller service operation, you must create a Distiller service client. If you are using the Java API, create a `DistillerServiceClient` object. If you are using the web service API, create a `DistillerServiceService` object.

Retrieve the file to convert

You must retrieve the file that you want to convert. For example, to convert a PS file to a PDF document, you must retrieve the PS file.

Invoke the PDF creation operation

After you create the service client, you can then invoke the PDF creation operation. This operation will need information about the document to be converted, including the path to the target document.

Save the PDF document

You can save the PDF document as a PDF file.

See also

"Convert a PostScript file to PDF using the Java API" on page 768

"Converting a PostScript file to PDF using the web service API" on page 769

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Output Service Java API Quick Start(SOAP)" on page 239

Convert a PostScript file to PDF using the Java API

Convert a PostScript file to PDF document by using the Distiller Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-distiller-client.jar, in your Java project's class path.

2 Create a Distiller service client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `DistillerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the file to convert.

- Create a `java.io.FileInputStream` object that represents the file to convert by using its constructor and passing a string value that specifies the location of the file.

- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Invoke the PDF creation operation.

Invoke the `DistillerServiceClient` object's `createPDF` method and pass the following values:

- The `com.adobe.idp.Document` object that represents the PS, EPS, or PRN file to be converted
- A `java.lang.String` object that contains the name of the file to be converted
- A `java.lang.String` object that contains the name of the Adobe PDF settings to be used
- A `java.lang.String` object that contains the name of the security settings to be used
- An optional `com.adobe.idp.Document` object that contains settings to be applied while generating the PDF document
- An optional `com.adobe.idp.Document` object that contains metadata information to be applied to the PDF document

The `createPDF` method returns a `CreatePDFResult` object that contains the new PDF document and a log file that may be generated. The log file typically contains error or warning messages that are generated by the conversion request.

5 Save the PDF document.

To obtain the newly created PDF document, perform the following actions:

- Invoke the `CreatePDFResult` object's `getCreatedDocument` method. This returns a `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document.

Similarly, to obtain the log document, perform the following actions.

- Invoke the `CreatePDFResult` object's `getLogDocument` method. This returns a `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the log document.

See also

"Summary of steps" on page 767

"Quick Start (SOAP mode): Converting a PostScript file to a PDF document using the Java API" on page 90

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Converting a PostScript file to PDF using the web service API

Convert a PostScript file to PDF document by using the Distiller Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/DistillerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Distiller service client.

- Create a `DistillerServiceClient` object by using its default constructor.

- Create a `DistillerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/DistillerService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. However, specify `?blob=mtom` to use MTOM.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DistillerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DistillerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DistillerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the file to convert.

- Create a `BLOB` object by using its constructor. This `BLOB` object is used to store the file to convert to a PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Invoke the PDF creation operation.

Invoke the `DistillerServiceService` object's `CreatePDF2` method and pass the following required values:

- The `BLOB` object that represents the PS file to convert
- A string that contains the path name of the file to be convert
- A string object that contains the Adobe PDF settings to be used (for example, `Standard`)
- A string object that contains the security settings to be used (for example, `No Security`)
- An optional `BLOB` object that contains settings to be applied while generating the PDF document
- An optional `BLOB` object that contains metadata information to be applied to the PDF document
- A `BLOB` output parameter used to store the PDF document
- A `BLOB` output parameter used to store the log

5 Save the PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the signed PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `CreatePDF2` method (the output parameter). Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 767

Quick Start (MTOM): Converting a PostScript file to a PDF document using the web service API

Quick Start (SwaRef): Converting a PostScript file to a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Converting PDF to Postscript and Image Files

About the Convert PDF Service

The Convert PDF service converts PDF documents to PostScript and to a number of image formats (JPEG, JPEG 2000, PNG, and TIFF). Converting a PDF document to PostScript is useful for unattended server-based printing on any PostScript printer. Converting a PDF document to a multipage TIFF file is practical when archiving documents in content management systems that do not support PDF documents.

You can accomplish these tasks using the Convert PDF service:

- Convert PDF documents to PostScript.
- Convert PDF documents to image formats.

Note: *For more information about the Convert PDF service, see Services Reference for AEM Forms.*

Converting PDF Documents to PostScript

This topic describes how you can use the Convert PDF Service API (Java and web service) to programmatically convert PDF documents to PostScript files. The PDF document that is converted to a PostScript file must be a non-interactive PDF document. That is, if you attempt to convert an interactive PDF document to a PostScript file, an exception is thrown.

Note: *For more information about the Convert PDF service, see Services Reference for AEM Forms.*

Summary of steps

To convert a PDF document to a PostScript file, perform the following steps:

1 Include project files.

2 Create a Convert PDF service client.

3 Reference the PDF document to convert to a PostScript file.

4 Set conversion run-time options.

5 Convert the PDF document to a PostScript file.

6 Save the PostScript file.

Include project files

Include the necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure you include the proxy files.

Create a Convert PDF client

Before you can programmatically perform a Convert PDF service operation, you must create a Convert PDF service client. If you are using the Java API, create a `ConvertPdfServiceClient` object. If you are using the web service API, create a `ConvertPDFServiceService` object.

This section uses web service functionality that is introduced in AEM Forms. To access new functionality, you have to construct your proxy object using the `lc_version` attribute. (See "Accessing new functionality using web services" in "Invoking AEM Forms using Web Services" on page 514.)

Reference the PDF document to convert to a PostScript file

Reference the PDF document that you want to convert to a PostScript file. As stated earlier in this topic, the PDF document must be a non-interactive PDF document. If you attempt to convert an interactive PDF document to a PostScript file, an exception is thrown.

Set conversion run-time options

When converting a PDF document to a PostScript file, you can define run-time options that specify the PostScript type that is created. For example, you can define a level 3 PostScript file.

Typically, the generated PostScript file will reflect the size of input PDF document. If you select the `ShrinkToFit` option (which shrinks the output of the PostScript file to fit the page), you will not see a difference between the input PDF document and the generated PostScript file. The `ShrinkToFit` option takes effect only if you select to print on a smaller page size than the input PDF document. To select a smaller page size, define the `PageSize` option. In addition, it is recommended that you set the `RotateAndCenter` option to `true` to obtain the correct PostScript output.

Likewise, if you select the `ExpandToFit` option (which expands the output of the PostScript file to fit the page), it takes effect only if you select to print on a larger page size than the input PDF document. To select a larger page size, define the `PageSize` option. In addition, it is recommended that you set the `RotateAndCenter` option to `true` to obtain the correct PostScript output.

Note: *For information about the run-time values that you can set, see the `ToPSOptionsSpec` class reference in AEM Forms API Reference.*

Convert the PDF document to a PostScript file

After you create the service client and set run-time options, you can invoke the PostScript conversion operation. This operation will need information about the document to convert, including the preferred PostScript level for the target document.

Save the PostScript file

After you convert the PDF document to PostScript, you can save the output as a PostScript file.

See also

"Convert a PDF document to PS using the Java API" on page 773

"Convert a PDF document to PS using the web service API" on page 774

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Convert PDF Service Java API Quick Start(SOAP)" on page 80

Convert a PDF document to PS using the Java API

Convert a PDF document to PostScript by using the Convert PDF Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-convertpdf-client.jar, in your Java project's class path.

2 Create a Convert PDF client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ConvertPdfServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference the PDF document to convert to a PostScript file.

- Create a `java.io.FileInputStream` object by using its constructor and pass a string value that specifies the location of the PDF document to convert.
- Create a `com.adobe.idp.Document` object that stores the PDF document by using the `com.adobe.idp.Document` constructor. Pass the `java.io.FileInputStream` object that contains the PDF document.

4 Set conversion run-time options.

- Create a `ToPSOptionsSpec` object by invoking its constructor.
- Set run-time options by invoking an appropriate method that belongs to the `ToPSOptionsSpec` object. For example, to define the PostScript level that is created, invoke the `ToPSOptionsSpec` object's `setPsLevel` method and pass a `PSLevel` enumeration value that specifies the PostScript level. For information about all run-time values that you can set, see the `ToPSOptionsSpec` class reference in AEM Forms API Reference.

5 Convert the PDF document to a PostScript file.

Invoke the `ConvertPdfServiceClient` object's `toPS2` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the PDF document to convert to a PostScript file.
- A `ToPSOptionsSpec` object that specifies PostScript run-time options.

The `toPS2` method returns a `Document` object that contains the new PostScript document.

6 Save the PostScript file.

- Create a `java.io.File` object and ensure that the file name extension is .ps.
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `toPS2` method).

See also

"Summary of steps" on page 771

"Quick Start (SOAP mode): Converting a PDF document to PostScript using the Java API" on page 80

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert a PDF document to PS using the web service API

Convert a PDF document to PostScript by using the Convert PDF Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/ConvertPDFService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Convert PDF client.

- Create a `ConvertPdfServiceClient` object by using its default constructor.
- Create a `ConvertPdfServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/ConvertPDFService?blob=mtom`.) You do not need to use the `lc_version` attribute. However, specify `?blob=mtom`.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `ConvertPdfServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `ConvertPdfServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `ConvertPdfServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference the PDF document to convert to a PostScript file.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is converted to a PostScript file.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to convert and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, starting position, and stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Set conversion run-time options.

- Create a `ToPSOptionsSpec` object by invoking its constructor.

- Set run-time options by assigning a value to the `ToPSOptionsSpec` object's data member. For example, to define the PostScript level that is created, assign a `PSLevel` enumeration value to the `ToPSOptionsSpec` object's `psLevel` data member.

5 Convert the PDF document to a PostScript file.

Invoke the `GeneratePDFServiceService` object's `toPS2` method and pass the following values:

- A `BLOB` object that represents the PDF document to convert to a PostScript file
- A `ToPSOptionsSpec` object that specifies run-time options

After the conversion is complete, extract the binary data that represents the PostScript document by accessing its `BLOB` object's `MTOM` property. This returns a byte array that you can write out to a PostScript file.

6 Save the PostScript file.

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the PS file.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `encryptPDFUsingPassword` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to the PostScript file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 771

Quick Start (MTOM): Converting a PDF document to PostScript using the web service API

Quick Start (SwaRef): Converting a PDF document to PostScript using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Converting PDF Documents to Image Formats

You can use the Convert PDF service to programmatically convert PDF documents to image formats, which include JPEG, JPEG 2000, TIFF, and PNG. By converting a PDF document to an image file, you can use the PDF document as an image file. For example, you can place the image in an enterprise content management system for storage.

When converting a PDF document to an image, the Convert PDF service creates a separate image for each page in the document. That is, if the document has 20 pages, the Convert PDF service creates 20 image files. When converting a PDF document to an image format, you can create individual images for each page within the PDF document or a single image file for the entire PDF document.

Note: *For more information about the Convert PDF service, see Services Reference for AEM Forms.*

Summary of steps

To convert a PDF document to any of the supported types, perform the following steps:

1 Include project files.

2 Create a Convert PDF service client.

3 Retrieve the PDF document to convert.

4 Set run-time options.

5 Convert the PDF to an image.

6 Retrieve the image files from a collection.

Include project files

Include the necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure you include the proxy files.

Create a Convert PDF client

Before you can programmatically perform a Convert PDF service operation, you must create a Convert PDF service client. If you are using the Java API, create a `ConvertPdfServiceClient` object. If you are using the web service API, create a `ConvertPDFServiceService` object.

Retrieve the PDF document to convert

You must retrieve the PDF document to convert to an image. You cannot convert an interactive PDF document to an image. If you attempt to do so, an exception is thrown. To convert an interactive PDF document to an image file, you must flatten the PDF document before you convert it. (See "Flattening PDF Documents" on page 732.)

Set run-time options

You must set run-time options such as the image format and the resolution values. For information about the run-time values, see the `ToImageOptionsSpec` class reference in AEM Forms API Reference.

Convert the PDF to an image

After you create the service client and set run-time options, you can convert the PDF document to an image. A collection object that contains the images is returned.

Retrieve the image files from a collection

You can retrieve image files from a collection object that the Convert PDF service returns. Each element in the collection is a `com.adobe.idp.Document` instance (or a `BLOB` instance if you are using web services) that you can save as an image file, such as a JPG file.

The format of the image file is dependent on the `ImageConvertFormat` run-time option. That is, if you set the `ImageConvertFormat` run-time option to `ImageConvertFormat.JPEG`, you can save image files as JPG files.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Convert PDF Service Java API Quick Start(SOAP)" on page 80

Convert a PDF document to image files using the Java API

Convert a PDF document to an image format by using the Convert PDF service API (Java):

1 Include project files.

Include client JAR files, such as adobe-convertpdf-client.jar, in your Java project's class path.

2 Create a Convert PDF client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ConvertPdfServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the PDF document to convert.

- Create a `java.io.FileInputStream` object that represents the PDF document to convert by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set run-time options.

- Create a `ToImageOptionsSpec` object by using its constructor.
- Invoke methods that belong to this object as required. For example, set the image type by invoking the `setImageConvertFormat` method and passing an `ImageConvertFormat` enum value that specifies the format type.

Note: *Setting the `ImageConvertFormat` enumeration value is mandatory.*

5 Convert the PDF to an image.

Invoke the `ConvertPdfServiceClient` object's `toImage2` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the PDF file to convert.
- A `com.adobe.livecycle.converpdfservice.client.ToImageOptionsSpec` object that contains the various preferences about the target image format.

The `toImage2` method returns a `java.util.List` object that contains images. Each element in the collection is a `com.adobe.idp.Document` instance.

6 Retrieve the image files from a collection.

Iterate through the `java.util.List` object to determine whether images are present. Each element is a `com.adobe.idp.Document` instance. Save the image by invoking the `com.adobe.idp.Document` object's `copyToFile` method and passing a `java.io.File` object.

See also

"Quick Start (SOAP mode): Converting a PDF document to JPEG files using the Java API" on page 83

Convert a PDF document to image files using the web service API

Convert a PDF document to an image format by using the Convert PDF Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/ConvertPDFService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a convert PDF client.

- Create a `ConvertPdfServiceClient` object by using its default constructor.

- Create a `ConvertPdfServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/ConvertPDFService?blob=mtom`.) You do not need to use the `lc_version` attribute. However, specify `?blob=mtom`.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `ConvertPdfServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `ConvertPdfServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `ConvertPdfServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the PDF document to convert.

- Create a `BLOB` object by using its constructor. This `BLOB` object is used to store the PDF form.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that specifies the location of the PDF form and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. Determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Set run-time options.

- Create a `ToImageOptionsSpec` object by using its constructor.
- Invoke methods that belong to this object as required. For example, set the image type by invoking the `setImageConvertFormat` method and passing an `ImageConvertFormat` enumeration value that specifies the format type.

Note: *Setting the `ImageConvertFormat` enumeration value is mandatory.*

5 Convert the PDF to an image.

Invoke the `ConvertPDFServiceService` object's `toImage2` method and pass the following values:

- A `BLOB` object that represents the file to be converted
- A `ToImageOptionsSpec` object that contains the various preferences about the target image format

The `toImage2` method returns a `MyArrayOfBLOB` object that contains the newly created image files.

6 Retrieve the image files from a collection.

- Determine the number of elements in the `MyArrayOfBLOB` object by getting the value of its `Count` field. Each element is a `BLOB` object that contains the image.

- Iterate through the `MyArrayOfBLOB` object and save each image file.

See also

Quick Start (MTOM): Converting a PDF document to a set of JPEG files using the web service API

Quick Start (SwaRef): Converting a PDF document to a set of JPEG files using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Converting Between File Formats and PDF

About the Generate PDF Service

The Generate PDF service converts native file formats to PDF. It also converts PDF to other file formats and optimizes the size of PDF documents.

The Generate PDF service uses native applications to convert the following file formats to PDF. Unless otherwise indicated, only the German, French, English, and Japanese versions of these applications are supported. *Windows only* indicates support for only Windows Server® 2003 and Windows Server 2008.

- Microsoft Office 2003 and 2007 to convert DOC, DOCX, RTF, TXT, XLS, XLSX, PPT, PPTX, VSD, MPP, MPPX, XPS, and PUB (Windows only)

  Note: *Acrobat® 9.2 or later is required to convert Microsoft XPS format to PDF.*

- Autodesk AutoCAD 2005, 2006, 2007, 2008, and 2009 to convert DWF, DWG, and DXW (English only)
- Corel WordPerfect 12 and X4 to convert WPD, QPW, SHW (English only)
- OpenOffice 2.0, 2.4, 3.0.1, and 3.1 to convert ODT, ODS, ODP, ODG, ODF, SXW, SXI, SXC, SXD, DOC, DOCX, RTF, TXT, XLS, XLSX, PPT, PPTX, VSD, MPP, MPPX, and PUB

  Note: *The Generate PDF service does not support the 64-bit versions of OpenOffice.*

- Adobe Photoshop® CS2 to convert PSD (Windows only)

  Note: *Photoshop CS3 and CS4 are not supported because they do not support Windows Server 2003 or Windows Server 2008.*

- Adobe FrameMaker® 7.2 and 8 to convert FM (Windows only)
- Adobe PageMaker® 7.0 to convert PMD, PM6, P65, and PM (Windows only)
- Native formats supported by third-party applications (requires development of setup files specific for the application) (Windows only)

The Generate PDF service converts the following standards-based file formats to PDF.

- Video formats: SWF, FLV (Windows only)
- Image formats: JPEG, JPG, JP2, J2Kí, JPC, J2C, GIF, BMP, TIFF, TIF, PNG, JPF
- HTML (Windows, Sun™ Solaris™, and Linux®)

The Generate PDF service converts PDF to the following file formats (Windows only):

- Encapsulated PostScript (EPS)
- HTML 3.2
- HTML 4.01 with CSS 1.0

- DOC (Microsoft Word format)
- RTF
- Text (both accessible and plain)
- XML
- PDF/A-1a that uses only the DeviceRGB color space
- PDF/A-1b that uses only the DeviceRGB color space
- PDF/E-1 that uses only the DeviceRGB color space

The Generate PDF service requires that you perform these administrative tasks:

- Install required native applications on the computer hosting AEM Forms
- Install Adobe Acrobat Professional or Acrobat Pro Extended 9.2 on the computer hosting AEM Forms
- Perform post-installation setup tasks

These tasks are described in Installing and Deploying AEM forms Using JBoss Turnkey.

You can accomplish these tasks using the Generate PDF service:

- Convert from native file formats to PDF.
- Convert HTML documents to PDF documents.
- Convert PDF documents to file formats.

Note: *For more information about the Generate PDF service, see Services Reference for AEM Forms.*

Converting Word Documents to PDF Documents

This section describes how you can use the Generate PDF API to programmatically convert a Microsoft Word document to a PDF document.

Note: *For more information about additional file formats, see "Adding Support for Additional Native File Formats" on page 791.*

Note: *For more information about the Generate PDF service, see Services Reference for AEM Forms.*

Summary of steps

To convert a Microsoft Word document to a PDF document, perform the following tasks:

1. Include project files.
2. Create a Generate PDF client.
3. Retrieve the file to convert to a PDF document.
4. Convert the file to a PDF document.
5. Retrieve the results.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Generate PDF client

Before you can programmatically perform a Generate PDF operation, create a Generate PDF service client. If you are using the Java API, create a `GeneratePdfServiceClient` object. If you are using the web service API, create a `GeneratePDFServiceService` object.

Retrieve the file to convert to a PDF document

Retrieve the Microsoft Word document to convert to a PDF document.

Convert the file to a PDF document

After you create the Generate PDF service client, you can invoke the `createPDF2` method. This method needs information about the document to convert, including the file extension.

Retrieve the results

After the file is converted to a PDF document, you can retrieve the results. For example, after you convert a Word file to a PDF document, you can retrieve and save the PDF document.

See also

"Convert Word documents to PDF documents using the Java API" on page 781

"Convert Word documents to PDF documents using the web service API" on page 782

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Generate PDF Service Java API Quick Start(SOAP)" on page 211

Convert Word documents to PDF documents using the Java API

Convert a Microsoft Word document to a PDF document by using the Generate PDF API (Java):

1 Include project files.

   Include client JAR files, such as adobe-generatepdf-client.jar, in your Java project's class path.

2 Create a Generate PDF client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `GeneratePdfServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the file to convert to a PDF document.

   - Create a `java.io.FileInputStream` object that represents the Word file to convert by using its constructor. Pass a string value that specifies the file location.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Convert the file to a PDF document.

   Convert the file to a PDF document by invoking the `GeneratePdfServiceClient` object's `createPDF2` method and passing the following values:

   - A `com.adobe.idp.Document` object that represents the file to convert.
   - A `java.lang.String` object that contains the file extension.

- A `java.lang.String` object that contains the file type settings to be used in the conversion. File type settings provide conversion settings for different file types, such as .doc or .xls.
- A `java.lang.String` object that contains the name of the PDF settings to be used. For example, you can specify `Standard`.
- A `java.lang.String` object that contains the name of the security settings to be used.
- An optional `com.adobe.idp.Document` object that contains settings to be applied while generating the PDF document.
- An optional `com.adobe.idp.Document` object that contains metadata information to be applied to the PDF document.

The `createPDF2` method returns a `CreatePDFResult` object that contains the new PDF document and a log information. The log file typically contains error or warning messages generated by the conversion request.

5 Retrieve the results.

To obtain the PDF document, perform the following actions:

- Invoke the `CreatePDFResult` object's `getCreatedDocument` method, which returns a `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document from the object created in the previous step.

If you used the `createPDF2` method to obtain the log document (not applicable to HTML conversions), perform the following actions:

- Invoke the `CreatePDFResult` object's `getLogDocument` method. This returns a `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the log document.

See also

"Summary of steps" on page 780

"Quick Start (SOAP mode): Converting a Microsoft Word document to a PDF document using the Java API" on page 212

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert Word documents to PDF documents using the web service API

Convert a Microsoft Word document to a PDF document by using the Generate PDF API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/GeneratePDFService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Generate PDF client.

- Create a `GeneratePDFServiceClient` object by using its default constructor.

- Create a `GeneratePDFServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/GeneratePDFService?blob=mtom`.) You do not need to use the `lc_version` attribute. However, specify `?blob=mtom`.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `GeneratePDFServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `GeneratePDFServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `GeneratePDFServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the file to convert to a PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the file that you want to convert to a PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the file to convert and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning to its `MTOM` property the contents of the byte array.

4 Convert the file to a PDF document.

Convert the file to a PDF document by invoking the `GeneratePDFServiceService` object's `CreatePDF2` method and passing the following values:

- A `BLOB` object that represents the file to be converted.
- A string that contains the file extension.
- A `java.lang.String` object that contains the file type settings to be used in the conversion. File type settings provide conversion settings for different file types, such as .doc or .xls.
- A string object that contains the PDF settings to be used. You can specify `Standard`.
- A string object that contains the security settings to be used. You can specify `No Security`.
- An optional `BLOB` object that contains settings to be applied while generating the PDF document.
- An optional `BLOB` object that contains metadata information to be applied to the PDF document.

- An output parameter of type `BLOB` that is populated by the `CreatePDF2` method. The `CreatePDF2` method populates this object with the converted document. (This parameter value is required only for web service invocation).
- An output parameter of type `BLOB` that is populated by the `CreatePDF2` method. The `CreatePDF2` method populates this object with the log document. (This parameter value is required only for web service invocation).

5 Retrieve the results.

- Retrieve the converted PDF document by assigning the `BLOB` object's `MTOM` field to a byte array. The byte array represents the converted PDF document. Ensure you use the `BLOB` object that is used as the output parameter for the `createPDF2` method.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the converted PDF document.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 780

Quick Start (MTOM): Converting a Microsoft Word document to a PDF document using the web service API

Quick Start (SwaRef): Converting a Microsoft Word document to a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Converting HTML Documents to PDF Documents

This section describes how you can use the Generate PDF API to programmatically convert HTML documents to PDF documents.

Note: *For more information about the Generate PDF service, see Services Reference for AEM Forms.*

Summary of steps

To convert an HTML document to a PDF document, perform the following tasks:

1 Include project files.

2 Create a Generate PDF client.

3 Retrieve the HTML content to convert to a PDF document.

4 Convert the HTML content to a PDF document.

5 Retrieve the results.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Generate PDF client

Before you can programmatically perform a Generate PDF operation, you must create a Generate PDF service client. If you are using the Java API, create a `GeneratePdfServiceClient` object. If you are using the web service API, create a `GeneratePDFServiceService`.

Retrieve the HTML content to convert to a PDF document

Reference HTML content that you want to convert to a PDF document. You can reference HTML content such as an HTML file or HTML content that is accessible using a URL.

Convert the HTML content to a PDF document

After you create the service client, you can invoke the appropriate PDF creation operation. This operation needs information about the document to be converted, including the path to the target document.

Retrieve the results

After the HTML content is converted to a PDF document, you can retrieve the results and save the PDF document.

See also

"Convert HTML content to a PDF document using the Java API" on page 785

"Convert HTML content to a PDF document using the web service API" on page 786

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Generate PDF Service Java API Quick Start(SOAP)" on page 211

Convert HTML content to a PDF document using the Java API

Convert an HTML document to a PDF document using the Generate PDF API (Java):

1 Include project files.

   Include client JAR files, such as adobe-generatepdf-client.jar, in your Java project's class path.

2 Create a Generate PDF client.

   Create a `GeneratePdfServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Retrieve the HTML content to convert to a PDF document.

   Retrieve HTML content by creating a string variable and assigning a URL that points to HTML content.

4 Convert the HTML content to a PDF document.

   Invoke the `GeneratePdfServiceClient` object's `htmlToPDF2` method and pass the following values:

   - A `java.lang.String` object that contains the URL of the HTML file to be converted.
   - A `java.lang.String` object that contains the file type settings to be used in the conversion. File type settings can include spidering levels.
   - A `java.lang.String` object that contains the name of the security settings to be used.
   - An optional `com.adobe.idp.Document` object that contains settings to be applied while generating the PDF document. If this information is not supplied, the settings are automatically chosen based on the previous three parameters.

- An optional `com.adobe.idp.Document` object that contains metadata information to be applied to the PDF document.

5 Retrieve the results.

The `htmlToPDF2` method returns an `HtmlToPdfResult` object that contains the new PDF document that was generated. To obtain the newly created PDF document, perform the following actions:

- Invoke the `HtmlToPdfResult` object's `getCreatedDocument` method. This returns a `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document from the object created in the previous step.

See also

"Converting HTML Documents to PDF Documents" on page 784

Quick Start (SOAP mode): Converting HTML content to a PDF document using the Java API

"Quick Start (SOAP mode): Converting HTML content to a PDF document using the Java API" on page 214

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert HTML content to a PDF document using the web service API

Convert HTML content to a PDF document by using the Generate PDF API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/GeneratePDFService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Generate PDF client.

- Create a `GeneratePDFServiceClient` object by using its default constructor.
- Create a `GeneratePDFServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/GeneratePDFService?blob=mtom`.) You do not need to use the `lc_version` attribute. However, specify `?blob=mtom`.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `GeneratePDFServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `GeneratePDFServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `GeneratePDFServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.

- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the HTML content to convert to a PDF document.

Retrieve HTML content by creating a string variable and assigning a URL that points to HTML content.

4 Convert the HTML content to a PDF document.

Convert the HTML content to a PDF document by invoking the `GeneratePDFServiceService` object's `HtmlToPDF2` method and pass the following values:

- A string that contains the HTML content to convert.
- A `java.lang.String` object that contains the file type settings to be used in the conversion.
- A string object that contains the security settings to be used.
- An optional `BLOB` object that contains settings to be applied while generating the PDF document.
- An optional `BLOB` object that contains metadata information to be applied to the PDF document.
- An output parameter of type `BLOB` that is populated by the `CreatePDF2` method. The `CreatePDF2` method populates this object with the converted document. (This parameter value is required only for web service invocation).

5 Retrieve the results.

- Retrieve the converted PDF document by assigning the `BLOB` object's `MTOM` field to a byte array. The byte array represents the converted PDF document. Ensure you use the `BLOB` object that is used as the output parameter for the `HtmlToPDF2` method.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the converted PDF document.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Converting HTML Documents to PDF Documents" on page 784

Quick Start (MTOM): Converting an HTML document to a PDF document using the web service API

Quick Start (SwaRef): Converting an HTML document to a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Converting PDF Documents to Non-image Formats

This section describes how you can use the Generate PDF Java API and web service API to programmatically convert a PDF document to an RTF file, which is an example of a non-image format. Other non-image formats include HTML, text, DOC, and EPS. When converting a PDF document to RTF, ensure that the PDF document does not contain form elements, such as a submit button. Form elements are not converted.

Note: *For more information about the Generate PDF service, see Services Reference for AEM Forms.*

Summary of steps

To convert a PDF document to any of the supported types, perform the following steps:

1 Include project files.

2 Create a Generate PDF client.

3 Retrieve the PDF document to convert.

4 Convert the PDF document.

5 Save the converted file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a Generate PDF client

Before you can programmatically perform a Generate PDF operation, you must create a Generate PDF service client. If you are using the Java API, create a `GeneratePdfServiceClient` object. If you are using the web service API, create a `GeneratePDFServiceService` object.

Retrieve the PDF document to convert

Retrieve the PDF document to convert to a non-image format.

Convert the PDF document

After you create the service client, you can invoke the PDF export operation. This operation needs information about the document to be converted, including the path to the target document.

Save the converted file

Save the converted file. For example, if you convert a PDF document to an RTF file, save the converted document to an RTF file.

See also

"Convert a PDF document to a RTF file using the Java API" on page 788

"Convert a PDF document to a RTF file using the web service API" on page 789

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Generate PDF Service Java API Quick Start(SOAP)" on page 211

Convert a PDF document to a RTF file using the Java API

Convert a PDF document to an RTF file by using the Generate PDF API (Java):

1 Include project files.

Include client JAR files, such as adobe-generatepdf-client.jar, in your Java project's class path.

2 Create a Generate PDF client.

Create a `GeneratePdfServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Retrieve the PDF document to convert.

- Create a `java.io.FileInputStream` object that represents the PDF document to convert by using its constructor. Pass a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Convert the PDF document.

Invoke the `GeneratePdfServiceClient` object's `exportPDF2` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the PDF file to convert.
- A `java.lang.String` object that contains the name of the file to convert.
- A `java.lang.String` object that contains the name of the Adobe PDF settings.
- A `ConvertPDFFormatType` object that specifies the target file type for the conversion.
- An optional `com.adobe.idp.Document` object that contains settings to be applied while generating the PDF document.

The `exportPDF2` method returns an `ExportPDFResult` object that contains the converted file.

5 Convert the PDF document.

To obtain the newly created file, perform the following actions:

- Invoke the `ExportPDFResult` object's `getConvertedDocument` method. This returns a `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the new document.

See also

"Summary of steps" on page 788

"Quick Start (SOAP mode): Converting HTML content to a PDF document using the Java API" on page 214

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert a PDF document to a RTF file using the web service API

Convert a PDF document to an RTF file by using the Generate PDF API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/GeneratePDFService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Generate PDf client.

- Create a `GeneratePDFServiceClient` object by using its default constructor.
- Create a `GeneratePDFServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/GeneratePDFService?blob=mtom`.) You do not need to use the `lc_version` attribute. However, specify `?blob=mtom`.

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `GeneratePDFServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `GeneratePDFServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `GeneratePDFServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the PDF document to convert.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is converted.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning to its `MTOM` property the contents of the byte array.

4 Convert the PDF document.

Invoke the `GeneratePDFServiceServiceWse` object's `ExportPDF2` method and pass the following values:

- A `BLOB` object that represents the PDF file to convert.
- A string that contains the path name of the file to convert.
- A `java.lang.String` object that specifies the file location.
- A string object that specifies the target file type for the conversion. Specify `RTF`.
- An optional `BLOB` object that contains settings to be applied while generating the PDF document.
- An output parameter of type `BLOB` that is populated by the `ExportPDF2` method. The `ExportPDF2` method populates this object with the converted document. (This parameter value is required only for web service invocation).

5 Save the converted file.

- Retrieve the converted RTF document by assigning the `BLOB` object's `MTOM` field to a byte array. The byte array represents the converted RTF document. Ensure you use the `BLOB` object that is used as the output parameter for the `ExportPDF2` method.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the location of the RTF file.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

- Write the contents of the byte array to a RTF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 788

Quick Start (MTOM): Converting a PDF document to an RTF file using the web service API

Quick Start (SwaRef): Converting a PDF document to an RTF file using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Adding Support for Additional Native File Formats

This section explains how to add support for additional native file formats. It provides an overview of the interactions between the Generate PDF service and the native applications that this service uses to convert native file formats into PDF.

This section also explains the following:

- How to modify the response that the Generate PDF service provides to the native applications that this product already uses to convert native file formats into PDF
- The interactions between the Generate PDF service, the Generate PDF service Application Monitor (AppMon) component, and native applications, such as Microsoft Word
- The roles that XML grammars play in those interactions

Component interactions

The Generate PDF service converts native file formats by invoking the application associated with the file format and then interacting with the application to print the document using the default printer. The default printer must be set up as the Adobe PDF printer.

This illustration shows the components and drivers involved with native application support. It also mentions the XML grammars that influence the interactions.

Component interactions for native file conversion

This document uses the term *native application* to indicate the application used to produce a native file format, such as Microsoft Word.

AppMon is an enterprise component that interacts with a native application in the same way a user would navigate through the dialog boxes presented by that application. The XML grammars used by AppMon to instruct an application, such as Microsoft Word, to open and print a file involve these sequential tasks:

1. Opening the file by selecting File > Open
2. Ensuring that the Open dialog box appears; if not, handling the error
3. Providing the file name in the File Name field and then clicking the Open button
4. Ensuring that the file actually opens
5. Opening the Print dialog box by selecting File > Print
6. Ensuring that the Print dialog box appears

AppMon uses standard Win32 APIs to interact with third-party applications in order to transfer UI events such as key-strokes and mouse clicks, which is useful to control these applications to produce PDF files from them.

Due to a limitation with these Win32 APIs, AppMon is not able to dispatch these UI events to some specific kinds of windows, such as floating menu-bars (found in some applications such as TextPad), and certain kind of dialogs whose contents cannot be retrieved using the Win32 APIs.

It is easy to visually identify a floating menu-bar; however it might not be possible to identify the special types of dialogs just by visual inspection. You would require a third-party application such as Microsoft Spy++ (part of the Microsoft Visual C++ development environment) or its equivalent WinID (that can be downloaded free of cost from http://www.dennisbabkin.com/php/download.php?what=WinID) to examine a dialog to determine if AppMon would be able to interact with it using standard Win32 APIs.

If WinID is able to extract the dialog contents such as the text, sub-windows, window class ID, and so on, then AppMon would also be able to do the same.

This table lists the type of information used in printing native file formats.

Information type	Description	Modifying/creating entries related to native files
Administrative settings	Includes PDF settings, security settings, and file type settings. File type settings associate file name extensions with the corresponding native applications. File type settings also specify native application settings used to print native files.	To change settings for an already supported native application, the system administrator sets the File Type Settings in the administration console. To add support for a new native file format, you must manually edit the file. (See "Adding or modifying support for a native file format" on page 795.)
Script	Specifies interactions between the Generate PDF service and a native application. Such interactions usually direct the application to print a file to the Adobe PDF driver. The script contains instructions that direct the native application to open specific dialog boxes and that supply specific responses to fields and buttons in those dialog boxes.	The Generate PDF service includes script files for all supported native applications. You can modify these files using an XML editing application. To add support for a new native application, you must create a new script file. (See "Creating or modifying an additional dialog XML file for a native application" on page 799.)
Generic dialog box instructions	Specifies how to respond to dialog boxes that are common to multiple applications. Such dialog boxes are generated by operating systems, helper applications (such as PDFMaker), and drivers. The file that contains this information is appmon.global.en_US.xml.	Do not modify this file.
Application-specific dialog box instructions	Specifies how to respond to application-specific dialog boxes. The file that contains this information is appmon.*[appname]*.dialog.*[locale]*.xml (for example, appmon.word.en_US.xml).	Do not modify this file. To add dialog box instructions for a new native application, see "Creating or modifying an additional dialog XML file for a native application" on page 799.
Additional application-specific dialog box instructions	Specifies overrides and additions to application-specific dialog box instructions. The section presents an example of such information. The file that contains this information is appmon.*[appname]*.addition.*[locale]*.xml. An example is appmon.addition.en_US.xml.	Files of this type can be created and modified using an XML editing application. (See "Creating or modifying an additional dialog XML file for a native application" on page 799.) **Important**: You must create additional application-specific dialog box instructions for each native application your server will support.

About the script and dialog XML files

Script XML files direct the Generate PDF service to navigate through application dialog boxes in the same way a user would navigate through the application dialog boxes. Script XML files also direct the Generate PDF service to respond to dialog boxes by performing actions such as pressing buttons, selecting or deselecting check boxes, or selecting menu items.

In contrast, dialog XML files simply respond to dialog boxes with the same types of actions used in script XML files.

Dialog box and window element terminology

This section and the next section use different terminology for dialog boxes and the components they contain, depending on the perspective being described. Dialog box components are items such as buttons, fields, and combo boxes.

When this section and the next section describe dialog boxes and their components from the perspective of a user, terms such as *dialog box*, *button*, *field*, and *combo box* are used.

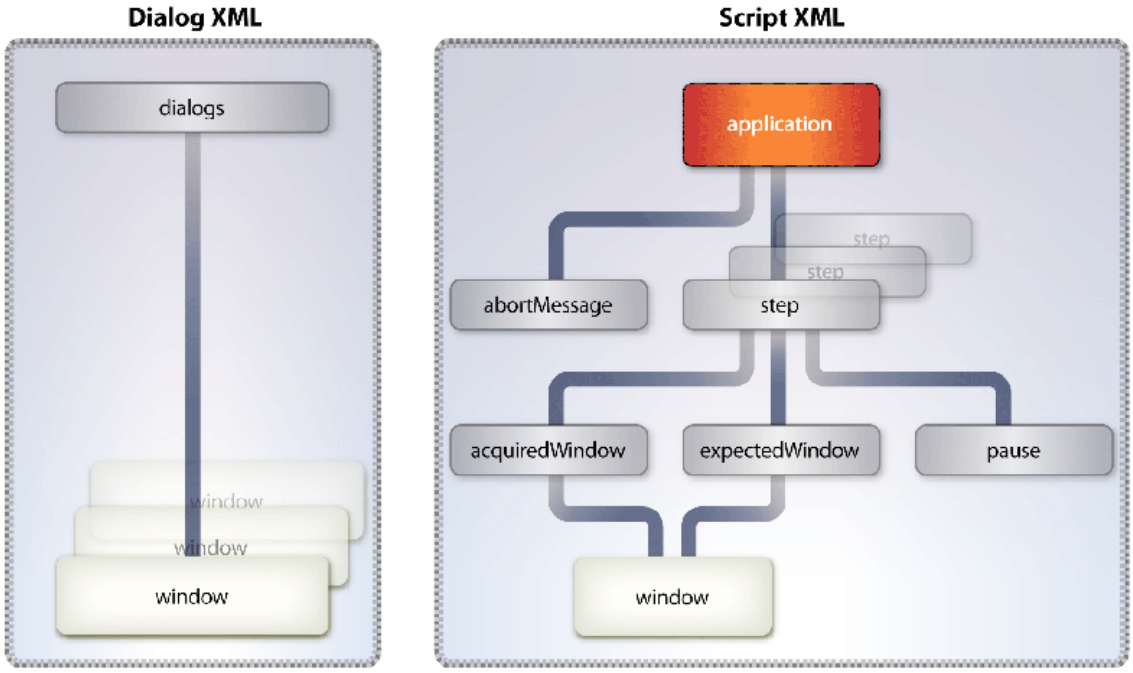
When this section and the next section describe dialog boxes and their components from the perspective of their internal representation, the term *window element* is used. The internal representation of window elements is a hierarchy, where each window element instance is identified by labels. The window element instance also describes its physical characteristics and behavior.

From a user's perspective, the dialog boxes and their components show different behaviors, where some dialog box elements are hidden until activated. From an internal representation perspective, no such issue of behavior exists. For example, the internal representation of a dialog box looks similar to that of the components it contains, with the exception that the components are nested within the dialog box.

This section describes XML elements that provide AppMon with instructions. These elements have names such as the `dialog` element and the `window` element. This document uses a monospaced font to distinguish XML elements. The *`dialog`* element identifies a dialog box that an XML script file can cause to be displayed, either intentionally or unintentionally. The *`window`* element identifies a window element (dialog box or the components of a dialog box).

Hierarchy

This diagram shows the hierarchy of script and dialog XML. A script XML file conforms to the script.xsd schema, which includes (in the XML sense) the window.xsd schema. Similarly, a dialog XML file conforms to the dialogs.xsd schema, which also includes the window.xsd schema.



Hierarchy of script and dialog XML

Script XML files

A *script XML file* specifies a series of steps that direct the native application to navigate to certain window elements and then supply responses to those elements. Most responses are text or keystrokes that correspond to the input a user would provide to a field, combo box, or button in the corresponding dialog box.

The intent of the Generate PDF service's support for script XML files is to direct a native application to print a native file. However, script XML files can be used to accomplish any task that a user can perform when interacting with the native application's dialog boxes.

The steps in a script XML file are executed in order, without any opportunity for branching. The only conditional test supported is for time-out/retry, which causes a script to terminate if a step does not complete successfully within a specific period of time and after a specific number of retries.

In addition to steps being sequential, the instructions within a step are also executed in order. You must ensure that the steps and instructions reflect the order in which a user would perform those same steps.

Each step in a script XML file identifies the window element that is expected to appear if the step's instructions are successfully performed. If an unexpected dialog box appears while executing a script step, the Generate PDF service searches the dialog XML files as described in the next section.

Dialog XML files

Running native applications displays different dialog boxes, which appear regardless of whether the native applications are in a visible or invisible mode. The dialog boxes can be generated by the operating system or by the application itself. When native applications are running under control of the Generate PDF service, system and native application dialog boxes are displayed in an invisible window.

A *dialog XML file* specifies how the Generate PDF service responds to system or native application dialog boxes. The dialog XML files allow the Generate PDF service to respond to unprompted dialog boxes in a way that facilitates the conversion process.

When the system or native application displays a dialog box that is not handled by the currently executing script XML file, the Generate PDF service searches the dialog XML files in this order, stopping when it finds a match:

- appmon.*[appname]*.additional.*[locale]*.xml
- appmon.*[appname]*.*[locale]*.xml (Do not modify this file.)
- appmon.global.*[locale]*.xml (Do not modify this file.)

If the Generate PDF service finds a match for the dialog box, it dismisses it by sending it the keystroke or other action specified for the dialog box. If the instructions for the dialog box specify an abort message, the Generate PDF service terminates the currently executing job and generates an error message. Such an abort message would be specified in the `abortMessage` element in the script XML grammar.

If the Generate PDF service encounters a dialog box that is not described in any of the previously-listed files, the Generate PDF service incorporates the dialog box's caption into the log file entry. The currently executing job eventually times out. You can then use the information in the log file to compose new instructions in the additional dialog XML file for the native application.

Adding or modifying support for a native file format

This section describes the tasks you must perform to support other native file formats or to modify support for an already supported native file format.

Before you can add or modify support, you must complete the following tasks.

Choosing a tool for identifying window elements

The dialog and script XML files require you to identify the window element (dialog box, field, or other dialog component) to which your dialog or script element is responding. For example, after a script invokes a menu for a native application, the script must identify the window element on that menu to which keystrokes or an action are to be applied.

You can easily identify a dialog box by the caption it displays in its title bar. However, you must use a tool such as Microsoft Spy++ to identify lower-level window elements. The lower-level window elements can be identified through a variety of attributes, which are not obvious. Additionally, each native application may identify its window element differently. As a result, there are multiple ways of identifying a window element. Here is the suggested order for considering window element identification:

1. Caption itself if it is unique
2. Control ID, which may or may not be unique for a given dialog box
3. Class name, which may or may not be unique

Any one or a combination of these three attributes can be used to identify a window.

If the attributes fail to identify a caption, you can instead identify a window element by using its index with respect to its parent. An *index* specifies the position of the window element relative to its sibling window elements. Frequently, indexes are the only way to identify combo boxes.

Be aware of these issues:

- Microsoft Spy++ displays captions by using an ampersand (&) to identify the caption's hot key. For example, Spy++ shows the caption for one Print dialog box as `Pri&nt`, which indicates that the hotkey is *n*. Caption titles in script and dialog XML files must omit ampersands.
- Some captions include line breaks. the Generate PDF service cannot identify line breaks. If a caption includes a line break, include enough of the caption to differentiate it from the other menu items and then use regular expressions for the omitted part. An example is (`^Long caption title$`).]. (See "Using regular expressions in caption attributes" on page 798.)
- Use character entities (also called escape sequences) for reserved XML characters. For example, use `&` for ampersands, `<` and `>` for less than and greater than symbols, `&apos;` for apostrophes, and `&quot;` for quotation marks.

If you plan to work on dialog or script XML files, you should install the application Microsoft Spy++.

Unpackaging the dialog and script files

The dialog and script files reside in the appmondata.jar file. Before you can modify any of these files or add new script or dialog files, you must unpackage this JAR file. For example, assume that you want to add support for the EditPlus application. You create two XML files, named appmon.editplus.script.en_US.xml and appmon.editplus.script.addition.en_US.xml. These XML scripts must be added to the adobe-appmondata.jar file in two locations, as specified below:

- adobe-livecycle-native-jboss-x86_win32.ear > adobe-Native2PDFSvc.war\WEB-INF\lib > adobe-native.jar > Native2PDFSvc-native.jar\bin > adobe-appmondata.jar\com\adobe\appmon. The adobe-livecycle-native-jboss-x86_win32.ear file is in the export folder at *[AEM forms install directory]*\configurationManager. (if AEM Forms is deployed on another J2EE application server, replace the adobe-livecycle-native-jboss-x86_win32.ear file with the EAR file that corresponds to your J2EE application server.)
- adobe-generatepdf-dsc.jar > adobe-appmondata.jar\com\adobe\appmon (the adobe-appmondata.jar file is within the adobe-generatepdf-dsc.jar file). The adobe-generatepdf-dsc.jar file is in the *[AEM forms install directory]*\deploy folder.

After you add these XML files to the adobe-appmondata.jar file, you must redeploy the GeneratePDF component. To add dialog and script XML files to the adobe-appmondata.jar file, perform these tasks:

1. Using a tool such as WinZip or WinRAR, open the adobe-livecycle-native-jboss-x86_win32.earfile > adobe-Native2PDFSvc.war\WEB-INF\lib > adobe-native.jar > Native2PDFSvc-native.jar\bin > adobe-appmondata.jar file.
2. Add the dialog and script XML files to the appmondata.jar file or modify existing XML files in this file. (See "Creating or modifying a script XML file for a native application" on page 797and "Creating or modifying an additional dialog XML file for a native application" on page 799.)
3. Using a tool such as WinZip or WinRAR, open adobe-generatepdf-dsc.jar > adobe-appmondata.jar.
4. Add the dialog and script XML files to the appmondata.jar file or modify existing XML files in this file. (See "Creating or modifying a script XML file for a native application" on page 797and "Creating or modifying an additional dialog XML file for a native application" on page 799.) After you add the XML files to the adobe-appmondata.jar file, place the new adobe-appmondata.jar file into the adobe-generatepdf-dsc.jar file.
5. If you added support for an additional native file format, create a system environment variable that provides the path of the application (See "Creating an environment variable to locate the native application" on page 801.)

To redeploy the GeneratePDF component

1. Log in to Workbench.
2. Select **Window** > **Show Views** > **Components**. This action adds the Components view to Workbench.
3. Right-click the GeneratePDF component, and then select **Stop Component**.
4. When the component has stopped, right-click and select Uninstall Component to remove it.
5. Right-click the **Components** icon and select **Install Component**.
6. Browse for and select the modified adobe-generatepdf-dsc.jar file and then click Open. Notice that a red square appears next to the GeneratePDF component.
7. Expand the GeneratePDF component, select Service Descriptors, and then right-click GeneratePDFService and select Activate Service.
8. In the configuration dialog box that appears, enter applicable configuration values. If you leave these values blank, default configuration values are used.
9. Right-click GeneratePDF and select Start Component.
10. Expand Active Services. A green arrow appears next to the service name if it is running. Otherwise, the service is in a stopped state.
11. If the service is in a stopped state, right-click the service name and select Start Service.

Creating or modifying a script XML file for a native application

If you want to direct files to a new native application, you must create a script XML file for that application. If you want to modify how the Generate PDF service interacts with a native application that is already supported, you must modify the script for that application.

The script contains instructions that navigate through the native application's window elements and that supply specific responses to those elements. The file that contains this information is appmon.*[appname]*.script.*[locale]*.xml. An example is appmon.notepad.script.en_US.xml.

Identifying steps the script must execute

Using the native application, determine the window elements that you must navigate and each response you must perform to print the document. Notice the dialog boxes that result from any response. The steps will be similar to these steps:

1. Select File > Open.
2. Specify the path and then click Open.
3. Select File > Print on the menu bar.
4. Specify the properties required for the printer.
5. Select Print and wait for the Save As dialog box to appear. The Save As dialog box is required for the Generate PDF service to specify the destination for the PDF file.

Identifying the dialogs specified in caption attributes

Use Microsoft Spy++ to obtain the identities of window element properties in the native application. You must have these identities to write scripts.

Using regular expressions in caption attributes

You can use regular expressions in caption specifications. The Generate PDF service uses the `java.util.regex.Matcher` class to support regular expressions. That utility supports the regular expressions described in `java.util.regex.Pattern`. (Go to the Java website at http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html.)

Regular expression accommodating the file name prepended to Notepad in the Notepad banner

```
<!-- The regular expression ".*Notepad" means any number of non-terminating characters followed
by Notepad. -->
<step>
    <expectedWindow>
        <window caption=".*Notepad"/>
    </expectedWindow>
</step>
```

Regular expression differentiating Print from Print Setup

```
<!-- This regular expression differentiates the Print dialog box from the Print Setup dialog
box. The "^" specifies the beginning of the line, and the "$" specifies the end of the line. -->
<windowList>
    <window controlID="0x01" caption="^Print$" action="press"/>
</windowList>
```

Ordering the window and windowList elements

You must order `window` and `windowList` elements as follows:

- When multiple `window` elements appear as children in a `windowList` or `dialog` element, order those `window` elements in descending order, with the lengths of the `caption` names indicating the position in the order.
- When multiple `windowList` elements appear in a `window` element, order those `windowList` elements in descending order, with the lengths of the `caption` attributes of the first `indexes/`element indicating the position in the order.

Ordering window elements in a dialog file

```
<!-- The caption attribute in the following window element is 40 characters long. It is the
longest caption in this example, so its parent window element appears before the others. -->
<window caption="Unexpected Failure in DebugActiveProcess">
    <…>
</window>

<!-- Caption length is 33 characters. -->
<window caption="Adobe Acrobat - License Agreement">
    <…>
</window>

<!-- Caption length is 33 characters. -->
<window caption="Microsoft Visual.*Runtime Library">
    <…>
</window>

<!-- The caption attribute in the following window element is 28 characters long. It is the
shortest caption in this example, so its parent window element appears after the others. -->
<window caption="Adobe Acrobat - Registration">
    <…>
</window>
```

Ordering window elements within a windowList element

```
<!-- The caption attribute in the following indexes element is 56 characters long. It is the
longest caption in this example, so its parent window element appears before the others. -->
<windowList>
    <window caption="Can&apos;t exit design mode because.* cannot be created"/>
    <window className="Button" caption="OK" action="press"/>
</windowList>
<windowList>
    <window caption="Do you want to continue loading the project?"/>
    <window className="Button" caption="No" action="press"/>
</windowList>
<windowList>
    <window caption="The macros in this project are disabled"/>
    <window className="Button" caption="OK" action="press"/>
</windowList>
```

Creating or modifying an additional dialog XML file for a native application

If you create a script for a native application that was not previously supported, you must also create an additional dialog XML file for that application. Every native application that AppMon uses must have only one additional dialog XML file. The additional dialog XML file is required even if no unsolicited dialog boxes are expected. The additional dialog box must have at least one `window` element, even if that `window` element is merely a placeholder.

Note: *In this context, the term additional means the contents of the appmon.[applicationname].addition.[locale].xml file. Such a file specifies overrides and additions to the dialog XML file.*

You can also modify the additional dialog XML file for a native application for these purposes:

- To override the dialog XML file for an application with a different response
- To add a response to a dialog box that is not addressed in the dialog XML file for that application

The file name that identifies an additional dialogXML file is appmon.*[appname]*.addition.*[locale]*.xml. An example is appmon.excel.addition.en_US.xml.

The name of the additional dialog XML file must use the format appmon.*[applicationname]*.addition.*[locale]*.xml, where *applicationname* must exactly match the application name used in the XML configuration file and in the script.

Note: *None of the generic applications specified in the native2pdfconfig.xml configuration file have a primary dialog XML file. The section "Adding or modifying support for a native file format" on page 795describes such specifications.*

You must order `windowList` elements that appear as children in a `window` element. (See "Ordering the window and windowList elements" on page 798.)

Modifying the general dialog XML file

You can modify the general dialog XML file to respond to dialog boxes that are generated by the system or to respond to dialog boxes that are common to multiple applications.

Adding a filetype entry in the XML configuration file

This procedure explains how to update the Generate PDF service configuration file to associate file types with native applications. To update this configuration file, you must use administration console to export the configuration data to a file. The default file name for the configuration data is native2pdfconfig.xml.

Update the Generate PDF service configuration file

1. Select **Home** > **Services** > **Adobe PDF Generator**> **Configuration Files**, and then select **Export Configuration**.
2. Modify the `filetype-settings` element in the native2pdfconfig.xml file, as needed.
3. Select **Home** > **Services** > **Adobe PDF Generator**> **Configuration Files**, and then select **Import Configuration**. The configuration data is imported into the Generate PDF service, replacing previous settings.

Note: *The name of the application is specified as the value of the `GenericApp` element's `name` attribute. This value must exactly match the corresponding name specified in the script that you develop for that application. Likewise, the `GenericApp` element's `displayName` attribute should exactly match the corresponding script's `expectedWindow` window caption. Such equivalency is evaluated after resolving any regular expressions that appear in the `displayName` or `caption` attributes.*

In this example, the default configuration data supplied with the Generate PDF service was modified to specify that Notepad (not Microsoft Word) should be used to process files with the file name extension .txt. Before this modification, Microsoft Word was specified as the native application that should process such files.

Modifications for directing text files to Notepad (native2pdfconfig.xml)

```
<filetype-settings>

<!-- Some native app file types were omitted for brevity. -->
<!-- The following GenericApp element specifies Notepad as the native application that should
be used to process files that have a txt file name extension. -->
            <GenericApp
                extensions="txt"
                name="Notepad" displayName=".*Notepad"/>
            <GenericApp
                extensions="wpd"
                name="WordPerfect" displayName="Corel WordPerfect"/>
            <GenericApp extensions="pmd,pm6,p65,pm"
                name="PageMaker" displayName="Adobe PageMaker"/>
            <GenericApp extensions="fm"
                name="FrameMaker" displayName="Adobe FrameMaker"/>
            <GenericApp extensions="psd"
                name="Photoshop" displayName="Adobe Photoshop"/>
        </settings>
    </filetype-settings>
```

Creating an environment variable to locate the native application

Create an environment variable that specifies the location of the native application executable. The variable must use the format *[applicationname]*_PATH, where *applicationname* must exactly match the application name used in the XML configuration file and in the script, and where the path contains the path to the executable in double quotation marks. An example of such an environment variable is `Photoshop_PATH`.

After creating the new environment variable, you must restart the server on which the Generate PDF service is deployed.

Create a system variable in the Windows XP environment

1. Select **Control Panel > System**.
2. In the System Properties dialog box, click the **Advanced** tab and then click **Environment Variables**.
3. Under System Variables in the Environment Variables dialog box, click **New**.
4. In the New System Variable dialog box, in the **Variable name** box, type a name that uses the format *[applicationname]*_PATH.
5. In the **Variable value** box, type the full path and file name of the application's executable file and then click **OK**. For example, type: `c:\windows\Notepad.exe`
6. In the Environment Variables dialog box, click **OK**.

Create a system variable from the command line

1. In a command line window, type the variable definition, using this format:

   *[applicationname]*_PATH=*[Full path name]*

   For example, type: `NotePad_PATH=C:\WINDOWS\NOTEPAD.EXE`
2. Start a fresh command line prompt for the system variable to take effect.

XML files

AEM Forms includes sample XML files that cause the Generate PDF service to use Notepad to process any files with the file name extension .txt. This code is included in this section. In addition, you must make the other modifications described in this section.

Additional dialog XML file

This example contains the additional dialog boxes for the Notepad application. These dialog boxes can be in addition to the ones specified by the Generate PDF service.

Notepad dialog boxes(appmon.notepad.addition.en_US.xml)

```
<dialogs app="Notepad" locale="en_US" version="7.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="dialogs.xsd">
    <window caption="Caption Title">
        <windowList>
            <window className="Button" caption="OK" action="press"/>
        </windowList>
    </window>
</dialogs>
```

Script XML file

This example specifies how the Generate PDF service should interact with Notepad to print files by using the Adobe PDF printer.

Notepad script XML file (appmon.notepad.script.en_US.xml)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--
*
* ADOBE CONFIDENTIAL
* ___________________
* Copyright 2004 - 2005 Adobe Systems Incorporated
* All Rights Reserved.
*
* NOTICE:  All information contained herein is, and remains
* the property of Adobe Systems Incorporated and its suppliers,
* if any.  The intellectual and technical concepts contained
* herein are proprietary to Adobe Systems Incorporated and its
* suppliers and may be covered by U.S. and Foreign Patents,
* patents in process, and are protected by trade secret or copyright law.
* Dissemination of this information or reproduction of this material
* is strictly forbidden unless prior written permission is obtained
* from Adobe Systems Incorporated.
*-->

<!-- This file automates printing of text files via notepad to Adobe PDF printer. In order to
see the complete hierarchy we recommend using the Microsoft Spy++ which details the properties
of windows necessary to write scripts. In this sample there are total of eight steps-->

<application name="Notepad" version="9.0" locale="en_US"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="scripts.xsd">
```

```
    <!-- In this step we wait for the application window to appear -->
    <step>
        <expectedWindow>
            <window caption=".*Notepad"/>
        </expectedWindow>
    </step>

    <!-- In this step, we acquire the application window and send File->Open menu bar, menu
item commands and the expectation is the windows Open dialog-->
    <step>
        <acquiredWindow>
            <window caption=".*Notepad">
                <virtualInput>
                    <menuBar>
                        <selection>
                            <name>File</name>
                        </selection>
                        <selection>
                            <name>Open...</name>
                        </selection>
                    </menuBar>
                </virtualInput>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Open"/>
        </expectedWindow>
    </step>

    <!-- In this step, we acquire the Open window and then select the 'Edit' widget and input
the source path followed by clicking on the 'Open' button . The expectation of this 'action'
is that the Open dialog will disappear -->
    <step>
        <acquiredWindow>
            <window caption="Open">
                <windowList>
                    <window className="ComboBoxEx32">
                        <windowList>
                            <window className="ComboBox">
                                <windowList>
                                <window className="Edit" action="inputSourcePath"/>
                                </windowList>
                            </window>
                        </windowList>
                    </window>
                </windowList>
                <windowList>
                    <window className="Button" caption="Open" action="press"/>
                </windowList>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Open" action="disappear"/>
        </expectedWindow>
        <pause value="30"/>
    </step>
```

```
    <!-- In this step, we acquire the application window and send File->Print menu bar, menu
item commands and the expectation is the windows Print dialog-->
    <step>
        <acquiredWindow>
            <window caption=".*Notepad">
                <virtualInput>
                    <menuBar>
                        <selection>
                            <name>File</name>
                        </selection>
                        <selection>
                            <name>Print...</name>
                        </selection>
                    </menuBar>
                </virtualInput>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Print">
        </window>
        </expectedWindow>
    </step>

    <!-- In this step, we acquire the Print dialog and click on the 'Preferences' button and
the expected window in this case is the dialog with the caption '"Printing Preferences' -->
    <step>
        <acquiredWindow>
            <window caption="Print">
                <windowList>
                    <window caption="General">
                        <windowList>
                            <window className="Button" caption="Preferences" action="press"/>
                        </windowList>
                    </window>
                </windowList>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Printing Preferences"/>
        </expectedWindow>
    </step>

    <!-- In this step, we acquire the dialog "Printing Preferences' and select the combo box
which is the 10th child of window with caption '"Adobe PDF Settings' and select the first index.
(Note: All indeces start with 0.) Besides this we uncheck the box which  has the caption '"View
Adobe PDF results' and we click on the button OK. The expectation is that 'Printing Preferences'
dialog disappears. -->
    <step>
        <acquiredWindow>
            <window caption="Printing Preferences">
                <windowList>
                    <window caption="Adobe PDF Settings">
                        <windowList>
                            <window className="Button" caption="View Adobe PDF results"
action="uncheck"/>
                        </windowList>
```

```
                        <windowList>
                            <window className="Button" caption="Ask to Replace existing PDF
file" action="uncheck"/>
                        </windowList>
                    </window>
                </windowList>
                <windowList>
                    <window className="Button" caption="OK" action="press"/>
                </windowList>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Printing Preferences" action="disappear"/>
        </expectedWindow>
    </step>

    <!-- In this step, we acquire the 'Print' dialog and click on the Print button. The
expectation is that the dialog with caption 'Print' disappears. In this case we use the regular
expression '^Print$' for specifying the caption given there could be multiple dialogs with
caption that includes the word Print. -->
    <step>
        <acquiredWindow>
            <window caption="Print">
                <windowList>
                    <window caption="General"/>
                    <window className="Button" caption="^Print$" action="press"/>
                </windowList>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Print" action="disappear"/>
        </expectedWindow>
    </step>
    <step>
        <expectedWindow>
            <window caption="Save PDF File As"/>
        </expectedWindow>
    </step>
    <!-- Finally in this step, we acquire the dialog with caption "Save PDF File As" and in
the Edit widget type the destination path for the output PDF file and click on the Save button.
The expectation is that the dialog disappears-->
```

```
    <step>
        <acquiredWindow>
            <window caption="Save PDF File As">
                <windowList>
                    <window className="Edit" action="inputDestinationPath"/>
                </windowList>
                <windowList>
                    <window className="Button" caption="Save" action="press"/>
                </windowList>
            </window>
        </acquiredWindow>
        <expectedWindow>
            <window caption="Save PDF File As" action="disappear"/>
        </expectedWindow>
    </step>

    <!-- We can always set a retry count or a maximum time for a step. In case we surpass these
limitations, PDF Generator generates this abort message and terminates processing. -->
    <abortMessage msg="15078"/>
</application>
```

Encrypting and Decrypting PDF Documents

About the Encryption Service

The Encryption service lets you encrypt and decrypt documents. When a document is encrypted, its contents become unreadable. An authorized user can decrypt the document to obtain access to the contents. If a PDF document is encrypted with a password, the user must specify the open password before the document can be viewed in Adobe Reader or Adobe Acrobat. Likewise, if a PDF document is encrypted with a certificate, the user must decrypt the PDF document with the public key that corresponds to the certificate (private key) that was used to encrypt the PDF document.

You can accomplish these tasks using the Encryption service:

- Encrypt a PDF document with a password. (See "Encrypting PDF Documents with a Password" on page 806.)
- Encrypt a PDF document with a certificate. (See "Encrypting PDF Documents with Certificates" on page 811.)
- Remove password-based encryption from a PDF document. (See "Removing Password Encryption" on page 821.)
- Remove certificate-based encryption from a PDF document. (See "Removing Certificate Based Encryption" on page 817.)
- Unlock the PDF document so that other service operations can be performed. For example, after a password-encrypted PDF document is unlocked, you can apply a digital signature to it. (See "Unlocking Encrypted PDF Documents" on page 824.)
- Determine the encryption type of a secured PDF document. (See "Determining Encryption Type" on page 828.)

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Encrypting PDF Documents with a Password

When you encrypt a PDF document with a password, a user must specify the password to open the PDF document in Adobe Reader or Acrobat. Also, before another AEM Forms operation, such as digitally signing the PDF document, can be performed on the document, a password-encrypted PDF document must be unlocked.

Important: *If you upload an encrypted PDF document to the AEM Forms repository, it cannot decrypt the PDF document and extract the XDP content. It is recommended that you do not encrypt a document prior to uploading it to the AEM Forms repository. (See "Writing Resources" on page 1037.)*

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Summary of steps

To encrypt a PDF document with a password, perform the following steps:

1. Include project files.
2. Create an Encryption Client API object.
3. Get a PDF document to encrypt.
4. Set encryption run-time options.
5. Add the password.
6. Save the encrypted PDF document as a PDF file.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-encryption-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create an Encryption Client API object

To programmatically perform an Encryption service operation, you must create an Encryption service client.

Get a PDF document to encrypt

You must obtain an unencrypted PDF document to encrypt the document with a password. If you attempt to secure a PDF document that is already encrypted, you cause an exception.

Set encryption run-time options

To encrypt a PDF document with a password, you specify four values, including two password values. The first password value is used to encrypt the PDF document and must be specified when opening the PDF document. The second password value, named the master password value, is used to remove encryption from the PDF document. Password values are case sensitive, and these two password values cannot be the same values.

You must specify the PDF document resources to encrypt. You can encrypt the entire PDF document, everything except for the document's metadata, or just the document's attachments. If you encrypt only the document's attachments, a user is prompted for a password when they attempt to access the file attachments.

When encrypting a PDF document, you can specify permissions that are associated with the secured document. By specifying permissions, you can control the actions that a user who opens a password-encrypted PDF document is allowed to perform. For example to successfully extract form data, you must set the following permissions:

- PASSWORD_EDIT_ADD
- PASSWORD_EDIT_MODIFY

Note: *Permissions are specified as* `PasswordEncryptionPermission` *enumeration values.*

Add the password

After you retrieve an unsecured PDF document and set encryption run-time values, you can add a password to the PDF document.

Save the encrypted PDF document as a PDF file

You can save the password-encrypted PDF document as a PDF file.

See also

"Encrypt a PDF document using the Java API" on page 808

"Encrypting a PDF document using the web service API" on page 809

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encryption Service Java API Quick Start(SOAP)" on page 117

"Encrypting PDF Documents with Certificates" on page 811

Encrypt a PDF document using the Java API

Encrypt a PDF document with a password by using the Encryption API (Java):

1 Include project files.

   Include client JAR files, such as adobe-encryption-client.jar, in your Java project's class path.

2 Create an Encryption Client API.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EncryptionServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get a PDF document to encrypt.

   - Create a `java.io.FileInputStream` object that represents the PDF document to encrypt by using its constructor and passing a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set encryption run-time options.

   - Create a `PasswordEncryptionOptionSpec` object by invoking its constructor.
   - Specify the PDF document resources to encrypt by invoking the `PasswordEncryptionOptionSpec` object's `setEncryptOption` method and passing a `PasswordEncryptionOption` enumeration value that specifies the document resources to encrypt. For example, to encrypt the entire PDF document, including its metadata and its attachments, specify `PasswordEncryptionOption.ALL`.

- Create a `java.util.List` object that stores the encryption permissions by using the `ArrayList` constructor.
- Specify a permission by invoking the `java.util.List` object 's `add` method and passing an enumeration value that corresponds to the permission that you want to set. For example, to set the permission that lets a user copy data located in the PDF document, specify `PasswordEncryptionPermission.PASSWORD_EDIT_COPY`. (Repeat this step for each permission to set).
- Specify the Acrobat compatibility option by invoking the `PasswordEncryptionOptionSpec` object's `setCompatability` method and passing an enumeration value that specifies the Acrobat compatibility level. For example, you can specify `PasswordEncryptionCompatability.ACRO_7`.
- Specify the password value that lets a user open the encrypted PDF document by invoking the `PasswordEncryptionOptionSpec` object's `setDocumentOpenPassword` method and passing a string value that represents the open password.
- Specify the master password value that lets a user remove encryption from the PDF document by invoking the `PasswordEncryptionOptionSpec` object's `setPermissionPassword` method and passing a string value that represents the master password.

5 Add the password.

Encrypt the PDF document by invoking the `EncryptionServiceClient` object's `encryptPDFUsingPassword` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the PDF document to encrypt with the password.
- The `PasswordEncryptionOptionSpec` object that contains encryption run-time options.

The `encryptPDFUsingPassword` method returns a `com.adobe.idp.Document` object that contains a password-encrypted PDF document.

6 Save the encrypted PDF document as a PDF file.

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that was returned by the `encryptPDFUsingPassword` method.

See also

"Summary of steps" on page 807

"Quick Start (SOAP mode): Encrypting a PDF document using the Java API" on page 118

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Encrypting a PDF document using the web service API

Encrypt a PDF document with a password by using the Encryption API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/EncryptionService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Encryption Client API object.

- Create an `EncryptionServiceClient` object by using its default constructor.

- Create an `EncryptionServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/EncryptionService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptionServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `EncryptionServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `EncryptionServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get a PDF document to encrypt.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is encrypted with a password.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to encrypt and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning the contents of the byte array to the `BLOB` object's `MTOM` data member.

4 Set encryption run-time options.

- Create a `PasswordEncryptionOptionSpec` object by using its constructor.
- Specify the PDF document resources to encrypt by assigning a `PasswordEncryptionOption` enumeration value to the `PasswordEncryptionOptionSpec` object's `encryptOption` data member. To encrypt the entire PDF, including its metadata and its attachments, assign `PasswordEncryptionOption.ALL` to this data member.
- Specify the Acrobat compatibility option by assigning a `PasswordEncryptionCompatability` enumeration value to the `PasswordEncryptionOptionSpec` object's `compatability` data member. For example, assign `PasswordEncryptionCompatability.ACRO_7` to this data member.
- Specify the password value that lets a user open the encrypted PDF document by assigning a string value that represents the open password to the `PasswordEncryptionOptionSpec` object's `documentOpenPassword` data member.
- Specify the password value that lets a user remove encryption from the PDF document by assigning a string value that represents the master password to the `PasswordEncryptionOptionSpec` object's `permissionPassword` data member.

5 Add the password.

Encrypt the PDF document by invoking the `EncryptionServiceClient` object's `encryptPDFUsingPassword` method and passing the following values:

- The `BLOB` object that contains the PDF document to encrypt with the password.
- The `PasswordEncryptionOptionSpec` object that contains encryption run-time options.

The `encryptPDFUsingPassword` method returns a `BLOB` object that contains a password-encrypted PDF document.

6 Save the encrypted PDF document as a PDF file.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the secured PDF document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `encryptPDFUsingPassword` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 807

Quick Start (MTOM): Encrypting a PDF document using the web service API

Quick Start (Java SwaRef): Encrypting a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Encrypting PDF Documents with Certificates

Certificate-based encryption lets you encrypt a document for specific recipients by means of public key technology. Various recipients can be given different permissions for the document. Many aspects of encryption are made possible by public key technology. An algorithm is used to generate two large numbers, known as *keys*, that have the following properties:

- One key is used to encrypt a set of data. Subsequently, only the other key can be used to decrypt the data.
- It is impossible to distinguish one key from the other.

One of the keys acts as a user's private key. It is important that only the user has access to this key. The other key is the user's public key, which can be shared with others.

A public key certificate contains a user's public key and identifying information. The X.509 format is used for storing certificates. Certificates are typically issued and digitally signed by a certificate authority (CA), which is a recognized entity that provides a measure of confidence in the validity of the certificate. Certificates have an expiration date, after which they are no longer valid. In addition, certificate revocation lists (CRLs) provide information about certificates that were revoked prior to their expiration date. CRLs are published periodically by certificate authorities. The revocation status of a certificate can also be retrieved through Online Certificate Status Protocol (OCSP) over the network.

Note: *If you upload an encrypted PDF document to the AEM Forms repository, it cannot decrypt the PDF document and extract the XDP content. It is recommended that you do not encrypt a document prior to uploading it to the AEM Forms repository. (See "Writing Resources" on page 1037.)*

Note: *Before you can encrypt a PDF document with a certificate, you must ensure that you add the certificate to AEM Forms. A certificate is added using administration console or programmatically using the Trust Manager API. (See "Importing Credentials by using the Trust Manager API" on page 1060.)*

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Summary of steps

To encrypt a PDF document with a certificate, perform the following steps:

1. Include project files.
2. Create an Encryption Client API object.
3. Get a PDF document to encrypt.
4. Reference the certificate.
5. Set encryption run-time options.
6. Create a certificate-encrypted PDF document.
7. Save the encrypted PDF document as a PDF file.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-encryption-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

Create an Encryption Client API object

To programmatically perform an Encryption service operation, you must create an Encryption service client. If you are using the Java Encryption Service API, create an `EncrytionServiceClient` object. If you are using the web service Encryption Service API, create an `EncryptionServiceService` object.

Get a PDF document to encrypt

You must obtain an unencrypted PDF document to encrypt. If you attempt to secure a PDF document that is already encrypted, an exception is thrown.

Reference the certificate

To encrypt a PDF document with a certificate, reference a certificate that is used to encrypt a PDF document. The certificate is a .cer file, a .crt file, or a .pem file. A PKCS#12 file is used to store private keys with corresponding certificates.

When encrypting a PDF document with a certificate, specify permissions that are associated with the secured document. By specifying permissions, you can control the actions that a user who opens a certificate-encrypted PDF document can perform.

Set encryption run-time options
Specify the PDF document resources to encrypt. You can encrypt the entire PDF document, everything except the document's metadata, or only the document's attachments.

Create a certificate-encrypted PDF document
After you retrieve an unsecured PDF document, reference the certificate, and set run-time options, you can create a certificate-encrypted PDF document. After the PDF document is encrypted, you need the corresponding public key to decrypt it.

Save the encrypted PDF document as a PDF file
You can save the encrypted PDF document as a PDF file.

See also
"Encrypt a PDF document with a certificate using the Java API" on page 813

"Encrypt a PDF document with a certificate using the web service API" on page 815

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encryption Service Java API Quick Start(SOAP)" on page 117

"Encrypting PDF Documents with a Password" on page 806

Encrypt a PDF document with a certificate using the Java API

Encrypt a PDF document with a certificate by using the Encryption API (Java):

1 Include project files.

   Include client JAR files, such as adobe-encryption-client.jar, in your Java project's class path.

2 Create an Encryption Client API object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EncryptionServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get a PDF document to encrypt.

   - Create a `java.io.FileInputStream` object that represents the PDF document to encrypt by using its constructor and passing a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference the certificate.

   - Create a `java.util.List` object that stores permission information by using its constructor.

- Specify the permission associated with the encrypted document by invoking the `java.util.List` object's `add` method and passing a `CertificateEncryptionPermissions` enumeration value that represents the permissions that are granted to the user who opens the secured PDF document. For example, to specify all permissions, pass `CertificateEncryptionPermissions.PKI_ALL_PERM`.
- Create a `Recipient` object by using its constructor.
- Create a `java.io.FileInputStream` object that represents the certificate that is used to encrypt the PDF document by using its constructor and passing a string value that specifies the location of the certificate.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object that represents the certificate.
- Invoke the `Recipient` object's `setX509Cert` method and pass the `com.adobe.idp.Document` object that contains the certificate. (In addition, the `Recipient` object can have a Truststore certificate alias or LDAP URL as a certificate source.)
- Create a `CertificateEncryptionIdentity` object that stores permission and certificate information by using its constructor.
- Invoke the `CertificateEncryptionIdentity` object's `setPerms` method and pass the `java.util.List` object that stores permission information.
- Invoke the `CertificateEncryptionIdentity` object's `setRecipient` method and pass the `Recipient` object that stores certificate information.
- Create a `java.util.List` object that stores certificate information by using its constructor.
- Invoke the `java.util.List` object's add method and pass the `CertificateEncryptionIdentity` object. (This `java.util.List` object is passed as a parameter to the `encryptPDFUsingCertificates` method.)

5 Set encryption run-time options.

- Create a `CertificateEncryptionOptionSpec` object by invoking its constructor.
- Specify the PDF document resources to encrypt by invoking the `CertificateEncryptionOptionSpec` object's `setOption` method and passing a `CertificateEncryptionOption` enumeration value that specifies the document resources to encrypt. For example, to encrypt the entire PDF document, including its metadata and its attachments, specify `CertificateEncryptionOption.ALL`.
- Specify the Acrobat compatibility option by invoking the `CertificateEncryptionOptionSpec` object's `setCompat` method and passing a `CertificateEncryptionCompatibility` enumeration value that specifies the Acrobat compatibility level. For example, you can specify `CertificateEncryptionCompatibility.ACRO_7`.

6 Create a certificate-encrypted PDF document.

Encrypt the PDF document with a certificate by invoking the `EncryptionServiceClient` object's `encryptPDFUsingCertificates` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the PDF document to encrypt.
- The `java.util.List` object that stores certificate information.
- The `CertificateEncryptionOptionSpec` object that contains encryption run-time options.

The `encryptPDFUsingCertificates` method returns a `com.adobe.idp.Document` object that contains a certificate-encrypted PDF document.

7 Save the encrypted PDF document as a PDF file.

- Create a `java.io.File` object and ensure that the file name extension is .pdf.

- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that was returned by the `encryptPDFUsingCertificates` method.

See also

"Summary of steps" on page 812

"Quick Start (SOAP mode): Encrypting a PDF document with a certificate using the Java API" on page 122

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Encrypt a PDF document with a certificate using the web service API

Encrypt a PDF document with a certificate by using the Encryption API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/EncryptionService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Encryption Client API object.

- Create an `EncryptionServiceClient` object by using its default constructor.
- Create an `EncryptionServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/EncryptionService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptionServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `EncryptionServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `EncryptionServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get a PDF document to encrypt.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is encrypted with a certificate.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to encrypt and the mode in which to open the file.

- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Reference the certificate.

- Create a `Recipient` object by using its constructor. This object will store certificate information.
- Create a `BLOB` object by using its constructor. This `BLOB` object will store the certificate that encrypts the PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the certificate and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning the contents of the byte array to the `BLOB` object's `MTOM` data member.
- Assign the `BLOB` object that stores the certificate to the `Recipient` object's `x509Cert` data member.
- Create a `CertificateEncryptionIdentity` object that stores certificate information by using its constructor.
- Assign the `Recipient` object that stores the certificate to the `CertificateEncryptionIdentity` object's recipient data member.
- Create an `Object` array and assign the `CertificateEncryptionIdentity` object to the first element of the `Object` array. This `Object` array is passed as a parameter to the `encryptPDFUsingCertificates` method.

5 Set encryption run-time options.

- Create a `CertificateEncryptionOptionSpec` object by using its constructor.
- Specify the PDF document resources to encrypt by assigning a `CertificateEncryptionOption` enumeration value to the `CertificateEncryptionOptionSpec` object's `option` data member. To encrypt the entire PDF document, including its metadata and its attachments, assign `CertificateEncryptionOption.ALL` to this data member.
- Specify the Acrobat compatibility option by assigning a `CertificateEncryptionCompatibility` enumeration value to the `CertificateEncryptionOptionSpec` object's `compat` data member. For example, assign `CertificateEncryptionCompatibility.ACRO_7` to this data member.

6 Create a certificate-encrypted PDF document.

Encrypt the PDF document with a certificate by invoking the `EncryptionServiceService` object's `encryptPDFUsingCertificates` method and passing the following values:

- The `BLOB` object that contains the PDF document to encrypt.
- The `Object` array that stores certificate information.
- The `CertificateEncryptionOptionSpec` object that contains encryption run-time options.

The `encryptPDFUsingCertificates` method returns a `BLOB` object that contains a certificate-encrypted PDF document.

7 Save the encrypted PDF document as a PDF file.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the secured PDF document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `encryptPDFUsingCertificates` method. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 812

Quick Start (MTOM): Encrypting a PDF document with a certificate using the web service API

Quick Start (Java SwaRef): Encrypting a PDF document with a certificate using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Removing Certificate Based Encryption

Certificate-based encryption can be removed from a PDF document so that users can open the PDF document in Adobe Reader or Acrobat. To remove encryption from a PDF document that is encrypted with a certificate, a public key must be referenced. After encryption is removed from a PDF document, it is no longer secure.

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Summary of steps

To remove certificate-based encryption from a PDF document, perform the following steps:

1 Include project files.

2 Create an encryption service client.

3 Get the encrypted PDF document.

4 Remove encryption.

5 Save the PDF document as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-encryption-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

Create an encryption service client

To programmatically perform an Encryption service operation, you must create an Encryption service client. If you are using the Java Encryption Service API, create an `EncrytionServiceClient` object. If you are using the web service Encryption Service API, create an `EncryptionServiceService` object.

Get the encrypted PDF document

You must obtain an encrypted PDF document to remove certificate-based encryption. If you attempt to remove encryption from a PDF document that is not encrypted, an exception is thrown. Likewise, if you attempt to remove certificate-based encryption from a password-encrypted document, an exception is thrown.

Remove encryption

To remove certificate-based encryption from an encrypted PDF document, you require both an encrypted PDF document and the private key that corresponds to the key that was used to encrypt the PDF document. The alias value of the private key is specified when removing certificate-based encryption from an encrypted PDF document. For information about the public key, see "Encrypting PDF Documents with Certificates" on page 811.

Note: *A private key is stored in the AEM Forms Trust Store. When a certificate is placed there, an alias value is specified.*

Save the PDF document

After certificate-based encryption is removed from an encrypted PDF document, you can save the PDF document as a PDF file. Users can open the PDF document in Adobe Reader or Acrobat.

See also

"Remove certificate-based encryption using the Java API" on page 818

"Remove certificate-based encryption using the web service API" on page 819

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encryption Service Java API Quick Start(SOAP)" on page 117

Remove certificate-based encryption using the Java API

Remove certificate-based encryption from a PDF document by using the Encryption API (Java):

1 Include project files.

   Include client JAR files, such as adobe-encryption-client.jar, in your Java project's class path.

2 Create an encryption service client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EncryptionServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the encrypted PDF document.

   - Create a `java.io.FileInputStream` object that represents the encrypted PDF document by using its constructor and passing a string value that specifies the location of the encrypted PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Remove encryption.

Remove certificate-based encryption from the PDF document by invoking the `EncryptionServiceClient` object's `removePDFCertificateSecurity` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the encrypted PDF document.
- A string value that specifies the alias name of the private key that corresponds to the key used to encrypt the PDf document.

The `removePDFCertificateSecurity` method returns a `com.adobe.idp.Document` object that contains an unsecured PDF document.

5 Save the PDF document.

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that was returned by the `removePDFCredentialSecurity` method.

See also

"Summary of steps" on page 817

"Quick Start (SOAP mode): Removing certificate-based encryption using the Java API" on page 125

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Remove certificate-based encryption using the web service API

Remove certificate-based encryption by using the Encryption API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/EncryptionService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an encryption service client.

- Create an `EncryptionServiceClient` object by using its default constructor.
- Create an `EncryptionServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/EncryptionService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptionServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `EncryptionServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `EncryptionServiceClient.ClientCredentials.UserName.Password`.

- Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the encrypted PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the encrypted PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the encrypted PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning the contents of the byte array to the `BLOB` object's `MTOM` data member.

4 Remove encryption.

Invoke the `EncryptionServiceClient` object's `removePDFCertificateSecurity` method and pass the following values:

- The `BLOB` object that contains file stream data that represents an encrypted PDF document.
- A string value that specifies the alias name of the public key that corresponds to the private key used to encrypt the PDf document.

The `removePDFCredentialSecurity` method returns a `BLOB` object that contains an unsecured PDF document.

5 Save the PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the unsecured PDF document.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `removePDFPasswordSecurity` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Summary of steps" on page 817

Quick Start (MTOM): Removing certificate-based encryption using the web service API

Quick Start (Java SWAref): Removing certificate-based encryption using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Removing Password Encryption

Password-based encryption can be removed from a PDF document so that users can open the PDF document in Adobe Reader or Acrobat without having to specify a password. After password-based encryption is removed from a PDF document, the document is no longer secure.

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Summary of steps

To remove password-based encryption from a PDF document, perform the following steps:

1. Include project files
2. Create an encryption service client.
3. Get the encrypted PDF document.
4. Remove the password.
5. Save the PDF document as a PDF file.

Include project files

Include the necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-encryption-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create an encryption service client

To programmatically perform an Encryption service operation, you must create an Encryption service client. If you are using the Java Encryption Service API, create an `EncrytionServiceClient` object. If you are using the web service Encryption Service API, create an `EncryptionServiceService` object.

Get the encrypted PDF document

You must obtain an encrypted PDF document to remove password-based encryption. If you attempt to remove encryption from a PDF document that is not encrypted, an exception is thrown.

Remove the password

To remove password-based encryption from an encrypted PDF document, you require both an encrypted PDF document and a master password value that is used to remove encryption from the PDF document. The password that is used to open a password-encrypted PDF document cannot be used to remove encryption. A master password is specified when the PDF document is encrypted with a password. (See "Encrypting PDF Documents with a Password" on page 806.)

Save the PDF document

After the Encryption service removes password-based encryption from a PDF document, you can save the PDF document as a PDF file. Users can open the PDF document in Adobe Reader or Acrobat without specifying a password.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encryption Service Java API Quick Start(SOAP)" on page 117

"Encrypting PDF Documents with a Password" on page 806

Remove password-based encryption using the Java API

Remove password-based encryption from a PDF document by using the Encryption API (Java):

1 Include project files.

Include client JAR files, such as the adobe-encryption-client.jar, in your Java project's class path.

2 Create an encryption service client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `EncryptionServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the encrypted PDF document.

- Create a `java.io.FileInputStream` object that represents the encrypted PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Remove the password.

Remove password-based encryption from the PDF document by invoking the `EncryptionServiceClient` object's `removePDFPasswordSecurity` method and passing the following values:

- A `com.adobe.idp.Document` object that contains the encrypted PDF document.
- A string value that specifies the master password value that is used to remove encryption from the PDF document.

The `removePDFPasswordSecurity` method returns a `com.adobe.idp.Document` object that contains an unsecured PDF document.

5 Save the PDF document.

- Create a `java.io.File` object and ensure that the file name extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `Document` object to the file. Ensure that you use the `Document` object that was returned by the `removePDFPasswordSecurity` method.

See also

"Quick Start (SOAP mode): Removing password-based encryption using the Java API" on page 120

Remove password-based encryption using the web service API

Remove password-based encryption by using the Encryption API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/EncryptionService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an encryption service client.

- Create an `EncryptionServiceClient` object by using its default constructor.
- Create an `EncryptionServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/EncryptionService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptionServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `EncryptionServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `EncryptionServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the encrypted PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a password-encrypted PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the encrypted PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning the contents of the byte array to the `BLOB` object's `MTOM` data member.

4 Remove the password.

Invoke the `EncryptionServiceService` object's `removePDFPasswordSecurity` method and pass the following values:

- The `BLOB` object that contains file stream data that represents an encrypted PDF document.
- A string value that specifies the password value that is used to remove encryption from the PDF document. This value is specified when encrypting the PDF document with a password.

The `removePDFPasswordSecurity` method returns a `BLOB` object that contains an unsecured PDF document.

5 Save the PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the unsecured PDF document.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `removePDFPasswordSecurity` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

Quick Start (MTOM): Removing password-based encryption using the web service API

Quick Start (Java SWAref): Removing password-based encryption using web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Unlocking Encrypted PDF Documents

A password-encrypted or certificate-encrypted PDF document must be unlocked before another AEM Forms operation can be performed on it. If you attempt to perform an operation on an encrypted PDF document, you will generate an exception. After you unlock an encrypted PDF document, you can perform one or more operations on it. These operations can belong to other services, such as the Acrobat Reader DC extensions Service.

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Summary of steps

To unlock an encrypted PDF document, perform the following steps:

1 Include project files.

2 Create an encryption service client.

3 Get the encrypted PDF document.

4 Unlock the document.

5 Perform an AEM Forms operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-encryption-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)

- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

Create an encryption service client

To programmatically perform an Encryption service operation, you must create an Encryption service client. If you are using the Java Encryption Service API, create an `EncrytionServiceClient` object. If you are using the web service Encryption Service API, create an `EncryptionServiceService` object.

Get the encrypted PDF document

You must obtain an encrypted PDF document in order to unlock it. If you attempt to unlock a PDF document that is not encrypted, an exception is thrown.

Unlock the document

To unlock a password-encrypted PDF document, you require both an encrypted PDF document and a password value that is used to open a password-encrypted PDF document. This value is specified when encrypting the PDF document with a password. (See "Encrypting PDF Documents with a Password" on page 806.)

To unlock a certificate-encrypted PDF document, you require both an encrypted PDF document and the alias value of the public key that corresponds to the private key that was used to encrypt the PDF document.

Perform an AEM Forms operation

After an encrypted PDF document is unlocked, you can perform another service operation on it, such as applying usage rights to it. This operation belongs to the Acrobat Reader DC Extensions service.

See also

"Unlock an encrypted PDF document using the Java API" on page 825

"Unlock an encrypted PDF document using the web service API" on page 826

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encryption Service Java API Quick Start(SOAP)" on page 117

Unlock an encrypted PDF document using the Java API

Unlock an encrypted PDF document by using the Encryption API (Java):

1. Include project files.

   Include client JAR files, such as adobe-encryption-client.jar, in your Java project's class path.

2. Create an encryption service client.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EncryptionServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Get the encrypted PDF document.
   - Create a `java.io.FileInputStream` object that represents the encrypted PDF document by using its constructor and passing a string value that specifies the location of the encrypted PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Unlock the document.

Unlock an encrypted PDF document by invoking the `EncryptionServiceClient` object's `unlockPDFUsingPassword` or `unlockPDFUsingCredential` method.

To unlock a PDF document that is encrypted with a password, invoke the `unlockPDFUsingPassword` method and pass the following values:

- A `com.adobe.idp.Document` object that contains the password-encrypted PDF document.
- A string value that specifies the password value that is used to open a password-encrypted PDF document. This value is specified when encrypting the PDF document with a password.

To unlock a PDF document that is encrypted with a certificate, invoke the `unlockPDFUsingCredential` method and pass the following values:

- A `com.adobe.idp.Document` object that contains the certificate-encrypted PDF document.
- A string value that specifies the alias name of the public key that corresponds to the private key used to encrypt the PDF document.

The `unlockPDFUsingPassword` and `unlockPDFUsingCredential` methods both return a `com.adobe.idp.Document` object that you pass to another AEM Forms Java method to perform an operation.

5 Perform a AEM Forms operation.

Perform a AEM Forms operation on the unlocked PDF document to meet your business requirements. For example, assuming that you want to apply usage rights to an unlocked PDF document, pass the `com.adobe.idp.Document` object that was returned by either the `unlockPDFUsingPassword` or `unlockPDFUsingCredential` methods to the `ReaderExtensionsServiceClient` object's `applyUsageRights` method.

See also

"Summary of steps" on page 824

"Quick Start (SOAP mode): Unlocking an encrypted PDF document using the Java API" on page 127 (SOAP mode)

"Applying Usage Rights to PDF Documents" on page 751

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Unlock an encrypted PDF document using the web service API

Unlock an encrypted PDF document by using the Encryption API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/EncryptionService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an encryption service client.

- Create an `EncryptionServiceClient` object by using its default constructor.
- Create an `EncryptionServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/EncryptionService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptionServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `EncryptionServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `EncryptionServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get an encrypted PDF document.

- Create a `BLOB` object by using its constructor.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the encrypted PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning the contents of the byte array to the `BLOB` object's `MTOM` data member.

4 Unlock the document.

Unlock an encrypted PDF document by invoking the `EncryptionServiceClient` object's `unlockPDFUsingPassword` or `unlockPDFUsingCredential` method.

To unlock a PDF document that is encrypted with a password, invoke the `unlockPDFUsingPassword` method and pass the following values:

- A `BLOB` object that contains the password-encrypted PDF document.
- A string value that specifies the password value that is used to open a password-encrypted PDF document. This value is specified when encrypting the PDF document with a password.

To unlock a PDF document that is encrypted with a certificate, invoke the `unlockPDFUsingCredential` method and pass the following values:

- A `BLOB` object that contains the certificate-encrypted PDF document.
- A string value that specifies the alias name of the public key that corresponds to the private key used to encrypt the PDf document.

The `unlockPDFUsingPassword` and `unlockPDFUsingCredential` methods both return a `com.adobe.idp.Document` object that you pass to another AEM Forms method to perform an operation.

5 Perform a AEM Forms operation.

Perform a AEM Forms operation on the unlocked PDF document to meet your business requirements. For example, assuming that you want to apply usage rights to the unlocked PDF document, pass the `BLOB` object that was returned by either the `unlockPDFUsingPassword` or `unlockPDFUsingCredential` methods to the `ReaderExtensionsServiceClient` object's `applyUsageRights` method.

See also

"Summary of steps" on page 824

Quick Start (MTOM): Unlocking an encrypted PDF document using the web service API

Quick Start (Java SwaRef): Unlocking an encrypted PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Determining Encryption Type

You can programmatically determine the type of encryption that is protecting a PDF document by using the Java Encryption Service API or the web service Encryption Service API. Sometimes it is necessary to dynamically determine whether a PDF document is encrypted and, if so, the encryption type. For example, you can determine whether a PDF document is protected with password-based encryption or a Rights Management policy.

A PDF document can be protected by the following encryption types:

- Password-based encryption
- Certificate-based encryption
- A policy that is created by the Rights Management service
- Another type of encryption

Note: *For more information about the Encryption service, see Services Reference for AEM Forms.*

Summary of steps

To determine the type of encryption that is protecting a PDF document, perform the following steps:

1. Include project files.
2. Create an encryption service client.
3. Get the encrypted PDF document.
4. Determine the encryption type.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-encryption-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

Create a service client

To programmatically perform an Encryption service operation, you must create an Encryption service client. If you are using the Java Encryption Service API, create an `EncrytionServiceClient` object. If you are using the web service Encryption Service API, create an `EncryptionServiceService` object.

Get the encrypted PDF document

You must obtain a PDF document to determine the type of encryption that is protecting it.

Determine the encryption type

You can determine the type of encryption that is protecting a PDF document. If the PDF document is not protected, then the Encryption service informs you that the PDF document is not secured.

See also

"Determine the encryption type using the Java API" on page 829

"Determine the encryption type using the web service API" on page 830

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encryption Service Java API Quick Start(SOAP)" on page 117

"Protecting Documents with Policies" on page 831

Determine the encryption type using the Java API

Determine the type of encryption that is protecting a PDF document by using the Encryption API (Java):

1. Include project files.

   Include client JAR files, such as adobe-encryption-client.jar, in your Java project's class path.

2. Create a service client.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EncryptionServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Get the encrypted PDF document.
   - Create a `java.io.FileInputStream` object that represents the PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Determine the encryption type.
   - Determine the encryption type by invoking the `EncryptionServiceClient` object's `getPDFEncryption` method and passing the`com.adobe.idp.Document` object that contains the PDF document. This method returns an `EncryptionTypeResult` object.
   - Invoke the `EncryptionTypeResult` object's `getEncryptionType` method. This method returns an `EncryptionType` enum value that specifies the encryption type. For example, if the PDF document is protected with password-based encryption, this method returns `EncryptionType.PASSWORD`.

See also

"Summary of steps" on page 828

"Quick Start (SOAP mode): Determining encryption type using the Java API" on page 128

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Determine the encryption type using the web service API

Determine the type of encryption that is protecting a PDF document by using the Encryption API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/EncryptionService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create a service client.

- Create an `EncryptionServiceClient` object by using its default constructor.
- Create an `EncryptionServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/EncryptionService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `EncryptionServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `EncryptionServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `EncryptionServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the encrypted PDF document.

- Create a `BLOB` object by using its constructor.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the encrypted PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning the contents of the byte array to the `BLOB` object's `MTOM` data member.

4 Determine the encryption type.

- Invoke the `EncryptionServiceClient` object's `getPDFEncryption` method and pass the `BLOB` object that contains the PDF document. This method returns an `EncryptionTypeResult` object.
- Get the value of the `EncryptionTypeResult` object's `encryptionType` data method. For example, if the PDF document is protected with password-based encryption, the value of this data member is `EncryptionType.PASSWORD`.

See also

"Summary of steps" on page 828

Quick Start (MTOM): Determining encryption type using the web service API

Quick Start (Java SWAref): Determining encryption type using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Protecting Documents with Policies

About the Document Security Service

The Document Security service enables users to dynamically apply confidentiality settings to Adobe PDF documents and to maintain control over the documents, no matter how widely they are distributed.

The Document Security service prevents information from spreading beyond the user's reach by enabling the users to maintain control over how recipients use the policy-protected PDF document. A user can specify who can open a document, limit how they can use it, and monitor the document after it is distributed. A user can also dynamically control access to a policy-protected document and can even dynamically revoke access to the document.

The Document Security service also protects other file types such as Microsoft Word files (DOC files). You can use the Document Security Client API to work with these file types. The following versions are supported:

- Microsoft Office 2003 files (DOC, XLS, PPT files)
- Microsoft Office 2007 files (DOCX, XLSX, PPTX files)
- PTC Pro/E files

For clarity, the following two sections discuss how to work with Word documents:

- "Applying Policies to Word Documents" on page 872
- "Removing Policies from Word Documents" on page 876

You can accomplish these tasks using the Document Security service:

- Create policies. For information, see "Creating Policies" on page 832.
- Modify policies. For information, see "Modifying Policies" on page 839.
- Delete policies. For information, see "Deleting Policies" on page 842.
- Apply policies to PDF documents. For information, see "Applying Policies to PDF Documents" on page 844.
- Remove policies from PDF documents. For information, see "Removing Policies from PDF Documents" on page 848.
- Inspect policy-protected documents. For information, see "Inspecting Policy Protected PDF Documents" on page 857.

- Revoke access to PDF documents. For information, see "Revoking Access to Documents" on page 851.
- Reinstate access to revoked documents. For information, see "Reinstating Access to Revoked Documents" on page 854.
- Create watermarks. For information, see "Creating Watermarks" on page 861.
- Search for events. For information, see "Searching for Events" on page 868.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms. For examples of working with policies, see the "Document Security Service API Quick Starts" in "Java API(SOAP) Quick Start (Code Examples)" on page 2.*

Creating Policies

You can programmatically create policies using the Document Security Java API or web service API. A *policy* is a collection of information that includes document security settings, authorized users, and usage rights. You can create and save any number of policies, using security settings appropriate for different situations and users.

Policies enable you to perform these tasks:

- Specify the individuals who can open the document. Recipients can either belong to or be external to your organization.
- Specify how recipients can use the document. You can restrict access to different Acrobat and Adobe Reader features. These features include the ability to print and copy text, add signatures, and add comments to a document.
- Change the access and security settings at any time, even after you distribute the policy-protected document.
- Monitor the use of the document after you distribute it. You can see how the document is being used and who is using it. For example, you can find out when someone has opened the document.

Creating a policy using web services

When creating a policy using the web service API, reference an existing Portable Document Rights Language (PDRL) XML file that describes the policy. Policy permissions and the principal are defined in the PDRL document. The following XML document is an example of a PDRL document.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Policy PolicyInstanceVersion="1" PolicyID="5DA3F847-DE76-F9CC-63EA-49A8D59154DE"
PolicyCreationTime="2004-08-30T00:02:28.294+00:00" PolicyType="1" PolicySchemaVersion="1.0"
PolicyName="SDK Test Policy -4344050357301573237" PolicyDescription="An SDK Test policy"
xmlns="http://www.adobe.com/schema/1.0/pdrl">
      <PolicyEntry>
         <ns1:Permission PermissionName="com.adobe.aps.onlineOpen" Access="ALLOW"
xmlns:ns1="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns2:Permission PermissionName="com.adobe.aps.offlineOpen" Access="ALLOW"
xmlns:ns2="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns3:Permission PermissionName="com.adobe.aps.pdf.editNotes" Access="ALLOW"
xmlns:ns3="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns4:Permission PermissionName="com.adobe.aps.pdf.fillAndSign" Access="ALLOW"
xmlns:ns4="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />
```

```
         <Principal PrincipalNameType="SYSTEM">
            <PrincipalDomain>EDC_SPECIAL</PrincipalDomain>

            <PrincipalName>all_internal_users</PrincipalName>
         </Principal>
      </PolicyEntry>

      <PolicyEntry>
         <ns5:Permission PermissionName="com.adobe.aps.onlineOpen" Access="ALLOW"
xmlns:ns5="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns6:Permission PermissionName="com.adobe.aps.offlineOpen" Access="ALLOW"
xmlns:ns6="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns7:Permission PermissionName="com.adobe.aps.pdf.copy" Access="ALLOW"
xmlns:ns7="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns8:Permission PermissionName="com.adobe.aps.pdf.printLow" Access="ALLOW"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex"
xmlns:ns8="http://www.adobe.com/schema/1.0/pdrl" />

         <ns9:Permission PermissionName="com.adobe.aps.policySwitch" Access="ALLOW"
xmlns:ns9="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns10:Permission PermissionName="com.adobe.aps.revoke" Access="ALLOW"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex"
xmlns:ns10="http://www.adobe.com/schema/1.0/pdrl" />

         <ns11:Permission PermissionName="com.adobe.aps.pdf.edit" Access="ALLOW"
xmlns:ns11="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns12:Permission PermissionName="com.adobe.aps.pdf.editNotes" Access="ALLOW"
xmlns:ns12="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns13:Permission PermissionName="com.adobe.aps.pdf.fillAndSign" Access="ALLOW"
xmlns:ns13="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <ns14:Permission PermissionName="com.adobe.aps.pdf.printHigh" Access="ALLOW"
xmlns:ns14="http://www.adobe.com/schema/1.0/pdrl"
xmlns="http://www.adobe.com/schema/1.0/pdrl-ex" />

         <Principal PrincipalNameType="SYSTEM">
            <PrincipalDomain>EDC_SPECIAL</PrincipalDomain>
```

```
            <PrincipalName>publisher</PrincipalName>
         </Principal>
      </PolicyEntry>

      <OfflineLeasePeriod>
         <Duration>P31D</Duration>
      </OfflineLeasePeriod>

      <AuditSettings isTracked="true" />

      <PolicyValidityPeriod isAbsoluteTime="false">
         <ValidityPeriodRelative>
            <NotBeforeRelative>PT0S</NotBeforeRelative>

            <NotAfterRelative>P20D</NotAfterRelative>
         </ValidityPeriodRelative>
      </PolicyValidityPeriod>
</Policy>
```

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To create a policy, perform the following steps:

1. Include project files.
2. Create a Document Security Client API object.
3. Set the policy's attributes.
4. Create a policy entry.
5. Register the policy.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-rightsmanagement-client.jar
- namespace.jar (if AEM Forms is deployed on JBoss)
- jaxb-api.jar (if AEM Forms is deployed on JBoss)
- jaxb-impl.jar (if AEM Forms is deployed on JBoss)
- jaxb-libs.jar (if AEM Forms is deployed on JBoss)
- jaxb-xjc.jar (if AEM Forms is deployed on JBoss)
- relaxngDatatype.jar (if AEM Forms is deployed on JBoss)
- xsdlib.jar (if AEM Forms is deployed on JBoss)
- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar

- jbossall-client.jar (use a different JAR file if AEM Forms is not deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, create a Document Security service client object.

Set the policy's attributes

To create a policy, set policy attributes. A mandatory attribute is the policy name. Policy names must be unique for each policy set. A policy set is simply a collection of policies. There can be two policies with the same name if the policies belong to separate policy sets. However, two policies within a single policy set cannot have the same policy name.

Another useful attribute to set is the validity period. A validity period is the time period during which a policy-protected document is accessible to authorized recipients. If you do not set this attribute, then the policy is always valid.

A validity period can be set to one of these options:

- A set number of days that the document is accessible from the time which the document is published
- An end date after which the document is not accessible
- A specific date range for which the document is accessible
- Always valid

You can specify just a start date, which results in the policy being valid after the start date. If you specify just an end date, the policy is valid until the end date. However, an exception is thrown if both a start date and an end date are not defined.

When setting attributes that belong to a policy, you can also set encryption settings. These encryption settings take affect when the policy is applied to a document. You can specify the following encryption values:

- **AES256**: Represents the AES encryption algorithm with a 256-bit key.
- **AES128**: Represents the AES encryption algorithm with a 128-bit key.
- **NoEncryption:** Represents no encryption.

When specifying the `NoEncryption` option, you cannot set the `PlaintextMetadata` option to `false`. If you attempt to do so, an exception is thrown.

Note: *For information about other attributes that you can set, see the* `Policy` *interface description in the AEM Forms API Reference.*

Create a policy entry

A policy entry attaches principals, which are groups and users, and permissions to a policy. A policy must have at least one policy entry. Assume, for example, that you perform these tasks:

- Create and register a policy entry that enables a group to only view a document while online and prohibits recipients from copying it.
- Attach the policy entry to the policy.
- Secure a document with the policy by using Acrobat.

These actions result in recipients only being able to view the document online and not being able to copy it. The document remains secure until security is removed from it.

Register the policy

A new policy must be registered before it can be used. After you register a policy, you can use it to protect documents.

Create a policy using the Java API

Create a policy by using the Document Security API (Java):

1 Include project files.

Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Document Security Client API object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `DocumentSecurityClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set the policy's attributes.

- Create a `Policy` object by invoking the `InfomodelObjectFactory` object's static `createPolicy` method. This method returns a `Policy` object.
- Set the policy's name attribute by invoking the `Policy` object's `setName` method and passing a string value that specifies the policy name.
- Set the policy's description by invoking the `Policy` object's `setDescription` method and passing a string value that specifies the policy's description.
- Set the policy set to which the new policy belongs by invoking the `Policy` object's `setPolicySetName` method and passing a string value that specifies the policy set name. (You can specify `null` for this parameter value that results in the policy being added to the *My Policies* policy set.)
- Create the policy's validity period by invoking the `InfomodelObjectFactory` object's static `createValidityPeriod` method. This method returns a `ValidityPeriod` object.
- Set the number of days for which a policy-protected document is accessible by invoking the `ValidityPeriod` object's `setRelativeExpirationDays` method and passing an integer value that specifies the number of days.
- Set the policy's validity period by invoking the `Policy` object's `setValidityPeriod` method and passing the `ValidityPeriod` object.

4 Create a policy entry.

- Create a policy entry by invoking the `InfomodelObjectFactory` object's static `createPolicyEntry` method. This method returns a `PolicyEntry` object.
- Specify the policy's permissions by invoking the `InfomodelObjectFactory` object's static `createPermission` method. Pass a static data member that belongs to the `Permission` interface that represents the permission. This method returns a `Permission` object. For example, to add the permission that enables users to copy data from a policy-protected PDF document, pass `Permission.COPY`. (Repeat this step for each permission to add).
- Add the permission to the policy entry by invoking the `PolicyEntry` object's `addPermission` method and passing the `Permission` object. (Repeat this step for each `Permission` object that you created).
- Create the policy principal by invoking the `InfomodelObjectFactory` object's static `createSpecialPrincipal` method. Pass a data member that belongs to the `InfomodelObjectFactory` object that represents the principal. This method returns a `Principal` object. For example, to add the publisher of the document as the principal, pass `InfomodelObjectFactory.PUBLISHER_PRINCIPAL`.
- Add the principal to the policy entry by invoking the `PolicyEntry` object's `setPrincipal` method and passing the `Principal` object.

- Add the policy entry to the policy by invoking the `Policy` object's `addPolicyEntry` method and passing the `PolicyEntry` object.

5 Register the policy.

- Create a `PolicyManager` object by invoking the `DocumentSecurityClient` object's `getPolicyManager` method.
- Register the policy by invoking the `PolicyManager` object's `registerPolicy` method and passing the following values:
  - The `Policy` object that represents the policy to register.
- A string value that represents the policy set that the policy belongs to.

If you use a AEM forms administrator account within connection settings to create the `DocumentSecurityClient` object, then specify the policy set name when you invoke the `registerPolicy` method. If you pass a `null` value for the policy set, the policy is created in the administrators *My Policies* policy set.

If you use a Document Security user within connection settings, then you can invoke the overloaded `registerPolicy` method that accepts only the policy. That is, you do not need to specify the policy set name. However, the policy is added to the policy set named *My Policies*. If you do not want to add the new policy to this policy set, then specify a policy set name when you invoke the `registerPolicy` method.

Note: *When creating a policy, reference an existing policy set. If you specify a policy set that does not exist, then an exception is thrown.*

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (SOAP mode): Creating a policy using the Java API"

Create a policy using the web service API

Create a policy by using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `DocumentSecurityServiceClient` object by using its default constructor.
- Create a `DocumentSecurityServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.

- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Set the policy's attributes.

- Create a `PolicySpec` object by using its constructor.
- Set the policy's name by assigning a string value to the `PolicySpec` object's `name` data member.
- Set the policy's description by assigning a string value to the `PolicySpec` object's `description` data member.
- Set the policy set to which the policy will belong by assigning a string value to the `PolicySpec` object's `policySetName` data member. You must specify an existing policy set name. (You can specify `null` for this parameter value that results in the policy being added to *My Policies*.)
- Set the policy's offline lease period by assigning an integer value to the `PolicySpec` object's `offlineLeasePeriod` data member.
- Set the `PolicySpec` object's `policyXml` data member with a string value that represents PDRL XML data. To perform this task, create a .NET `StreamReader` object by using its constructor. Pass the location of a PDRL XML file that represents the policy to the `StreamReader` constructor. Next, invoke the `StreamReader` object's `ReadLine` method and assign the return value to a string variable. Iterate through the `StreamReader` object until the `ReadLine` method returns null. Assign the string variable to the `PolicySpec` object's `policyXml` data member.

4 Create a policy entry.

It is not necessary to create a policy entry when creating a policy using the Document Security web service API. The policy entry is defined in the PDRL document.

5 Register the policy.

Register the policy by invoking the `DocumentSecurityServiceClient` object's `registerPolicy` method and passing the following values:

- The `PolicySpec` object that represents the policy to register.
- A string value that represents the policy set that the policy belongs to. You can specify a `null` value which results in the policy being added to the *MyPolices* policy set.

If you use a AEM forms administrator account within connection settings to create the `DocumentSecurityClient` object, specify the policy set name when you invoke the `registerPolicy` method.

If you use a Document SecurityDocument Security user within connection settings, then you can invoke the overloaded `registerPolicy` method that accepts only the policy. That is, you do not need to specify the policy set name. However, the policy is added to the policy set named *My Policies*. If you do not want to add the new policy to this policy set, then specify a policy set name when you invoke the `registerPolicy` method.

Note: *When creating a policy and you specify a policy set, ensure that you specify an existing policy set. If you specify a policy set that does not exist, then an exception is thrown.*

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (MTOM): Creating a policy using the web service API"
- "Quick Start (SwaRef): Creating a policy using the web service API"

More Help topics

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

PDRL Reference

"Applying Policies to PDF Documents" on page 844

Modifying Policies

You can modify an existing policy using the Document Security Java API or web service API. To make changes to an existing policy, you retrieve it, modify it, and then update the policy on the server. For example, assume that you retrieve an existing policy and extend its validity period. Before the change takes effect, you must update the policy.

You can modify a policy when business requirements change and the policy no longer reflects these requirements. Instead of creating a new policy, you can simply update an existing policy.

To modify policy attributes using a web service (for example, using Java proxy classes that were created with JAX-WS), you must ensure that the policy is registered with the Document Security service. You can then reference the existing policy by using the `PolicySpec.getPolicyXml` method and modify the policy attributes by using the applicable methods. For example, you can modify the offline lease period by invoking the `PolicySpec.setOfflineLeasePeriod` method.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To modify an existing policy, perform the following steps:

1. Include project files.
2. Create a Document Security Client API object.
3. Retrieve an existing policy.
4. Change policies attributes.
5. Update the policy.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Securityservice operation, you must create a Document Security service client object. If you are using the Java API, create a `RightsManagementClient` object. If you are using the Document Security web service API, create a `RightsManagementServiceService` object.

Retrieve an existing policy

You must retrieve an existing policy in order to modify it. To retrieve a policy, specify the policy name and the policy set to which the policy belongs. If you specify a `null` value for the policy set name, the policy is retrieved from the *My Policies* policy set.

Set the policy's attributes

To modify a policy, you modify the value of policy attributes. The only policy attribute that you cannot change is the name attribute. For example, to change the policy's offline lease period, you can modify the value of the policy's offline lease period attribute.

When modifying a policy's offline lease period using a web service, the `offlineLeasePeriod` field on the `PolicySpec` interface is ignored. To update the offline lease period, modify the `OfflineLeasePeriod` element in the PDRL XML document. Then reference the updated PDRL XML document by using the `PolicySpec` interface's `policyXML` data member.

Note: *For information about other attributes that you can set, see the `Policy` interface description in the AEM Forms API Reference.*

Update the policy

Before the changes that you make to a policy take affect, you must update the policy with the Document Security service. Changes to policies that are protecting documents are updated the next time that the policy-protected document is synchronized with the Document Security service.

Modify existing policies using the Java API

Modify an existing policy by using the Document Security API (Java):

1. Include project files.

   Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2. Create a Document Security Client API object.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `RightsManagementClient` object by using its constructor and passing the `ServiceClientFactory` object.

3. Retrieve an existing policy.
   - Create a `PolicyManager` object by invoking the `RightsManagementClient` object's `getPolicyManager` method.
   - Create a `Policy` object that represents the policy to update by invoking the `PolicyManager` object's `getPolicy` method and passing the following values"
     - A string value that represents the policy set name to which the policy belongs. You can specify `null` that results in the `MyPolicies` policy set being used.
     - A string value that represents the policy name.

4. Set the policy's attributes.

   Change the policy's attributes to meet your business requirements. For example, to change the policy's offline lease period, invoke the `Policy` object's `setOfflineLeasePeriod` method.

5. Update the policy.

   Update the policy by invoking `PolicyManager` object's `updatePolicy` method. Pass the `Policy` object that represents the policy to update.

Code examples

For code examples using the Document Security service, see theQuick Start(SOAP mode): Modifying a policy using the Java API in "*Java API(SOAP) Quick Start (Code Examples)*" on page 2.

Modify existing policies using the web service API

Modify an existing policy by using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `RightsManagementServiceClient` object by using its default constructor.
- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve an existing policy.

Create a `PolicySpec` object that represents the policy to modify by invoking the `RightsManagementServiceClient` object's `getPolicy` method and passing the following values:

- A string value that specifies the policy set name to which the policy belongs. You can specify `null` that results in the `MyPolicies` policy set being used.
- A string value that specifies the name of the policy.

4 Set the policy's attributes.

Change the policy's attributes to meet your business requirements.

5 Update the policy.

Update the policy by invoking the `RightsManagementServiceClient` object's `updatePolicyFromSDK` method and passing the `PolicySpec` object that represents the policy to update.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (MTOM): Modifying a policy using the web service API"
- "Quick Start (SwaRef): Modifying a policy using the web service API"

More Help topics

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Deleting Policies

You can delete an existing policy using the Document Security Java API or web service API. After a policy is deleted, it can no longer be used to protect documents. However, existing policy-protected documents that are using the policy are still protected. You can delete a policy when a newer one becomes available.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To delete an existing policy, perform the following steps:

1. Include project files
2. Create a Document Security Client API object.
3. Delete the policy.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, you must create a Document Security service client object. If you are using the Java API, create a `RightsManagementClient` object. If you are using the Document Security web service API, create a `RightsManagementServiceService` object.

Delete the policy

To delete a policy, you specify the policy to delete and the policy set to which the policy belongs. The user whose settings are used to invoke AEM Forms must have permission to delete the policy; otherwise an exception occurs. Likewise, if you attempt to delete a policy that does not exist, an exception occurs.

Delete policies using the Java API

Delete a policy by using the Document Security API (Java):

1. Include project files.

   Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2. Create a Document Security Client API object.
   - Create a `ServiceClientFactory` object that contains connection properties.

- Create a `RightsManagementClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Delete the policy.

- Create a `PolicyManager` object by invoking the `RightsManagementClient` object's `getPolicyManager` method.
- Delete the policy by invoking the `PolicyManager` object's `deletePolicy` method and passing the following values:
  - A string value that specifies the policy set name to which the policy belongs. You can specify `null` that results in the `MyPolicies` policy set being used.
  - A string value that specifies the name of the policy to delete.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (SOAP mode): Deleting a policy using the Java API"

Delete policies using the web service API

Delete a policy by using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `RightsManagementServiceClient` object by using its default constructor.
- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Delete the policy.

Delete a policy by invoking the `RightsManagementServiceClient` object's `deletePolicy` method and passing the following values:

- A string value that specifies the policy set name to which the policy belongs. You can specify `null` that results in the `MyPolicies` policy set being used.
- A string value that specifies the name of the policy to delete.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Deleting a policy using the web service API"
- "Quick Start (SwaRef): Deleting a policy using the web service API"

More Help topics

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Applying Policies to PDF Documents

You can apply a policy to a PDF document in order to secure the document. By applying a policy to a PDF document, you restrict access to the document. You cannot apply a policy to a document if the document is already secured with a policy.

While the document is open, you can also restrict access to Acrobat and Adobe Reader features, including the ability to print and copy text, make changes, and add signatures and comments to a document. In addition, you can revoke a policy-protected PDF document when you no longer want users to access the document.

You can monitor the use of a policy-protected document after you distribute it. That is, you can see how the document is being used and who is using it. For example, you can find out when somebody has opened the document.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To apply a policy to a PDF document, perform the following steps:

1 Include project files.

2 Create a Document Security Client API object.

3 Retrieve a PDF document to which a policy is applied.

4 Apply an existing policy to the PDF document.

5 Save the policy-protected PDF document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client APIobject

Before you can programmatically perform a Document Security service operation, create a Document Security service client object. If you are using the Java API, create a `DocumentSecurityClient` object. If you are using the Document Security web service API, create a `DocumentSecurityServiceService` object.

Retrieve a PDF document

You can retrieve a PDF document in order to apply a policy. After you apply a policy to the PDF document, users are restricted when using the document. For example, if the policy does not enable the document to be opened while offline, then users must be online to open the document.

Apply an existing policy to the PDF document

To apply a policy to a PDF document, reference an existing policy and specify which policy set the policy belongs to. The user who is setting the connection properties must have access to the specified policy. If not, an exception occurs.

Save the PDF document

After the Document Security service applies a policy to a PDF document, you can save the policy-protected PDF document as a PDF file.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Revoking Access to Documents" on page 851

Apply a policy to a PDF document using the Java API

Apply a policy to a PDF document by using the Document Security API (Java):

1 Include project files.

   Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Document Security Client API object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `RightsManagementClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve a PDF document.

   - Create a `java.io.FileInputStream` object that represents the PDF document by using its constructor. Pass a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Apply an existing policy to the PDF document.

   - Create a `DocumentManager` object by invoking the `RightsManagementClient` object's `getDocumentManager` method.
   - Apply a policy to the PDF document by invoking the `DocumentManager` object's `protectDocument` method and passing the following values:
     - The `com.adobe.idp.Document` object that contains the PDF document to which the policy is applied.
     - A string value that specifies the name of the document.

- A string value that specifies the name of the policy set to which the policy belongs. You can specify a `null` value that results in the `MyPolicies` policy set being used.
- A string value that specifies the policy name.
- A string value that represents the name of the user manager domain of the user who is the publisher of the document. This parameter value is optional and can be null (if this parameter is null, then the next parameter value must be null).
- A string value that represents the name of the canonical name of the user manager user who is the publisher of the document. This parameter value is optional and can be `null` (if this parameter is null, then the previous parameter value must be `null`).
- A `com.adobe.livecycle.rightsmanagement.Locale` that represents the locale that is used for selecting the MS Office template. This parameter value is optional and not used for PDF documents. To secure a PDF document, specify `null`.

The `protectDocument` method returns a `RMSecureDocumentResult` object that contains the policy-protected PDF document.

5 Save the PDF document.

- Invoke the `RMSecureDocumentResult` object's `getProtectedDoc` method to get the policy-protected PDF document. This method returns a `com.adobe.idp.Document` object.
- Create a `java.io.File` object and ensure that the file extension is PDF.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `getProtectedDoc` method).

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (EJB mode): Applying a policy to a PDF document using the Java API"
- "Quick Start (SOAP mode): Applying a policy to a PDF document using the Java API"

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Apply a policy to a PDF document using the web service API

Apply a policy to a PDF document by using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `RightsManagementServiceClient` object by using its default constructor.

- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document to which a policy is applied.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. Determine the byte array size by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Apply an existing policy to the PDF document.

Apply a policy to the PDF document by invoking the `RightsManagementServiceClient` object's `protectDocument` method and passing the following values:

- The `BLOB` object that contains the PDF document to which the policy is applied.
- A string value that specifies the name of the document.
- A string value that specifies the name of the policy set to which the policy belongs. You can specify a `null` value that results in the `MyPolicies` policy set being used.
- A string value that specifies the policy name.
- A string value that represents the name of the user manager domain of the user who is the publisher of the document. This parameter value is optional and can be null (if this parameter is null, then the next parameter value must be `null`).
- A string value that represents the name of the canonical name of the user manager user who is the publisher of the document. This parameter value is optional and can be null (if this parameter is null, then the previous parameter value must be `null`).
- A `RMLocale` value that specifies the locale value (for example, `RMLocale.en`).

- A string output parameter that is used to store the policy identifier value.
- A string output parameter that is used to store the policy-protected identifier value.
- A string output parameter that is used to store the mime type (for example, `application/pdf`).

The `protectDocument` method returns a `BLOB` object that contains the policy-protected PDF document.

5 Save the PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the policy-protected PDF document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `protectDocument` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Applying a policy to a PDF document using the web service API"
- "Quick Start (SwaRef): Applying a policy to a PDF document using the web service API "

Removing Policies from PDF Documents

You can remove a policy from a policy-protected document in order to remove security from the document. That is, if you no longer want the document to be protected by a policy. If you want to update a policy-protected document with a newer policy, then instead of removing the policy and adding the updated policy, it is more efficient to switch the policy.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To remove a policy from a policy-protected PDF document, perform the following steps:

1 Include project files

2 Create a Document Security Client API object.

3 Retrieve a policy-protected PDF document.

4 Remove the policy from the PDF document.

5 Save the unsecured PDF document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, create a Document Security service client object.

Retrieve a policy-protected PDF document

You can retrieve a policy-protected PDF document in order to remove a policy. If you attempt to remove a policy from a PDF document that is not protected by a policy, you will cause an exception.

Remove the policy from the PDF document

You can remove a policy from a policy-protected PDF document provided that an administrator is specified in the connection settings. If not, then the policy used to secure a document must contain the `SWITCH_POLICY` permission in order to remove a policy from a PDF document. Also, the user specified in the AEM Forms connection settings must also have that permission. Otherwise, an exception is thrown.

Save the unsecured PDF document

After the Document Security service removes a policy from a PDF document, you can save the unsecured PDF document as a PDF file.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Applying Policies to PDF Documents" on page 844

Remove a policy from a PDF document using the Java API

Remove a policy from a policy-protected PDF document by using the Document Security API (Java):

1. Include project files.

   Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2. Create a Document Security Client API object.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `DocumentSecurityClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Retrieve a policy-protected PDF document.
   - Create a `java.io.FileInputStream` object that represents the policy-protected PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Remove the policy from the PDF document.
   - Create a `DocumentManager` object by invoking the `DocumentSecurityClient` object's `getDocumentManager` method.
   - Remove a policy from the PDF document by invoking the `DocumentManager` object's `removeSecurity` method and passing the `com.adobe.idp.Document` object that contains the policy-protected PDF document. This method returns a `com.adobe.idp.Document` object that contains an unsecured PDF document.
5. Save the unsecured PDF document.
   - Create a `java.io.File` object and ensure that the file extension is PDF.
   - Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `removeSecurity` method).

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (SOAP mode): Removing a policy from a PDF document using the Java API"

Remove a policy using the web service API

Remove a policy from a policy-protected PDF document using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `DocumentSecurityServiceClient` object by using its default constructor.
- Create a `DocumentSecurityServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DocumentSecurityServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a policy-protected PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the policy-protected PDF document from which the policy is removed.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Remove the policy from the PDF document.

   Remove the policy from the PDF document by invoking the `DocumentSecurityServiceClient` object's `removePolicySecurity` method and passing the `BLOB` object that contains the policy-protected PDF document. This method returns a `BLOB` object that contains an unsecured PDF document.

5 Save the unsecured PDF document.

   - Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the unsecured PDF document.
   - Create a byte array that stores the data content of the `BLOB` object that was returned by the `removePolicySecurity` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
   - Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Removing a policy from a PDF document using the web service API "
- "Quick Start (SwaRef): Removing a policy from a PDF document using the web service API"

See also

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Revoking Access to Documents

You can revoke access to a policy-protected PDF document resulting in all copies of the document being inaccessible to users. When a user attempts to open a revoked PDF document, they are redirected to a specified URL where a revised document can be viewed. The URL to where the user is redirected must be programmatically specified. When you revoke access to a document, the change takes effect the next time the user synchronizes with the Document Security service by opening the policy-protected document online.

The ability to revoke access to a document provides additional security. For example, assume a newer version of a document is available and you no longer want anyone viewing the outdated version. In this situation, access to the older document can be revoked, and nobody can view the document unless access is reinstated.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To revoke a policy-protected document, perform the following steps:

1 Include project files.

2 Create a Document Security Client API object.

3 Retrieve a policy-protected PDF document.

4 Revoke the policy-protected document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, you must create a Document Security service client object.

Retrieve a policy-protected PDF document

You must retrieve a policy-protected PDF document in order to revoke it. You cannot revoke a document that has already been revoked or is not a policy-protected document.

If you know the license identifier value of the policy-protected document, then it is not necessary to retrieve the policy-protected PDF document. However, in most cases, you will need to retrieve the PDF document in order to obtain the license identifier value.

Revoke the policy-protected document

To revoke a policy-protected document, specify the license identifier of the policy-protected document. In addition, you can specify the URL of a document that the user can view when they attempt to open the revoked document. That is, assume that an outdated document is revoked. When a user attempts to open the revoked document, they will see an updated document instead of the revoked document.

Note: *If you attempt to revoke a document that is already revoked, an exception is thrown.*

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Applying Policies to PDF Documents" on page 844

"Reinstating Access to Revoked Documents" on page 854

Revoke access to documents using the Java API

Revoke access to a policy-protected PDF document by using the Document Security API (Java):

1 Include project files

Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Document Security Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `DocumentSecurityClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve a policy-protected PDF document

- Create a `java.io.FileInputStream` object that represent the policy-protected PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Revoke the policy-protected document

- Create a `DocumentManager` object by invoking the `DocumentSecurityClient` object's `getDocumentManager` method.

- Retrieve the license identifier value of the policy-protected document by invoking the `DocumentManager` object's `getLicenseId` method. Pass the `com.adobe.idp.Document` object that represents the policy-protected document. This method returns a string value that represents the license identifier value.
- Create a `LicenseManager` object by invoking the `DocumentSecurityClient` object's `getLicenseManager` method.
- Revoke the policy-protected document by invoking the `LicenseManager` object's `revokeLicense` method and passing the following values:
  - A string value that specifies the license identifier value of the policy-protected document (specify the return value of the `DocumentManager` object's `getLicenseId` method).
  - A static data member of the `License` interface that specifies the reason to revoke the document. For example, you can specify `License.DOCUMENT_REVISED`.
  - A `java.net.URL` value that specifies the location to where a revised document is located. If you do not want to redirect a user to another URL, then you can pass `null`.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (SOAP mode): Revoking a document using the Java API"

Revoke access to documents using the web service API

Revoke access to a policy-protected PDF document by using the Document Security API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object

- Create a `DocumentSecurityServiceClient` object by using its default constructor.
- Create a `DocumentSecurityServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DocumentSecurityServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.Password`.

- Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.

- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a policy-protected PDF document

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a policy-protected PDF document that is revoked.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the policy-protected PDF document to revoke and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Revoke the policy-protected document

- Retrieve the license identifier value of the policy-protected document by invoking the `DocumentSecurityServiceClient` object's `getLicenseID` method and passing the `BLOB` object that represents the policy-protected document. This method returns a string value that represents the license identifier.
- Revoke the policy-protected document by invoking the `DocumentSecurityServiceClient` object's `revokeLicense` method and passing the following values:
  - A string value that specifies the license identifier value of the policy-protected document (specify the return value of the `DocumentSecurityServiceService` object's `getLicenseId` method).
  - A static data member of the `Reason` enum that specifies the reason to revoke the document. For example, you can specify `Reason.DOCUMENT_REVISED`.
  - A `string` value that specifies the URL location to where a revised document is located. If you do not want to redirect a user to another URL, then you can pass `null`.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Revoking a document using the web service API"
- "Quick Start (SwaRef): Revoking a document using the web service API"

See also

"Removing Policies from Word Documents" on page 876

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Reinstating Access to Revoked Documents

You can reinstate access to a revoked PDF document, resulting in all copies of the revoked document being accessible to users. When a user opens a reinstated document that was revoked, the user is able to view the document.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To reinstate access to a revoked PDF document, perform the following steps:

1 Include project files.

2 Create a Document Security Client API object.

3 Retrieve the license identifier of the revoked PDF document.

4 Reinstate access to the revoked PDF document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, you must create a Document Security service client object. If you are using the Java API, create a `DocumentSecurityClient` object. If you are using the Document Security web service API, create a `DocumentSecurityServiceService` object.

Retrieve the license identifier of the revoked PDF document

You must retrieve the license identifier of the revoked PDF document in order to reinstate a revoked PDF document. After you obtain the license identifier value, you can reinstate a revoked document. If you attempt to reinstate a document that is not revoked, you will cause an exception.

Reinstate access to the revoked PDF document

To reinstate access to a revoked PDF document, you must specify the license identifier of the revoked document. If you attempt to reinstate access to a PDF document that is not revoked, you will cause an exception.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Applying Policies to PDF Documents" on page 844

"Revoking Access to Documents" on page 851

Reinstate access to revoked documents using the Java API

Reinstate access to a revoked document by using the Document Security API (Java):

1 Include project files.

Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Document Security Client API object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `DocumentSecurityClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the license identifier of the revoked PDF document.

- Create a `java.io.FileInputStream` object that represents the revoked PDF document by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
- Create a `DocumentManager` object by invoking the `DocumentSecurityClient` object's `getDocumentManager` method.
- Retrieve the license identifier value of the revoked document by invoking the `DocumentManager` object's `getLicenseId` method and passing the `com.adobe.idp.Document` object that represents the revoked document. This method returns a string value that represents the license identifier.

4 Reinstate access to the revoked PDF document.

- Create a `LicenseManager` object by invoking the `DocumentSecurityClient` object's `getLicenseManager` method.
- Reinstate access to the revoked PDF document by invoking the `LicenseManager` object's `unrevokeLicense` method and passing the license identifier value of the revoked document.

Code examples

For code examples using the Document Security service, see the following Quick Starts in "*Java API(SOAP) Quick Start (Code Examples)*" on page 2:

- "Quick Start (SOAP mode): Reinstating access to a revoked document using the web service API"

Reinstate access to revoked documents using the web service API

Reinstate access to a revoked document using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `DocumentSecurityServiceClient` object by using its default constructor.
- Create a `DocumentSecurityServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DocumentSecurityServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.UserName`.

- Assign the corresponding password value to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.Password`.
- Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.

- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the license identifier of the revoked PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a revoked PDF document to which access is reinstated.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the revoked PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reinstate access to the revoked PDF document.

- Retrieve the license identifier value of the revoked document by invoking the `DocumentSecurityServiceClient` object's `getLicenseID` method and passing the `BLOB` object that represents the revoked document. This method returns a string value that represents the license identifier.
- Reinstate access to the revoked PDF document by invoking the `DocumentSecurityServiceClient` object's `unrevokeLicense` method and passing a string value that specifies the license identifier value of the revoked PDF document (pass the return value of the `DocumentSecurityServiceClient` object's `getLicenseId` method).

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Reinstating access to a revoked document using the web service API"
- "Quick Start (SwaRef): Reinstating access to a revoked document using the web service API"

See also

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Inspecting Policy Protected PDF Documents

You can use the Document Security Service API (Java and web service) to inspect policy-protected PDF documents. Inspecting policy-protected PDF documents returns information about the policy-protected PDF document. You can, for example, determine the policy that was used to secure the document and the date when the document was secured.

You cannot perform this task if your version of LiveCycle is 8.x or an earlier version. Support for inspecting policy-protected documents is added in AEM Forms. If you attempt to inspect a policy-protected document using LiveCycle 8.x (or earlier), an exception is thrown.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To inspect a policy-protected PDF document, perform the following steps:

1 Include project files.

2 Create a Document Security Client API object.

3 Retrieve a policy-protected document to inspect.

4 Obtain information about the policy-protected document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, create a Document Security service client object. If you are using the Java API, create a `RightsManagementClient` object. If you are using the Document Security web service API, create a `RightsManagementServiceService` object.

Retrieve a policy-protected document to inspect

To inspect a policy-protected document, retrieve it. If you attempt to inspect a document that is not secured with a policy or is revoked, an exception is thrown.

Inspect the document

After you retrieve a polciy-protected document, you can inspect it.

Obtain information about the policy-protected document

After you inspect a policy-protected PDF document, you can obtain information about it. For example, you can determine the policy that is used to secure the document.

If you secure a document with a policy that belongs to My Policies and then call `RMInspectResult.getPolicysetName` or `RMInspectResult.getPolicysetId`, null is returned.

If the document is secured using a policy that is contained in a policy set (other than My Policies) then `RMInspectResult.getPolicysetName` and `RMInspectResult.getPolicysetId` return valid strings.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Inspect Policy Protected PDF Documents using the Java API

Inspect a policy-protected PDF document by using the Document Security Service API (Java):

1 Include project files.

Include client JAR files, such as the adobe-rightsmanagement-client.jar, in your Java project's class path. For information about the location of these files, see "Including AEM Forms Java library files" on page 491.

2 Create a Document Security Client API object.

- Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)

- Create a `RightsManagementClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve a policy-protected document to inspect.

- Create a `java.io.FileInputStream` object that represents the policy-protected PDF document by using its constructor. Pass a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Inspect the document.

- Create a `DocumentManager` object by invoking the `RightsManagementClient` object's `getDocumentManager` method.
- Inspect the policy-protected document by invoking the `LicenseManager` object's `inspectDocument` method. Pass the `com.adobe.idp.Document` object that contains the policy-protected PDF document. This method returns a `RMInspectResult` object that contains information about the policy-protected document.

5 Obtain information about the policy-protected document.

To obtain information about the policy-protected document, invoke the appropriate method that belongs `RMInspectResult` object. For example, to retrieve the policy name, invoke the `RMInspectResult` object's `getPolicyName` method.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (SOAP mode): Inspecting policy protected PDF documents using the Java API"

Inspect Policy Protected PDF Documents using the web service API

Inspect a policy-protected PDF document by using the Document Security Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `RightsManagementServiceClient` object by using its default constructor.
- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.

- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a policy-protected document to inspect.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document to inspect.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, starting position, and stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Inspect the document.

Inspect the policy-protected document by invoking the `RightsManagementServiceClient` object's `inspectDocument` method. Pass the `BLOB` object that contains the policy-protected PDF document. This method returns a `RMInspectResult` object that contains information about the policy-protected document.

5 Obtain information about the policy-protected document.

To obtain information about the policy-protected document, get the value of the appropriate field that belongs to the `RMInspectResult` object. For example, to retrieve the policy name, get the value of the `RMInspectResult` object's `policyName` field.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Inspecting policy protected PDF documents using the web service API"
- "Quick Start (SwaRef): Inspecting policy protected PDF documents using the web service API"

See also

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Creating Watermarks

Watermarks help ensure the security of a document by uniquely identifying the document and controlling copyright infringement. For example, you can create and place a watermark that states Confidential on all pages of a document. After a watermark is created, you can include it as part of a policy. That is, you can set the policy's watermark attribute with the newly created watermark. After a policy that contains a watermark is applied to a document, the watermark appears in the policy-protected document.

Note: *Only users with Document Security administrative privileges can create watermarks. That is, you must specify such a user when defining connection settings required to create a Document Security service client object.*

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To create a watermark, perform the following steps:

1. Include project files.
2. Create a Document Security Client API object.
3. Set the watermarks attributes.
4. Register the watermark with the Document Security service.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, you must create a Document Security service client object. If you are using the Java API, create a `RightsManagementClient` object. If you are using the Document Security web service API, create a `RightsManagementServiceService` object.

Set the watermarks attributes

To create a new watermark, you must set watermark attributes. The name attribute must always be defined. In addition to the name attribute, you must set at least one of the following attributes:

- Custom Text
- DateIncluded
- UserIdIncluded
- UserNameIncluded

The following table lists key and value pairs that are required when creating a watermark using web services.

Key Name	Description	Value
`WaterBackCmd:IS_USERNAME_ENABLED`	Specifies if the user name of the user opening the document is part of the watermark.	True or False
`WaterBackCmd:IS_USERID_ENABLED`	Specifies if the identification of the user opening the document is part of the watermark.	True or False
`WaterBackCmd:IS_CURRENTDATE_ENABLED`	Specifies if the current date is part of the watermark.	True or False

Key Name	Description	Value
`WaterBackCmd:IS_CUSTOMTEXT_ENABLED`	If this value is true, then the value of the custom text must be specified using `WaterBackCmd:SRCTEXT`.	True or False
`WaterBackCmd:OPACITY`	Specifies the opacity of the watermark. The default value is 0.5 if it is not specified.	A value between 0.0 and 1.0.
`WaterBackCmd:ROTATION`	Specifies the rotation of the watermark. The default value is 0 degrees.	A value between 0 and 359.
`WaterBackCmd:SCALE`	If this value is specified, then `WaterBackCmd:IS_SIZE_ENABLED` must be present and the value must be true. If this attribute is not specified, the default behavior is fit to page.	A value greater than 0.0 and less than or equal to 1.0.
`WaterBackCmd:HORIZ_ALIGN`	Specifies the watermark's horizontal alignment. The default value is center.	left, center, or right
`WaterBackCmd:VERT_ALIGN`	Specifies the watermark's vertical alignment. The default value is center.	top, center, or bottom
`WaterBackCmd:IS_USE_BACKGROUND`	Specifies if the watermark is a background. The default value is false.	True or False
`WaterBackCmd:IS_SIZE_ENABLED`	True if a custom scale is specified. If this value is true, SCALE must also be specified. If this value is false, then the default is fit to page.	True or False
`WaterBackCmd:SRCTEXT`	Specifies the custom text for a watermark. If this value is present, then `WaterBackCmd:IS_CUSTOMTEXT_ENABLED` must also be present and set to true.	True or False

All watermarks must have one of the following attributes defined:

- `WaterBackCmd:IS_USERNAME_ENABLED`
- `WaterBackCmd:IS_USERID_ENABLED`
- `WaterBackCmd:IS_CURRENTDATE_ENABLED`
- `WaterBackCmd:IS_CUSTOMTEXT_ENABLED`

All other attributes are optional.

Register the watermark

A new watermark must be registered with the Document Security service before it can be used. After you register a watermark, you can use it within policies.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Applying Policies to PDF Documents" on page 844

Create watermarks using the Java API

Create a watermark by using the Document Security API (Java):

1 Include project files.

Include client JAR files, such as the adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Document Security Client API object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `RightsManagementClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set the watermark attributes

- Create a `Watermark` object by invoking the `InfomodelObjectFactory` object's static `createWatermark` method. This method returns a `Watermark` object.
- Set the watermark's name attribute by invoking the `Watermark` object's `setName` method and passing a string value that specifies the policy name.
- Set the watermark's background attribute by invoking the `Watermark` object's `setBackground` method and passing `true`. By setting this attribute, the watermark appears in the background of the document.
- Set the watermark's custom text attribute by invoking the `Watermark` object's `setCustomText` method and passing a string value that represents the watermark's text.
- Set the watermark's opacity attribute by invoking the `Watermark` object's `setOpacity` method and passing an integer value that specifies the opacity level. A value of 100 indicates the watermark is completely opaque and a value of 0 indicates the watermark is completely transparent.

4 Register the watermark.

- Create a `WatermarkManager` object by invoking the `RightsManagementClient` object's `getWatermarkManager` method. This method returns a `WatermarkManager` object.
- Register the watermark by invoking the `WatermarkManager` object's `registerWatermark` method and passing the `Watermark` object that represents the watermark to register. This method returns a string value that represents the watermark's identification value.

Code examples

For code examples using the Document Security service, see the following Quick Starts in "*Java API(SOAP) Quick Start (Code Examples)*" on page 2:

- "Quick Start (SOAP mode): Creating a watermark using the Java API"

Create watermarks using the web service API

Create a watermark by using the Document Security API (web service):

1 Create a Document Security Client API object.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `RightsManagementServiceClient` object by using its default constructor.

- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Set the watermark attributes.

- Create a `WatermarkSpec` object by invoking the `WatermarkSpec` constructor.
- Set the watermark's name by assigning a string value to the `WatermarkSpec` object's `name` data member.
- Set the watermark's `id` attribute by assigning a string value to the `WatermarkSpec` object's `id` data member.
- For each watermark property to set, create a separate `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Set the key value by assigning a value to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` data member (for example, `WaterBackCmd:OPACITY)`.
- Set the value by assigning a value to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` data member (for example, `.25`).
- Create a `MyArrayOf_xsd_anyType` object. For each `MyMapOf_xsd_string_To_xsd_anyType_Item` object, invoke the `MyArrayOf_xsd_anyType` object's `Add` method. Pass the `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Assign the `MyArrayOf_xsd_anyType` object to the `WatermarkSpec` object's `values` data member.

4 Register the watermark.

Register the watermark by invoking the `RightsManagementServiceClient` object's `registerWatermark` method and passing the `WatermarkSpec` object that represents the watermark to register.

Code examples

For code examples using the Document Security service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Creating a watermark using the web service API"
- "Quick Start (SwaRef): Creating a watermark using the web service API"

See also

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Modifying Watermarks

You can modify an existing watermark using the Document Security Java API or web service API. To make changes to an existing watermark, you retrieve it, modify its attributes, and then update it on the server. For example, assume that you retrieve an watermark and modify its opacity attribute. Before the change takes effect, you must update the watermark.

When you modify a watermark, the change impacts future documents that have the watermark applied to them. That is, existing PDF documents that contain the watermark are not affected.

Note: *Only users with Document Security administrative privileges can modify watermarks. That is, you must specify such a user when defining connection settings required to create a Document Security service client object.*

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To modify a watermark, perform the following steps:

1. Include project files.
2. Create a Document Security Client API object.
3. Retrieve the watermark to modify.
4. Set the watermarks attributes.
5. Update the watermark.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, you must create a Document Security service client object. If you are using the Java API, create a `DocumentSecurityClient` object. If you are using the Document Security web service API, create a `DocumentSecurityServiceService` object.

Retrieve the watermark to modify

To modify a watermark, you must retrieve an existing watermark. You can retrieve a watermark by specifying its name or by specifying its identifier value.

Set the watermarks attributes

To modify an existing watermark, change the value of one or more watermark attributes. When programmatically updating a watermark using a web service, you must set all of the attributes that were originally set, even if the value does not change. For example, assume the following watermark attributes are set: `WaterBackCmd:IS_USERID_ENABLED`, `WaterBackCmd:IS_CUSTOMTEXT_ENABLED`, `WaterBackCmd:OPACITY`, and `WaterBackCmd:SRCTEXT`. Although the only attribute that you want to modify is `WaterBackCmd:OPACITY`, you must set the other values are well.

Note: *When using the Java API to modify a watermark, you do not need to specify all attributes. Set the watermark attribute that you want to modify.*

Note: *For information about the watermark attribute names, see "Creating Watermarks" on page 861.*

Update the watermark

After you modify a watermark's attributes, you must update the watermark.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Creating Watermarks" on page 861

Modify watermarks using the Java API

Modify a watermark by using the Document Security API (Java):

1. Include project files.

   Include client JAR files, such as the adobe-rightsmanagement-client.jar, in your Java project's class path.

2. Create a Document Security Client API object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `DocumentSecurityClient` object by using its constructor and passing the `ServiceClientFactory` object.

3. Retrieve the watermark to modify.

   Create a `WatermarkManager` object by invoking the `DocumentSecurityClient` object's `getWatermarkManager` method and pass a string value that specifies the watermark name. This method returns a `Watermark` object that represents the watermark to modify.

4. Set the watermark attributes.

   Set the watermark's opacity attribute by invoking the `Watermark` object's `setOpacity` method and passing an integer value that specifies the opacity level. A value of 100 indicates the watermark is completely opaque and a value of 0 indicates the watermark is completely transparent.

   Note: *This example modifies only the opacity attribute.*

5. Update the watermark.

   - Update the watermark by invoking the `WatermarkManager` object's `updateWatermark` method and pass the `Watermark` object whose attribute was modified.

Code examples

For code examples using the Document Security service, see the Quick Start(SOAP mode): Modifying a watermark using the Java API in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2

Modify watermarks using the web service API

Modify a watermark by using the Document Security API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.

- Create a `DocumentSecurityServiceClient` object by using its default constructor.
- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/DocumentSecurityService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DocumentSecurityServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve the watermark to modify.

Retrieve the watermark to modify by invoking the `DocumentSecurityServiceClient` object's `getWatermarkByName` method. Pass a string value that specifies the watermark name. This method returns a `WatermarkSpec` object that represents the watermark to modify.

4 Set the watermark attributes.

- For each watermark property to update, create a separate `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Set the key value by assigning a value to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` data member (for example, `WaterBackCmd:OPACITY)`.
- Set the value by assigning a value to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` data member (for example, `.50`).
- Create a `MyArrayOf_xsd_anyType` object. For each `MyMapOf_xsd_string_To_xsd_anyType_Item` object, invoke the `MyArrayOf_xsd_anyType` object's `Add` method. Pass the `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Assign the `MyArrayOf_xsd_anyType` object to the `WatermarkSpec` object's `values` data member.

5 Update the watermark.

Update the watermark by invoking the `DocumentSecurityServiceClient` object's `updateWatermark` method and passing the `WatermarkSpec` object that represents the watermark to modify.

Code examples

For code examples using the Document Security service, see the following Quick Start in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Modifying a watermark using the web service API"

Searching for Events

The Rights Management service tracks specific actions as they occur, such as applying a policy to a document, opening a policy-protected document, and revoking access to documents. Event auditing must be enabled for the Rights Management service or events are not tracked.

Events fall into one of the following categories:

- Administrator events are actions related to an administrator, such as creating a new administrator account.
- Document events are actions related to a document, such as closing a policy-protected document.
- Policy events are actions related to a policy, such as creating a new policy.
- Service events are actions related to the Rights Management service, such as synchronizing with the user directory.

You can search for specify specific events by using the Rights Management Java API or web service API. By searching for events, you can perform tasks, such as creating a log file of certain events.

Note: *For more information about the Rights Management service, see Services Reference for AEM Forms.*

Summary of steps

To search for a Rights Management event, perform the following steps:

1 Include project files.

2 Create a Rights Management Client API object.

3 Specify the event for which to search.

4 Search for the event.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Rights Management Client API object

Before you can programmatically perform a Rights Management service operation, you must create a Rights Management service client object. If you are using the Java API, create a `DocumentSecurityClient` object. If you are using the Rights Management web service API, create a `DocumentSecurityServiceService` object.

Specify the events to search for

You must specify the event to search for. For example, you can search for the policy create event, which occurs when a new policy is created.

Search for the event

After you specify the event to search for, you can use either the Rights Management Java API or the Rights Management web service API to search for the event.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Search for events using the Java API

Search for events by using the Rights Management API (Java):

1 Include project files

Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Rights Management Client API object

Create a `DocumentSecurityClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the events to search for

- Create an `EventManager` object by invoking the `DocumentSecurityClient` object's `getEventManager` method. This method returns an `EventManager` object.
- Create an `EventSearchFilter` object by invoking its constructor.
- Specify the event for which to search by invoking the `EventSearchFilter` object's `setEventCode` method and passing a static data member that belongs to the `EventManager` class that represents the event for which to search. For example, to search for the policy create event, pass `EventManager.POLICY_CREATE_EVENT`.

Note: *You can define additional search criteria by invoking `EventSearchFilter` object methods. For example, invoke the `setUserName` method to specify a user associated with the event.*

4 Search for the event

Search for the event by invoking the `EventManager` object's `searchForEvents` method and passing the `EventSearchFilter` object that defines the event search criteria. This method returns an array of `Event` objects.

Code examples

For code examples using the Rights Management service, see the following Quick Starts in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (SOAP): Searching for events using the Java API"

Search for events using the web service API

Search for events by using the Rights Management API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Rights Management Client API object

- Create a `DocumentSecurityServiceClient` object by using its default constructor.

- Create a `DocumentSecurityServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DocumentSecurityServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Specify the events to search for

- Create an `EventSpec` object by using its constructor.
- Specify the start of the time period during which the event occurred by setting the `EventSpec` object's `firstTime.date` data member with `DataTime` instance that represents the start of the date range when the event occurred.
- Assign the value `true` to the `EventSpec` object's `firstTime.dateSpecified` data member.
- Specify the end of the time period during which the event occurred by setting the `EventSpec` object's `lastTime.date` data member with `DataTime` instance that represents the end of the date range when the event occurred.
- Assign the value `true` to the `EventSpec` object's `lastTime.dateSpecified` data member.
- Set the event to search for by assigning a string value to the `EventSpec` object's `eventCode` data member. The following table lists the numeric values that you can assign to this property:

Event type	Value
`ALL_EVENTS`	999
`USER_CHANGE_PASSWORD_EVENT`	1000
`USER_REGISTER_EVENT`	1001
`USER_PREREGISTER_EVENT`	1002
`USER_ACTIVATE_EVENT`	1003
`USER_DEACTIVATE_EVENT`	1004
`USER_AUTHENTICATE_EVENT`	1005
`USER_AUTHENTICATE_DENY_EVENT`	1006
`USER_ACCOUNT_LOCK_EVENT`	1007

Event type	Value
USER_DELETE_EVENT	1008
USER_UPDATE_PROFILE_EVENT	1009
DOCUMENT_VIEW_EVENT	2000
DOCUMENT_PRINT_LOW_EVENT	2001
DOCUMENT_PRINT_HIGH_EVENT	2002
DOCUMENT_SIGN_EVENT	2003
DOCUMENT_ADD_ANNOTATION_EVENT	2004
DOCUMENT_FORM_FILL_EVENT	2005
DOCUMENT_CLOSE_EVENT	2006
DOCUMENT_MODIFY_EVENT	2007
DOCUMENT_CHANGE_SECURITY_HANDLER_EVENT	2008
DOCUMENT_SWITCH_POLICY_EVENT	2009
DOCUMENT_REVOKE_EVENT	2010
DOCUMENT_UNREVOKE_EVENT	2011
DOCUMENT_SECURE_EVENT	2012
DOCUMENT_UNKNOWN_CLIENT_EVENT	2013
DOCUMENT_CHANGE_REVOKE_URL_EVENT	2014
POLICY_CHANGE_EVENT	3000
POLICY_ENABLE_EVENT	3001
POLICY_DISABLE_EVENT	3002
POLICY_CREATE_EVENT	3003
POLICY_DELETE_EVENT	3004
POLICY_CHANGE_OWNER_EVENT	3005
SERVER_CLIENT_SYNC_EVENT	4000
SERVER_SYNC_DIR_INFO_EVENT	4001
SERVER_SYNC_DIR_COMPLETE_EVENT	4002
SERVER_VERSION_MISMATCH_EVENT	4003
SERVER_CONFIG_CHANGE_EVENT	4004
SERVER_ENABLE_OFFLINE_ACCESS_EVENT	4005
ADMIN_ADD_EVENT	5000
ADMIN_DELETE_EVENT	5001
ADMIN_EDIT_EVENT	5002
ADMIN_ACTIVATE_EVENT	5003
ADMIN_DEACTIVATE_EVENT	5004

Event type	Value
ERROR_DIRECTORY_SERVICE_EVENT	6000
CREATED_POLICYSET_EVENT	7000
DELETED_POLICYSET_EVENT	7001
MODIFIED_POLICYSET_EVENT	7002

4 Search for the event

Search for the event by invoking the `DocumentSecurityServiceClient` object's `searchForEvents` method and passing the `EventSpec` object that represents the event for which to search and the maximum number of results. This method returns a `MyArrayOf_xsd_anyType` collection where each element is an `AuditSpec` instance. Using an `AuditSpec` instance, you can obtain information about the event such as the time that it occurred. The `AuditSpec` instance contains a `timestamp` data member that specifies this information.

Code examples

For code examples using the Rights Management service, see the following Quick Starts in "*Java API(SOAP) Quick Start (Code Examples)*" on page 2:

- "Quick Start (MTOM): Searching for events using the web service API"
- "Quick Start (SwaRef): Searching for events using the web service API"

See also

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Applying Policies to Word Documents

In addition to PDF documents, the Rights Mangement service supports additional document formats such as a Microsoft Word document (DOC file) and other Micosoft office file formats. For example, you can apply a policy to a Word document in order to secure it. By applying a policy to a Word document, you restrict access to the document. You cannot apply a policy to a document if the document is already secured with a policy.

You can monitor the use of a policy-protected Word document after you distribute it. That is, you can see how the document is being used and who is using it. For example, you can find out when somebody has opened the document.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To apply a policy to a Word document, perform the following steps:

1 Include project files.

2 Create a Document Security Client API object.

3 Retrieve a Word document to which a policy is applied.

4 Apply an existing policy to the Word document.

5 Save the policy-protected Word document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client APIobject

Before you can programmatically perform a Document Security service operation, you must create a Document Security service client object.

Retrieve a Word document

You must retrieve a Word document in order to apply a policy. After you apply a policy to the Word document, users are restricted when using the document. For example, if the policy does not enable the document to be opened while offline, then users must be online to open the document.

Apply an existing policy to the Word document

To apply a policy to a Word document, you must reference an existing policy and specify which policy set the policy belongs to. The user who is setting the connection properties must have access to the specified policy. If not, an exception occurs.

Save the Word document

After the Document Security service applies a policy to a Word document, you can save the policy-protected Word document as a DOC file.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Revoking Access to Documents" on page 851

Apply a policy to a Word document using the Java API

Apply a policy to a Word document by using the Document Security API (Java):

1 Include project files.

   Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2 Create a Document Security Client API object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `DocumentSecurityClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve a Word document.

   - Create a `java.io.FileInputStream` object that represents the Word document by using its constructor and passing a string value that specifies the location of the Word document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Apply an existing policy to the Word document.

   - Create a `DocumentManager` object by invoking the `DocumentSecurityClient` object's `getDocumentManager` method.

- Apply a policy to the Word document by invoking the `DocumentManager` object's `protectDocument` method and passing the following values:
  - The `com.adobe.idp.Document` object that contains the Word document to which the policy is applied.
  - A string value that specifies the name of the document.
  - A string value that specifies the name of the policy set to which the policy belongs. You can specify a `null` value that results in the `MyPolicies` policy set being used.
  - A string value that specifies the policy name.
  - A string value that represents the name of the user manager domain of the user who is the publisher of the document. This parameter value is optional and can be null (if this parameter is null, then the next parameter value must be null).
  - A string value that represents the name of the canonical name of the user manager user who is the publisher of the document. This parameter value is optional and can be `null` (if this parameter is `null`, then the previous parameter value must be `null`).
  - A `com.adobe.livecycle.rightsmanagement.Locale` that represents the locale that is used for selecting the MS Office template. This parameter value is optional and you can specify `null`.

  The `protectDocument` method returns a `RMSecureDocumentResult` object that contains the policy-protected Word document.

5 Save the Word document.
  - Invoke the `RMSecureDocumentResult` object's `getProtectedDoc` method to get the policy-protected Word document. This method returns a `com.adobe.idp.Document` object.
  - Create a `java.io.File` object and ensure that the file extension is DOC.
  - Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `getProtectedDoc` method).

Code examples

For code examples using the Document Security service, see the following Quick Start in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (SOAP mode): Applying a policy to a Word document using the Java API"

Apply a policy to a Word document using the web service API

Apply a policy to a Word document by using the Document Security API (web service):

1 Include project files.

  Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/DocumentSecurityService?WSDL&lc_version=9.0.1`.

  Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object.
  - Create a `DocumentSecurityServiceClient` object by using its default constructor.
  - Create a `DocumentSecurityServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/DocumentSecurityService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DocumentSecurityServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DocumentSecurityServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a Word document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a Word document to which a policy is applied.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the Word document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. Determine the byte array size by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Apply an existing policy to the Word document.

Apply a policy to the Word document by invoking the `DocumentSecurityServiceClient` object's `protectDocument` method and passing the following values:

- The `BLOB` object that contains the Word document to which the policy is applied.
- A string value that specifies the name of the document.
- A string value that specifies the name of the policy set to which the policy belongs. You can specify a `null` value that results in the `MyPolicies` policy set being used.
- A string value that specifies the policy name.
- A string value that represents the name of the user manager domain of the user who is the publisher of the document. This parameter value is optional and can be null (if this parameter is null, then the next parameter value must be `null`).
- A string value that represents the name of the canonical name of the user manager user who is the publisher of the document. This parameter value is optional and can be null (if this parameter is null, then the previous parameter value must be `null`).
- A `RMLocale` value that specifies the locale value (for example, `RMLocale.en`).
- A string output parameter that is used to store the policy identifier value.
- A string output parameter that is used to store the policy-protected identifier value.
- A string output parameter that is used to store the mime type (for example, `application/doc`).

The `protectDocument` method returns a `BLOB` object that contains the policy-protected Word document.

5 Save the Word document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the policy-protected Word document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `protectDocument` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a Word file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

Code examples

For code examples using the Document Security service, see the following Quick Start in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (MTOM): Applying a policy to a Word document using the web service API "

Removing Policies from Word Documents

You can remove a policy from a policy-protected Word document in order to remove security from the document. That is, if you no longer want the document to be protected by a policy. If you want to update a policy-protected Word document with a newer policy, then instead of removing the policy and adding the updated policy, it is more efficient to switch the policy.

Note: *For more information about the Document Security service, see Services Reference for AEM Forms.*

Summary of steps

To remove a policy from a policy-protected Word document, perform the following steps:

1 Include project files

2 Create a Document Security Client API object.

3 Retrieve a policy-protected Word document.

4 Remove the policy from the Word document.

5 Save the unsecured Word document.s

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a Document Security Client API object

Before you can programmatically perform a Document Security service operation, create a Document Security service client object.

Retrieve a policy-protected Word document

You must retrieve a policy-protected Word document in order to remove a policy. If you attempt to remove a policy from a Word document that is not protected by a policy, you will cause an exception.

Remove the policy from the Word document

You can remove a policy from a policy-protected Word document provided that an administrator is specified in the connection settings. If not, then the policy used to secure a document must contain the `SWITCH_POLICY` permission in order to remove a policy from a Word document. Also, the user specified in the AEM Forms connection settings must also have that permission. Otherwise, an exception is thrown.

Save the unsecured Word document

After the Document Security service removes a policy from a Word document, you can save the unsecured Word document as a DOC file.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Applying Policies to Word Documents" on page 872

Remove a policy from a Word document using the Java API

Remove a policy from a policy-protected Word document by using the Document Security API (Java):

1. Include project files

   Include client JAR files, such as adobe-rightsmanagement-client.jar, in your Java project's class path.

2. Create a Document Security Client API object
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `RightsManagementClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Retrieve a policy-protected Word document
   - Create a `java.io.FileInputStream` object that represents the policy-protected Word document by using its constructor and passing a string value that specifies the location of the Word document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Remove the policy from the Word document
   - Create a `DocumentManager` object by invoking the `RightsManagementClient` object's `getDocumentManager` method.
   - Remove a policy from the Word document by invoking the `DocumentManager` object's `removeSecurity` method and passing the `com.adobe.idp.Document` object that contains the policy-protected Word document. This method returns a `com.adobe.idp.Document` object that contains an unsecured Word document.
5. Save the unsecured Word document
   - Create a `java.io.File` object and ensure that the file extension is DOC.
   - Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file (ensure that you use the `Document` object that was returned by the `removeSecurity` method).

Code examples

For code examples using the Document Security service, see the following Quick Start in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (SOAP mode): Removing a policy from a Word document using the Java API "

Remove a policy from a Word document using the web service API

Remove a policy from a policy-protected Word document by using the Document Security API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/RightsManagementService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Document Security Client API object

- Create a `RightsManagementServiceClient` object by using its default constructor.
- Create a `RightsManagementServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/RightsManagementService?WSDL`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `RightsManagementServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `RightsManagementServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `RightsManagementServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Retrieve a policy-protected Word document

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the policy-protected Word document from which the policy is removed.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the Word document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Remove the policy from the Word document

Remove the policy from the Word document by invoking the `RightsManagementServiceClient` object's `removePolicySecurity` method and passing the `BLOB` object that contains the policy-protected Word document. This method returns a `BLOB` object that contains an unsecured Word document.

5 Save the unsecured Word document

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the unsecured Word document.
- Create a byte array that stores the data content of the `BLOB` object that was returned by the `removePolicySecurity` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

Code examples

For code examples using the Document Security service, see the following Quick Start in *"Java API(SOAP) Quick Start (Code Examples)"* on page 2:

- "Quick Start (MTOM): Removing a policy from a Word document using the web service API"

See also

"Invoking AEM Forms using MTOM" on page 529

Digitally Signing and Certifying Documents

About the Signature Service

The Signature service lets your organization protect the security and privacy of Adobe PDF documents that it distributes and receives. This service uses digital signatures and certification to ensure that only intended recipients can alter documents. Because security features are applied to the document itself, the document remains secure and controlled for its entire life cycle. A document remains secure beyond the firewall, when it is downloaded offline, and when it is submitted back to your organization.

Note: *You can create a custom signature handler for the Signature service that is invoked when certain operations are invoked, such as signing a PDF document. (See Creating Signature Handlers.)*

Signature field names

Some Signature service operations require that you specify the name of the signature field on which an operation is performed. For example, when signing a PDF document, you specify the name of the signature field to sign. Assume that the full name of a signature field is `form1[0].Form1[0].SignatureField1[0]`. You can specify `SignatureField1[0]` instead of `form1[0].Form1[0].SignatureField1[0]`.

Sometimes a conflict causes the Signature service to sign (or perform another operation that requires the signature field name) the wrong field. This conflict is the result of the name `SignatureField1[0]` appearing in two or more places in the same PDF document. For example, consider a PDF document that contains two signature fields named `form1[0].Form1[0].SignatureField1[0]` and `form1[0].Form1[0].SubForm1[0].SignatureField1[0]` and you specify `SignatureField1[0]`. In this situation, the Signature service signs the first signature field that it finds while iterating over all the signature fields in the document.

If there are multiple signature fields located within a PDF document, it is recommended that you specify the full names of the signature fields. That is, specify `form1[0].Form1[0].SignatureField1[0]` instead of `SignatureField1[0]`.

You can accomplish these tasks using the Signature service:

- Add and delete digital signature fields to a PDF document. (See "Adding Signature Fields" on page 880.)
- Retrieve the names of signature fields located in a PDF document. (See "Retrieving Signature Field Names" on page 884.)
- Modify signature fields. (See "Modifying Signature Fields" on page 887.)
- Digitally sign PDF documents. (See "Digitally Signing PDF Documents" on page 892.)
- Certify PDF documents. (See "Certifying PDF Documents" on page 904.)
- Validate digital signatures located in a PDF document. (See "Verifying Digital Signatures" on page 911.)
- Validate all digital signatures located in a PDF document. (See "Verifying Multiple Digital Signatures" on page 916.)
- Remove a digital signature from a signature field. (See "Removing Digital Signatures" on page 921.)

  Note: *For more information about the Signature service, see Services Reference for AEM Forms.*

Adding Signature Fields

Digital signatures appear in signature fields, which are form fields that contain a graphic representation of the signature. Signature fields can be visible or invisible. Signers can use a preexisting signature field, or a signature field can be programmatically added. In either case, the signature field must exist before a PDF document can be signed.

You can programmatically add a signature field by using the Signature service Java API or Signature web service API. You can add more than one signature field to a PDF document; however, each signature field name must be unique.

Note: *Some PDF document types do not let you programmatically add a signature field. For more information about the Signature service and adding signature fields, see Services Reference for AEM Forms.*

Summary of steps

To add a signature field to a PDF document, perform the following tasks:

1. Include project files.
2. Create a Signature client.
3. Get a PDF document to which a signature field is added.
4. Add a signature field.
5. Save the PDF document as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create a Signature client

Before you can programmatically perform a Signature service operation, you must create a Signature service client.

Get a PDF document to which a signature field is added

You must obtain a PDF document to which a signature field is added.

Add a signature field

To successfully add a signature field to a PDF document, you specify coordinate values that identify the location of the signature field. (If you add an invisible signature field, these values are not required.) Also, you can specify which fields in the PDF document are locked after a signature is applied to the signature field.

Save the PDF document as a PDF file

After the Signature service adds a signature field to the PDF document, you can save the document as a PDF file so that users can open it in Acrobat or Adobe Reader.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Digitally Signing PDF Documents" on page 892

Add signature fields using the Java API

Add a signature field by using the Signature API (Java):

1. Include project files

   Include client JAR files, such as adobe-signatures-client.jar, in your Java project's classpath.

2. Create a Signature client
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Get a PDF document to which a signature field is added
   - Create a `java.io.FileInputStream` object that represents the PDF document to which a signature field is added by using its constructor and passing a string value that specifies the location of the PDF document.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Add a signature field
   - Create a `PositionRectangle` object that specifies the signature field location by using its constructor. Within the constructor, specify coordinate values.
   - If desired, create a `FieldMDPOptions` object that specifies the fields that are locked when a digital signature is applied to the signature field.
   - Add a signature field to a PDF document by invoking the `SignatureServiceClient` object's `addSignatureField` method and passing the following values:
     - A `com.adobe.idp.Document` object that represents the PDF document to which a signature field is added.
     - A string value that specifies the name of the signature field.

- A `java.lang.Integer` value that represents the page number to which a signature field is added.
- A `PositionRectangle` object that specifies the location of the signature field.
- A `FieldMDPOptions` object that specifies fields in the PDF document that are locked after a digital signature is applied to the signature field. This parameter value is optional, and you can pass `null`.

- A `PDFSeedValueOptions` object that specifies various run-time values. This parameter value is optional, and you can pass `null`.

  The `addSignatureField` method returns a `com.adobe.idp.Document` object that represents a PDF document that contains a signature field.

Note: *You can invoke the `SignatureServiceClient` object's `addInvisibleSignatureField` method to add an invisible signature field.*

5 Save the PDF document as a PDF file

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that was returned by the `addSignatureField` method.

See also

"Signature Service Java API Quick Start(SOAP)" on page 372

Add signature fields using the web service API

To add a signature field by using the Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.

- Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get a PDF document to which a signature field is added

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the PDF document that will contain a signature field.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Add a signature field

Add a signature field to the PDF document by invoking the `SignatureServiceClient` object's `addSignatureField` method and passing the following values:

- A `BLOB` object that represents the PDF document to which a signature field is added.
- A string value that specifies the signature field name.
- An integer value that represents the page number to which a signature field is added.
- A `PositionRect` object that specifies the location of the signature field.
- A `FieldMDPOptions` object that specifies fields in the PDF document that are locked after a digital signature is applied to the signature field. This parameter value is optional, and you can pass `null`.
- A `PDFSeedValueOptions` object that specifies various run-time values. This parameter value is optional, and you can pass `null`.

The `addSignatureField` method returns a `BLOB` object that represents a PDF document that contains a signature field.

5 Save the PDF document as a PDF file

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document that will contain the signature field and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `addSignatureField` method. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

Quick Start (MTOM):Adding a signature field using the web service API

Quick Start (SwaRef):Adding a signature field using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Retrieving Signature Field Names

You can retrieve the names of all signature fields that are located in a PDF document that you want to sign or certify. If you are unsure of the signature field names that are located in a PDF document or you want to verify the names, you can programmatically retrieve them. The Signature service returns the fully qualified name of the signature field, such as `form1[0].grantApplication[0].page1[0].SignatureField1[0]`.

Note: *For more information about the Signature service, see Services Reference for AEM Forms*

Summary of steps

To retrieve signature field names, perform the following tasks:

1. Include project files.
2. Create a Signature client.
3. Get the PDF document that contains signature fields.
4. Retrieve the signature field names.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Signature client

Before you can programmatically perform a Signature service operation, you must create a Signature service client.

Get the PDF document that contains signature fields

Retrieve a PDF document that contains signature fields.

Retrieve the signature field names

You can retrieve signature field names after you retrieve a PDF document that contains one or more signature fields.

See also

"Retrieve signature field names using the Java API" on page 885

"Retrieve signature field using the web service API" on page 885

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Adding Signature Fields" on page 880

Retrieve signature field names using the Java API

Retrieve signature field names by using the Signature API (Java):

1 Include project files

Include client JAR files, such as the adobe-signatures-client.jar, in your Java project's classpath.

2 Create a Signature client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document that contains signature fields

- Create a `java.io.FileInputStream` object that represents the PDF document that contains signature fields by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Retrieve the signature field names

- Retrieve the signature field names by invoking the `SignatureServiceClient` object's `getSignatureFieldList` method and passing the `com.adobe.idp.Document` object that contains the PDF document that contains signature fields. This method returns a `java.util.List` object, in which each element contains a `PDFSignatureField` object. Using this object, you can obtain additional information about a signature field, such as whether it is visible.
- Iterate through the `java.util.List` object to determine if there are signature field names. For each signature field in the PDF document, you can obtain a separate `PDFSignatureField` object. To obtain the name of the signature field, invoke the `PDFSignatureField` object's `getName` method. This method returns a string value that specifies the signature field name.

See also

"Retrieving Signature Field Names" on page 884

"Quick Start (SOAP mode): Retrieving signature field names using the Java API" on page 375

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve signature field using the web service API

Retrieve signature field names using the Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.

- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document that contains signature fields

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the PDF document that contains signature fields.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field the byte array contents.

4 Retrieve the signature field names

- Retrieve the signature field names by invoking `SignatureServiceClient` object's `getSignatureFieldList` method and passing the `BLOB` object that contains the PDF document that contains signature fields. This method returns a `MyArrayOfPDFSignatureField` collection object where each element contains a `PDFSignatureField` object.
- Iterate through the `MyArrayOfPDFSignatureField` object to determine whether there are signature field names. For each signature field in the PDF document, you can obtain a `PDFSignatureField` object. To obtain the name of the signature field, invoke the `PDFSignatureField` object's `getName` method. This method returns a string value that specifies the signature field name.

See also

"Retrieving Signature Field Names" on page 884

Quick Start (MTOM): Retrieving signature field names using the web service API

Quick Start (SwaRef): Retrieving signature field names using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Modifying Signature Fields

You can modify signature fields that are located in a PDF document by using the Java API and web service API. Modifying a signature field involves manipulating its signature field lock dictionary values or seed value dictionary values.

A *field lock dictionary* specifies a list of fields that are locked when the signature field is signed. A locked field prevents users from making changes to the field. A *seed value dictionary* contains constraining information that is used at the time the signature is applied. For example, you can change permissions that control the actions that can occur without invalidating a signature.

By modifying an existing signature field, you can make changes to the PDF document to reflect changing business requirements. For example, a new business requirement may require locking all document fields after the document is signed.

This section explains how to modify a signature field by amending both field lock dictionary and seed value dictionary values. Changes made to the signature field lock dictionary result in all fields in the PDF document being locked when a signature field is signed. Changes made to the seed value dictionary prohibit specific types of changes to the document.

Note: *For more information about the Signature service and modifying signature fields, see Services Reference for AEM Forms.*

Summary of steps

To modify signature fields located in a PDF document, perform the following tasks:

1. Include project files.
2. Create a Signature client.
3. Get the PDF document that contains the signature field to modify.
4. Set dictionary values.
5. Modify the signature field.
6. Save the PDF document as a PDF file.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Signature client

Before you can programmatically perform a Signature service operation, you must create a Signature service client.

Get the PDF document that contains the signature field to modify

Retrieve a PDF document that contains the signature field to modify.

Set dictionary values

To modify a signature field, assign values to its field lock dictionary or seed value dictionary. Specifying signature field lock dictionary values involves specifying PDF document fields that are locked when the signature field is signed. (This section discusses how to lock all fields.)

The following seed value dictionary values can be set:

- **Revision checking**: Specifies whether revocation checking is performed when a signature is applied to the signature field.
- **Certificate options**: Assigns values to the certificate seed value dictionary. Before specifying certificate options, it is recommended that you become familiar with a certificate seed value dictionary. (See PDF Reference.)
- **Digest options**: Assigns digest algorithms that are used for signing. Valid values are SHA1, SHA256, SHA384, SHA512, and RIPEMD160.
- **Filter**: Specifies the filter that is used with the signature field. For example, you can use the Adobe.PPKLite filter. (See PDF Reference.)
- **Flag options**: Specifies the flag values that are associated with this signature field. A value of 1 means that a signer must use only the specified values for the entry. A value of 0 means that other values are permitted. Here are the Bit positions:
  - **1(Filter):** The signature handler to be used to sign the signature field
  - **2 (SubFilter):** An array of names that indicate acceptable encodings to use when signing
  - **3 (V)**: The minimum required version number of the signature handler to be used to sign the signature field
  - **4 (Reasons):** An array of strings that specify possible reasons for signing a document
  - **5 (PDFLegalWarnings):** An array of strings that specify possible legal attestations
- **Legal attestations**: When a document is certified, it is automatically scanned for specific types of content that can make the visible contents of a document ambiguous or misleading. For example, an annotation can obscure text that is important for understanding what is being certified. The scanning process generates warnings that indicate the presence of this type of content. It also provides an additional explanation of the content that may have generated warnings.
- **Permissions**: Specifies permissions that can be used on a PDF document without invalidating the signature.
- **Reasons**: Specifies reasons why this document must be signed.
- **Time stamp**: Specifies time-stamping options. You can, for example, set the URL of the time-stamping server that is used.
- **Version**: Specifies the minimum version number of the signature handler to be used to sign the signature field.

Modify the signature field

After you create a Signature service client, retrieve the PDF document that contains the signature field to modify, and set dictionary values, you can instruct the Signature service to modify the signature field. The Signature service then returns a PDF document that contains the modified signature field. The original PDF document is not affected.

Save the PDF document as a PDF file

Save the PDF document that contains the modified signature field as a PDF file so that users can open it in Acrobat or Adobe Reader.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Signature Service Java API Quick Start(SOAP)" on page 372

"Digitally Signing PDF Documents" on page 892

Modify signature fields using the Java API

Modify a signature field by using the Signature API (Java):

1 Include project files

Include client JAR files, such as the adobe-signatures-client.jar, in your Java project's class path.

2 Create a Signature client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document that contains the signature field to modify

- Create a `java.io.FileInputStream` object that represents the PDF document that contains the signature field to modify by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set dictionary values

- Create a `PDFSignatureFieldProperties` object by using its constructor. A `PDFSignatureFieldProperties` object stores signature field lock dictionary and seed value dictionary information.
- Create a `PDFSeedValueOptionSpec` object by using its constructor. This object lets you set seed value dictionary values.
- Disallow changes to the PDF document by invoking the `PDFSeedValueOptionSpec` object's `setMdpValue` method and passing the `MDPPermissions.NoChanges` enumeration value.
- Create a `FieldMDPOptionSpec` object by using its constructor. This object lets you set signature field lock dictionary values.
- Lock all fields in the PDF document by invoking the `FieldMDPOptionSpec` object's `setMdpValue` method and passing the `FieldMDPAction.ALL` enumeration value.
- Set seed value dictionary information by invoking the `PDFSignatureFieldProperties` object's `setSeedValue` method and passing the `PDFSeedValueOptionSpec` object.
- Set signature field lock dictionary information by invoking the `PDFSignatureFieldProperties` object's `setFieldMDP` method and passing the `FieldMDPOptionSpec` object.

Note: *To see all seed value dictionary values that you can set, see the* `PDFSeedValueOptionSpec` *class reference. (See AEM Forms API Reference.)*

5 Modify the signature field

Modify the signature field by invoking the `SignatureServiceClient` object's `modifySignatureField` method and passing the following values:

- The `com.adobe.idp.Document` object that stores the PDF document that contains the signature field to modify

- A string value that specifies the name of the signature field
- The `PDFSignatureFieldProperties` object that stores signature field lock dictionary and seed value dictionary information

The `modifySignatureField` method returns a `com.adobe.idp.Document` object that stores a PDF document that contains the modified signature field.

6 Save the PDF document as a PDF file

- Create a `java.io.File` object and ensure that the file name extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that the `modifySignatureField` method returned.

Modify signature fields using the web service API

Modify a signature field by using the Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document that contains the signature field to modify

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the PDF document that contains the signature field to modify.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.

- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.

4 Set dictionary values

- Create a `PDFSignatureFieldProperties` object by using its constructor. This object stores signature field lock dictionary and seed value dictionary information.
- Create a `PDFSeedValueOptionSpec` object by using its constructor. This object lets you set seed value dictionary values.
- Disallow changes to the PDF document by assigning the `MDPPermissions.NoChanges` enumeration value to the `PDFSeedValueOptionSpec` object's `mdpValue` data member.
- Create a `FieldMDPOptionSpec` object by using its constructor. This object lets you set signature field lock dictionary values.
- Lock all fields in the PDF document by assigning the `FieldMDPAction.ALL` enumeration value to the `FieldMDPOptionSpec` object's `mdpValue` data member.
- Set seed value dictionary information by assigning the `PDFSeedValueOptionSpec` object to the `PDFSignatureFieldProperties` object's `seedValue` data member.
- Set signature field lock dictionary information by assigning the `FieldMDPOptionSpec` object to the `PDFSignatureFieldProperties` object's `fieldMDP` data member.

Note: *To see all seed value dictionary values that you can set, see the `PDFSeedValueOptionSpec` class reference. (See AEM Forms API Reference).*

5 Modify the signature field

Modify the signature field by invoking the `SignatureServiceClient` object's `modifySignatureField` method and passing the following values:

- The `BLOB` object that stores the PDF document that contains the signature field to modify
- A string value that specifies the name of the signature field
- The `PDFSignatureFieldProperties` object that stores signature field lock dictionary and seed value dictionary information

The `modifySignatureField` method returns a `BLOB` object that stores a PDF document that contains the modified signature field.

6 Save the PDF document as a PDF file

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document that will contain the signature field, and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that the `addSignatureField` method returns. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

Quick Start (MTOM): Modifying a signature field using the web service API

Quick Start (SwaRef): Modifying a signature field using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Digitally Signing PDF Documents

Digital signatures can be applied to PDF documents to provide a level of security. Digital signatures, like handwritten signatures, provide a means by which signers identify themselves and make statements about a document. The technology used to digitally sign documents helps to ensure that both the signer and recipients are clear about what was signed and confident that the document was not altered since it was signed.

PDF documents are signed by means of public-key technology. A signer has two keys: a public key and a private key. The private key is stored in a user's credential that must be available at the time of signing. The public key is stored in the user's certificate that must be available to recipients to validate the signature. Information about revoked certificates is found in certificate revocation lists (CRLs) and Online Certificate Status Protocol (OCSP) responses distributed by Certificate Authorities (CAs). The time of signing can be obtained from a trusted source known as a Timestamping Authority.

Note: *Before you can digitally sign a PDF document, you must ensure that you add the certificate to AEM Forms. A certificate is added using administration console or programmatically using the Trust Manager API. (See "Importing Credentials by using the Trust Manager API" on page 1060.)*

You can programmatically digitally sign PDF documents. When digitally signing a PDF document, you must reference a security credential that exists in AEM Forms. The credential is the private key used for signing.

The Signature service performs the following steps when a PDF document is signed:

1. The Signature service retrieves the credential from the Truststore by passing the alias specified in the request.
2. The Truststore searches for the specified credential.
3. The credential is returned to the Signature service and is used to sign the document. The credential is also cached against the alias for future requests.

For information about handling the security credential, see the *Installing and Deploying AEM Forms* guide for your application server.

Note: *There are differences between signing and certifying documents. (See "Certifying PDF Documents" on page 904.)*

Note: *Not all PDF documents support signing. For more information about the Signature service and digitally signing documents, see Services Reference for AEM Forms.*

Note: *The Signature service does not support XDP files with embedded PDF data as input to an operation, such as certifying a document. This action results in the Signature service throwing a* `PDFOperationException`*. To resolve this issue, convert the XDP file to a PDF file by using the PDF Utilities service and then pass the converted PDF file to a Signature service operation. (See "Working with PDF Utilities" on page 997.)*

nCipher nShield HSM credential

When using an nCipher nShield HSM credential to sign or certify a PDF document, the new credential cannot be used until the J2EE application server that AEM Forms is deployed on is restarted. However, you can set a configuration value, resulting in the sign or certify operation working without restarting the J2EE application server.

You can add the following configuration value in the cknfastrc file, which is located at /opt/nfast/cknfastrc (or c:\nfast\cknfastrc):

```
CKNFAST_ASSUME_SINGLE_PROCESS=0
```

After you add this configuration value to the cknfastrc file, the new credential can be used without restarting the J2EE application server.

Signature is not trusted

When certifying and signing the same PDF document, if the certifying signature is not trusted, a yellow triangle appears against the first signature when opening the PDF document in Acrobat or Adobe Reader. The certifying signature must be trusted in order to avoid this situation.

Signing documents that are XFA based forms

If you attempt to sign a XFA based form using the Signature service API, the data may be missing from the `ViewSignedVersion` located in Acrobat. For example, consider the following workflow:

- Using an XDP file created by using Designer, you merge a form design that contains a signature field and XML data that contains form data. You use the Forms service to generate an interactive PDF document.
- You sign the PDF document using the Signature service API.

Summary of steps

To digitally sign a PDF document, perform the following tasks:

1. Include project files.
2. Create a Signature service client.
3. Get the PDF document to sign.
4. Sign the PDF document.
5. Save the signed PDF document as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create a Signatures client

Before you can programmatically perform a Signature service operation, you must create a Signature service client.

Get the PDF document to sign

To sign a PDF document, you must obtain a PDF document that contains a signature field. If a PDF document does not contain a signature field, it cannot be signed. A signature field can be added by using Designer or programmatically.

Sign the PDF document

When signing a PDF document, you can set run-time options that are used by the Signature service. You can set the following options:

- Appearance options
- Revocation checking
- Time stamping values

You set appearance options by using a `PDFSignatureAppearanceOptionSpec` object. For example, you can display the date within a signature by invoking the `PDFSignatureAppearanceOptionSpec` object's `setShowDate` method and passing `true`.

You can also specify whether or not to perform a revocation check that determines whether the certificate that is used to digitally sign a PDF document has been revoked. To performing revocation checking, you can specify one of the following values:

- **NoCheck**: Do not perform revocation checking.
- **BestEffort**: Always attempt to check for revocation of all certificates in the chain. If any problem occurs in checking, the revocation is assumed to be valid. If any failure happens, assume that the certificate is not revoked.
- **CheckIfAvailable:** Check for revocation of all certificates in the chain if revocation information is available. If any problem occurs in checking, the revocation is assumed to be invalid. If any failure happens, assume the certificate is revoked and invalid. (This is the default value.)
- **AlwaysCheck**: Check for revocation of all certificates in the chain. If revocation information is not present in any certificate, revocation is assumed to be invalid.

To perform revocation checking on a certificate, you can specify a URL to a certificate revocation list (CRL) server by using a `CRLOptionSpec` object. However, if you want to perform revocation checking and you do not specify a URL to a CRL server, then the Signature service obtains the URL from the certificate.

Instead of using a CRL server, you can use an online certificate status protocol (OCSP) server when performing revocation checking. Typically when using an OCSP server as opposed to a CRL server, the revocation check is performed faster. (See "Online Certificate Status Protocol" at http://tools.ietf.org/html/rfc2560.)

You can set the CRL and OCSP server order that the Signature service uses using Adobe Applications and Services. For example, if the OCSP server is set first in Adobe Applications and Services, then the OCSP server is checked, followed by the CRL server. (See "Managing certificates and credentials using Trust Store" in AAC Help).

If you specify not to perform revocation checking, then the Signature service does not check to see if the certificate used to sign or certify a document has been revoked. That is, CRL and OCSP server information is ignored.

Note: *Although a CRL or an OCSP server may be specified in the certificate, you can override the URL specified in the certificate by using a* `CRLOptionSpec` *and an* `OCSPOptionSpec` *object. For example, to override the CRL server, you can invoke the* `CRLOptionSpec` *object's* `setLocalURI` *method.*

Time stamping refers to the process of tracking the time when a signed or certified document was modified. Once a document is signed, it should not be modified, even by the document owner. Time stamping helps enforce the validity of a signed or certified document. You can set time stamping options using a `TSPOptionSpec` object. For example, you can specify the URL of a time stamping provider (TSP) server.

Note: *In the Java and web service walk through sections and the corresponding quick starts, revocation checking is used. Because no CRL or OCSP server information is specified, the server information is obtained from the certificate used to digitally sign the PDF document.*

To successfully sign a PDF document, you can specify the fully qualified name of the signature field that will contain the digital signature, such as `form1[0].#subform[1].SignatureField3[3]`. When using an XFA form field, the partial name of the signature field can also be used: `SignatureField3[3]`.

You must also reference a security credential to digitally sign a PDF document. To reference a security credential, you specify an alias. The alias is a reference to an actual credential that may be in a PKCS#12 file (with a .pfx extension), or a hardware security module (HSM). For information about the security credential, see the *Installing and Deploying AEM Forms* guide for your application server.

Save the signed PDF document

After the Signature service digitally signs the PDF document, you can save it as a PDF file so that users can open it in Acrobat or Adobe Reader.

See also

"Digitally sign PDF documents using the Java API" on page 895

"Digitally signing PDF documents using the web service API" on page 896

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Adding Signature Fields" on page 880

"Retrieving Signature Field Names" on page 884

Digitally sign PDF documents using the Java API

Digitally sign a PDF document by using the Signature API (Java):

1 Include project files

Include client JAR files, such as adobe-signatures-client.jar, in your Java project's classpath.

2 Create a Signatures client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document to sign

- Create a `java.io.FileInputStream` object that represents the PDF document to digitally sign by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Sign the PDF document

Sign the PDF document by invoking the `SignatureServiceClient` object's `sign` method and passing the following values:

- A `com.adobe.idp.Document` object that represents the PDF document to sign.
- A string value that represents the name of the signature field that will contain the digital signature.

- A `Credential` object that represents the credential that is used to digitally sign the PDF document. Create a `Credential` object by invoking the `Credential` object's static `getInstance` method and passing a string value that specifies the alias value that corresponds to the security credential.
- A `HashAlgorithm` object that specifies a static data member that represents the hash algorithm to use to digest the PDF document. For example, you can specify `HashAlgorithm.SHA1` to use the SHA1 algorithm.
- A string value that represents the reason why the PDF document was digitally signed.
- A string value that represents the signer's contact information.
- A `PDFSignatureAppearanceOptions` object that controls the appearance of the digital signature. For example, you can use this object to add a custom logo to a digital signature.
- A `java.lang.Boolean` object that specifies whether to perform revocation checking on the signer's certificate.
- An `OCSPOptionSpec` object that stores preferences for Online Certificate Status Protocol (OCSP) support. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `CRLPreferences` object that stores certificate revocation list (CRL) preferences. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `TSPPreferences` object that stores preferences for time stamp provider (TSP) support. This parameter is optional and can be `null`. For more information, see AEM Forms API Reference.

The `sign` method returns a `com.adobe.idp.Document` object that represents the signed PDF document.

5 Save the signed PDF document

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method and pass `java.io.File` to copy the contents of the `Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that was returned by the `sign` method.

See also

"Digitally Signing PDF Documents" on page 892

"Quick Start (SOAP mode): Digitally signing a PDF document using the Java API" on page 380

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Digitally signing PDF documents using the web service API

To digitally sign a PDF document by using the Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Signatures client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)

- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document to sign

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is signed.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to sign, and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.

4 Sign the PDF document

Sign the PDF document by invoking the `SignatureServiceClient` object's `sign` method and passing the following values:

- A `BLOB` object that represents the PDF document to sign.
- A string value that represents the name of the signature field that will contain the digital signature.
- A `Credential` object that represents the credential that is used to digitally sign the PDF document. Create a `Credential` object by using its constructor and specify the alias by assigning a value to the `Credential` object's `alias` property.
- A `HashAlgorithm` object that specifies a static data member that represents the hash algorithm to use to digest the PDF document. For example, you can specify `HashAlgorithm.SHA1` to use the SHA1 algorithm.
- A Boolean value that specifies whether the hash algorithm is used.
- A string value that represents the reason why the PDF document was digitally signed.
- A string value that represents the signer's location.
- A string value that represents the signer's contact information.
- A `PDFSignatureAppearanceOptions` object that controls the appearance of the digital signature. For example, you can use this object to add a custom logo to a digital signature.
- A `System.Boolean` object that specifies whether to perform revocation checking on the signer's certificate. If this revocation checking is done, it is embedded in the signature. The default is `false`.

- An `OCSPOptionSpec` object that stores preferences for Online Certificate Status Protocol (OCSP) support. If revocation checking is not done, this parameter is not used and you can specify `null`. For information about this object, see AEM Forms API Reference.
- A `CRLPreferences` object that stores certificate revocation list (CRL) preferences. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `TSPPreferences` object that stores preferences for time stamp provider (TSP) support. This parameter is optional and can be `null`.

The `sign` method returns a `BLOB` object that represents the signed PDF document.

5 Save the signed PDF document

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the signed PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `sign` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Digitally Signing PDF Documents" on page 892

Quick Start (MTOM): Digitally signing a PDF document using the web service API

Quick Start (SwaRef): Digitally signing a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Digitally Signing Interactive Forms

You can sign an interactive form that the Forms service creates. For example, consider the following workflow:

- You merge an XFA-based PDF form created by using Designer and form data located in an XML document using the Forms service. The Forms server renders an interactive form.
- You sign the interactive form using the Signature service API.

The result is a digitally signed interactive PDF form. When signing a PDF form that is based on an XFA form, ensure that you save the PDF file as an Adobe Static PDF form. If you attempt to sign a PDF form that is saved as an Adobe Dynamic PDF form, an exception occurs. Because you are signing the form that is returned from the Forms service, ensure that the form contains a signature field.

Note: *Before you can digitally sign an interactive form, you must ensure that you add the certificate to AEM Forms. A certificate is added using administration console or programmatically using the Trust Manager API. (See "Importing Credentials by using the Trust Manager API" on page 1060.)*

When using the Forms Service API, set the `GenerateServerAppearance` run-time option to `true`. This run-time option ensures that the appearance of the form that is generated on the server remains valid when opened in Acrobat or Adobe Reader. It is recommended that you set this run-time option when generating an interactive form to sign by using the Forms API.

Note: *Before reading Digitally Signing Interactive Forms, it is recommended that you are familiar with signing PDF documents. (See "Digitally Signing PDF Documents" on page 892.)*

Summary of steps

To digitally sign an interactive form the Forms service returns, perform the following tasks:

1. Include project files.
2. Create a Forms and Signatures client.
3. Obtain the interactive form using the Forms service.
4. Sign the interactive form.
5. Save the signed PDF document as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-forms-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Forms and Signatures client

Because this workflow invokes both the Forms and Signature services, create both a Forms service client and Signature service client.

Obtain the interactive form using the Forms service

You can use the Forms service to obtain the interactive PDF form to sign. As of AEM Forms, you can pass a `com.adobe.idp.Document` object to the Forms service that contains the form to render. The name of this method is `renderPDFForm2`. This method returns a `com.adobe.idp.Document` object that contains the form to sign. You can pass this `com.adobe.idp.Document` instance to the Signature service.

Likewise, if you are using web services, you can pass the `BLOB` instance that the Forms service returns to the Signature service.

Note: *The quick start associated with Digitally Signing Interactive Forms section invokes the `renderPDFForm2` method.*

Sign the interactive form

When signing a PDF document, you can set run-time options that the Signature service uses. You can set the following options:

- Appearance options
- Revocation checking

- Time stamping values

You set appearance options by using a `PDFSignatureAppearanceOptionSpec` object. For example, you can display the date within a signature by invoking the `PDFSignatureAppearanceOptionSpec` object's `setShowDate` method and passing `true`.

Save the signed PDF document

After the Signature service digitally signs the PDF document, you can save it as a PDF file. The PDF file can be opened in Acrobat or Adobe Reader.

See also

"Digitally sign an interactive form using the Java API" on page 900

"Digitally sign an interactive form using the web service API" on page 902

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Digitally Signing PDF Documents" on page 892

"Rendering Interactive PDF Forms" on page 582

Digitally sign an interactive form using the Java API

Digitally sign an interactive form by using the Forms and Signature API (Java):

1 Include project files

Include client JAR files, such as adobe-signatures-client.jar and adobe-forms-client.jar, in your Java project's classpath.

2 Create a Forms and Signatures client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
- Create a `FormsServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Obtain the interactive form using the Forms service

- Create a `java.io.FileInputStream` object that represents the PDF document to pass to the Forms service by using its constructor. Pass a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
- Create a `java.io.FileInputStream` object that represents the XML document that contains form data to pass to the Forms service by using its constructor. Pass a string value that specifies the location of the XML file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
- Create a `PDFFormRenderSpec` object that is used to set run-time options. Invoke the `PDFFormRenderSpec` object's `setGenerateServerAppearance` method and pass `true`.
- Invoke the `FormsServiceClient` object's `renderPDFForm2` method and pass the following values:
  - A `com.adobe.idp.Document` object that contains the PDF form to render.
  - A `com.adobe.idp.Document` object that contains data to merge with the form.

- A `PDFFormRenderSpec` object that stores run-time options.
- A `URLSpec` object that contains URI values that are required by the Forms service. You can specify `null` for this parameter value.
- A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.

The `renderPDFForm2` method returns a `FormsResult` object that contains a form data stream

- Retrieve the PDF form by invoking the `FormsResult` object's `getOutputContent` method. This method returns a `com.adobe.idp.Document` object that represents the interactive form.

4 Sign the interactive form

Sign the PDF document by invoking the `SignatureServiceClient` object's `sign` method and passing the following values:

- A `com.adobe.idp.Document` object that represents the PDF document to sign. Ensure that this object is the `com.adobe.idp.Document` object obtained from the Forms service.
- A string value that represents the name of the signature field that is signed.
- A `Credential` object that represents the credential that is used to digitally sign the PDF document. Create a `Credential` object by invoking the `Credential` object's static `getInstance` method. Pass a string value that specifies the alias value that corresponds to the security credential.
- A `HashAlgorithm` object that specifies a static data member that represents the hash algorithm to use to digest the PDF document. For example, you can specify `HashAlgorithm.SHA1` to use the SHA1 algorithm.
- A string value that represents the reason why the PDF document was digitally signed.
- A string value that represents the signer's contact information.
- A `PDFSignatureAppearanceOptions` object that controls the appearance of the digital signature. For example, you can use this object to add a custom logo to a digital signature.
- A `java.lang.Boolean` object that specifies whether to perform revocation checking on the signer's certificate.
- An `OCSPPreferences` object that stores preferences for Online Certificate Status Protocol (OCSP) support. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `CRLPreferences` object that stores certificate revocation list (CRL) preferences. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `TSPPreferences` object that stores preferences for time stamp provider (TSP) support. This parameter is optional and can be `null`.

The `sign` method returns a `com.adobe.idp.Document` object that represents the signed PDF document.

5 Save the signed PDF document

- Create a `java.io.File` object and ensure that the filename extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method and pass `java.io.File` to copy the contents of the `Document` object to the file. Ensure that you use the `com.adobe.idp.Document` object that the `sign` method returned.

See also

"Digitally Signing Interactive Forms" on page 898

"Quick Start (SOAP mode): Digitally signing a PDF document using the Java API" on page 380

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Digitally sign an interactive form using the web service API

Digitally sign an interactive form by using the Forms and Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Because this client application invokes two AEM Forms services, create two service references. Use the following WSDL definition for the service reference associated with the Signature service: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Use the following WSDL definition for the service reference associated with the Forms service: `http://localhost:8080/soap/services/FormsService?WSDL&lc_version=9.0.1`.

Because the `BLOB` data type is common to both service references, fully qualify the `BLOB` data type when using it. In the corresponding web service quick start, all `BLOB` instances are fully qualified.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Forms and Signatures client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

Note: *Repeat these steps for the Forms service client.*

3 Obtain the interactive form using the Forms service

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is signed.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to sign, and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.

- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.
- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store form data.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the XML file that contains form data, and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.
- Create a `PDFFormRenderSpec` object that is used to set run-time options. Assign the value `true` to the `PDFFormRenderSpec` object's `generateServerAppearance` field.
- Invoke the `FormsServiceClient` object's `renderPDFForm2` method and pass the following values:
  - A `BLOB` object that contains the PDF form to render.
  - A `BLOB` object that contains data to merge with the form.
  - A `PDFFormRenderSpec` object that stores run-time options.
  - A `URLSpec` object that contains URI values that are required by the Forms service. You can specify `null` for this parameter value.
  - A `java.util.HashMap` object that stores file attachments. This is an optional parameter and you can specify `null` if you do not want to attach files to the form.
  - A long output parameter used to store the number of pages in the form.
  - A string output parameter that is used for the locale value.
  - A `FormResult` value that is an output parameter that is used to store the interactive form.
- Retieve the PDF form by invoking the `FormsResult` object's `outputContent` field. This field stores a `BLOB` object that represents the interactive form.

4 Sign the interactive form

Sign the PDF document by invoking the `SignatureServiceClient` object's `sign` method and passing the following values:

- A `BLOB` object that represents the PDF document to sign. Use the `BLOB` instance returned by the Forms service.
- A string value that represents the name of the signature field that is signed.
- A `Credential` object that represents the credential that is used to digitally sign the PDF document. Create a `Credential` object by using its constructor and specify the alias by assigning a value to the `Credential` object's `alias` property.
- A `HashAlgorithm` object that specifies a static data member that represents the hash algorithm to use to digest the PDF document. For example, you can specify `HashAlgorithm.SHA1` to use the SHA1 algorithm.
- A Boolean value that specifies whether the hash algorithm is used.
- A string value that represents the reason why the PDF document was digitally signed.
- A string value that represents the signer's location.
- A string value that represents the signer's contact information.
- A `PDFSignatureAppearanceOptions` object that controls the appearance of the digital signature. For example, you can use this object to add a custom logo to a digital signature.

- A `System.Boolean` object that specifies whether to perform revocation checking on the signer's certificate. If this revocation checking is done, it is embedded in the signature. The default is `false`.
- An `OCSPPreferences` object that stores preferences for Online Certificate Status Protocol (OCSP) support. If revocation checking is not done, this parameter is not used and you can specify `null`. For information about this object, see AEM Forms API Reference.
- A `CRLPreferences` object that stores certificate revocation list (CRL) preferences. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `TSPPreferences` object that stores preferences for time stamp provider (TSP) support. This parameter is optional and can be `null`.

The `sign` method returns a `BLOB` object that represents the signed PDF document.

5 Save the signed PDF document

- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the signed PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `sign` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Digitally Signing Interactive Forms" on page 898

Quick Start (MTOM): Digitally signing a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

Certifying PDF Documents

You can secure a PDF document by certifying it with a particular type of signature called a certified signature. A certified signature is distinguished from a digital signature in these ways:

- It must be the first signature applied to the PDF document; that is, at the time the certified signature is applied, any other signature fields in the document must be unsigned. Only one certified signature is permitted in a PDF document. If you want to sign and certify a PDF document, you must certify it before signing it. After you certify a PDF document, you can digitally sign additional signature fields.
- The author or originator of the document can specify that the document can be modified in certain ways without invalidating the certified signature. For example, the document may permit filling in forms or commenting. If the author specifies that a certain modification is not permitted, Acrobat restricts users from modifying the document in that way. If such modifications are made, such as by using another application, the certified signature is invalid and Acrobat issues a warning when a user opens the document. (With non-certified signatures, modifications are not prevented, and normal editing operations do not invalidate the original signature.)
- At the time of signing, the document is scanned for specific types of content that could make the contents of a document ambiguous or misleading. For example, an annotation could obscure some text on a page that is important for understanding what is being certified. An explanation (legal attestation) can be provided about such content.

You can programmatically certify PDF documents by using the Signature service Java API or the Signature web service API. When certifying a PDF document, you must reference a security credential that exists in the Credential service. For information about the security credential, see the *Installing and Deploying AEM Forms* guide for your application server.

Note: *When certifying and signing the same PDF document, if the certify signature is not trusted, a yellow triangle appears next to the first sign signature when you open the PDF document in Acrobat or Adobe Reader. The certifying signature must be trusted to avoid this situation.*

Note: *When using an nCipher nShield HSM credential to sign or certify a PDF document, the new credential cannot be used until the J2EE application server on which AEM Forms is deployed is restarted. However, you can set a configuration value, resulting in the sign or certify operation working without restarting the J2EE application server.*

You can add the following configuration value in the cknfastrc file, which is located at /opt/nfast/cknfastrc (or c:\nfast\cknfastrc):

```
CKNFAST_ASSUME_SINGLE_PROCESS=0
```

After you add this configuration value to the cknfastrc file, the new credential can be used without restarting the J2EE application server.

Note: *For more information about the Signature service and certifying a document, see Services Reference for AEM Forms.*

Summary of steps

To certify a PDF document, perform the following tasks:

1. Include project files.
2. Create a Signature client.
3. Get the PDF document to certify.
4. Certify the PDF document.
5. Save the certified PDF document as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Signature client

Before you can programmatically perform a Signature operation, you must create a Signature client.

Get the PDF document to certify

To certify a PDF document, you must obtain a PDF document that contains a signature field. If a PDF document does not contain a signature field, it cannot be certified. A signature field can be added by using Designer or programmatically. For information about programmatically adding a signature field, see "Adding Signature Fields" on page 880.

Certify the PDF document

To successfully certify a PDF document, you require the following input values that are used by the Signature service to certify a PDF document:

- **PDF document**: A PDF document that contains a signature field, which is a form field that contains a graphic representation of the certified signature. A PDF document must contain a signature field before it can be certified. A signature field can be added by using Designer or programmatically. (See "Adding Signature Fields" on page 880.)
- **Signature field name**: The fully-qualified name of the signature field that is certified. The following value is an example: `form1[0].#subform[1].SignatureField3[3]`. When using an XFA form field, the partial name of the signature field can also be used: `SignatureField3[3]`. If a null value is passed for the field name, an invisible signature field is dynamically created and certified.
- **Security credential**: A credential that is used to certify the PDF document. This security credential contains a password and an alias, which must match an alias that appears in the credential that is located within the Credential service. The alias is a reference to an actual credential that may be in a PKCS#12 file (with a .pfx extension) or a hardware security module (HSM).
- **Hash algorithm**: A hash algorithm to use to digest the PDF document.
- **Reason for signing**: A value that is displayed in Acrobat or Adobe Reader so that other users know the reason why the PDF document was certified.
- **Location of the signer**: The location of the signer specified by the credential.
- **Contact information**: Contact information, such as address and telephone number, of the signer.
- **Permission information**: Permissions that control the actions that an end user can perform on a document without causing the certified signature to be invalid. For example, you can set the permission so that any change to the PDF document causes the certified signature to be invalid.
- **Legal explanation**: When a document is certified, it is automatically scanned for specific types of content that could make the contents of a document ambiguous or misleading. For example, an annotation could obscure some text on a page that is important for understanding what is being certified. The scanning process generates warnings about these types of content. This value provides an additional explanation of the content that may have generated warnings.
- **Appearance options**: Options that control the appearance of the certified signature. For example, the certified signature can display date information.
- **Revocation checking**: This value specifies whether revocation checking is done for the signer's certificate. The default setting of `false` means that revocation checking is not done.
- **OCSP settings**: Settings for Online Certificate Status Protocol (OCSP) support, which provides information about the status of the credential that is used to certify the PDF document. You can, for example, specify the URL of the server that provides information about the credential that you are using to sign on to the PDF document.
- **CRL settings**: Settings for certificate revocation list (CRL) preferences if revocation checking is done. For example, you can specify to always check whether a credential was revoked.

- **Time stamping**: Settings that define time stamping information that is applied to the certified signature. A time stamp indicates that specific data was established before a certain time. This knowledge helps build a trusting relationship between the signer and verifier.

Save the certified PDF document as a PDF file

After the Signature service certifies the PDF document, you can save it as a PDF file so that users can open it in Acrobat or Adobe Reader.

See also

"Certify PDF documents using the Java API" on page 907

"Certify PDF documents using the web service API" on page 908

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Adding Signature Fields" on page 880

Certify PDF documents using the Java API

Certify a PDF document by using the Signature API (Java):

1 Include project files

Include client JAR files, such as adobe-signatures-client.jar, in your Java project's class path.

2 Create a Signature client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document to certify

- Create a `java.io.FileInputStream` object that represents the PDF document to certify by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Certify the PDF document

Certify the PDF document by invoking the `SignatureServiceClient` object's `certify` method and passing the following values:

- The `com.adobe.idp.Document` object that represents the PDF document to certify.
- A string value that represents the name of the signature field that will contain the signature.
- A `Credential` object that represents the credential that is used to certify the PDF document. Create a `Credential` object by invoking the `Credential` object's static `getInstance` method and passing a string value that specifies the alias value that corresponds to the security credential.
- A `HashAlgorithm` object that specifies a static data member that represents the hash algorithm used to digest the PDF document. For example, you can specify `HashAlgorithm.SHA1` to use the SHA1 algorithm.
- A string value that represents the reason why the PDF document was certified.
- A string value that represents the signer's contact information.

- A `MDPPermissions` object that specifies actions that can be performed on the PDF document that invalidates the signature.
- A `PDFSignatureAppearanceOptions` object that controls the appearance of the certified signature. If desired, modify the appearance of the signature by invoking a method, such as `setShowDate`.
- A string value that provides an explanation of what actions invalidate the signature.
- A `java.lang.Boolean` object that specifies whether to perform revocation checking on the signer's certificate. If this revocation checking is done, it is embedded in the signature. The default is `false`.
- A `java.lang.Boolean` object that specifies whether the signature field being certified is locked. If the field is locked, the signature field is marked as read only, its properties cannot be modified, and it cannot be cleared by anyone who does not have the required permissions. The default is `false`.
- An `OCSPPreferences` object that stores preferences for Online Certificate Status Protocol (OCSP) support. If revocation checking is not done, this parameter is not used and you can specify `null`. For information about this object, See AEM Forms API Reference.
- A `CRLPreferences` object that stores certificate revocation list (CRL) preferences. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `TSPPreferences` object that stores preferences for time stamp provider (TSP) support. For example, after you create a `TSPPreferences` object, you can set the URL of the TSP server by invoking the `TSPPreferences` object's `setTspServerURL` method. This parameter is optional and can be `null`. For more information, see Services Reference for AEM Forms.

The `certify` method returns a `com.adobe.idp.Document` object that represents the certified PDF document.

5 Save the certified PDF document as a PDF file

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file.

See also

"Certifying PDF Documents" on page 904

"Quick Start (SOAP mode): Certifying a PDF document using the Java API" on page 386

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Certify PDF documents using the web service API

Certify a PDF document by using the Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.

- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document to certify

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that is certified.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document to certify and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` data member the contents of the byte array.

4 Certify the PDF document

Certify the PDF document by invoking the `SignatureServiceClient` object's `certify` method and passing the following values:

- The `BLOB` object that represents the PDF document to certify.
- A string value that represents the name of the signature field that will contain the signature.
- A `Credential` object that represents the credential that is used to certify the PDF document. Create a `Credential` object by using its constructor, and specify the alias by assigning a value to the `Credential` object's `alias` property.
- A `HashAlgorithm` object that specifies a static data member that represents the hash algorithm used to digest the PDF document. For example, you can specify `HashAlgorithm.SHA1` to use the SHA1 algorithm.
- A Boolean value that specifies whether the hash algorithm is used.
- A string value that represents the reason why the PDF document was certified.
- A string value that represents the signer's location.
- A string value that represents the signer's contact information.

- An `MDPPermissions` object's static data member that specifies actions that can be performed on the PDF document that invalidate the signature.
- A Boolean value that specifies whether to use the `MDPPermissions` object that was passed as the previous parameter value.
- A string value that explains what actions invalidate the signature.
- A `PDFSignatureAppearanceOptions` object that controls the appearance of the certified signature. Create a `PDFSignatureAppearanceOptions` object by using its constructor. You can modify the appearance of the signature by setting one of its data members.
- A `System.Boolean` object that specifies whether to perform revocation checking on the signer's certificate. If this revocation checking is done, it is embedded in the signature. The default is `false`.
- A `System.Boolean` object that specifies whether the signature field being certified is locked. If the field is locked, the signature field is marked as read only, its properties cannot be modified, and it cannot be cleared by anyone who does not have the required permissions. The default is `false`.
- A `System.Boolean` object that specifies whether the signature field is locked. That is, if you pass `true` to the previous parameter, then pass `true` to this parameter.
- An `OCSPPreferences` object that stores preferences for Online Certificate Status Protocol (OCSP) support, which provides information about the status of the credential that is used to certify the PDF document. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `CRLPreferences` object that stores certificate revocation list (CRL) preferences. If revocation checking is not done, this parameter is not used and you can specify `null`.
- A `TSPPreferences` object that stores preferences for time stamp provider (TSP) support. For example, after you create a `TSPPreferences` object, you can set the URL of the TSP by setting the `TSPPreferences` object's `tspServerURL` data member. This parameter is optional and can be `null`.

The `certify` method returns a `BLOB` object that represents the certified PDF document.

5 Save the certified PDF document as a PDF file

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document that will contain the certified PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `certify` method. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Certifying PDF Documents" on page 904

Quick Start (MTOM): Certifying a PDF document using the web service API

Quick Start (SwaRef): Certifying a PDF document using the web service API

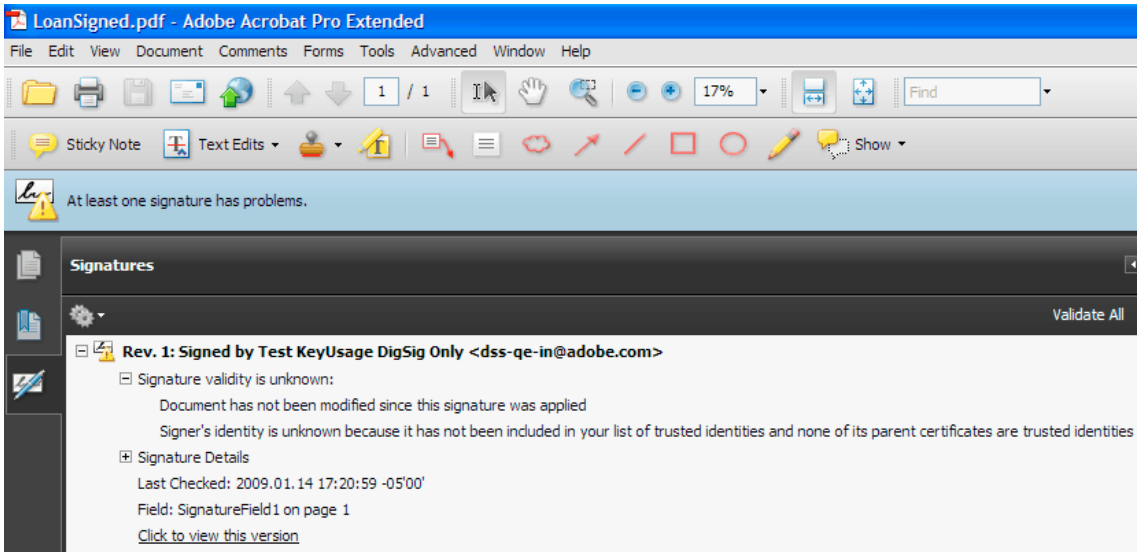
"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Verifying Digital Signatures

Digital signatures can be verified to ensure that a signed PDF document was not modified and that the digital signature is valid. When verifying a digital signature, you can check the signature's status and the signature's properties, such as the signer's identity. Before trusting a digital signature, it is recommended that you verify it. When verifying a digital signature, reference a PDF document that contains a digital signature.

Assume that the identity of the signer is unknown. When you open the PDF document in Acrobat, a warning message states that the signer's identity is unknown, as shown in the following illustration.



Likewise, when you programmatically verify a digital signature, you can determine the status of the signer's identity. For example, if you verify the digital signature in the document shown in the previous illustration, the result would be that the signer's identity is unknown.

Note: *For more information about the Signature service and verifying digital signatures, see Services Reference for AEM Forms.*

Summary of steps

To verify a digital signature, perform the following tasks:

1. Include project files.
2. Create a Signature client.
3. Get the PDF document that contains the signature to verify.
4. Set PKI run-time options.
5. Verify the digital signature.
6. Determine the status of the signature.
7. Determine the identity of the signer.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Signature client

Before you programmatically perform a Signature service operation, create a Signature service client.

Get the PDF document that contains the signature to verify

To verify a signature used to digitally sign or certify a PDF document, obtain a PDF document that contains a signature.

Set PKI run-time options

Set these PKI run-time options that the Signature service uses when verifying signatures in a PDF document:

- Verification time
- Revocation checking
- Time-stamping values

As part of setting these options, you can specify verification time. For example, you can select current time (the time on the validator's computer), which indicates to use the current time. For information about the different time values, see the `VerificationTime` enumeration value in AEM Forms API Reference.

You can also specify whether to perform revocation checking as part of the verification process. For example, you can perform a revocation check to determine whether the certificate is revoked. For information about the revocation-checking options, see the `RevocationCheckStyle` enumeration value in AEM Forms API Reference.

To perform revocation checking on a certificate, specify a URL to a certificate revocation list (CRL) server by using a `CRLOptionSpec` object. However, if you do not specify a URL to CRL server, the Signature service obtains the URL from the certificate.

Instead of using a CRL server, you can use an online certificate status protocol (OCSP) server when performing revocation checking. Typically, when using an OCSP server as opposed to a CRL server, the revocation check is performed faster. (See Online Certificate Status Protocol.)

You can set the CRL and OCSP server order that the Signature service uses by using Adobe Applications and Services. For example, if the OCSP server is set first in Adobe Applications and Services, then the OCSP server is checked, followed by the CRL server.

If you do not perform revocation checking, the Signature service does not check whether the certificate is revoked. That is, CRL and OCSP server information is ignored.

Note: *You can override the URL specified in the certificate by using a* `CRLOptionSpec` *and an* `OCSPOptionSpec` *object. For example, to override the CRL server, you can invoke the* `CRLOptionSpec` *object's* `setLocalURI` *method.*

Time stamping is the process of tracking the time when a signed or certified document was modified. After a document is signed, no one can modify it. Time stamping helps enforce the validity of a signed or certified document. You can set time stamping options using a `TSPOptionSpec` object. For example, you can specify the URL of a time stamping provider (TSP) server.

Note: *In the Java and web service quick starts, the verification time is set to* `VerificationTime.CURRENT_TIME` *and revocation checking is set to* `RevocationCheckStyle.BestEffort`*. Because no CRL or OCSP server information is specified, the server information is obtained from the certificate.*

Verify the digital signature

To successfully verify a signature, specify the fully qualified name of the signature field that contains the signature, such as `form1[0].#subform[1].SignatureField3[3]`. When using an XFA form field, you can also use the partial name of the signature field : `SignatureField3`.

By default, the Signature service limits the amount of time that a document can be signed after validation time to 65 min. If a user attempts to verify a signature at current time and the sign time is later than the current time and is within 65 min, the Signature service does not create a verification error.

Note: *For other values that you require when verifying a signature, see AEM Forms API Reference.*

Determine the status of the signature

As part of verifying a digital signature, you can check the status of the signature.

Determine the identity of the signer

You can determine the identity of the signer, which can be one of the following values:

- **Unknown**: This signer is unknown because the signer verification cannot be performed.
- **Trusted**: This signer is trusted.
- **Not trusted**: This signer is not trusted.

See also

"Verify digital signatures using the Java API" on page 913

"Verify digital signatures using the web service API" on page 915

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Verify digital signatures using the Java API

Verify a digital signature by using the Signature Service API (Java):

1 Include project files

Include client JAR files, such as adobe-signatures-client.jar, in your Java project's classpath.

2

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document that contains the signature to verify

- Create a `java.io.FileInputStream` object that represents the PDF document that contains the signature to verify by using its constructor. Pass a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set PKI run-time options

- Create a `PKIOptions` object by using its constructor.
- Set the verification time by invoking the `PKIOptions` object's `setVerificationTime` method and passing a `VerificationTime` enumeration value that specifies the verification time.
- Set the revocation-checking option by invoking `PKIOptions` object's `setRevocationCheckStyle` method and passing a `RevocationCheckStyle` enumeration value that specifies whether to perform revocation checking.

5 Verify the digital signature

Verify the signature by invoking the `SignatureServiceClient` object's `verify2` method and passing the following values:

- A `com.adobe.idp.Document` object that contains a digitally signed or certified PDF document.
- A string value that represents the signature field name that contains the signature to verify.
- A `PKIOptions` object that contains PKI run-time options.
- A `VerifySPIOptions` instance that contains SPI information. You can specify `null` for this parameter.

The `verify2` method returns a `PDFSignatureVerificationInfo` object that contains information that can be used to verify the digital signature.

6 Determine the status of the signature

- Determine the signature's status by invoking the `PDFSignatureVerificationInfo` object's `getStatus` method. This method returns a `SignatureStatus` object that specifies the signature status. For example, if a signed PDF document is not modified, this method returns `SignatureStatus.DocumentSigNoChanges`.

7 Determine the identity of the signer

- Determine the signer's identity by invoking the `PDFSignatureVerificationInfo` object's `getSigner` method. This method returns an `IdentityInformation` object.
- Invoke the `IdentityInformation` object's `getStatus` method to determine the signer's identity. This method returns an `IdentityStatus` enumeration value that specifies the identity. For example, if the signer is trusted, this method returns `IdentityStatus.TRUSTED`.

See also

"Verifying Digital Signatures" on page 911

"Quick Start (SOAP mode): Verifying a digital signature using the Java API" on page 389

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Verify digital signatures using the web service API

Verify a digital signature by using the Signature Service API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document that contains the signature to verify

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that contains a digital or certified signature to verify.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the signed PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.

4 Set PKI run-time options

- Create a `PKIOptions` object by using its constructor.
- Set the verification time by assigning the `PKIOptions` object's `verificationTime` data member a `VerificationTime` enumeration value that specifies the verification time.
- Set the revocation-checking option by assigning the `PKIOptions` object's `revocationCheckStyle` data member a `RevocationCheckStyle` enumeration value that specifies whether to perform revocation checking.

5 Verify the digital signature

Verify the signature by invoking the `SignatureServiceClient` object's `verify2` method and passing the following values:

- The `BLOB` object that contains a digitally signed or certified PDF document.
- A string value that represents the signature field name that contains the signature to verify.
- A `PKIOptions` object that contains PKI run-time options.
- A `VerifySPIOptions` instance that contains SPI information. You can specify `null` for this parameter.

The `verify2` method returns a `PDFSignatureVerificationInfo` object that contains information that can be used to verify the digital signature.

6 Determine the status of the signature

Determine the signature's status by getting the value of the `PDFSignatureVerificationInfo` object's `status` data member. This data member stores a `SignatureStatus` object that specifies the signature's status. For example, if a signed PDF document is modified, the `status` data member stores the value `SignatureStatus.DocumentSigNoChanges`.

7 Determine the identity of the signer

- Determine the signer's identity by retrieving the value of the `PDFSignatureVerificationInfo` object's `signer` data member. This member returns an `IdentityInformation` object.
- Retrieve the `IdentityInformation` object's `status` data member to determine the signer's identity. This data member returns an `IdentityStatus` enumeration value that specifies the identity. For example, if the signer is trusted, this member returns `IdentityStatus.TRUSTED`.

See also

"Verifying Digital Signatures" on page 911

Quick Start (MTOM): Verifying a digital signature using the web service API

Quick Start (SwaRef): Verifying a digital signature using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Verifying Multiple Digital Signatures

AEM Forms provides the means to verify all digital signatures that are located in a PDF document. Assume that a PDF document contains multiple digital signatures as a result of a business process that requires signatures from multiple signers. For example, consider a financial transaction that requires both a loan officer's and a manager's signature. You can use the Signature service Java API or web service API to verify all signatures within the PDF document. When verifying multiple digital signatures, you can check the status and properties of each signature. Before you trust a digital signature, it is recommended that you verify it. It is recommended that you are familiar with verifying a single digital signature.

Note: *For more information about the Signature service and verifying digital signatures, see Services Reference for AEM Forms.*

Summary of steps

To verify multiple digital signature, perform the following tasks:

1 Include project files.

2 Create a Signature client.

3 Get the PDF document that contains the signatures to verify.

4 Set PKI run-time options.

5 Retrieve all digital signatures.

6 Iterate through all signatures.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Signature client

Before you programmatically perform a Signature service operation, create a Signature service client.

Get the PDF document that contains the signatures to verify

To verify a signature used to digitally sign or certify a PDF document, obtain a PDF document that contains a signature.

Set PKI runtime options

Set these PKI run-time options that the Signature service uses when verifying all signatures in a PDF document:

- Verification time
- Revocation checking
- Time-stamping values

As part of setting these options, you can specify verification time. For example, you can select current time (the time on the validator's computer), which indicates to use the current time. For information about the different time values, see the `VerificationTime` enumeration value in AEM Forms API Reference.

You can also specify whether to perform revocation checking as part of the verification process. For example, you can perform a revocation check to determine whether the certificate is revoked. For information about the revocation-checking options, see the `RevocationCheckStyle` enumeration value in AEM Forms API Reference.

To perform revocation checking on a certificate, specify a URL to a certificate revocation list (CRL) server by using a `CRLOptionSpec` object. However, if you do not specify a URL to a CRL server, the Signature service obtains the URL from the certificate.

Instead of using a CRL server, you can use an online certificate status protocol (OCSP) server when performing revocation checking. Typically, when using an OCSP server instead of a CRL server, the revocation check is performed faster. (See Online Certificate Status Protocol.)

You can set the CRL and OCSP server order that the Signature service uses by using Adobe Applications and Services. For example, if the OCSP server is set first in Adobe Applications and Services, the OCSP server is checked, followed by the CRL server.

If you do not perform revocation checking, the Signature service does not check whether the certificate is revoked. That is, CRL and OCSP server information is ignored.

Note: *You can override the URL specified in the certificate by using a* `CRLOptionSpec` *and an* `OCSPOptionSpec` *object. For example, to override the CRL server, you can invoke the* `CRLOptionSpec` *object's* `setLocalURI` *method.*

Time stamping is the process of tracking the time when a signed or certified document was modified. After a document is signed, no one can modify it. Time stamping helps enforce the validity of a signed or certified document. You can set time stamping options by using a `TSPOptionSpec` object. For example, you can specify the URL of a time stamping provider (TSP) server.

Note: *In the Java and web service quick starts, the verification time is set to* `VerificationTime.CURRENT_TIME` *and revocation checking is set to* `RevocationCheckStyle.BestEffort`*. Because no CRL or OCSP server information is specified, the server information is obtained from the certificate.*

Retrieve all digital signatures

To verify all digital signatures located in a PDF document, retrieve the digital signatures from the PDF document. All signatures are returned in a list. As part of verifying a digital signature, check the status of the signature.

Note: *Unlike when you verify a single digital signature, when you verify multiple signatures, you are not required to specify the signature field name.*

Iterate through all signatures

Iterate through each signature. That is, for each signature, verify the digital signature, and check the signer's identity and the status of each signature. (See "Verifying Digital Signatures" on page 911.)

Note: *You do not need to iterate through all the signatures if the requirement is the entire document.*

See also

"Verify multiple digital signatures using the Java API" on page 918

"Verifying multiple digital signatures using the web service API" on page 919

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Verify multiple digital signatures using the Java API

Verify multiple digital signatures by using the Signature Service API (Java):

1 Include project files

   Include client JAR files, such as adobe-signatures-client.jar, in your Java project's classpath.

2 Create a Signature client

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document that contains the signatures to verify

- Create a `java.io.FileInputStream` object that represents the PDF document that contains multiple digital signatures to verify by using its constructor. Pass a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set PKI runtime options

- Create a `PKIOptions` object by using its constructor.
- Set the verification time by invoking the `PKIOptions` object's `setVerificationTime` method and passing a `VerificationTime` enumeration value that specifies the verification time.
- Set the revocation checking option by invoking `PKIOptions` object's `setRevocationCheckStyle` method and passing a `RevocationCheckStyle` enumeration value that specifies whether to perform revocation checking.

5 Retrieve all digital signatures

Invoke the `SignatureServiceClient` object's `verifyPDFDocument` method and pass the following values:

- A `com.adobe.idp.Document` object that contains a PDF document that contains multiple digital signatures.
- A `PKIOptions` object that contains PKI run-time options.
- A `VerifySPIOptions` instance that contains SPI information. You can specify `null` for this parameter.

The `verifyPDFDocument` method returns a `PDFDocumentVerificationInfo` object that contains information about all the digital signatures located in the PDF document.

6 Iterate through all signatures

- Iterate through all signatures by invoking the `PDFDocumentVerificationInfo` object's `getVerificationInfos` method. This method returns a `java.util.List` object where each element is a `PDFSignatureVerificationInfo` object. Use a `java.util.Iterator` object to iterate through the list of signatures.
- Using the `PDFSignatureVerificationInfo` object, you can perform tasks such as determining the status of the signature by invoking the `PDFSignatureVerificationInfo` object's `getStatus` method. This method returns a `SignatureStatus` object whose static data member informs you about the status of the signature. For example, if the signature is unknown, this method returns `SignatureStatus.DocumentSignatureUnknown`.

See also

"Verifying Multiple Digital Signatures" on page 916

"Quick Start (SOAP mode): Verifying multiple digital signatures using the Java API" on page 393

"Including AEM Forms Java library files" on page 491

"Verifying Digital Signatures" on page 911

"Setting connection properties" on page 500

Verifying multiple digital signatures using the web service API

Verify multiple digital signatures by using the Signature Service API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition:
`http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document that contains the signatures to verify

- Create a `BLOB` object by using its constructor. The `BLOB` object stores a PDF document that contains multiple digital signatures to verify.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.

4 Set PKI runtime options

- Create a `PKIOptions` object by using its constructor.
- Set the verification time by assigning the `PKIOptions` object's `verificationTime` data member a `VerificationTime` enumeration value that specifies the verification time.
- Set the revocation checking option by assigning the `PKIOptions` object's `revocationCheckStyle` data member a `RevocationCheckStyle` enumeration value that specifies whether to perform revocation checking.

5 Retrieve all digital signatures

Invoke the `SignatureServiceClient` object's `verifyPDFDocument` method and pass the following values:

- A `BLOB` object that contains a PDF document that contains multiple digital signatures.
- A `PKIOptions` object that contains PKI run-time options.

- A `VerifySPIOptions` instance that contains SPI information. You can specify null for this parameter.

The `verifyPDFDocument` method returns a `PDFDocumentVerificationInfo` object that contains information about all the digital signatures located in the PDF document.

6 Iterate through all signatures

- Iterate through all signatures by getting the `PDFDocumentVerificationInfo` object's `verificationInfos` data member. This data member returns an `Object` array where each element is a `PDFSignatureVerificationInfo` object.
- Using the `PDFSignatureVerificationInfo` object, you can perform tasks like determining the status of the signature by getting the `PDFSignatureVerificationInfo` object's `status` data member. This data member returns a `SignatureStatus` object whose static data member informs you about the status of the signature. For example, if the signature is unknown, this method returns `SignatureStatus.DocumentSignatureUnknown`.

See also

"Verifying Multiple Digital Signatures" on page 916

Quick Start (MTOM): Verifying multiple digital signatures using the web service API

Quick Start (MTOM): Verifying multiple digital signatures using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Removing Digital Signatures

Digital signatures must be removed from a signature field before a newer digital signature can be applied. A digital signature cannot be overwritten. If you attempt to apply a digital signature to a signature field that contains a signature, an exception occurs.

Note: *For more information about the Signature service, see Services Reference for AEM Forms.*

Summary of steps

To remove a digital signature from a signature field, perform the following tasks:

1 Include project files.

2 Create a Signature client.

3 Get the PDF document that contains a signature to remove.

4 Remove the digital signature from the signature field.

5 Save the PDF document as a PDF file.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-signatures-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)

- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a Signature client
Before you can programmatically perform a Signature service operation, you must create a Signature service client.

Get the PDF document that contains a signature to remove
To remove a signature from a PDF document, you must obtain a PDF document that contains a signature.

Remove the digital signature from the signature field
To successfully remove a digital signature from a PDF document, you must specify the name of the signature field that contains the digital signature. Also, you must have permission to remove the digital signature; otherwise, an exception occurs.

Save the PDF document as a PDF file
After the Signature service removes a digital signature from a signature field, you can save the PDF document as a PDF file so that users can open it in Acrobat or Adobe Reader.

See also
"Remove digital signatures using the Java API" on page 922

"Remove digital signatures using the web service API" on page 923

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Adding Signature Fields" on page 880

Remove digital signatures using the Java API

Remove a digital signature by using the Signature API (Java):

1 Include project files

Include client JAR files, such as adobe-signatures-client.jar, in your Java project's class path.

2

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `SignatureServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Get the PDF document that contains a signature to remove

- Create a `java.io.FileInputStream` object that represents the PDF document that contains the signature to remove by using its constructor and passing a string value that specifies the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Remove the digital signature from the signature field

Remove a digital signature from a signature field by invoking the `SignatureServiceClient` object's `clearSignatureField` method and passing the following values:

- A `com.adobe.idp.Document` object that represents the PDF document that contains the signature to remove.

- A string value that specifies the name of the signature field that contains the digital signature.

The `clearSignatureField` method returns a `com.adobe.idp.Document` object that represents the PDF document from which the digital signature was removed.

5 Save the PDF document as a PDF file

- Create a `java.io.File` object and ensure that the file extension is .pdf.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method. Pass the `java.io.File` object to copy the contents of the `com.adobe.idp.Document` object to the file. Ensure that you use the `Document` object that was returned by the `clearSignatureField` method.

See also

"Removing Digital Signatures" on page 921

"Quick Start (SOAP mode): Removing a digital signature using the Java API" on page 396

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Remove digital signatures using the web service API

Remove a digital signature by using the Signature API (web service):

1 Include project files

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/SignatureService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a Signature client

- Create a `SignatureServiceClient` object by using its default constructor.
- Create a `SignatureServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/SignatureService?WSDL`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `SignatureServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `SignatureServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `SignatureServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Get the PDF document that contains a signature to remove

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document that contains a digital signature to remove.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the signed PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Remove the digital signature from the signature field

Remove the digital signature by invoking the `SignatureServiceClient` object's `clearSignatureField` method and passing the following values:

- A `BLOB` object that contains the signed PDF document.
- A string value that represents the name of the signature field that contains the digital signature to remove.

The `clearSignatureField` method returns a `BLOB` object that represents the PDF document from which the digital signature was removed.

5 Save the PDF document as a PDF file

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document that contains an empty signature field and the mode in which to open the file.
- Create a byte array that stores the content of the `BLOB` object that was returned by the `sign` method. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to the PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Removing Digital Signatures" on page 921

Quick Start (MTOM): Removing a digital signature using the web service API

Quick Start (SwaRef): Removing a digital signature using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Assembling PDF Documents

About the Assembler Service

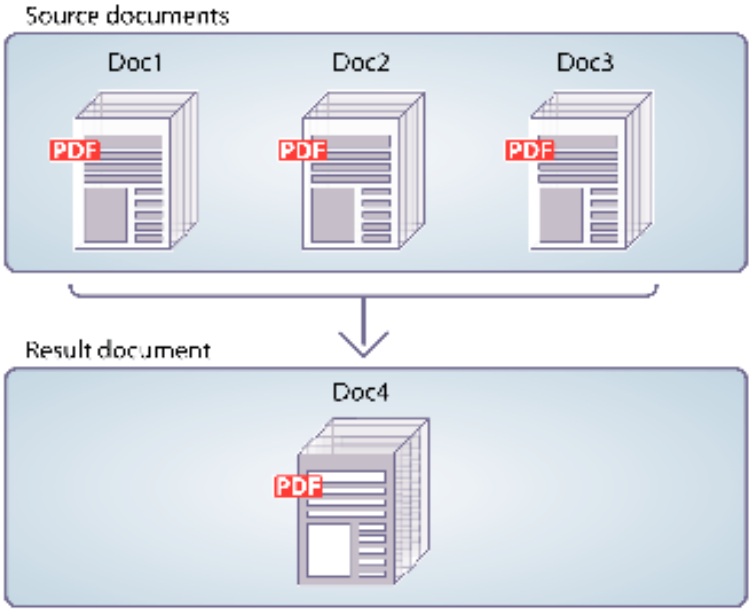
The Assembler service can assemble multiple PDF documents into one PDF document or disassemble one PDF document into multiple PDF documents. The Assembler service can manipulate documents in various ways, such as changing page size and rotating contents. It can insert additional content, such as headers, footers, and a table of contents, and can preserve, import, or export existing content, such as annotations, file attachments, and bookmarks.

Starting with LiveCycle ES 8.0 and later, support for PDF packages is available in the Assembler service.

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Programmatically Assembling PDF Documents

You can use the Assembler Service API to assemble multiple PDF documents into a single PDF document. The following illustration shows three PDF documents being merged into a single PDF document.



To assemble two or more PDF documents into a single PDF document, you need a DDX document. A DDX document describes the PDF document that the Assembler service produces. That is, the DDX document instructs the Assembler service what actions to perform.

For the purpose of this discussion, assume that the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    <PDF result="out.pdf">
        <PDF source="map.pdf" />
        <PDF source="directions.pdf" />
    </PDF>
</DDX>
```

This DDX document merges two PDF documents named *map.pdf* and *directions.pdf* into a single PDF document.

Note: *To see a DDX document that disassembles a PDF document, see "Programmatically Disassembling PDF Documents" on page 932.*

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, seeAssembler Service and DDX Reference.*

Considerations when invoking Assembler service using web services

When you add headers and footers during the assembling of large documents, you may encounter an `OutOfMemory` error and the files will not be assembled. To reduce the chance of this problem occurring, add a `DDXProcessorSetting` element to your DDX document, as shown in the following example.

```
<DDXProcessorSetting name="checkpoint" value="2000" />
```

You can add this element as a child of the `DDX` element or as a child of a `PDF result` element. The default value for this setting is 0 (zero), which turns checkpointing off and the DDX behaves as if the `DDXProcessorSetting` element is not present. If you have encountered an `OutOfMemory` error, you may need to set the value to an integer, typically between 500 and 5000. A small checkpoint value results in more frequent checkpointing.

Summary of steps

To assemble a single PDF document from multiple PDF documents, perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference input PDF documents.
5. Set run-time options.
6. Assemble the input PDF documents.
7. Extract the results.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler client.

Reference an existing DDX document

A DDX document must be referenced to assemble a PDF document. For example, consider the DDX document that was introduced in this section. This DDX document instructs the Assembler service to merge two PDF documents into a single PDF document.

Reference input PDF documents

Reference input PDF documents that you want to pass to the Assembler service. For example, if you want to pass two input PDF documents named Map and Directions, you must pass the corresponding PDF files.

Both the map.pdf file and the directions.pdf file must be placed in a collection object. The name of the key must match the value of the PDF source attribute in the DDX document. It does not matter what the name of the PDF file is if the key and the source attribute in the DDX document match.

Note: *An* `AssemblerResult` *object, which contains a collection object, is returned if you invoke the* `invokeDDX` *operation. This operation is used when you pass two or more input PDF documents to the Assembler service. However, if you pass only one input PDF to the Assembler service and expect only one return document, invoke the* `invokeOneDocument` *operation. When invoking this operation, a single document is returned. For information about using this operation, see "Assembling Encrypted PDF Documents" on page 937.*

Set run-time options

You can set run-time options that control the behaviour of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered. For information about the run-time options that you can set, see the `AssemblerOptionSpec` class reference in AEM Forms API Reference.

Assemble the input PDF documents

After you create the service client, reference a DDX file, create a collection object that stores input PDF documents, and set run-time options, you can invoke the DDX operation. When using the DDX document specified in this section, the map.pdf and direction.pdf files are merged into one PDF document.

Extract the results

The Assembler service returns a `java.util.Map` object, which can be obtained from the `AssemblerResult` object, and that contains operation results. The returned `java.util.Map` object contains the resultant documents and any exceptions.

The following table summarizes some of the key values and object types that can be located in the returned `java.util.Map` object.

Key value	Object type	Description
documentName	`com.adobe.idp.Document`	Contains the resultant documents that are specified in a DDX resultant element
documentName	`Exception`	Contains any exception for the document
`OutputMapConstants.LOG_NAME`	`com.adobe.idp.Documen`	Contains the job log

See also

"Assemble PDF documents using the Java API" on page 928

"Assemble PDF documents using the web service API" on page 929

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Disassembling PDF Documents" on page 932

Assemble PDF documents using the Java API

Assemble a PDF document by using the Assembler Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

- Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference input PDF documents.

- Create a `java.util.Map` object that is used to store input PDF documents by using a `HashMap` constructor.
- For each input PDF document, create a `java.io.FileInputStream` object by using its constructor and passing the location of the input PDF document.
- For each input PDF document, create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the PDF document.
- For each input document, add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the PDF source element specified in the DDX document.
  - A `com.adobe.idp.Document` object (or `java.util.List` object that specifies multiple documents) that contains the source PDF document.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

6 Assemble the input PDF documents.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to be used
- A `java.util.Map` object that contains the input PDF files to be assembled
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including default font and job log level

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the results of the job and any exceptions that occurred.

7 Extract the results.

To obtain the newly created PDF document, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object. (You can use the PDF result element specified in the DDX document to get the document.)
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document.

Note: *If `LOG_LEVEL` was set to produce a log, you can extract the log by using the `AssemblerResult` object's `getJobLog` method.*

See also

"Programmatically Assembling PDF Documents" on page 925

"Quick Start (SOAP mode): Assembling a PDF document using the Java API" on page 24

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Assemble PDF documents using the web service API

Assemble PDF documents by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Reference input PDF documents.

- For each input PDF document, create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.
- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store input PDF documents.
- For each input PDF document, create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object. For example, if two input PDF documents are used, create two `MyMapOf_xsd_string_To_xsd_anyType_Item` objects.
- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the PDF source element specified in the DDX document. (Perform this task for each input PDF document.)
- Assign the `BLOB` object that stores the PDF document to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field. (Perform this task for each input PDF document.)
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object's `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object. (Perform this task for each input PDF document.)

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

6 Assemble the input PDF documents.

Invoke the `AssemblerServiceClient` object's `invoke` method and pass the following values:

- A `BLOB` object that represents the DDX document.

- The `mapItem` array that contains the input PDF documents. Its keys must match the names of the PDF source files, and its values must be the `BLOB` objects that correspond to those files.
- An `AssemblerOptionSpec` object that specifies run-time options.

The `invoke` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that may have occurred.

7 Extract the results.

To obtain the newly created PDF document, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the result PDF documents.
- Iterate through the `Map` object until you find the key that matches the name of the resultant document. Then cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` property. This returns an array of bytes that you can write out to a PDF file.

Note: *If `LOG_LEVEL` was set to produce a log, you can extract the log by getting the value of the `AssemblerResult` object's `jobLog` data member.*

See also

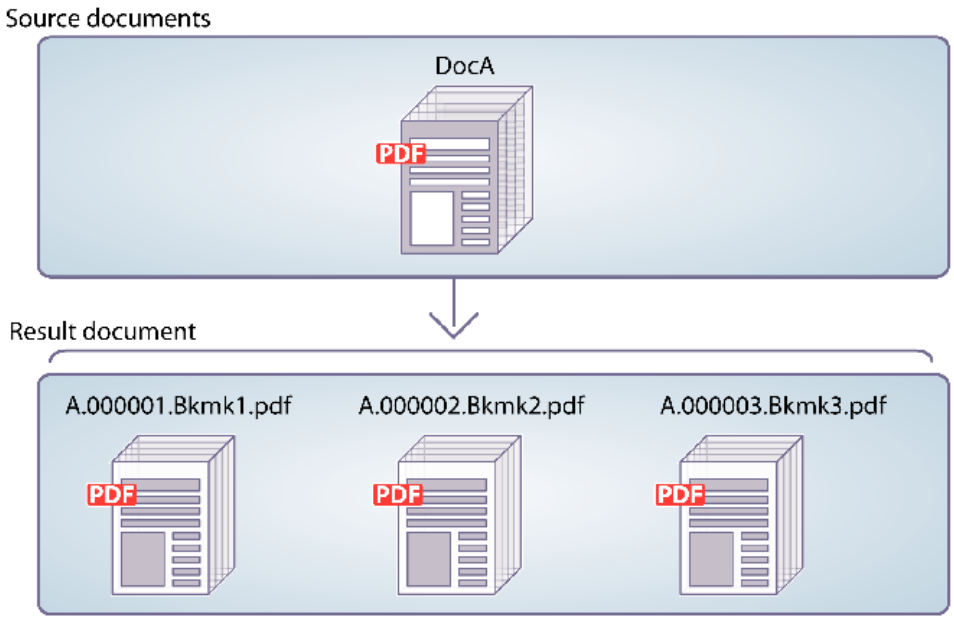
"Programmatically Assembling PDF Documents" on page 925

Quick Start (MTOM): Assembling a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

Programmatically Disassembling PDF Documents

You can disassemble a PDF document by passing it to the Assembler service. Typically, this task is useful when the PDF document was originally created from many individual documents, such as a collection of statements. In the following illustration, DocA is divided into multiple resultant documents, where the first level 1 bookmark on a page identifies the start of a new resultant document.



To disassemble a PDF document, ensure that the `PDFsFromBookmarks` element is located in the DDX document. The `PDFsFromBookmarks` element is a resultant element and can be only a child element of the `DDX` element. It does not have a `result` attribute because it can result in the generation of multiple documents.

The `PDFsFromBookmarks` element causes a single document to be generated for each level 1 bookmark in the source document.

For the purpose of this discussion, assume the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
     <PDFsFromBookmarks prefix="stmt">
    <PDF source="AssemblerResultPDF.pdf"/>
</PDFsFromBookmarks>
</DDX>
```

Before reading this section, it is recommended that you be familiar with assembling PDF documents by using the Assembler service. (See "Programmatically Assembling PDF Documents" on page 925.)

***Note:** When passing a single PDF document to the Assembler service and getting back a single document, you can invoke the `invokeOneDocument` operation. However, to disassemble a PDF document, use the `invokeDDX` operation because although one input PDF document is passed to the Assembler service, the Assembler service returns a collection object that contains one or more documents.*

***Note:** For more information about the Assembler service, see Services Reference for AEM Forms.*

***Note:** For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To disassemble a PDF document, perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference a PDF document to disassemble.
5. Set run-time options.
6. Disassemble the PDF document.
7. Save the disassembled PDF documents.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server that is not JBoss, you must replace adobe-utilities.jar and jbossall-client.jar with JAR files that are specific to the J2EE application server on which AEM Forms is deployed.

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to disassemble a PDF document. This DDX document must contain the `PDFsFromBookmarks` element.

Reference a PDF document to disassemble

To disassemble a PDF document, reference a PDF file that represents the PDF document to disassemble. When passed to the Assembler service, a separate PDF document is returned for each level 1 bookmark in the document.

Set run-time options

You can set run-time options that control the behaviour of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered.

Disassemble the PDF document

After you create the Assembler service client, reference the DDX document, reference a PDF document to disassemble, and set run-time options, you can disassemble a PDF document by invoking the `invokeDDX` method. Provided that the DDX document contains instructions to disassemble the PDF document, the Assembler service returns disassembled PDF documents within a collection object.

Save the disassembled PDF documents

All disassembled PDF documents are returned within a collection object. Iterate through the collection object and save each PDF document as a PDF file.

See also

"Disassemble a PDF document using the Java API" on page 934

"Disassemble a PDF document using the web service API" on page 935

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Disassemble a PDF document using the Java API

Disassemble a PDF document by using the Assembler Service API (Java):

1. Include project files.

   Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2. Create a PDF Assembler client.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Reference an existing DDX document.
   - Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Reference a PDF document to disassemble.
   - Create a `java.util.Map` object that is used to store input PDF documents by using a `HashMap` constructor.
   - Create a `java.io.FileInputStream` object by using its constructor and passing the location of the PDF document to disassemble.
   - Create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the PDF document to disassemble.
   - Add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
     - A string value that represents the key name. This value must match the value of the PDF source element specified in the DDX document.
     - A `com.adobe.idp.Document` object that contains the PDF document to disassemble.
5. Set run-time options.
   - Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
   - Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.
6. Disassemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to use
- A `java.util.Map` object that contains the PDF document to disassemble
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including the default font and the job log level

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the disassembled PDF documents and any exceptions that occurred.

7 Save the disassembled PDF documents.

To obtain the disassembled PDF documents, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document.

See also

"Programmatically Disassembling PDF Documents" on page 932

"Quick Start (SOAP mode): Disassembling a PDF document using the Java API" on page 27

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Disassemble a PDF document using the web service API

Disassemble a PDF document by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition when setting a service reference: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.

- Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
- Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
- Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the DDX document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Reference a PDF document to disassemble.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document. This `BLOB` object is passed to the `invokeOneDocument` as an argument.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field the contents of the byte array.
- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store the PDF to disassemble.
- Create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the PDF source element specified in the DDX document.
- Assign the `BLOB` object that stores the PDF document to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field.
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object' `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` field.

6 Disassemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the DDX document that disassembles the PDF document
- The `MyMapOf_xsd_string_To_xsd_anyType` object that contains the PDF document to disassemble
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeDDX` method returns an `AssemblerResult` object that contains the job results and any exceptions that occurred.

7 Save the disassembled PDF documents.

To obtain the newly created PDF documents, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the disassembled PDF documents.
- Iterate through the `Map` object to obtain each resultant document. Then, cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` property. This returns an array of bytes that you can write out to a PDF file.

See also

"Programmatically Disassembling PDF Documents" on page 932

Quick Start (MTOM): Disassembling a PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

Assembling Encrypted PDF Documents

You can encrypt a PDF document with a password by using the Assembler service. After a PDF document is encrypted with a password, a user must specify the password to view the PDF document in Adobe Reader or Acrobat. To encrypt a PDF document with a password, the DDX document must contain encryption element values that are required to encrypt a PDF document.

For the purpose of this discussion, assume that the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
       <PDF result="EncryptLoan.pdf" encryption="userProtect">
        <PDF source="inDoc" />
    </PDF>
    <PasswordEncryptionProfile name="userProtect" compatibilityLevel="Acrobat7">
        <OpenPassword>AdobeOpen</OpenPassword>
       </PasswordEncryptionProfile>
</DDX>
```

Within this DDX document, notice that the source attribute is assigned the value `inDoc`. In situations where only one input PDF document is passed to the Assembler service and one PDF document is returned, and you invoke the `invokeOneDocument` operation, assign the value `inDoc` to the PDF source attribute. When invoking the `invokeOneDocument` operation, the `inDoc` value is a predefined key that must be specified in the DDX document.

In contrast, when passing two or more input PDF documents to the Assembler service, you can invoke the `invokeDDX` operation. In this situation, assign the file name of the input PDF document to the `source` attribute.

The Encryption service does not have to be part of your AEM forms installation to encrypt a PDF document with a password. See "Encrypting and Decrypting PDF Documents" on page 806.

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To assemble an encrypted PDF document, perform the following steps:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference an unsecured PDF document.
5. Set run-time options.
6. Encrypt the document.
7. Save the encrypted PDF document.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server that AEM Forms is deployed on. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create an Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to assemble a PDF document. For example, consider the DDX document that was introduced in this section. To encrypt a PDF document, the DDX document must contain the `PasswordEncryptionProfile` element.

Reference an unsecured PDF document

An unsecured PDF document must be referenced and passed to the Assembler service to encrypt it. If you reference a PDF document that is already encrypted, an exception is thrown.

Set run-time options

You can set run-time options that control the behaviour of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered. For information about the run-time options that you can set, see the `AssemblerOptionSpec` class reference in AEM Forms API Reference.

Encrypt the document

After you create the Assembler service client, reference the DDX document that contains encryption information, reference an unsecured PDF document, and set run-time options, you can invoke the `invokeOneDocument` operation. Because only one input PDF document is being passed to the Assembler service (and one document is being returned), you can use the `invokeOneDocument` operation instead of the `invokeDDX` operation.

Save the encrypted PDF document

If only a single PDF document is being passed to the Assembler service, the Assembler service returns a single document instead of a collection object. That is, when invoking the `invokeOneDocument` operation, a single document is returned. Because the DDX document referenced in this section contains encryption information, the Assembler service returns a PDF document that is encrypted with a password.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Assemble an encrypted PDF document using the Java API

1 Include project files.

Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create an Assembler client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

- Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference an unsecured PDF document.

- Create a `java.io.FileInputStream` object by using its constructor and passing the location of an unsecured PDF document.
- Create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the PDF document. This `com.adobe.idp.Document` object is passed to the `invokeOneDocument` method.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

6 Encrypt the document.

Invoke the `AssemblerServiceClient` object's `invokeOneDocument` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the DDX document. Ensure that this DDX document contains the value `inDoc` for the PDF source element.
- A `com.adobe.idp.Document` object that contains the unsecured PDF document.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including default font and job log level.

The `invokeOneDocument` method returns a `com.adobe.idp.Document` object that contains a password-encrypted PDF document.

7 Save the encrypted PDF document.

- Create a `java.io.File` object and ensure that the file name extension is .pdf.
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file. Ensure that you use the `Document` object that the `invokeOneDocument` method returned.

See also

"Quick Start (SOAP mode): Assembling an encrypted PDF document using the Java API" on page 29

Assemble an encrypted PDF document using the web service API

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition when setting a service reference: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reference an unsecured PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document. This `BLOB` object is passed to the `invokeOneDocument` as an argument.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

6 Encrypt the document.

Invoke the `AssemblerServiceClient` object's `invokeOneDocument` method and pass the following values:

- A `BLOB` object that represents the DDX document
- A `BLOB` object that represents the unsecured PDF document
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeOneDocument` method returns a `BLOB` object that contains an encrypted PDF document.

7 Save the encrypted PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the encrypted PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `BLOB` object that the `invokeOneDocument` method returned. Populate the byte array by getting the value of the `BLOB` object's `MTOM` data member.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

Quick Start (MTOM): Assembling an encrypted PDF document using the web service API

"Invoking AEM Forms using MTOM" on page 529

Assembling Non-Interactive PDF Documents

You can assemble a non-interactive PDF document when using an interactive PDF form as input. That is, assume that you have a form that users can use to enter data into its fields. You can pass that form to the Assembler service, resulting in the Assembler service returning a PDF document that prevents users from entering data into its fields. This document is a non-interactive PDF form. For example, the following illustration shows a mortgage application that represents an interactive form.

For the purpose of this discussion, assume that the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
     <PDF result="out.pdf">
       <PDF source="inDoc"/>
       <NoXFA/>
     </PDF>
</DDX>
```

Within this DDX document, notice that the source attribute is assigned the value `inDoc`. In situations where only one input PDF document is passed to the Assembler service and one PDF document is returned, and you invoke the `invokeOneDocument` operation, assign the value `inDoc` to the PDF source attribute. When invoking the `invokeOneDocument` operation, the `inDoc` value is a predefined key that must be specified in the DDX document.

In contrast, when passing two or more input PDF documents to the Assembler service, you can invoke the `invokeDDX` operation. In this situation, assign the file name of the input PDF document to the `source` attribute.

This DDX document contains the `NoXFA` element, which instructs the Assembler service to return a non-interactive PDF document.

The Assembler service can assemble non-interactive PDF documents without the Output service being part of your AEM forms installation if the input PDF document is based on an Acrobat form or a static XFA form. However, if the input PDF document is a dynamic XFA form, the Output service must be part of your AEM forms installation. If the Output service is not part of your AEM forms installation when a dynamic XFA form is assembled, an exception is thrown. See "Creating Document Output Streams" on page 680.

Before reading this section, it is recommended that you be familiar with assembling PDF documents using the Assembler service. This section does not discuss concepts, such as creating a collection object that contains input documents or learning how to extract the results from the returned collection object. (See "Programmatically Assembling PDF Documents" on page 925.)

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To assemble a non-interactive PDF document, perform the following tasks:

1 Include project files.

2 Create a PDF Assembler client.

3 Reference an existing DDX document.

4 Reference an interactive PDF document.

5 Set run-time options.

6 Assemble the PDF document.

7 Save the non-interactive PDF document.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server that AEM Forms is deployed on.

Create an Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to assemble a PDF document. This DDX document must contain the `NoXFA` element, which instructs the Assembler service to return a non-interactive PDF document.

Reference an interactive PDF document

An interactive PDF document must be referenced and passed to the Assembler service to get back a non-interactive PDF document.

Set run-time options

You can set run-time options that control the behavior of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered.

Assemble the PDF document

After you create the Assembler service client, reference the DDX document, reference an interactive PDF document, and set run-time options, you can invoke the `invokeOneDocument` operation. Because only one input PDF document is passed to the Assembler service and a single document is returned, you can use the `invokeOneDocument` operation as opposed to the `invokeDDX` operation.

Save the non-interactive PDF document

If only a single PDF document is passed to the Assembler service, the Assembler service returns a single document instead of a collection object. That is, when invoking the `invokeOneDocument` operation, a single document is returned. Because the DDX document referenced in this section contains instructions to create a non-interactive PDF document, the Assembler service returns a non-interactive PDF document that can be saved as a PDF file.

See also

"Assemble a non-interactive PDF document using the Java API" on page 944

"Assemble a non-interactive PDF document using the web service API" on page 945

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Assemble a non-interactive PDF document using the Java API

Assemble a non-interactive PDF document by using the Assembler Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create an Assembler client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

- Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference an interactive PDF document.

- Create a `java.io.FileInputStream` object by using its constructor and passing the location of an interactive PDF document.
- Create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the PDF document. This `com.adobe.idp.Document` object is passed to the `invokeOneDocument` method.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

6 Assemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeOneDocument` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the DDX document. Ensure that this DDX document contains the value `inDoc` for the PDF source element.

- A `com.adobe.idp.Document` object that contains the interactive PDF document.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including default font and job log level.

The `invokeOneDocument` method returns a `com.adobe.idp.Document` object that contains a non-interactive PDF document.

7 Save the non-interactive PDF document.

- Create a `java.io.File` object and ensure that the file name extension is .pdf.
- Invoke the `Document` object's `copyToFile` method to copy the contents of the `Document` object to the file. Ensure that you use the `Document` object that the `invokeOneDocument` method returned.

For code examples, see these Assembler Service Quick Starts in "*Java API(SOAP) Quick Start (Code Examples)*" on page 2:

- "Quick Start (SOAP mode): Assembling a non-interactive PDF document using the Java API"

Assemble a non-interactive PDF document using the web service API

Assemble a non-interactive PDF document by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create an Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reference an interactive PDF document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document. This `BLOB` object is passed to the `invokeOneDocument` as an argument.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

6 Assemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeOneDocument` method and pass the following values:

- A `BLOB` object that represents the DDX document
- A `BLOB` object that represents the interactive PDF document
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeOneDocument` method returns a `BLOB` object that contains a non-interactive PDF document.

7 Save the non-interactive PDF document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the non-interactive PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `BLOB` object that the `invokeOneDocument` method returned. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

For code examples, see this Assembler Service Quick Start in *"Java API(SOAP) Quick Start (Code Examples)" on page 2*:

- "Quick Start (MTOM): Assembling a non-interactive PDF document using the web service API".

See also

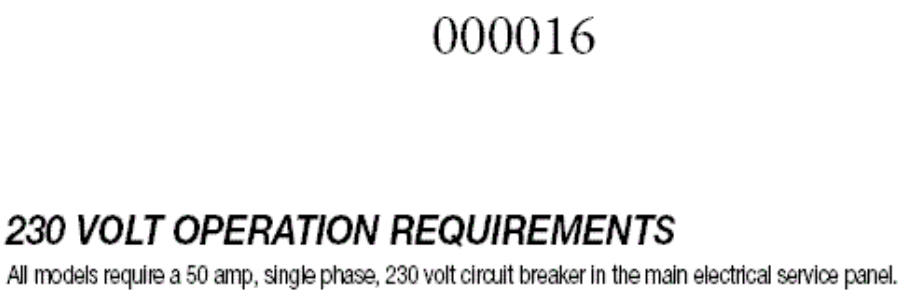
"Assembling Non-Interactive PDF Documents" on page 942

"Invoking AEM Forms using MTOM" on page 529

Assembling Documents Using Bates Numbering

You can assemble PDF documents that contain unique page identifiers by using Bates numbering. *Bates numbering* is a method of applying unique identifies to a batch of related documents. Each page in the document (or set of documents) is assigned a Bates number that uniquely identifies the page. For example, manufacturing documents that contain bill of material information and are associated with the production of an assembly can contain an identifier. A Bates number contains a sequentially incremented numeric value and an optional prefix and suffix. The prefix + numeric + suffix is referred to as a *bates pattern*.

The following illustration shows a PDF document that contains a unique identifier located in the document's header.



For the purpose of this discussion, the unique page identifier is placed in a document's header. Assume that the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
       <PDF result="out.pdf">
       <Header>
        <Center>
            <StyledText>
                <p font-size="20pt"><BatesNumber/></p>
            </StyledText>
        </Center>
    </Header>
          <PDF source="map.pdf" />
         <PDF source="directions.pdf" />
         </PDF>
</DDX>
```

This DDX document merges two PDF documents named *map.pdf* and *directions.pdf* into a single PDF document. The resultant PDF document contains a header that consists of a unique page identifier. For example, the document in the above illustration shows 000016.

Before reading this section, it is recommended that you be familiar with assembling PDF documents using the Assembler service. This section does not discuss the concepts, such as creating a collection object that contains input documents, or extracting the results from the returned collection object. (See "Programmatically Assembling PDF Documents" on page 925.)

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To assemble a PDF document that contains a unique page identifier (Bates numbering), perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference input PDF documents.
5. Set the initial Bates number value.
6. Assemble the input PDF documents.
7. Extract the results.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server on which AEM Forms is deployed. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to assemble a PDF document. For example, consider the DDX document that was introduced in this section. To assemble a PDF document that contains unique page identifiers, the DDX document must contain the `BatesNumber` element.

Reference input PDF documents

Input PDF documents must be referenced to assemble a PDF document. For example, the map.pdf and directions.pdf documents must be referenced to assemble these PDF documents into a single PDF document.

Set the initial Bates number value

You can set the initial Bates number value to meet your business requirements. For example, assume that it is a requirement to set the initial value to 000100. If you do not set the initial value, the value of the first page is 000000.

Assemble the input PDF documents

After you create the Assembler service client, reference the DDX document that contains `BatesNumber` element information, reference an input PDF document, and set run-time options, you can invoke the `invokeDDX` operation that results in the Assembler service assembling a PDF document that contains unique page identifiers.

Extract the results

The Assembler service returns a collection object that contains the job results. You can extract the resultant PDF document and any exceptions that are thrown. In this situation, an encrypted PDF document is located within the collection object.

Note: *A collection object is returned if you invoke the* `invokeDDX` *operation. This operation is used when passing two or more input PDF documents to the Assembler service. However if you pass only one input PDF document to the Assembler service, you should invoke the* `invokeOneDocument` *operation. For information about using this operation, see* *"Assembling Encrypted PDF Documents" on page 937.*

See also

"Assemble documents with Bates numbering using the Java API" on page 949

"Assemble documents with Bates numbering using the web service API" on page 950

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Assemble documents with Bates numbering using the Java API

Assemble a PDF document that uses unique page identifiers (Bates numbering) by using the Assembler Service API (Java):

1 Include project files.

   Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

   - Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference input PDF documents.

   - Create a `java.util.Map` object used to store input PDF documents by using a `HashMap` constructor.
   - For each input PDF document, create a `java.io.FileInputStream` object by using its constructor and passing the location of the input PDF document. In this situation, pass the location of an unsecured PDF document.
   - For each input PDF document, create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the PDF document.

- Add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the PDF source element specified in the DDX document. For example, the name of the PDF source file specified in the DDX document that is introduced in this section is Loan.pdf.
  - A `com.adobe.idp.Document` object that contains the unsecured PDF document.

5 Set the initial Bates number value.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set the initial Bates number by invoking the `AssemblerOptionSpec` object's `setFirstBatesNumber` and passing a numeric value that specifies the initial value.

6 Assemble the input PDF documents.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document.
- A `java.util.Map` object that contains the input unsecured PDF file.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including default font and job log level.

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains a password-encrypted PDF document.

7 Extract the results.

To obtain the newly created PDF document, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This action returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document.

See also

"Assembling Documents Using Bates Numbering" on page 947

"Quick Start (SOAP mode): Assembling a PDF document with bates numbering using the Java API" on page 32

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Assemble documents with Bates numbering using the web service API

Assemble a PDF document that uses unique page identifiers (Bates numbering) by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.

- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reference input PDF documents.

- For each input PDF document, create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.
- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store the input PDF documents.
- For each input PDF document, create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object. For example, if two input PDF documents are used, create two `MyMapOf_xsd_string_To_xsd_anyType_Item` objects.

- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the PDF source element specified in the DDX document. (Perform this task for each input PDF document.)
- Assign the `BLOB` object that stores the PDF document to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field. (Perform this task for each input PDF document.)
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object's `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object. (Perform this task for each input PDF document.)

5 Set the initial Bates number value.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set the initial Bates number by assigning a numeric value to the `firstBatesNumber` data member that belongs to the `AssemblerOptionSpec` object.

6 Assemble the input PDF documents.

Invoke the `AssemblerServiceClient` object's `invoke` method and pass the following values:

- A `BLOB` object that represents the DDX document.
- The `MyMapOf_xsd_string_To_xsd_anyType` object that contains the input PDF documents. Its keys must match the names of the PDF source files, and its values must be the `BLOB` objects that corresponds to those files.
- An `AssemblerOptionSpec` object that specifies run-time options.

The `invoke` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that occurred.

7 Extract the results.

To obtain the newly created PDF document, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the result PDF documents.
- Iterate through the `Map` object until you find the key that matches the name of the resultant document. Then cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` property. This returns an array of bytes that you can write out to a PDF file.

See also

"Assembling Documents Using Bates Numbering" on page 947

Quick Start (MTOM): Assembling a PDF document with bates numbering using the web service API

"Invoking AEM Forms using MTOM" on page 529

Determining Whether Documents Are PDF/A- Compliant

You can determine whether a PDF document is PDF/A-compliant by using the Assembler service. A PDF/A document exists as an archival format meant for long-term preservation of the document's content. The fonts are embedded within the document, and the file is uncompressed. As a result, a PDF/A document is typically larger than a standard PDF document. Also, a PDF/A document does not contain audio and video content.

The PDF/A-1 specification consists of two levels of conformance, namely A and B. The major difference between the two levels is the logical structure (accessibility) support, which is not required for conformance level B. Regardless of the conformance level, PDF/A-1 dictates that all fonts are embedded within the generated PDF/A document. At this time, only PDF/A-1b is supported in validation (and conversion).

For the purpose of this discussion, assume that the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
        <DocumentInformation source="Loan.pdf" result="Loan_result.xml">
        <PDFAValidation compliance="PDF/A-1b" resultLevel="Detailed"
ignoreUnusedResources="true" allowCertificationSignatures="true" />
    </DocumentInformation>
</DDX>
```

Within this DDX document, the `DocumentInformation` element instructs the Assembler service to return information about the input PDF document. Within the `DocumentInformation` element, the `PDFAValidation` element instructs the Assembler service to indicate whether the input PDF document is PDF/A-compliant.

The Assembler service returns information that specifies whether the input PDF document is PDF/A-compliant within an XML document that contains a `PDFAConformance` element. If the input PDF document is PDF/A-compliant, the value of the `PDFAConformance` element's `isCompliant` attribute is `true`. If the PDF document is not PDF/A-compliant, the value of the `PDFAConformance` element's `isCompliant` attribute is `false`.

Note: *Because the DDX document specified in this section contains a* `DocumentInformation` *element, the Assembler service returns XML data instead of a PDF document. That is, the Assembler service does not assemble or disassemble a PDF document; it returns information about the input PDF document within an XML document.*

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To determine whether a PDF document is PDF/A-compliant, perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference a PDF document used to determine PDF/A compliancy.
5. Set run-time options.
6. Retrieve information about the PDF document.
7. Save the returned XML document.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar

- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server that AEM Forms is deployed on. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to perform an Assembler service operation. To determine whether an input PDF document is PDF/A-compliant, ensure that the DDX document contains the `PDFAValidation` element within a `DocumentInformation` element. The `PDFAValidation` element instructs the Assembler service to return an XML document that specifies whether the input PDF document is PDF/A-compliant.

Reference a PDF document used to determine PDF/A compliancy

A PDF document must be referenced and passed to the Assembler service to determine whether the PDF document is PDF/A-compliant.

Set run-time options

You can set run-time options that control the behaviour of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered. For information about the run-time options that you can set, see the `AssemblerOptionSpec` class reference in AEM Forms API Reference.

Retrieve information about the PDF document

After you create the Assembler service client, reference the DDX document, reference an interactive PDF document, and set run-time options, you can invoke the `invokeDDX` operation. Because the DDX document contains the `DocumentInformation` element, the Assembler service returns XML data instead of a PDF document.

Save the returned XML document

The XML document that the Assembler service returns specifies whether the input PDF document is PDF/A-compliant. For example, if the input PDF document is not PDF/A-compliant, the Assembler service returns an XML document that contains the following element:

```
<PDFAConformance isCompliant="false" compliance="PDF/A-1b" resultLevel="Detailed"
ignoreUnusedResources="true" allowCertificationSignatures="true">
```

Save the XML document as an XML file so that you can open the file and view the results.

See also

"Determine whether a document is PDF/A compliant using the Java API" on page 955

"Determine whether a document is PDF/A compliant using the web service API" on page 956

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Determine whether a document is PDF/A compliant using the Java API

Determine whether a PDF document is PDF/A-compliant by using the Assembler Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

- Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file. To determine whether the PDF document is PDF/A-compliant, ensure that the DDX document contains the `PDFAValidation` element that is contained within a `DocumentInformation` element.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference a PDF document used to determine PDF/A compliancy.

- Create a `java.io.FileInputStream` object by using its constructor and passing the location of a PDF document that is used to determine PDF/A compliancy.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object that contains the PDF document.
- Create a `java.util.Map` object that is used to store the input PDF document by using a `HashMap` constructor.
- Add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the source element specified in the DDX document. For example, the value of the source element located in the DDX document that is introduced in this section is Loan.pdf.
  - A `com.adobe.idp.Document` object that contains the input PDF document.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

6 Retrieve information about the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to use
- A `java.util.Map` object that contains the input PDF file that is used to determine PDF/A compliancy
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains XML data that specifies whether the input PDF document is PDF/A-compliant.

7 Save the returned XML document.

To obtain XML data that specifies whether the input PDF document is a PDF/A document, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the XML document. Ensure that you save the XML data as an XML file.

See also

"Determining Whether Documents Are PDF/A- Compliant" on page 952

"Quick Start (SOAP mode): Determining whether a document is PDF/A compliant using the Java API" on page 37 (SOAP mode)

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Determine whether a document is PDF/A compliant using the web service API

Determine whether a PDF document is PDF/A-compliant by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.)
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reference a PDF document used to determine PDF/A compliancy.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.
- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store the PDF document.
- Create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the PDF source element specified in the DDX document.
- Assign the `BLOB` object that stores the PDF document to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field.
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object' `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object.

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

6 Retrieve information about the PDF document.

Invoke the `AssemblerServiceService` object's `invoke` method and pass the following values:

- A `BLOB` object that represents the DDX document.
- The `MyMapOf_xsd_string_To_xsd_anyType` object that contains the input PDF document. Its keys must match the names of the PDF source files, and its values must be the `BLOB` object that corresponds to the input PDF file.
- An `AssemblerOptionSpec` object that specifies run-time options.

The `invoke` method returns an `AssemblerResult` object that contains XML data that specifies whether the input PDF document is a PDF/A document.

7 Save the returned XML document.

To obtain XML data that specifies whether the input PDF document is a PDF/A document, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the XML data that specifies whether the input PDF document is a PDF/A document.
- Iterate through the `Map` object to obtain each resultant document. Then, cast that array member's value to a `BLOB`.
- Extract the binary data that represents the XML data by accessing its `BLOB` object's `MTOM` field. This field stores an array of bytes that you can write out to as a XML file.

See also

"Determining Whether Documents Are PDF/A- Compliant" on page 952

Quick Start (MTOM): Determining whether a document is PDF/A compliant using the web service API

"Invoking AEM Forms using MTOM" on page 529

Assembling PDF Documents with Bookmarks

You can assemble a PDF document that contains bookmarks. For example, assume that you have a PDF document that does not contain bookmarks and you want to modify it by providing bookmarks. Using the Assembler service, you can pass it a PDF document that does not contain bookmarks and get back a PDF document that contains bookmarks.

Bookmarks contain the following properties:

- A title that appears as text on the screen.
- An action that specifies what happens when a user clicks on the bookmark. The typical action for a bookmark is to move to another location in the current document or open another PDF document, although other actions can be specified.

For the purpose of this discussion, assume that the following DDX document is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
      <PDF result="FinalDoc.pdf">
         <PDF source="Loan.pdf">
            <Bookmarks source="doc2" />
         </PDF>
      </PDF>
</DDX>
```

Within this DDX document, notice that the source attribute is assigned the value `Loan.pdf`. This DDX document specifies that a single PDF document is passed to the Assembler service. When assembling a PDF document with bookmarks, you must specify a bookmark XML document that describes the bookmarks in the result document. To specify a bookmark XML document, ensure that the `Bookmarks` element is specified in your DDX document.

In this example DDX document, the `Bookmarks` element specifies `doc2` as the value. This value indicates that the input map passed to the Assembler service contains a key named `doc2`. The value of the `doc2` key is a `com.adobe.idp.Document` value that represents the bookmark XML document. (See "Bookmarks Language" in the Assembler Service and DDX Reference.)

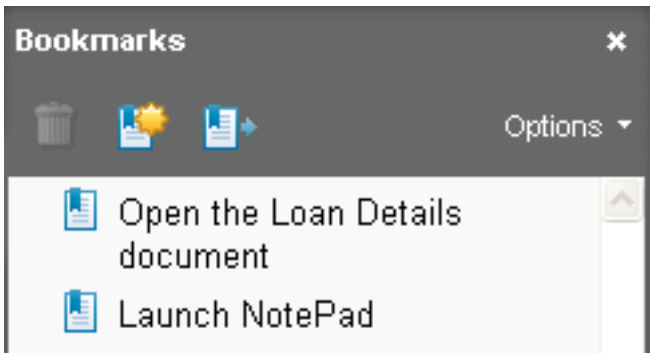
This topic uses the following XML bookmarks language to assemble a PDF document containing bookmarks.

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookmarks xmlns="http://ns.adobe.com/pdf/bookmarks" version="1.0">
      <Bookmark>
         <Action>
            <Launch NewWindow="true">
               <File Name="C:\Adobe\LoanDetails.pdf" />
            </Launch>
         </Action>
        <Title>Open the Loan document</Title>
      </Bookmark>
<Bookmark>
         <Action>
            <Launch>
               <Win Name="C:\WINDOWS\notepad.exe" />
            </Launch>
         </Action>
    <Title>Launch NotePad</Title>
      </Bookmark>
</Bookmarks>
```

Within this bookmark XML document, notice the Action element that defines the action that is performed when a user clicks the bookmark. Under the Action element is the Launch element that launches applications, such as NotePad and opens files, such as PDF files. To open a PDF file, you must use the File element that specifies the file to open. For example, in the bookmark XML file specified in this section, the name of the file that is opened is LoanDetails.pdf.

Important: *For complete details about supported actions, see "`Action` element" in the Assembler Service and DDX Reference.*

Given the DDX document specified in this section and bookmark XML file as input, the Assembler service assembles a PDF document that contains the following bookmarks.



When a user clicks on the *Open the Loan Details* bookmark, the LoanDetails.pdf is opened. Likewise, when the user clicks on the *Launch NotePad* bookmark, NotePad is started.

Before reading this section, it is recommended that you be familiar with assembling PDF documents using the Assembler service. This section does not discuss concepts, such as creating a collection object that contains input documents or learning how to extract the results from the returned collection object. (See "Programmatically Assembling PDF Documents" on page 925.)

Note: *For more information about the Assembler service, seeServices Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To assemble a PDF document that contains bookmarks, perform the following tasks:

1 Include project files.

2 Create a PDF Assembler client.

3 Reference an existing DDX document.

4 Reference a PDF document to which bookmarks are added.

5 Reference the bookmark XML document.

6 Add the PDF document and the bookmark XML document to a Map collection.

7 Set run-time options.

8 Assemble the PDF document.

9 Save the PDF document that contains bookmarks.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server that AEM Forms is deployed on. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to assemble a PDF document. This DDX document must contain the `Bookmarks` element, which instructs the Assembler service to assemble a PDF that contains bookmarks. (See the DDX document shown earlier in this section for an example.)

Reference a PDF document to which bookmarks are added

Reference a PDF document to which bookmarks are added. It does not matter whether the referenced PDF document already contains bookmarks. If the `Bookmarks` element is a child of the PDF source element, then the Bookmarks will replace those that already exist in the PDF source. However, if you want to keep the existing bookmarks, then ensure that `Bookmarks` is a sibling of the PDF source element. For example, consider the following example:

```
<PDF result="foo">
     <PDF source="inDoc"/>
     <Bookmarks source="doc2"/>
</PDF>
```

Reference the bookmark XML document

To assemble a PDF that contains new bookmarks, you must reference a bookmark XML document. The bookmark XML document is passed to the Assembler service within the Map collection object. (See the bookmark XML document shown earlier in this section for an example.)

Note: *See "Bookmarks Language" in the Assembler Service and DDX Reference.*

Add the PDF document and the bookmark XML document to a Map collection

You must add both the PDF document to which bookmarks are added and the bookmark XML document to the Map collection. Therefore the Map collection object contains two elements: a PDF document and the bookmark XML document.

Set run-time options

You can set run-time options that control the behavior of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered. For information about the run-time options that you can set, see the `AssemblerOptionSpec` class reference in AEM Forms API Reference.

Assemble the PDF document

To assemble a PDF document that contains new bookmarks, use the Assembler service's `invokeDDX` operation. The reason why you must use the `invokeDDX` operation as opposed to other Assembler service operations such as `invokeOneDocument` is because the Assembler service requires a bookmark XML document that is passed within the Map collection object. This object is a parameter of the `invokeDDX` operation.

Save the PDF document that contains bookmarks

You must extract the results from the returned map object and save the corresponding PDF document. (See "Extract the results" in "Programmatically Assembling PDF Documents" on page 925.)

See also

"Assemble PDF documents with bookmarks using the Java API" on page 961

"Assemble PDF documents with bookmarks using the web service API" on page 963

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Assemble PDF documents with bookmarks using the Java API

Assemble a PDF document with bookmarks by using the Assembler Service API (Java):

1 Include project files.

   Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

   - Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
   - Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

- Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference a PDF document to which bookmarks are added.

- Create a `java.io.FileInputStream` object by using its constructor and passing the location of the PDF document.
- Create a `com.adobe.idp.Document` object by using its constructor and pass the `java.io.FileInputStream` object that contains the PDF document.

5 Reference the bookmark XML document.

- Create a `java.io.FileInputStream` object by using its constructor and passing the location of the XML file that represents the bookmark XML document.
- Create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the PDF document.

6 Add the PDF document and the bookmark XML document to a Map collection.

- Create a `java.util.Map` object that is used to store both the input PDF document and the bookmark XML document.
- Add the input PDF document by invoking the `java.util.Map` object's `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the PDF source element specified in the DDX document.
  - A `com.adobe.idp.Document` object that contains the input PDF document.
- Add the bookmark XML document by invoking the `java.util.Map` object's `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the Bookmarks source element specified in the DDX document.
  - A `com.adobe.idp.Document` object that contains the bookmark XML document.

7 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

8 Assemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to be used
- A `java.util.Map` object that contains both the input PDF document and the bookmark XML document.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including default font and job log level

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the results of the job and any exceptions that occurred.

9 Save the PDF document that contains bookmarks.

To obtain the newly created PDF document, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object. (You can use the PDF result element specified in the DDX document to get the document.)
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document.

See also

"Assembling PDF Documents with Bookmarks" on page 958

"Quick Start (SOAP mode): Assembling PDF documents with bookmarks using the Java API" on page 43

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Assemble PDF documents with bookmarks using the web service API

Assemble a PDF document with bookmarks by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

4 Reference a PDF document to which bookmarks are added.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

5 Reference the bookmark XML document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the bookmark XML document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.

6 Add the PDF document and the bookmark XML document to a Map collection.

- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store the input PDF documents and the bookmark XML document.
- For each input PDF document and the bookmark XML document , create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the PDF source element specified in the DDX document.
- Assign the `BLOB` object that stores the PDF document to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field.
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object's `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object. (Perform this task for each input PDF document and the bookmark XML document.)

7 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.

- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

8 Assemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the DDX document
- The `MyMapOf_xsd_string_To_xsd_anyType` array that contains the input documents
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeDDX` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that may have occurred.

9 Save the PDF document that contains bookmarks.

To obtain the newly created PDF document, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the result PDF documents.
- Iterate through the `Map` object until you find the key that matches the name of the resultant document. Then cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` field. This returns an array of bytes that you can write out to a PDF file.

See also

"Assembling PDF Documents with Bookmarks" on page 958

Quick Start (MTOM): Assembling PDF documents with bookmarks using the web service API

"Invoking AEM Forms using MTOM" on page 529

Validating DDX Documents

You can programmatically validate a DDX document that is used by the Assembler service. That is, using the Assembler service API, you can determine whether or not a DDX document is valid. For example, if you upgraded from a previous AEM Forms version and you want to ensure that your DDX document is valid, you can validate it using the Assembler service API.

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To validate a DDX document, perform the following tasks:

1 Include project files.

2 Create an Assembler client.

3 Reference an existing DDX document.

4 Set run-time options to validate the DDX document.

5 Perform the validation.

6 Save the validation results in a log file.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

if AEM Forms is deployed on a supported J2EE application server other than JBoss, you must replace the adobe-utilities.jar and jbossall-client.jar files with JAR files that are specific to the J2EE application server that AEM Forms is deployed on.

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, you must create an Assembler service client.

Reference an existing DDX document

To validate a DDX document, you must reference an existing DDX document.

Set run-time options to validate the DDX document

When validating a DDX document, you must set specific run-time options that instruct the Assembler service to validate the DDX document as opposed to executing it. Also, you can increase the amount of information that the Assembler service writes to the log file.

Perform the validation

After you create the Assembler service client, reference the DDX document, and set run-time options, you can invoke the `invokeDDX` operation to validate the DDX document. When validating the DDX document, you can pass `null` as the map parameter (this parameter usually stores PDF documents that the Assembler requires to perform the operation(s) specified in the DDX document).

If validation fails, an exception is thrown and the log file contains details that explains why the DDX document is invalid can be obtained from the `OperationException` instance. Once past the basic XML parsing and schema checking, then the validation against the DDX specification is performed. All errors that are located in the DDX document are specified in the log.

Save the validation results in a log file

The Assembler service returns the validation results that you can write to a XML log file. The amount of detail that the Assembler service writes to the log file depends on the run-time option that you set.

See also

"Validate a DDX document using the Java API" on page 967

"Validate a DDX document using the web service API" on page 968

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Validate a DDX document using the Java API

Validate a DDX document by using the Assembler Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

- Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set run-time options to validate the DDX document.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set the run-time option that instructs the Assembler service to validate the DDX document by invoking the `AssemblerOptionSpec` object's setValidateOnly method and passing `true`.
- Set the amount of information that the Assembler service writes to the log file by invoking the `AssemblerOptionSpec` object's `getLogLevel` method and passing a string value meets your requirements. When validating a DDX document, you want more information written to the log file that will assist in the validation process. As a result, you can pass the value `FINE` or `FINER`.

5 Perform the validation.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the DDX document.
- The value `null` for the java.io.Map object that usually stores PDF documents.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options.

The `invokeDDX` method returns an `AssemblerResult` object that contains information that specifies whether the DDX document is valid.

6 Save the validation results in a log file.

- Create a `java.io.File` object and ensure that the file name extension is .xml.
- Invoke the `AssemblerResult` object's `getJobLog` method. This method returns a `com.adobe.idp.Document` instance that contains validation information.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to copy the contents of the `com.adobe.idp.Document` object to the file.

Note: *If the DDX document is invalid, an `OperationException` is thrown. Within the catch statement, you can invoke the `OperationException` object's `getJobLog` method.*

See also

"Validating DDX Documents" on page 965

"Quick Start (SOAP mode): Validating DDX documents using the Java API" on page 40 (SOAP mode)

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Validate a DDX document using the web service API

Validate a DDX document by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace localhost with the IP address of the forms server.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Set run-time options to validate the DDX document.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set the run-time option that instructs the Assembler service to validate the DDX document by assigning the value true to the `AssemblerOptionSpec` object's `validateOnly` data member.
- Set the amount of information that the Assembler service writes to the log file by assigning a string value to the `AssemblerOptionSpec` object's `logLevel` data member. method When validating a DDX document, you want more information written to the log file that will assist in the validation process. As a result, you can specify the value `FINE` or `FINER`. For information about the run-time options that you can set, see the `AssemblerOptionSpec` class reference in AEM Forms API Reference.

5 Perform the validation.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the DDX document.
- The value `null` for the `Map` object that usually stores PDF documents.
- An `AssemblerOptionSpec` object that specifies run-time options.

The `invokeDDX` method returns an `AssemblerResult` object that contains information that specifies whether the DDX document is valid.

6 Save the validation results in a log file.

- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the log file and the mode to open the file in. Ensure that the file name extension is .xml.
- Create a `BLOB` object that stores log information by getting the value of the `AssemblerResult` object's `jobLog` data member.
- Create a byte array that stores the content of the `BLOB` object. Populate the byte array by getting the value of the `BLOB` object's `MTOM` field.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

Note: *If the DDX document is invalid, an `OperationException` is thrown. Within the catch statement, you can get the value of the `OperationException` object's `jobLog` member.*

See also

"Validating DDX Documents" on page 965

Quick Start (MTOM): Validating DDX documents using the web service API

"Invoking AEM Forms using MTOM" on page 529

Dynamically Creating DDX Documents

You can dynamically create a DDX document that can be used to perform an Assembler operation. Dynamically creating a DDX document enables you to use values in the DDX document that are obtained during run-time. To dynamically create a DDX document, use classes that belong to the programming language that you are using. For example, if you are developing your client application using Java, use classes that belong to the `org.w3c.dom.*` package. Likewise, if you are using Microsoft .NET, use classes that belong to the `System.Xml` namespace.

Before you can pass the DDX document to the Assembler service, convert the XML from an `org.w3c.dom.Document` instance to a `com.adobe.idp.Document` instance. If you are using web services, convert the XML from the data type used to create the XML(for example, `XmlDocument`) to a `BLOB` instance.

For this discussion, assume that the following DDX document is dynamically created.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
     <PDFsFromBookmarks prefix="stmt">
    <PDF source="AssemblerResultPDF.pdf"/>
</PDFsFromBookmarks>
</DDX>
```

This DDX document disassembles a PDF document. It is recommended that you be familiar with disassembling PDF documents.

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To disassemble a PDF document by using a dynamically created DDX document, perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Create the DDX document.
4. Convert the DDX document.
5. Set run-time options.
6. Disassemble the PDF document.
7. Save the disassembled PDF documents.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, create an Assembler service client.

Create the DDX document

Create a DDX document using the programming language that you are using. To create a DDX document that disassembles a PDF document, ensure that it contains the `PDFsFromBookmarks` element. Convert the data type used to create the DDX document to a `com.adobe.idp.Document` instance if you are using the Java API. If you are using web services, convert the data type to a `BLOB` instance.

Convert the DDX document

A DDX document that is created by using `org.w3c.dom` classes must be converted to a `com.adobe.idp.Document` object. To perform this task when using the Java API, use Java XML transform classes. If you are using web services, convert the DDX document to a `BLOB` object.

Reference a PDF document to disassemble

To disassemble a PDF document, reference a PDF file that represents the PDF document to disassemble. When passed to the Assembler service, a separate PDF document is returned for each level 1 bookmark in the document.

Set run-time options

You can set run-time options that control the behavior of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered. To set run-time options, you use an `AssemblerOptionSpec` object.

Disassemble the PDF document

Disassemble the PDF document by invoking the `invokeDDX` operation. Pass the DDX document that was dynamically created. The Assembler service returns disassembled PDF documents within a collection object.

Save the disassembled PDF documents

All disassembled PDF documents are returned within a collection object. Iterate through the collection object and save each PDF document as a PDF file.

See also

"Dynamically create a DDX document using the Java API" on page 971

"Dynamically create a DDX document using the web service API" on page 974

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

"Programmatically Disassembling PDF Documents" on page 932

Dynamically create a DDX document using the Java API

Dynamically create a DDX document and disassemble a PDF document by using the Assembler Service API (Java):

1 Include project files.

   Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Create the DDX document.

- Create a Java `DocumentBuilderFactory` object by calling the `DocumentBuilderFactory` class' `newInstance` method.
- Create a Java `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.
- Call the `DocumentBuilder` object's `newDocument` method to instantiate a `org.w3c.dom.Document` object.
- Create the DDX document's root element by invoking the `org.w3c.dom.Document` object's `createElement` method. This method creates an `Element` object that represents the root element. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, set a value for the child element by calling its `setAttribute` method. Finally, append the element to the header element by calling the header element's `appendChild` method, and pass the child element object as an argument. The following lines of code show this application logic:

```
Element root = (Element)document.createElement("DDX");
root.setAttribute("xmlns","http://ns.adobe.com/DDX/1.0/");
document.appendChild(root);
```

- Create the `PDFsFromBookmarks` element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Set a value for the `PDFsFromBookmarks` element by calling its `setAttribute` method. Append the `PDFsFromBookmarks` element to the `DDX` element by calling the DDX element's `appendChild` method. Pass the `PDFsFromBookmarks` element object as an argument. The following lines of code show this application logic:

```
Element PDFsFromBookmarks = (Element)document.createElement("PDFsFromBookmarks");
PDFsFromBookmarks.setAttribute("prefix","stmt");
root.appendChild(PDFsFromBookmarks);
```

- Create a `PDF` element by calling the `Document` object's `createElement` method. Pass a string value that represents the element's name. Cast the return value to `Element`. Set a value for the `PDF` element by calling its `setAttribute` method. Append the `PDF` element to the `PDFsFromBookmarks` element by calling the `PDFsFromBookmarks` element's `appendChild` method. Pass the `PDF` element object as an argument. The following lines of code shows this application logic:

```
Element PDF = (Element)document.createElement("PDF");
PDF.setAttribute("source","AssemblerResultPDF.pdf");
PDFsFromBookmarks.appendChild(PDF);
```

4 Convert the DDX document.

- Create a `javax.xml.transform.Transformer` object by invoking the `javax.xml.transform.Transformer` object's static `newInstance` method.
- Create a `Transformer` object by invoking the `TransformerFactory` object's `newTransformer` method.
- Create a `ByteArrayOutputStream` object by using its constructor.
- Create a `javax.xml.transform.dom.DOMSource` object by using its constructor. Pass the `org.w3c.dom.Document` object that represents the DDX document.
- Create a `javax.xml.transform.dom.DOMSource` object by using its constructor and passing the `ByteArrayOutputStream` object.
- Populate the Java `ByteArrayOutputStream` object by invoking the `javax.xml.transform.Transformer` object's `transform` method. Pass the `javax.xml.transform.dom.DOMSource` and the `javax.xml.transform.stream.StreamResult` objects.
- Create a byte array and allocate the size of the `ByteArrayOutputStream` object to the byte array.

- Populate the byte array by invoking the `ByteArrayOutputStream` object's `toByteArray` method.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the byte array.

5 Reference a PDF document to disassemble.

- Create a `java.util.Map` object that is used to store input PDF documents by using a `HashMap` constructor.
- Create a `java.io.FileInputStream` object by using its constructor and passing the location of the PDF document to disassemble.
- Create a `com.adobe.idp.Document` object. Pass the `java.io.FileInputStream` object that contains the PDF document to disassemble.
- Add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the PDF source element specified in the DDX document. (In the DDX document that is dynamically created, the value is `AssemblerResultPDF.pdf`.)
  - A `com.adobe.idp.Document` object that contains the PDF document to disassemble.

6 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

7 Disassemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the dynamically created DDX document
- A `java.util.Map` object that contains the PDF document to disassemble
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including the default font and the job log level

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the disassembled PDF documents and any exceptions that occurred.

8 Save the disassembled PDF documents.

To obtain the disassembled PDF documents, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This method returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF document.

See also

"Dynamically Creating DDX Documents" on page 969

"Quick Start (SOAP mode): Dynamically creating a DDX document using the Java API" on page 46

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Dynamically create a DDX document using the web service API

Dynamically create a DDX document and disassemble a PDF document by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition when setting a service reference: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Create the DDX document.

- Create a `System.Xml.XmlElement` object by using its constructor.
- Create the DDX document's root element by invoking the `XmlElement` object's `CreateElement` method. This method creates an `Element` object that represents the root element. Pass a string value representing the name of the element to the `CreateElement` method. Set a value for the DDX element by calling its `SetAttribute` method. Finally, append the element to the DDX document by calling the `XmlElement` object's `AppendChild` method. Pass the DDX object as an argument. The following lines of code show this application logic:

```
System.Xml.XmlElement root = ddx.CreateElement("DDX");
root.SetAttribute("xmlns", "http://ns.adobe.com/DDX/1.0/");
ddx.AppendChild(root);
```

- Create the DDX document's `PDFsFromBookmarks` element by calling the `XmlElement` object's `CreateElement` method. Pass a string value representing the name of the element to the `CreateElement` method. Next, set a value for the element by calling its `SetAttribute` method. Append the `PDFsFromBookmarks` element to the root element by calling the `DDX` element's `AppendChild` method. Pass the `PDFsFromBookmarks` element object as an argument. The following lines of code show this application logic:

```
XmlElement PDFsFromBookmarks = ddx.CreateElement("PDFsFromBookmarks");
PDFsFromBookmarks.SetAttribute("prefix", "stmt");
root.AppendChild(PDFsFromBookmarks);
```

- Create the DDX document's `PDF` element by calling the `XmlElement` object's `CreateElement` method. Pass a string value representing the name of the element to the `CreateElement` method. Next, set a value for the child element by calling its `SetAttribute` method. Append the `PDF` element to the `PDFsFromBookmarks` element by calling the `PDFsFromBookmarks` element's `AppendChild` method. Pass the `PDF` element object as an argument. The following lines of code shows this application logic:

```
XmlElement PDF = ddx.CreateElement("PDF");
PDF.SetAttribute("source", "AssemblerResultPDF.pdf");
PDFsFromBookmarks.AppendChild(PDF);
```

4 Convert the DDX document.

- Create a `System.IO.MemoryStream` object by using its constructor.
- Populate the `MemoryStream` object with the DDX document by using the `XmlElement` object that represents the DDX document. Invoke the `XmlElement` object's `Save` method and pass the `MemoryStream` object.
- Create a byte array and populate it with data located in the `MemoryStream` object. The following code shows this application logic:

```
int bufLen = Convert.ToInt32(stream.Length);
byte[] byteArray = new byte[bufLen];
stream.Position = 0;
int count = stream.Read(byteArray, 0, bufLen);
```

- Create a `BLOB` object. Assign the byte array to the `BLOB` object's `MTOM` field.

5 Reference a PDF document to disassemble.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input PDF document. This `BLOB` object is passed to the `invokeOneDocument` as an argument.
- Create a `System.IO.FileStream` object by invoking its constructor. Pass a string value that represents the file location of the input PDF document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property the contents of the byte array.

6 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

7 Disassemble the PDF document.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the dynamically created DDX document
- The `mapItem` array that contains the input PDF document
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeDDX` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that occurred.

8 Save the disassembled PDF documents.

To obtain the newly created PDF documents, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the disassembled PDF documents.
- Iterate through the `Map` object to obtain each resultant document. Then, cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` property. This returns an array of bytes that you can write out to a PDF file.

See also

"Dynamically Creating DDX Documents" on page 969

Quick Start (MTOM mode): Dynamically creating a DDX document using the web service API

Quick Start (SwaRef mode): Dynamically creating a DDX document using the web service API

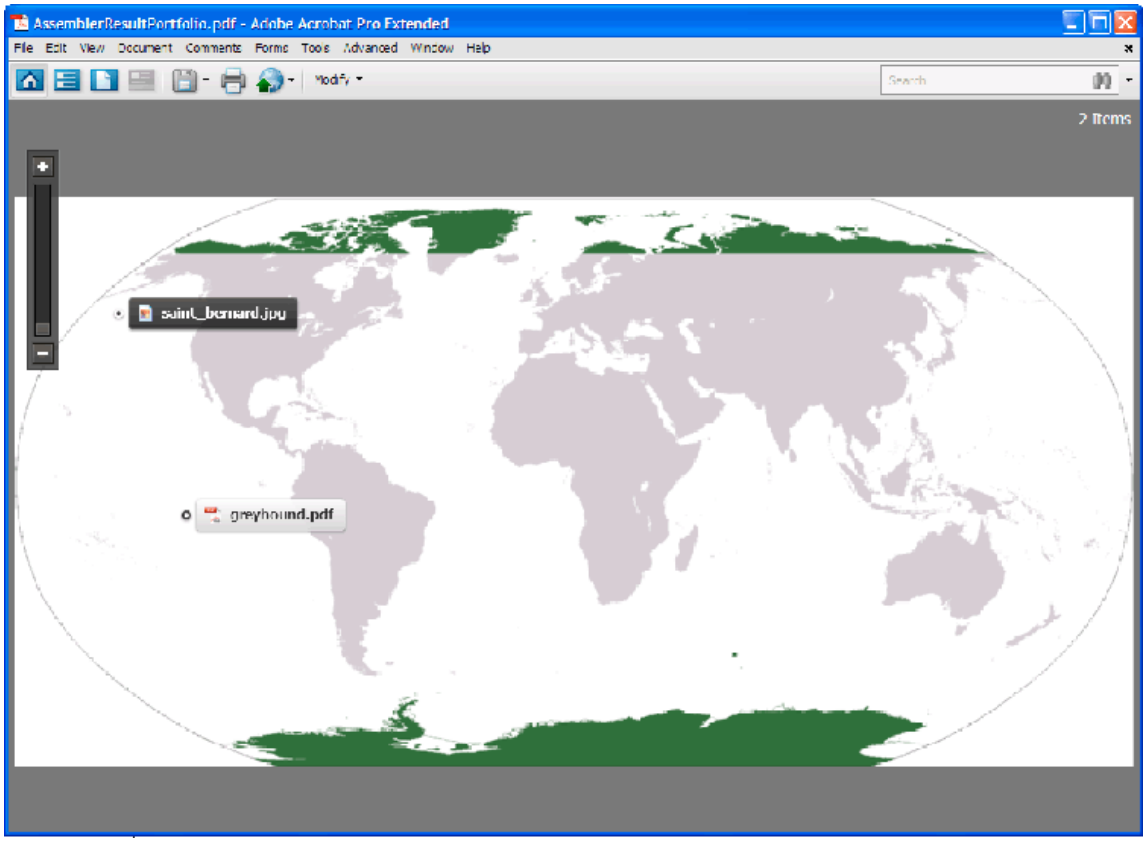
"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Assembling PDF Portfolios

You can assemble a PDF Portfolio using the Assembler Java and web service API. A portfolio can combine several documents of various types, including word file, image files (for example, a jpeg file), and PDF documents. The layout of the portfolio can be set to different styles like the *Grid with Preview*, the *On an Image* layout or even *Revolve*.

The following illustration is a screenshot of a portfolio with *On an Image* style layout.



Creating a PDF Portfolio serves as a paperless alternative to passing a collection of documents. Using AEM Forms you can create portfolios by invoking the Assembler service with a structured DDX document. The following DDX document is an example of a DDX document that creates a PDF Portfolio.

```
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
    <PDF result="portfolio1.pdf">
        <Portfolio>
            <Navigator source="myNavigator">
                <Resource name="navigator/image.xxx" source="myImage.png"/>
            </Navigator>
        </Portfolio>
        <PackageFiles source="dog1"  >
            <FieldData name="X">72</FieldData>
            <FieldData name="Y">72</FieldData>
            <File filename="saint_bernard.jpg" mimetype="image/jpeg"/>
        </PackageFiles>
        <PackageFiles source="dog2"  >
            <FieldData name="X">120</FieldData>
            <FieldData name="Y">216</FieldData>
            <File filename="greyhound.pdf"/>
        </PackageFiles>
    </PDF>
</DDX>
```

The DXX document must contain a `Portfolio` tag with a nested `Navigator` tag. Note the tag `<Resource name="navigator/image.xxx" source="myImage.png"/>` is only necessary if `myNavigator` is assigned as the onImage layout navigator: `AdobeOnImage.nav`. This tag allows the Assembler service to select the image to use as the portfolio background. Include `PackageFiles` and `File` tags to define the filename and MIME type of the packaged file.

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, seeAssembler Service and DDX Reference.*

Summary of steps

To create a PDF Portfolio, perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference the required documents.
5. Set run-time options.
6. Assemble the portfolio.
7. Save the assembled portfolio.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to assemble a PDF Portfolio. This DDX document must contain the `Portfolio`, `Navigator` and, `PackageFiles` elements.

Reference the required documents

To assemble a PDF Portfolio, reference all files that represents the documents to assemble. For example, pass all image files that are specified in the DDX document to the Assembler service. Notice that these files are referenced in the DDX document specified in this section: *myImage.png* and *saint_bernard.jpg*.

When assembling a PDF Portfolio, pass a NAV file (a navigator file) to the Assembler service. The NAV file that you pass to the Assembler service depends upon what type of PDF Portfolio to create. For example, to create an *On an Image* layout, pass the AdobeOnImage.nav file. You can locate NAV files in the following folder:

```
<Install folder>\Acrobat 9.0\Acrobat\Navigators
```

Copy the NAV file from the Acrobat 9 (or later) installation directory. Place the NAV file in a location where your client application can access it. All files are passed to the Assembler service within a Map collection object.

Note: *The quick starts that are associated with Assembling PDF Portfolios use AdobeOnImage.nav.*

Set run-time options

You can set run-time options that control the behavior of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered.

Assemble the portfolio

To assemble a PDF Portfolio, you call the `invokeDDX` operation. The Assembler service returns the PDF Portfolio within a collection object.

Save the assembled portfolio

A PDF Portfolio is returned within a collection object. Iterate through the collection object and save PDF Portfolio as a PDF file.

See also

"Assemble a PDF Portfolio using the Java API" on page 979

"Assemble a PDF Portfolio using the web service API" on page 980

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

Assemble a PDF Portfolio using the Java API

Assemble a PDF Portfolio by using the Assembler Service API (Java):

1 Include project files.

   Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2 Create a PDF Assembler client.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference an existing DDX document.

   - Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Reference the required documents.

   - Create a `java.util.Map` object that is used to store input PDF documents by using a `HashMap` constructor.
   - Create a `java.io.FileInputStream` object by using its constructor. Pass the location of the required NAV file (repeat this task for each file required to create a portfolio).

- Create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the NAV file (repeat this task for each file required to create a portfolio).
- Add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
  - A string value that represents the key name. This value must match the value of the source element specified in the DDX document. (repeat this task for each file required to create a portfolio).
  - A `com.adobe.idp.Document` object that contains the PDF document. (repeat this task for each file required to create a portfolio).

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.

6 Assemble the portfolio.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to use
- A `java.util.Map` object that contains the files required to build a PDF Portfolio.
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the runtime options, including the default font and the job log level

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the assembled PDF Portfolio and any exceptions that occurred.

7 Save the assembled portfolio.

To obtain the PDF Portfolio, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This method returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the PDF Portfolio.

See also

"Assembling PDF Portfolios" on page 976

"Quick Start (SOAP mode): Assembling PDF Portfolios using the Java API" on page 51

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Assemble a PDF Portfolio using the web service API

Assemble a PDF Portfolio by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition when setting a service reference: `http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AssemblerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AssemblerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Reference the required documents.

- For each input file, create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input file.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input file and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.
- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store input files required to create a PDF Portfolio.
- For each input file, create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object.

- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the element specified in the DDX document. (Perform this task for each input file.)
- Assign the `BLOB` object that stores the input file to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field. (Perform this task for each input PDF document.)
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object's `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object. (Perform this task for each input PDF document.)

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

6 Assemble the portfolio.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the DDX document
- The `MyMapOf_xsd_string_To_xsd_anyType` object that contains the required files
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeDDX` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that occurred.

7 Save the assembled portfolio.

To obtain the newly created PDF Portfolio, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the resultant PDF documents.
- Iterate through the `Map` object to obtain each resultant document. Then, cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` property. This returns an array of bytes that you can write out to a PDF file.

See also

"Assembling PDF Portfolios" on page 976

Quick Start (MTOM): Assembling PDF Portfolios using the web service API

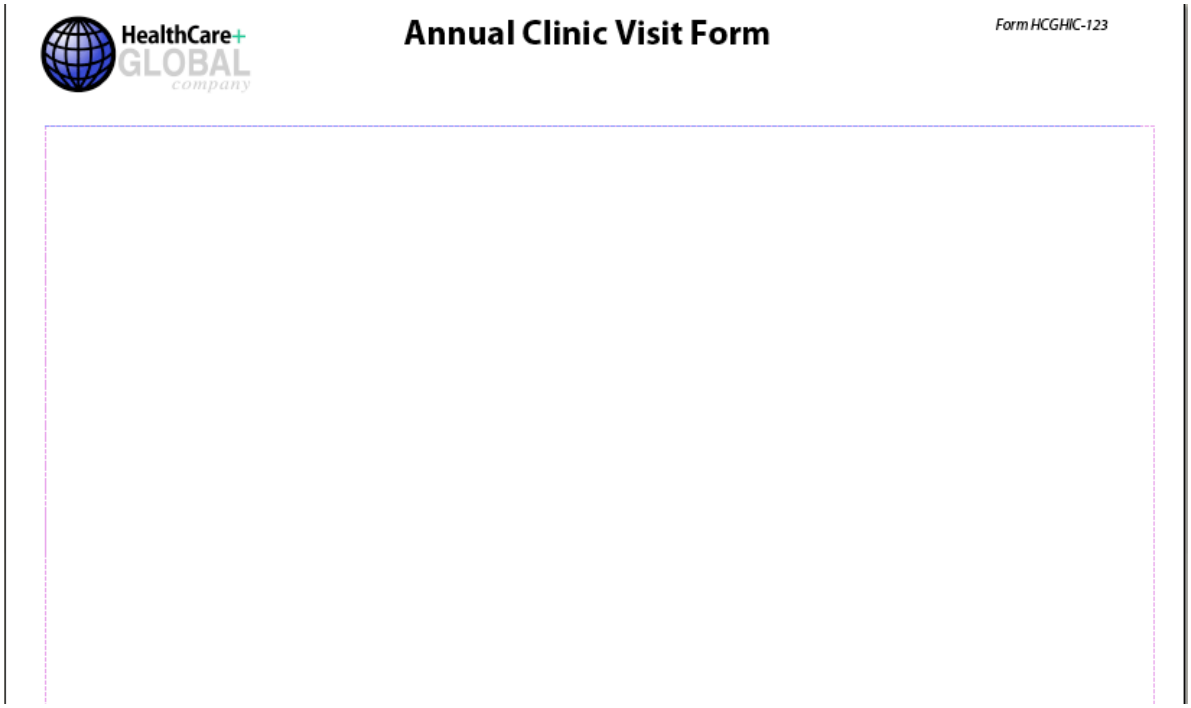
Quick Start (SwaRef mode): Assembling PDF Portfolios using the web service API

"Invoking AEM Forms using MTOM" on page 529

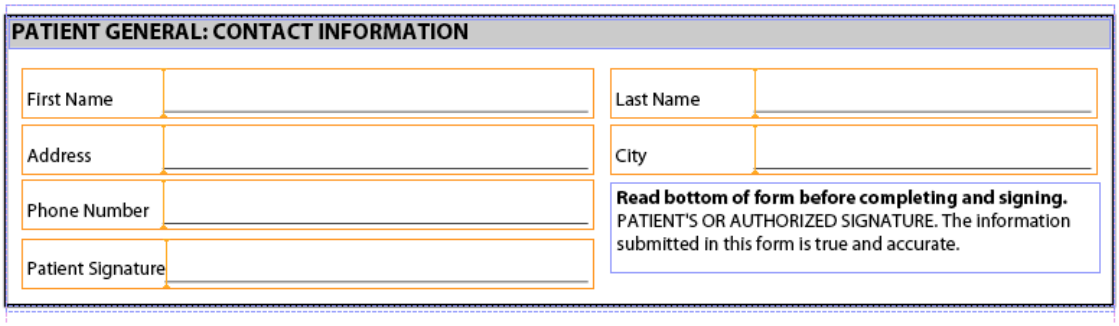
"Invoking AEM Forms using SwaRef" on page 531

Assembling Multiple XDP Fragments

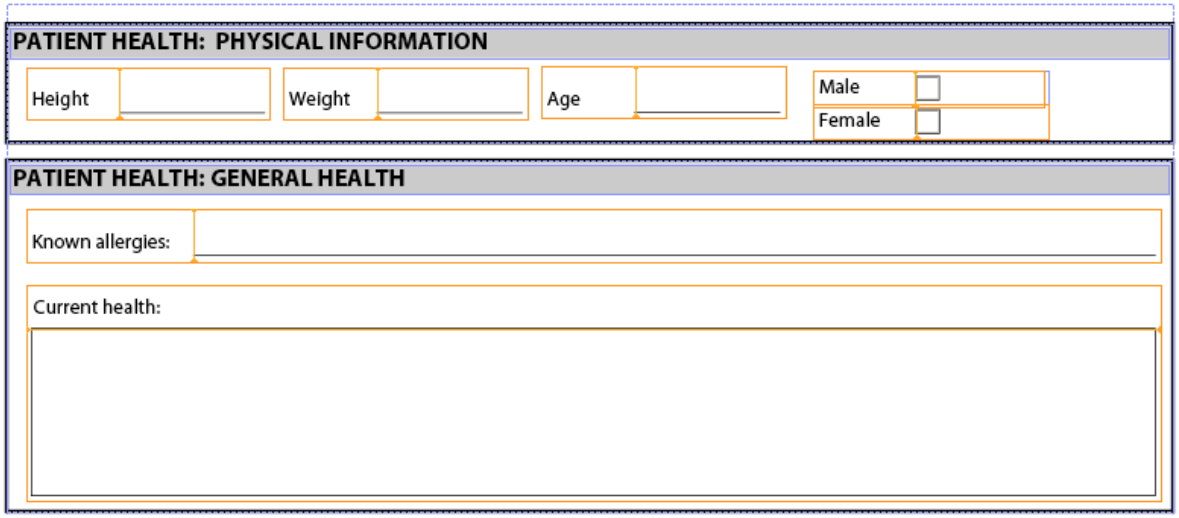
You can assemble multiple XDP fragments into a single XDP document. For example, consider XDP fragments where each XDP file contains one or more subforms used to create a health form. The following illustration shows the outline view (represents the tuc018_template_flowed.xdp file used in the *Assembling multiple XDP fragments* quick start):



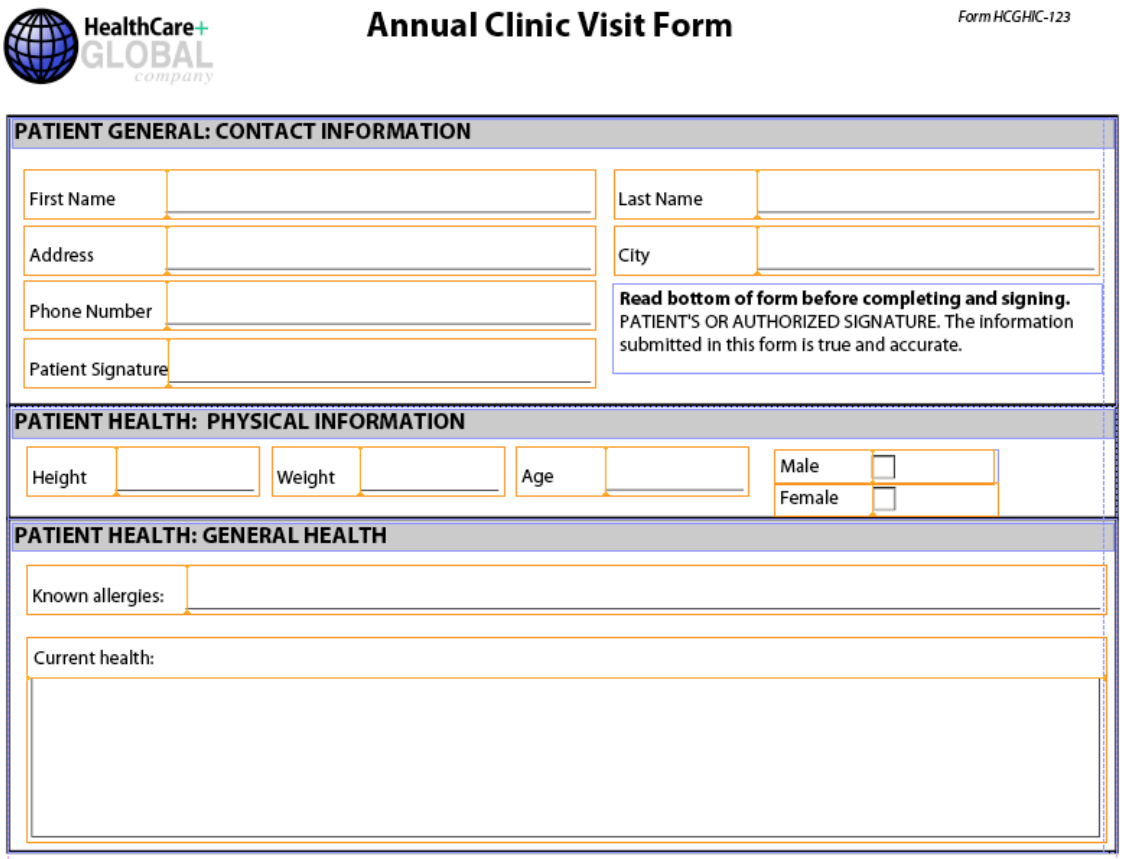
The following illustration shows the patient section (represents the tuc018_contact.xdp file used in the *Assembling multiple XDP fragments* quick start):



The following illustration shows the patient health section (represents the tuc018_patient.xdp file used in the *Assembling multiple XDP fragments* quick start):



This fragment contains two subforms named *subPatientPhysical* and *subPatientHealth*. Both of these sub forms are referenced in the DDX document that is passed to the Assembler service. Using the Assembler service, you can combine all of these XDP fragments into a single XDP document, as shown in the following illustration.



The following DDX document assembles multiple XDP fragments into an XDP document.

```
<?xml version="1.0" encoding="UTF-8"?>
<DDX xmlns="http://ns.adobe.com/DDX/1.0/">
        <XDP result="tuc018result.xdp">
           <XDP source="tuc018_template_flowed.xdp">
           <XDPContent insertionPoint="ddx_fragment" source="tuc018_contact.xdp"
fragment="subPatientContact" required="false"/>
              <XDPContent insertionPoint="ddx_fragment" source="tuc018_patient.xdp"
fragment="subPatientPhysical" required="false"/>
              <XDPContent insertionPoint="ddx_fragment" source="tuc018_patient.xdp"
fragment="subPatientHealth" required="false"/>
           </XDP>
        </XDP>
</DDX>
```

The DDX document contains an XDP `result` tag that specifies the name of the result. In this situation, the value is `tuc018result.xdp`. This value is referenced in the application logic that is used to retrieve the XDP document after the Assembler service returns the result. For example, consider the following Java application logic that is used to retrieve the assembled XDP document (notice the value is bolded):

```
//Iterate through the map object to retrieve the result XDP document
for (Iterator i = allDocs.entrySet().iterator(); i.hasNext();) {
    // Retrieve the Map object's value
    Map.Entry e = (Map.Entry)i.next();

    //Get the key name as specified in the
    //DDX document
    String keyName = (String)e.getKey();
    if (keyName.equalsIgnoreCase("tuc018result.xdp"))
                {
        Object o = e.getValue();
        outDoc = (Document)o;

        //Save the result PDF file
        File myOutFile = new File("C:\\AssemblerResultXDP.xdp");
        outDoc.copyToFile(myOutFile);
    }
}
```

The `XDP source` tag specifies the XDP file that represents a complete XDP document that can be used as a container for adding XDP fragments or as one of a number of documents that are appended together in order. In this situation, the XDP document is used only as a container (the first illustration shown in *Assembling Multiple XDP Fragments*). That is, the other XDP files are placed within the XDP container.

For each sub form, you can add an `XDPContent` element (this element is optional). In the above example, notice that there are three sub forms: `subPatientContact`, `subPatientPhysical`, and `subPatientHealth`. Both the `subPatientPhysical` subform and the `subPatientHealth` sub form are located in the same XDP file, tuc018_patient.xdp. The fragment element specifies the name of the sub form, as defined in Designer.

Note: *For more information about the Assembler service, see Services Reference for AEM Forms.*

Note: *For more information about a DDX document, see Assembler Service and DDX Reference.*

Summary of steps

To assemble multiple XDP fragments, perform the following tasks:

1. Include project files.
2. Create a PDF Assembler client.
3. Reference an existing DDX document.
4. Reference the XDP documents.
5. Set run-time options.
6. Assemble the multiple XDP documents.
7. Retrieve the assembled XDP document.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-assembler-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

Create a PDF Assembler client

Before you can programmatically perform an Assembler operation, create an Assembler service client.

Reference an existing DDX document

A DDX document must be referenced to assemble multiple XDP documents. This DDX document must contain `XDP result`, `XDP source`, and `XDPContent` elements.

Reference the XDP documents

To assemble multiple XDP documents, reference all XDP files that are used to assemble the result XDP document. Ensure that the name of the sub form contained in the XDP document that is referenced by the `source` attribute is specified in the `fragment` attribute. A sub form is defined in Designer. For example, consider the following XML.

```
<XDPContent insertionPoint="ddx_fragment" source="tuc018_contact.xdp"
fragment="subPatientContact" required="false"/>
```

The sub form named *subPatientContact* must be located in the XDP file named *tuc018_contact.xdp*.

Set run-time options

You can set run-time options that control the behavior of the Assembler service while it performs a job. For example, you can set an option that instructs the Assembler service to continue processing a job if an error is encountered.

Assemble the multiple XDP documents

To assemble multiple XDP files, call the `invokeDDX` operation. The Assembler service returns the assembled XDP document within a collection object.

Retrieve the assembled XDP document

An assembled XDP document is returned within a collection object. Iterate through the collection object and save the XDP document as an XDP file. You can also pass the XDP document to another AEM Forms service, such as Output.

See also

"Assemble multiple XDP fragments using the Java API" on page 987

"Assemble multiple XDP fragments using the web service API" on page 988

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Assembling PDF Documents" on page 925

"Creating PDF Documents Using Fragments" on page 703

Assemble multiple XDP fragments using the Java API

Assemble multiple XDP fragments by using the Assembler Service API (Java):

1. Include project files.

   Include client JAR files, such as adobe-assembler-client.jar, in your Java project's class path.

2. Create a PDF Assembler client.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `AssemblerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Reference an existing DDX document.
   - Create a `java.io.FileInputStream` object that represents the DDX document by using its constructor and passing a string value that specifies the location of the DDX file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.
4. Reference the XDP documents.
   - Create a `java.util.Map` object that is used to store input XDP documents by using a `HashMap` constructor.
   - Create a `com.adobe.idp.Document` object and pass the `java.io.FileInputStream` object that contains the input XDP file (repeat this task for each XDP file).
   - Add an entry to the `java.util.Map` object by invoking its `put` method and passing the following arguments:
     - A string value that represents the key name. This value must match the `source` element value specified in the DDX document (repeat this task for each XDP file).
     - A `com.adobe.idp.Document` object that contains the XDP document that corresponds to the `source` element (repeat this task for each XDP file).
5. Set the run-time options.
   - Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
   - Set run-time options to meet your business requirements by invoking a method that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, invoke the `AssemblerOptionSpec` object's `setFailOnError` method and pass `false`.
6. Assemble the multiple XDP documents.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following required values:

- A `com.adobe.idp.Document` object that represents the DDX document to use
- A `java.util.Map` object that contains the input XDP files
- A `com.adobe.livecycle.assembler.client.AssemblerOptionSpec` object that specifies the run-time options, including the default font and the job log level

The `invokeDDX` method returns a `com.adobe.livecycle.assembler.client.AssemblerResult` object that contains the assembled XDP document.

7 Retrieve the assembled XDP document.

To obtain the assembled XDP document, perform the following actions:

- Invoke the `AssemblerResult` object's `getDocuments` method. This method returns a `java.util.Map` object.
- Iterate through the `java.util.Map` object until you find the resultant `com.adobe.idp.Document` object.
- Invoke the `com.adobe.idp.Document` object's `copyToFile` method to extract the assembled XDP document.

See also

"Assembling Multiple XDP Fragments" on page 983

"Quick Start (SOAP mode): Assembling multiple XDP fragments using the Java API" on page 54

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Assemble multiple XDP fragments using the web service API

Assemble multiple XDP fragments by using the Assembler Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition when setting a service reference:

```
http://localhost:8080/soap/services/AssemblerService?WSDL&lc_version=9.0.1.
```

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create a PDF Assembler client.

- Create an `AssemblerServiceClient` object by using its default constructor.
- Create an `AssemblerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service, such as `http://localhost:8080/soap/services/AssemblerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AssemblerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the `AssemblerServiceClient.ClientCredentials.UserName.UserName` field.

- Assign the corresponding password value to the `AssemblerServiceClient.ClientCredentials.UserName.Password` field.
- Assign the `HttpClientCredentialType.Basic` constant value to the `BasicHttpBindingSecurity.Transport.ClientCredentialType` field.
- Assign the `BasicHttpSecurityMode.TransportCredentialOnly` constant value to the `BasicHttpBindingSecurity.Security.Mode` field.

3 Reference an existing DDX document.

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the DDX document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the DDX document and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, starting position, and stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` property with the contents of the byte array.

4 Reference the XDP documents.

- For each input XDP file, create a `BLOB` object by using its constructor. The `BLOB` object is used to store the input file.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the input file and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method. Pass the byte array, starting position, and stream length to read.
- Populate the `BLOB` object by assigning its `MTOM` field with the contents of the byte array.
- Create a `MyMapOf_xsd_string_To_xsd_anyType` object. This collection object is used to store input files required to create an assembled XDP document.
- For each input file, create a `MyMapOf_xsd_string_To_xsd_anyType_Item` object.
- Assign a string value that represents the key name to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `key` field. This value must match the value of the element specified in the DDX document. (Perform this task for each input XDP file.)
- Assign the `BLOB` object that stores the input file to the `MyMapOf_xsd_string_To_xsd_anyType_Item` object's `value` field. (Perform this task for each input XDP file.)
- Add the `MyMapOf_xsd_string_To_xsd_anyType_Item` object to the `MyMapOf_xsd_string_To_xsd_anyType` object. Invoke the `MyMapOf_xsd_string_To_xsd_anyType` object's `Add` method and pass the `MyMapOf_xsd_string_To_xsd_anyType` object. (Perform this task for each input XDP document.)

5 Set run-time options.

- Create an `AssemblerOptionSpec` object that stores run-time options by using its constructor.
- Set run-time options to meet your business requirements by assigning a value to a data member that belongs to the `AssemblerOptionSpec` object. For example, to instruct the Assembler service to continue processing a job when an error occurs, assign `false` to the `AssemblerOptionSpec` object's `failOnError` data member.

6 Assemble the multiple XDP documents.

Invoke the `AssemblerServiceClient` object's `invokeDDX` method and pass the following values:

- A `BLOB` object that represents the DDX document
- The `MyMapOf_xsd_string_To_xsd_anyType` object that contains the required files
- An `AssemblerOptionSpec` object that specifies run-time options

The `invokeDDX` method returns an `AssemblerResult` object that contains the results of the job and any exceptions that occurred.

7 Retrieve the assembled XDP document.

To obtain the newly created XDP document, perform the following actions:

- Access the `AssemblerResult` object's `documents` field, which is a `Map` object that contains the resultant PDF documents.
- Iterate through the `Map` object to obtain each resultant document. Then, cast that array member's `value` to a `BLOB`.
- Extract the binary data that represents the PDF document by accessing its `BLOB` object's `MTOM` property. This returns an array of bytes that you can write out to an XDP file.

See also

"Assembling Multiple XDP Fragments" on page 983

Quick Start (MTOM): Assembling multiple XDP fragments using the web service API

"Invoking AEM Forms using MTOM" on page 529

Working with PDF/A Documents

About the DocConverter Service

The DocConverter service can convert PDF documents to PDA/A documents. You can accomplish these tasks using this service:

- Convert PDF documents to PDF/A documents. (See "Converting Documents to PDF/A Documents" on page 990.)
- Determine if PDF documents are PDF/A documents. (See "Programmatically Determining PDF/A Compliancy" on page 994.)

Note: *For more information about the DocConverter service, see Services Reference for AEM Forms.*

Converting Documents to PDF/A Documents

You can use the DocConverter service to convert a PDF document to a PDF/A document. Because PDF/A is an archival format for long-term preservation of the document's content, all fonts are embedded and the file is uncompressed. As a result, a PDF/A document is typically larger than a standard PDF document. Also, a PDF/A document does not contain audio and video content. Before you convert a PDF document to a PDF/A document, ensure that the PDF document is not a PDF/A document.

The PDF/A-1 specification consists of two levels of conformance, namely A and B. The major difference between the two is regarding the logical structure (accessibility) support, which is not required for conformance level B. Regardless of the conformance level, PDF/A-1 dictates that all fonts are embedded within the generated PDF/A document. At this time, only PDF/A-1b is supported in validation (and conversion).

While PDF/A is the standard for archiving PDF documents, it is not mandatory that PDF/A be used for archiving if a standard PDF document meets your company's requirements. The purpose of the PDF/A standard is to establish a PDF file meant for long-term archiving and document-preservation needs.

Note: *For more information about the DocConverter service, see Services Reference for AEM Forms.*

Summary of steps

To convert a PDF document to a PDF/A document, perform the following steps:

1. Include project files.
2. Create a DocConvert client
3. Reference a PDF document to convert to a PDF/A document.
4. Set tracking information.
5. Convert the document.
6. Save the PDF/A document.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-docconverter-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a DocConvert client

Before you can programmatically perform an DocConverter operation, you must create a DocConverter client. If you are using the Java API, create a `DocConverterServiceClient` object. If you are using the DocConverter web service API, create a `DocConverterServiceService` object.

Reference a PDF document to convert to a PDF/A document

Retrieve a PDF document to convert to a PDF/A document. If you attempt to convert a PDF document, such as an Acrobat form, to a PDF/A document, you will cause an exception.

Set tracking information

You can set a run-time option that determines how much information is tracked during the conversion process. That is, you can set nine different levels that specify how much information the DocConverter service tracks when it converts a PDF document to a PDF/A document.

Convert the document

After you create the DocConverter service client, reference the PDF document to convert and set the run-time option that specifies how much information is tracked, you can convert the PDF document to a PDF/A document.

Save the PDF/A document

You can save the PDF/A document as a PDF file.

See also

"Convert documents to PDF/A documents using the Java API" on page 992

"Convert documents to PDF/A documents using the web service API" on page 993

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Programmatically Determining PDF/A Compliancy" on page 994

Convert documents to PDF/A documents using the Java API

Convert a PDF document to a PDF/A document by using the Java API:

1 Include project files

Include client JAR files, such as adobe-docconverter-client.jar, in your Java project's class path.

2 Create a DocConvert client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `DocConverterServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference a PDF document to convert to a PDF/A document

- Create a `java.io.FileInputStream` object that represents the PDF document to convert by using its constructor and passing a string value that specifies the location of the PDF file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set tracking information

- Create a `PDFAConversionOptionSpec` object by using its constructor.
- Set the information tracking level by invoking the `PDFAConversionOptionSpec` object's `setLogLevel` method and passing a string value that specifies the tracking level. For example, pass the value `FINE`. For information about the different values, see the `setLogLevel` method in the AEM Forms API Reference.

5 Convert the document

Convert the PDF document to a PDF/A document by invoking the `DocConverterServiceClient` object's `toPDFA` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the PDF document to convert
- The `PDFAConversionOptionSpec` object that specifies tracking information

The `toPDFA` method returns a `PDFAConversionResult` object that contains the PDF/A document.

6 Save the PDF/A document

- Retrieve the PDF/A document by invoking the `PDFAConversionResult` object's `getPDFA` method. This method returns a `com.adobe.idp.Document` object that represents the PDF/A document.
- Create a `java.io.File` object that represents the PDF/A file. Ensure that the file name extension is .pdf.
- Populate the file with PDF/A data by invoking the `com.adobe.idp.Document` object's `copyToFile` method and passing the `java.io.File` object.

See also

"Working with PDF/A Documents" on page 990

"Quick Start (SOAP mode): Converting a document to a PDF/A document using the Java API" on page 92

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert documents to PDF/A documents using the web service API

Convert a PDF document to a PDF/A document by using the DocConverter API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the DocConverter WSDL.
- Reference the Microsoft .NET client assembly.

2 Create a DocConvert client

- Using the Microsoft .NET client assembly, create a `DocConverterServiceService` object by invoking its default constructor.
- Set the `DocConverterServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Reference a PDF document to convert to a PDF/A document

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the PDF document that is converted to a PDF/A document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.

4 Set tracking information

- Create a `PDFAConversionOptionSpec` object by using its constructor.
- Set the information tracking level by assigning a value that specifies the tracking level to the `PDFAConversionOptionSpec` object's `logLevel` data member. For example, assign the value `FINE` to this data member.

5 Convert the document

Convert the PDF document to a PDF/A document by invoking the `DocConverterServiceService` object's `toPDFA` method and passing the following values:

- The `BLOB` object that contains the PDF document to convert
- The `PDFAConversionOptionSpec` object that specifies tracking information

The `toPDFA` method returns a `PDFAConversionResult` object that contains the PDF/A document.

6 Save the PDF/A document

- Create a `BLOB` object that stores the PDF/A document by getting the value of the `PDFAConversionResult` object's `PDFADocument` data member.

- Create a byte array that stores the content of the `BLOB` object that was returned by using the `PDFAConversionResult` object. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF/A document.
- Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
- Write the contents of the byte array to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

See also

"Working with PDF/A Documents" on page 990

Quick Start (Base64): Converting a document to a PDF/A document using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Programmatically Determining PDF/A Compliancy

You can use the DocConverter service to determine whether a PDF document is PDF/A-compliant. For information about a PDF/A document and how to convert a PDF document to a PDF/A document, see "Converting Documents to PDF/A Documents" on page 990.

Note: *For more information about the DocConverter service, see Services Reference for AEM Forms.*

Summary of steps

To determine PDF/A compliancy, perform the following steps:

1. Include project files.
2. Create a DocConvert client
3. Reference a PDF document used to determine PDF/A compliancy.
4. Set run-time options.
5. Retrieve information about the PDF document.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-docconverter-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a DocConvert client

Before you can programmatically perform an DocConverter operation, you must create a DocConverter client. If you are using the Java API, create a `DocConverterServiceClient` object. If you are using the DocConverter web service API, create a `DocConverterServiceService` object.

Reference a PDF document used to determine PDF/A compliancy

A PDF document must be referenced and passed to the DocConverter service in order to determine whether the PDF document is PDF/A-compliant.

Set run-time options

You can set a run-time option that determines how much information is tracked during the conversion process. That is, you can set nine different level that specify how much information the DocConverter service tracks when it converts a PDF document to a PDF/A document.

Retrieve information about the PDF document

After you create the DocConverter service client, reference the PDF document, and set the run-time options, you can determine whether the PDF document is a PDF/A-compliant document.

See also

"Determine PDF/A compliancy using the Java API" on page 995

"Determine PDF/A compliancy using the web service API" on page 996

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Determine PDF/A compliancy using the Java API

Determine PDF/A compliancy by using the Java API:

1 Include project files

Include client JAR files, such as adobe-docconverter-client.jar, in your Java project's class path.

2 Create a DocConvert client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `DocConverterServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference a PDF document used to determine PDF/A compliancy

- Create a `java.io.FileInputStream` object that represents the PDF document to convert by using its constructor and passing a string value that specifies the location of the PDF file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4 Set run-time options

- Create a `PDFAValidationOptionSpec` object by using its constructor.
- Set the compliance level by invoking the `PDFAValidationOptionSpec` object's `setCompliance` method and passing `PDFAValidationOptionSpec.Compliance.PDFA_1B`.

- Set the information tracking level by invoking the `PDFAValidationOptionSpec` object's `setLogLevel` method and passing a string value that specifies the tracking level. For example, pass the value `FINE`. For information about the different values, see the `setLogLevel` method in the AEM Forms API Reference.

5 Retrieve information about the PDF document

Determine PDF/A compliancy by invoking the `DocConverterServiceClient` object's `isPDFA` method and passing the following values:

- The `com.adobe.idp.Document` object that contains the PDF document.
- The `PDFAValidationOptionSpec` object that specifies run-time options.

The `isPDFA` method returns a `PDFAValidationResult` object that contains the results of this operation.

See also

"Working with PDF/A Documents" on page 990

"Quick Start (SOAP mode): Determining PDF/A compliancy using the Java API" on page 94

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Determine PDF/A compliancy using the web service API

Determine PDF/A compliancy by using the web service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the DocConverter WSDL.
- Reference the Microsoft .NET client assembly.

2 Create a DocConvert client

- Using the Microsoft .NET client assembly, create a `DocConverterServiceService` object by invoking its default constructor.
- Set the `DocConverterServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Reference a PDF document used to determine PDF/A compliancy

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store the PDF document that is converted to a PDF/A document.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode to open the file in.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.

4 Set run-time options

- Create a `PDFAValidationOptionSpec` object by using its constructor.
- Set the compliance level by assigning the `PDFAValidationOptionSpec` object's `compliance` data member with the value `PDFAConversionOptionSpec_Compliance.PDFA_1B`.

- Set the information tracking level by assigning the `PDFAValidationOptionSpec` object's `resultLevel` data member with the value `PDFAValidationOptionSpec_ResultLevel.DETAILED`.

5 Retrieve information about the PDF document

Determine PDF/A compliancy by invoking the `DocConverterServiceService` object's `isPDFA` method and passing the following values:

- The `BLOB` object that contains the PDF document.
- The `PDFAValidationOptionSpec` object that contains run-time options.

The `isPDFA` method returns a `PDFAValidationResult` object that contains the results of this operation.

See also

"Working with PDF/A Documents" on page 990

Quick Start (Base64): Determining PDF/A compliancy using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Working with PDF Utilities

About the PDF Utilities Service

The PDF Utilities service can convert between PDF and XDP file formats, set and retrieve PDF document properties, and manipulate XMP metadata. For example, before converting a PDF document to another format, it is useful to inspect its properties to determine which service operation to invoke for the conversion.

You can accomplish these tasks using the PDF Utilities service:

- Convert PDF documents to XDP documents. (See Converting PDF Documents into XDP Documents.)
- Convert XDP documents to PDF documents. (See "Converting XDP Documents into PDF Documents" on page 999.)
- Retrieve PDF document properties. (See "Retrieving PDF Document Properties" on page 1000.)
- Save a PDF document and optimize it for fast web viewing. (See "Setting PDF Document Save Modes" on page 1002.)

Note: *For more information about the PDF Utilities service, see Services Reference for AEM Forms.*

Converting PDF Documents into XDP Documents

You can use the PDF Utilities Java and web service APIs to programmatically convert PDF documents into XDP documents.

Note: *For more information about the PDF Utilities service, see Services Reference for AEM Forms.*

Summary of steps

To convert a PDF document into an XDP document, perform the following steps:

1 Include project files.

2 Create a PDFUtilityService client.

3 Invoke the PDF to XDP conversion operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a PDFUtilityService client

Before you can programmatically perform a PDF Utilities operation, you must create a PDFUtilityService client. With the Java API, this is accomplished by creating a `PDFUtilityServiceClient` object. With the web service API, this is accomplished by using a `PDFUtilityServiceService` object.

Invoke the PDF to XDP conversion operation

After you create the service client, you can invoke the PDF to XDP conversion operation.

See also

"Convert PDF documents into XDP documents using the Java API" on page 998

"Convert PDF documents into XDP documents using the web service API" on page 998

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert PDF documents into XDP documents using the Java API

Convert PDF documents into XDP documents by using the PDF Utilities API(Java):

1 Include project files

Include client JAR files, such as the adobe-pdfutility-client.jar, in your Java project's class path..

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the PDF to XDP conversion operation

To perform the conversion, invoke the `PDFUtilityServiceClient` object's `convertPDFtoXDP` method and pass in a `com.adobe.idp.Document` object that represents the PDF file. The method returns a `com.adobe.idp.Document` object that represents the newly created XDP file.

See also

"Converting PDF Documents into XDP Documents" on page 997

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert PDF documents into XDP documents using the web service API

Convert PDF documents into XDP documents by using the PDF Utilities API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the PDF Utilities service WSDL file.
- Reference the Microsoft .NET client assembly.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceService` object by using your proxy class constructor.

3 Invoke the PDF to XDP conversion operation

Invoke the `PDFUtilityServiceService` object's `convertPDFtoXDP` method and pass in a `BLOB` object that represents the PDF file. The method returns a `BLOB` object that represents the newly created XDP file.

See also

"Converting PDF Documents into XDP Documents" on page 997

Quick Start (Base64): Converting a PDF document to an XDP document using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Converting XDP Documents into PDF Documents

You can use the PDF Utilities Java and web service APIs to programmatically convert XDP documents into PDF documents.

Note: *For more information about the PDF Utilities service, see Services Reference for AEM Forms.*

Summary of steps

To convert an XDP document into a PDF document, perform the following steps:

1 Include project files.

2 Create a PDFUtilityService client.

3 Invoke the XDP to PDF conversion operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a PDFUtilityService client

Before you can programmatically perform a PDF Utilities operation, you must create a PDFUtilityService client. With the Java API, this is accomplished by creating a `PDFUtilityServiceClient` object. With the web service API, this is accomplished by using a `PDFUtilityServiceService` object.

Invoke the XDP to PDF conversion operation

After you create the service client, you can invoke the XDP to PDF conversion operation.

See also

"Convert XDP documents into PDF documents using the Java API" on page 1000

"Converting XDP documents into PDF documents using the web service API" on page 1000

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Convert XDP documents into PDF documents using the Java API

Convert XDP documents into PDF documents by using the PDF Utilities API (Java):

1 Include project files

Include client JAR files, such as adobe-pdfutility-client.jar, in your Java project's class path.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the XDP to PDF conversion operation

To perform the conversion, invoke the `PDFUtilityServiceClient` object's `convertXDPtoPDF` method and pass in a `com.adobe.idp.Document` object that represents the XDP file. The method returns a `com.adobe.idp.Document` object that represents the newly created PDF file.

See also

"Converting XDP Documents into PDF Documents" on page 999

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Converting XDP documents into PDF documents using the web service API

Convert XDP documents into PDF documents by using the PDF Utilities API (web service API):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the PDF Utilities service WSDL file.
- Reference the Microsoft .NET client assembly.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceService` object by using your proxy class constructor.

3 Invoke the XDP to PDF conversion operation

To perform the conversion, invoke the `PDFUtilityServiceService` object's `convertXDPtoPDF` method and pass in a `BLOB` object that represents the XDP file. The method returns a `BLOB` object that represents the newly created PDF file.

See also

"Converting XDP Documents into PDF Documents" on page 999

Quick Start (Base64): Converting an XDP document to a PDF document using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Retrieving PDF Document Properties

You can use the PDF Utilities Java and web service APIs to programmatically retrieve PDF document properties, such as whether the document is a fillable form or the minimum Acrobat version required to read the document.

Note: *For more information about the PDF Utilities service, see Services Reference for AEM Forms*

Summary of steps

To retrieve PDF document properties, perform the following steps:

1 Include project files.

2 Create a PDFUtilityService client.

3 Invoke the properties retrieval operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a PDFUtilityService client

Before you can programmatically perform a PDF Utilities operation, you must create a PDFUtilityService client. With the Java API, this is accomplished by creating a `PDFUtilityServiceClient` object. With the web service API, this is accomplished using a `PDFUtilityServiceService` object.

Invoke the properties retrieval operation

After you create the service client, you can invoke the properties retrieval operation.

See also

"Retrieve PDF document properties using the Java API" on page 1001

"Retrieve PDF document properties using the web service API" on page 1002

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve PDF document properties using the Java API

Retrieve PDF document properties by using the PDF Utilities API (Java):

1 Include project files

Include client JAR files, such as adobe-pdfutility-client.jar, in your Java project's class path.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the properties retrieval operation

To perform the conversion, invoke the `PDFUtilityServiceClient` object's `getPDFProperties` method and pass in the following:

- A `com.adobe.idp.Document` object that represents the PDF document.
- A `PDFPropertiesOptionSpec` object that contains the properties to be evaluated.

The method returns a `PDFPropertiesResult` object that contains the results of the query.

See also

"Retrieving PDF Document Properties" on page 1000

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve PDF document properties using the web service API

Retrieve PDF document properties by using the PDF Utilities web service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the PDF Utilities service WSDL file.
- Reference the Microsoft .NET client assembly.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceService` object by using your proxy class constructor.

3 Invoke the properties retrieval operation

To perform the conversion, invoke the `PDFUtilityServiceService` object's `getPDFProperties` method and pass in the following:

- A `BLOB` object that represents the PDF document.
- A `PDFPropertiesOptionSpec` object that contains the properties to be evaluated.

The method returns a `PDFPropertiesResult` object that contains the results of the query.

See also

"Retrieving PDF Document Properties" on page 1000

Quick Start (Base64): Retrieving PDF document properties using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Setting PDF Document Save Modes

You can use the PDF Utilities service Java and web service APIs to programmatically set a save mode for a PDF document. When using the PDF Utilities service to set a save mode, the PDF Utilities service only sets the save mode and does not actually save the PDF document. The PDF document is saved when it is passed to another service operation. For example, you can use the PDF Utilities service to set a specific save mode and pass it to the Encryption service, where the PDF document is actually saved and encrypted.

Note: *For more information about the PDF Utilities service, see Services Reference for AEM Forms.*

Summary of steps

To set the save option for PDF documents, perform the following steps:

1 Include project files.

2 Create a PDFUtilityService client.

3 Set the save mode.

4 Invoke the save operation.

5 Pass the PDF document to another operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create a PDFUtilityService client

Before you can programmatically perform a PDF Utilities operation, you must create a PDFUtilityService client. With the Java API, this is accomplished by creating a `PDFUtilityServiceClient` object. With the web service API, this is accomplished using a `PDFUtilityServiceService` object.

Set the Save mode

You can choose one of the following save options:

- `INCREMENTAL`: To save incrementally to reduce the time required to save
- `FAST_WEB_VIEW`: save for fast web viewing
- `FULL`: To save using a full save (without optimizations)

Invoke the save style operation

After you create the service client, you can invoke the properties retrieval operation.

Pass the PDF document to another AEM Forms operation

Once the PDF Utilities service sets the specified Save mode, pass the PDF document to another AEM Forms operation. Once returned from that operation, the PDF document is saved in the specified mode. For example, if you use the PDF Utilities service to set the `FAST_WEB_VIEW` mode and then pass the PDF document to the Encryption service's `encryptUsingPassword` operation, the returned PDF document is encrypted with a password and save in the `FAST_WEB_VIEW` mode.

Note: *The Quick Start that is associated with this section sets the `FAST_WEB_VIEW` mode and then passes the PDF document to the Encryption service's `encryptUsingPassword` operation.*

See also

"Set PDF document save options using the Java API" on page 1003

"Set PDF document save options using the web service API" on page 1004

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Encrypting PDF Documents with a Password" on page 806

Set PDF document save options using the Java API

Set the PDF document save options by using the PDF Utilities API (Java):

1. Include project files

   Include client JAR files, such as adobe-pdfutility-client.jar, in your Java project's class path.

2. Create a PDFUtilityService client

   Create a `PDFUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3. Set the Save mode

   - Create a `PDFUtilitySaveMode` object by using its constructor.
   - Set the save mode by invoking the `PDFUtilitySaveMode` object's `setSaveStyle` method and passing a string value that specifies the save mode. For example, to save for fast web viewing, pass `FAST_WEB_VIEW`.

4. Invoke the save style operation

Invoke the `PDFUtilityServiceClient` object's `setSaveMode` method and pass the following values:

- A `com.adobe.idp.Document` object that represents the PDF document.
- A `PDFUtilitySaveMode` object that contains the save style to be used.
- A Boolean value used to determine whether to override any previous settings.

The method returns a `com.adobe.idp.Document` object formatted using the specified save style.

5 Pass the PDF document to another AEM Forms operation

- Pass the returned `com.adobe.idp.Document` object to another AEM Forms operation.

See also

"Setting PDF Document Save Modes" on page 1002

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Set PDF document save options using the web service API

Set the PDF document save options by using the PDF Utilities AP (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the PDF Utilities service WSDL file.
- Reference the Microsoft .NET client assembly.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceService` object by using your proxy class constructor.

3 Set the Save mode

- Create a `PDFUtilitySaveMode` object by using its constructor.
- Set the save mode by assigning a string value to the `PDFUtilitySaveMode` object's `saveStyle` method that specifies the save mode. For example, to save for fast web viewing, specify `FAST_WEB_VIEW`.

4 Invoke the save style operation

Invoke the `PDFUtilityServiceService` object's `setSaveMode` method and pass the following values:

- A `BLOB` object that represents the PDF document.
- A `PDFUtilitySaveMode` object that contains the save style to be used.
- A Boolean value used to determine whether to override any previous settings.

The method returns a `BLOB` object formatted using the specified save style. You can then save that object as a PDF document.

5 Pass the PDF document to another Forms operation

- Pass the returned `BLOB` object to another AEM Forms operation.

See also

"Setting PDF Document Save Modes" on page 1002

Quick Start (Base64): Setting the save style for a PDF document using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Sanitizing PDF Documents

You can use the PDF Utilities Java APIs to programmatically convert PDF documents into XDP documents.

Note: *For more information about the PDF Utilities service, see Services Reference for AEM Forms.*

Summary of steps

To sanitize PDF document, perform the following steps:

1 Include project files.

2 Create a PDFUtilityService client.

3 Invoke the sanitization operation.

Include project files

Include necessary files into your development project. To create a client application using Java, include the necessary JAR files.

Create a PDFUtilityService client

Before you can programmatically perform a sanitization operation, you must create a PDFUtilityService client. With the Java API, this is accomplished by creating a `PDFUtilityServiceClient` object.

Invoke the PDF to XDP conversion operation

After you create the service client, you can invoke the sanitization operation.

See also

"Convert PDF documents into XDP documents using the Java API" on page 998

"Convert PDF documents into XDP documents using the web service API" on page 998

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Sanitize PDF documents using the Java API

Sanitize documents by using the PDF Utilities API (Java):

1 Include project files

Include client JAR files, such as the adobe-pdfutility-client.jar, in your Java project's class path.

2 Create a PDFUtilityService client

Create a `PDFUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the PDF to XDP conversion operation

To perform the conversion, invoke the `PDFUtilityServiceClient` object's `convertPDFtoXDP` method and pass in a `com.adobe.idp.Document` object that represents the PDF file. The method returns a `com.adobe.idp.Document` object that represents the newly created XDP file.

See also

"Quick Start (SOAP mode): Sanitizing PDF documents" on page 282

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500S

Working with XMP Utilities

About the XMP Utilities Service

PDF documents contain metadata, which is information about the document as distinguished from the contents of the document, such as text and graphics. Adobe Extensible Metadata Platform (XMP) is a standard for handling document metadata.

The XMP Utilities service can retrieve and save XMP metadata from PDF documents, and import XMP metadata into PDF documents.

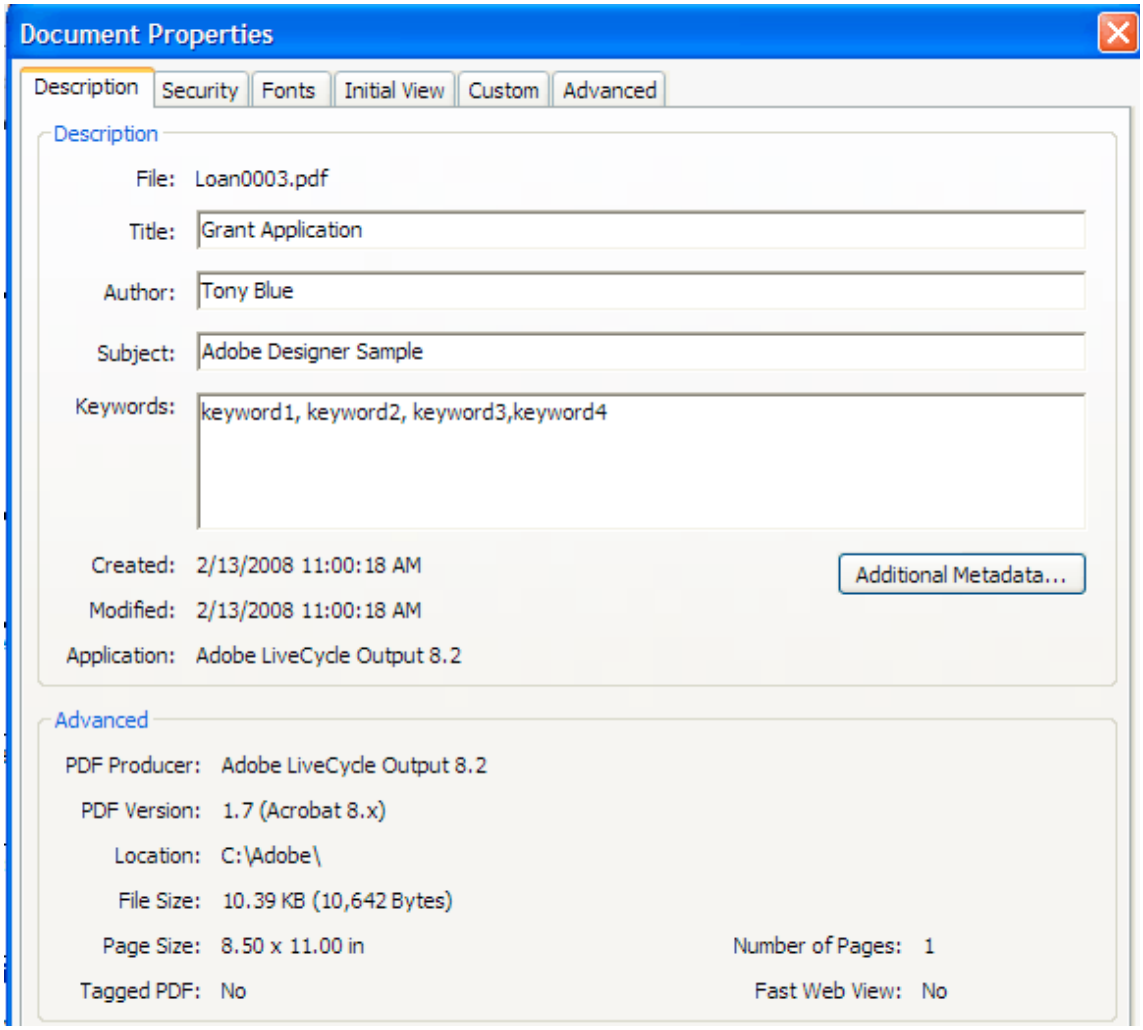
You can accomplish these tasks using the XMP Utilities service:

- Import metadata into PDF documents. (See "Importing Metadata into PDF Documents" on page 1007.)
- Export metadata from PDF documents. (See "Exporting Metadata from PDF Documents" on page 1011.)

Note: *For more information about the XMP Utilities service, see Services Reference for AEM Forms.*

Importing Metadata into PDF Documents

You can use the XMP Utilities Java and web service APIs to programmatically import XMP metadata into a PDF document. Metadata provides information about a PDF document such as the document's author and keywords related to the document. Metadata can be located in the document's Document Properties dialog, as shown in the following illustration.



To programmatically import metadata into a PDF document, you can use an existing XML document that specifies the metadata values or you can use an object of type `XMPUtilityMetadata`. (See AEM Forms API Reference.)

Note: *This section discusses how to use an XML document to import metadata into a PDF document.*

The following XML code contains metadata values that correspond to the previous illustration. For example, notice the bold items, which specify keywords.

```
<?xpacket begin="?" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="Adobe XMP Core 4.2-jc015 52.349034, 2008 Jun 20
00:30:39-PDT (debug)">
      <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
         <rdf:Description rdf:about=""
               xmlns:xmp="http://ns.adobe.com/xap/1.0/">
            <xmp:MetadataDate>2008-10-22T10:52:21-04:00</xmp:MetadataDate>
            <xmp:CreatorTool>AEM Forms</xmp:CreatorTool>
            <xmp:ModifyDate>2008-10-22T10:52:21-04:00</xmp:ModifyDate>
            <xmp:CreateDate>2008-02-13T11:00:18-05:00</xmp:CreateDate>
         </rdf:Description>
         <rdf:Description rdf:about=""
               xmlns:pdf="http://ns.adobe.com/pdf/1.3/">
            <pdf:Producer>AEM Forms</pdf:Producer>
            <pdf:Keywords>keyword1, keyword2, keyword3,keyword4</pdf:Keywords>
         </rdf:Description>
         <rdf:Description rdf:about=""
               xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/">
            <xmpMM:DocumentID>uuid:1cce1f84-331e-4d8d-8538-15441c271dd7</xmpMM:DocumentID>
            <xmpMM:InstanceID>uuid:cdda0ca6-7c91-4771-9dc9-796c8fe59350</xmpMM:InstanceID>
         </rdf:Description>
         <rdf:Description rdf:about=""
               xmlns:dc="http://purl.org/dc/elements/1.1/">
            <dc:format>application/pdf</dc:format>
            <dc:description>
               <rdf:Alt>
                  <rdf:li xml:lang="x-default">Adobe Designer Sample</rdf:li>
               </rdf:Alt>
            </dc:description>
            <dc:title>
               <rdf:Alt>
                  <rdf:li xml:lang="x-default">Grant Application</rdf:li>
               </rdf:Alt>
            </dc:title>
            <dc:creator>
               <rdf:Seq>
                  <rdf:li>Tony Blue</rdf:li>
               </rdf:Seq>
            </dc:creator>
            <dc:subject>
               <rdf:Bag>
```

```
                <rdf:li>keyword1</rdf:li>
                <rdf:li>keyword2</rdf:li>
                <rdf:li>keyword3</rdf:li>
                <rdf:li>keyword4</rdf:li>
             </rdf:Bag>
          </dc:subject>
       </rdf:Description>
       <rdf:Description rdf:about=""
             xmlns:desc="http://ns.adobe.com/xfa/promoted-desc/">
          <desc:version rdf:parseType="Resource">
             <rdf:value>1.0</rdf:value>
             <desc:ref>/template/subform[1]</desc:ref>
          </desc:version>
          <desc:contact rdf:parseType="Resource">
             <rdf:value>Adobe Systems Incorporated</rdf:value>
             <desc:ref>/template/subform[1]</desc:ref>
          </desc:contact>
       </rdf:Description>
    </rdf:RDF>
</x:xmpmeta>
```

Note: *For more information about the XMP Utilities service, see Services Reference for AEM Forms.*

Summary of steps

To import XMP metadata into a PDF document, perform the following steps:

1. Include project files.
2. Create an XMPUtilityService client.
3. Invoke the XMP metadata import operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create an XMPUtilityService client

Before you can programmatically perform an XMP Utilities operation, you must create an XMPUtilityService client. With the Java API, this is accomplished by creating an `XMPUtilityServiceClient` object. With the web service API, this is accomplished by using an `XMPUtilityServiceService` object.

Invoke the XMP metadata import operation

After you create the service client, you can invoke one of the XMP metadata import operations to import the XMP metadata into the specified PDF document.

See also

"Import XMP metadata using the Java API" on page 1010

"Importing XMP metadata using the web service API" on page 1010

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Import XMP metadata using the Java API

Import XMP metadata by using the XMP Utilities API (Java):

1 Include project files

Include client JAR files, such as adobe-pdfutility-client.jar, in your Java project's class path.

Note: *The adobe-pdfutility-client.jar file contains classes that enable you to programmatically invoke the XMP Utilities service.*

2 Create an XMPUtilityService client

Create an `XMPUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the XMP metadata import operation

To modify the XMP metadata, invoke either the `XMPUtilityServiceClient` object's `importMetadata` method or its `importXMP` method.

If you use the `importMetadata` method, pass in the following values:

- A `com.adobe.idp.Document` object that represents the PDF file.
- An `XMPUtilityMetadata` object that contains the metadata to be imported.

If you use the `importXMP` method, pass in the following values:

- A `com.adobe.idp.Document` object that represents the PDF file.
- A `com.adobe.idp.Document` object that represents an XML file that contains the metadata to be imported.

In either case, the returned value is a `com.adobe.idp.Document` object that represents the PDF file with the newly imported metadata. You can then save this object to disk.

See also

"Importing Metadata into PDF Documents" on page 1007

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Importing XMP metadata using the web service API

To programmatically import XMP metadata using the XMP Utilities web service API, perform the following tasks:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the XMP Utilities service WSDL file. (See "Invoking AEM Forms using Base64 encoding" on page 525.)
- Reference the Microsoft .NET client assembly. (See "Creating a .NET client assembly that uses Base64 encoding" on page 525.)

2 Create an XMPUtilityService client

Create an `XMPUtilityServiceService` object by using your proxy class constructor.

3 Invoke the XMP metadata import operation

To modify the XMP metadata, invoke either the `XMPUtilityServiceService` object's `importMetadata` method or its `importXMP` method.

If you use the `importMetadata` method, pass in the following values:

- A `BLOB` object that represents the PDF file.
- An `XMPUtilityMetadata` object that contains the metadata to be imported.

If you use the `importXMP` method, pass in the following values:

- A `BLOB` object that represents the PDF file.
- A `BLOB` object that represents an XML file that contains the metadata to be imported.

In either case, the returned value is a `BLOB` object that represents the PDF file with the newly imported metadata. You can then save this object to disk.

See also

"Importing Metadata into PDF Documents" on page 1007

Quick Start (Base64): Importing XMP metadata using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Exporting Metadata from PDF Documents

You can use the XMP Utilities Java and web service APIs to programmatically retrieve and save XMP metadata from a PDF document.

Note: *For more information about the XMP Utilities service, see Services Reference for AEM Forms.*

Summary of steps

To export XMP metadata from a PDF document, perform the following steps:

1. Include project files.
2. Create an XMPUtilityService client.
3. Invoke the XMP metadata export operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, ensure that you include the proxy files.

Create an XMPUtilityService client

Before you can programmatically perform an XMP Utilities operation, you must create an XMPUtilityService client. With the Java AP,I this is accomplished by creating an `XMPUtilityServiceClient` object. With the web service API, this is accomplished using an `XMPUtilityServiceService` object.

Invoke the XMP metadata export operation

After you create the service client, you can invoke one of the XMP metadata export operations, which can be used to inspect the XMP metadata or save it to disk.

See also

"Import XMP metadata using the Java API" on page 1010

"Importing XMP metadata using the web service API" on page 1010

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Export XMP metadata using the Java API

Export XMP metadata by using the XMP Utilities API (Java):

1 Include project files

Include client JAR files, such as adobe-pdfutility-client.jar, in your Java project's class path.

Note: *The adobe-pdfutility-client.jar file contains classes that enable you to programmatically invoke the XMP Utility service.*

2 Create an XMPUtilityService client

Create an `XMPUtilityServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the XMP metadata import operation

To inspect the XMP metadata, invoke the `XMPUtilityServiceClient` object's `exportMetadata` method and pass in a `com.adobe.idp.Document` object that represents the PDF file. The method returns an `XMPUtilityMetadata` object that contains the retrieved metadata.

To retrieve and save the XMP metadata, invoke the `XMPUtilityServiceClient` object's `exportXMP` method and pass in a `com.adobe.idp.Document` object that represents the PDF file. The method returns a `com.adobe.idp.Document` object that contains the retrieved metadata, which you can subsequently save to disk as an XML file.

See also

"Exporting Metadata from PDF Documents" on page 1011

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Export XMP metadata using the web service API

Export XMP metadata by using the XMP Utilities API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the XMP Utilities service WSDL file.
- Reference the Microsoft .NET client assembly.

2 Create an XMPUtilityService client

Create an `XMPUtilityServiceService` object by using your proxy class constructor.

3 Invoke the XMP metadata import operation

To inspect the XMP metadata, invoke the `XMPUtilityServiceClient` object's `exportMetadata` method and pass in a `BLOB` object that represents the PDF file. The method returns an `XMPUtilityMetadata` object that contains the retrieved metadata.

To retrieve and save the XMP metadata, invoke the `XMPUtilityServiceClient` object's `exportXMP` method and pass in a `BLOB` object that represents the PDF file. The method returns a `BLOB` object that contains the retrieved metadata, which you can subsequently save to disk as an XML file.

See also

"Exporting Metadata from PDF Documents" on page 1011

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Managing Users

About User Management

You can use the User Management API to create client applications that can manage roles, permissions, and principals (which can be users or groups), as well as authenticate users. User Management API consists of the following AEM Forms APIs:

- Directory Manager Service API
- Authentication Manager Service API
- Authorization Manager Service API

User Management enables you to assign, remove, and determine roles and permissions. It also enables you to assign, remove, and query domains, users, and groups. Finally, you can use User Management to authenticate users.

In "Adding Users" on page 1015 you will understand how to programmatically add users. This section uses the Directory Manager Service API.

In "Deleting Users" on page 1019 you will understand how to programmatically delete users. This section uses the Directory Manager Service API.

In "Managing Users and Groups" on page 1023 you will understand the difference between a local user and a directory user, and see examples of how to use the Java and web service APIs to programmatically manage users and groups. This section uses the Directory Manager Service API.

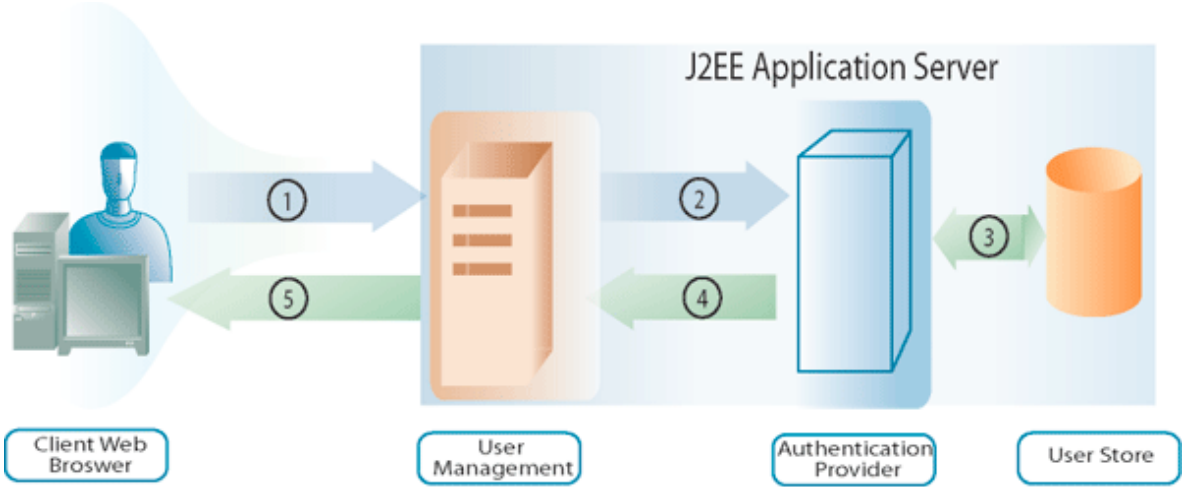
In "Managing Roles and Permissions" on page 1026 you will learn about the system roles and permissions and what you can do programmatically to augment them, and see examples of how to use the Java and web service APIs to programmatically manage roles and permissions. This section uses both the Directory Manager Service API and Authorization Manager Service API.

In "Authenticating Users" on page 1028 you will see examples of how to use the Java and web service APIs to programmatically authenticate users. This section uses the Authorization Manager Service API.

Understanding the authentication process

User Management provides built-in authentication functionality, and also provides you with the ability to connect it with your own authentication provider. When User Management receives an authentication request (for example, a user attempts to log in), it passes user information to the authentication provider to authenticate. User Management receives the results from the authentication provider after it authenticates the user.

The following diagram shows the interaction among an end user attempting to log in, User Management, and the authentication provider.



The following table describes each step of the authentication process.

Step	Description
1	A user attempts to log into a service that invokes User Management. The user specifies a user name and password.
2	User Management sends the user name and password, as well as configuration information, to the authentication provider.
3	The authentication provider connects to the user store and authenticates the user.
4	The authentication provider returns the results to User Management.
5	User Management either lets the user log in or denies access to the product.

Note: *If the server time zone is different from the client time zone, when consuming the WSDL for the AEM Forms Generate PDF service on a native SOAP stack using a .NET client on a WebSphere Application Server cluster, the following User Management authentication error may occur:*

```
[com.adobe.idp.um.webservices.WSSecurityHandler] errorCode:12803 errorCodeHEX:0x3203
message:WSSecurityHandler: UM authenticate returns exception : An error was discovered
processing the <wsse:Security> header. (WSSecurityEngine: Invalid timestamp The security
semantics of message have expired).
```

Understanding directory management

User Management is packaged with a directory service provider (the DirectoryManagerService) that supports connections to LDAP directories. If your organization uses a non-LDAP repository to store user records, you can create your own directory service provider that works with your repository.

Directory service providers retrieve records from a user store at the request of User Management. User Management regularly caches user and group records in the database to improve performance.

The directory service provider can be used to synchronize the User Management database with the user store. This step ensures that all user directory information and all user and group records are up to date.

In addition, the DirectoryManagerService provides you with the ability to create and manage domains. Domains define different user bases. The boundary of a domain is usually defined according to the way your organization is structured or how your user store is set up. User Management domains provide configuration settings that authentication providers and directory service providers use.

In the configuration XML that User Management exports, the root node that has the attribute value of `Domains` contains an XML element for each domain defined for User Management. Each of these elements contain other elements that define aspects of the domain associated with specific service providers.

Understanding objectSID values

When using Active Directory, it is important to understand that an `objectSID` value is not a unique attribute across multiple domains. This value stores the security identifier of an object. In a multiple domain environment (for example, a tree of domains) the `objectSID` value can be different.

An `objectSID` value would change if an object is moved from one Active Directory domain to another domain. Some objects have the same `objectSID` value anywhere in the domain. For example, groups like BUILTIN\Administrators, BUILTIN\Power Users and so on would have the same `objectSID` value regardless of the domains. These `objectSID` values are well known.

Adding Users

You can use the Directory Manager Service API (Java and web service) to programmatically add users to AEM Forms. After you add a user, you can use that user when performing a service operation that requires a user. For example, you can assign a task to the new user. (See "Assigning Tasks" on page 1089.)

Summary of steps

To add a user, perform the following steps:

1. Include project files.
2. Create a DirectoryManagerService client.
3. Define user information.
4. Add the user to AEM Forms.
5. Verify that the user is added.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create a DirectoryManagerService client

Before you can programmatically perform a Directory Manager service operation, create a Directory Manager Service API client.

Define user information

When you add a new user by using the Directory Manager Service API, define information for that user. Typically, when you add a new user, you define the following values:

- **Domain name**: The domain to which the user belongs (for example, `DefaultDom`).
- **User identifier value**: The identifier value of the user (for example, `wblue`).
- **Principal type**: The type of user (for example, you can specify `USER`).

- **Given name**: A given name for the user (for example, `Wendy`).
- **Family name**: The family name for the user (for example, `Blue)`.
- **Locale**: Locale information for the user.

Add the user to AEM Forms

After you define user information, you can add the user to AEM Forms. To add a user, invoke the `DirectoryManagerServiceClient` object's `createLocalUser` method.

Verify that the user was added

You can verify that the user was added to ensure that no issues occurred. Locate the new user by using the user identifier value.

See also

"Add users using the Java API" on page 1016

"Add users using the web service API" on page 1017

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Deleting Users" on page 1019

Add users using the Java API

Add users by using the Directory Manager Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path.

2 Create a DirectoryManagerServices client.

Create a `DirectoryManagerServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Define user information.

- Create a `UserImpl` object by using its constructor.
- Set the demain name by invoking the `UserImpl` object's `setDomainName` method. Pass a string value that specifies the domain name.
- Set the principal type by invoking the `UserImpl` object's `setPrincipalType` method. Pass a string value that specifies the type of user. For example, you can specify `USER`.
- Set the user identifier value by invoking the `UserImpl` object's `setUserid` method. Pass a string value that specifies the user identifier value. For example, you can specify `wblue`.
- Set the canonical name by invoking the `UserImpl` object's `setCanonicalName` method. Pass a string value that specifies the user's canonical name. For example, you can specify `wblue`.
- Set the given name by invoking the `UserImpl` object's `setGivenName` method. Pass a string value that specifies the user's given name. For example, you can specify `Wendy`.
- Set the family name by invoking the `UserImpl` object's `setFamilyName` method. Pass a string value that specifies the user's family name. For example, you can specify `Blue`.

Note: *Invoke a method that belongs to the* `UserImpl` *object to set other values. For example, you can set the locale value by invoking the* `UserImpl` *object's* `setLocale` *method.*

4 Add the user to AEM Forms.

Invoke the `DirectoryManagerServiceClient` object's `createLocalUser` method and pass the following values:

- The `UserImpl` object that represents the new user
- A string value that represents the user's password

The `createLocalUser` method returns a string value that specifies the local user identifier value.

5 Verify that the user was added.

- Create a `PrincipalSearchFilter` object by using its constructor.
- Set the user identifier value by invoking the `PrincipalSearchFilter` object's `setUserId` method. Pass a string value that represents the user identifier value.
- Invoke the `DirectoryManagerServiceClient` object's `findPrincipals` method and pass the `PrincipalSearchFilter` object. This method returns a `java.util.List` instance, where each element is a `User` object. Iterate through the `java.util.List` instance to locate the user.

See also

"Summary of steps" on page 1015

Quick Start (SOAP mode): Adding users using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Add users using the web service API

Add users by using the Directory Manager Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition for the service reference:
`http://localhost:8080/soap/services/DirectoryManagerService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create a DirectoryManagerService client.

- Create a `DirectoryManagerServiceClient` object by using its default constructor.
- Create a `DirectoryManagerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/DirectoryManagerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. Ensure that you specify `?blob=mtom`.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DirectoryManagerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.

- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DirectoryManagerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DirectoryManagerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Define user information.

- Create a `UserImpl` object by using its constructor.
- Set the demain name by assigning a string value to the `UserImpl` object's `domainName` field.
- Set the principal type by assigning a string value to the `UserImpl` object's `principalType` field. For example, you can specify `USER`.
- Set the user identifier value by assigning a string value to the `UserImpl` object's `userid` field.
- Set the canonical name value by assigning a string value to the `UserImpl` object's `canonicalName` field.
- Set the given name value by assigning a string value to the `UserImpl` object's `givenName` field.
- Set the family name value by assigning a string value to the `UserImpl` object's `familyName` field.

4 Add the user to AEM Forms.

Invoke the `DirectoryManagerServiceClient` object's `createLocalUser` method and pass the following values:

- The `UserImpl` object that represents the new user
- A string value that represents the user's password

The `createLocalUser` method returns a string value that specifies the local user identifier value.

5 Verify that the user was added.

- Create a `PrincipalSearchFilter` object by using its constructor.
- Set the user identifier value of the user by assigning a string value that represents the user identifier value to the `PrincipalSearchFilter` object's `userId` field.
- Invoke the `DirectoryManagerServiceClient` object's `findPrincipals` method and pass the `PrincipalSearchFilter` object. This method returns a `MyArrayOfUser` collection object, where each element is a `User` object. Iterate through the `MyArrayOfUser` collection to locate the user.

See also

"Summary of steps" on page 1015

Quick Start (MTOM): Adding users using the web service API

Quick Start (SwaRef): Adding users using the web service API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Deleting Users

You can use the Directory Manager Service API (Java and web service) to programmatically delete users from AEM Forms. After you delete a user, the user can no longer be used to perform a service operation that requires a user. For example, you cannot assign a task to a deleted user. (See "Assigning Tasks" on page 1089.)

Summary of steps

To delete a user, perform the following steps:

1 Include project files.

2 Create a DirectoryManagerService client.

3 Specify the user to delete.

4 Delete the user from AEM Forms.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create a DirectoryManagerService client

Before you can programmatically perform a Directory Manager Service API operation, create a Directory Manager service client.

Specify the user to delete

You can specify a user to delete by using the user's identifier value.

Delete the user from AEM Forms

To delete a user, invoke the `DirectoryManagerServiceClient` object's `deleteLocalUser` method.

See also

"Delete users using the Java API" on page 1019

"Delete users using the web service API" on page 1020

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Adding Users" on page 1015

Delete users using the Java API

Delete users by using the Directory Manager Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path.

2 Create a DirectoryManagerService client.

Create a `DirectoryManagerServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the user to delete.

- Create a `PrincipalSearchFilter` object by using its constructor.

- Set the user identifier value by invoking the `PrincipalSearchFilter` object's `setUserId` method. Pass a string value that represents the user identifier value.
- Invoke the `DirectoryManagerServiceClient` object's `findPrincipals` method and pass the `PrincipalSearchFilter` object. This method returns a `java.util.List` instance, where each element is a `User` object. Iterate through the `java.util.List` instance to locate the user to delete.

4 Delete the user from AEM Forms.

Invoke the `DirectoryManagerServiceClient` object's `deleteLocalUser` method and pass the value of the `User` object's `oid` field. Invoke the `User` object's `getOid` method. Use the `User` object retrieved from the `java.util.List` instance.

See also

"Summary of steps" on page 1019

"Quick Start (SOAP mode): Deleting users using the Java API" on page 421

Quick Start (SOAP mode): Deleting users using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Delete users using the web service API

Delete users by using the Directory Manager Service API (web service):

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path.

2 Create a DirectoryManagerService client.

- Create a `DirectoryManagerServiceClient` object by using its default constructor.
- Create a `DirectoryManagerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/DirectoryManagerService?blob=mtom`). You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference. Ensure that you specify `blob=mtom`.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `DirectoryManagerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `DirectoryManagerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `DirectoryManagerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Specify the user to delete.

- Create a `PrincipalSearchFilter` object by using its constructor.
- Set the user identifier value by assigning a string value to the `PrincipalSearchFilter` object's `userId` field.
- Invoke the `DirectoryManagerServiceClient` object's `findPrincipals` method and pass the `PrincipalSearchFilter` object. This method returns a `MyArrayOfUser` collection object, where each element is a `User` object. Iterate through the `MyArrayOfUser` collection to locate the user. The `User` object retrieved from the `MyArrayOfUser` collection object is used to delete the user.

4 Delete the user from AEM Forms.

Delete the user by passing the `User` object's `oid` field value to the `DirectoryManagerServiceClient` object's `deleteLocalUser` method.

See also

"Summary of steps" on page 1019

Quick Start (MTOM): Deleting users using the Java API

Quick Start (SwaRef): Deleting users using the Java API

"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Creating Groups

You can use the Directory Manager Service API (Java and web service) to programmatically create AEM Forms groups. After you create a group, you can use that group to perform a service operation that requires a group. For example, you can assign a user to the new group. (See Managing Users and Groups.)

Summary of steps

To create a group, perform the following steps:

1 Include project files.

2 Create a DirectoryManagerService client.

3 Determine that the group does not exist.

4 Create the group.

5 Perform an action with the group.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a DirectoryManagerService client

Before you can programmatically perform a Directory Manager service operation, create a Directory Manager Service API client.

Determine whether the group exists

When you create a group, ensure that the group does not exist in the same domain. That is, two groups cannot have the same name within the same domain. To perform this task, perform a search and filter the search results based on two values. Set the principal type to `com.adobe.idp.um.api.infomodel.Principal.PRINCIPALTYPE_GROUP` to ensure that only groups are returned. Also, sure that you specify the domain name.

Create the group

After you determine that the group does not exist in the domain, create the group and specify the following attributes:

- **CommonName**: The name of the group.
- **Domain**: The domain in which the group is added.
- **Description**: A description of the group.

Perform an action with the group

After you create a group, you can perform an action using the group. For example, you can add a user to the group. To add a user to a group, retrieve the unique identifier value of both the user and the group. Pass these values to the `addPrincipalToLocalGroup` method.

See also

"Create groups using the Java API" on page 1022

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Adding Users" on page 1015

"Deleting Users" on page 1019

Create groups using the Java API

Create a group by using the Directory Manager Service API (Java):

1 Include project files.

   Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path.

2 Create a DirectoryManagerService client.

   Create a `DirectoryManagerServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Determine whether the group exists.

   - Create a `PrincipalSearchFilter` object by using its constructor.
   - Set the principal type by invoking the `PrincipalSearchFilter` object's `setPrincipalType` object. Pass the value `com.adobe.idp.um.api.infomodel.Principal.PRINCIPALTYPE_GROUP`.
   - Set the domain by invoking the `PrincipalSearchFilter` object's `setSpecificDomainName` object. Pass a string value that specifies the domain name.

- To find a group, invoke the `DirectoryManagerServiceClient` object's `findPrincipals` method (a principal can be a group). Pass the `PrincipalSearchFilter` object that specifies the principal type and the domain name. This method returns a `java.util.List` instance where each element is a `Group` instance. Each group instance conforms to the filter specified by using the `PrincipalSearchFilter` object.
- Iterate through the `java.util.List` instance. For each element, retrieve the group name. Ensure that the group name does not equal the new group name.

4 Create the group.

- If the group does not exist, invoke the `Group` object's `setCommonName` method and pass a string value that specifies the group name.
- Invoke the `Group` object's `setDescription` method and pass a string value that specifies the group description.
- Invoke the `Group` object's `setDomainName` method and pass a string value that specifies the domain name.
- Invoke the `DirectoryManagerServiceClient` object's `createLocalGroup` method and pass the `Group` instance.

The `createLocalUser` method returns a string value that specifies the local user identifier value.

5 Perform an action with the group.

- Create a `PrincipalSearchFilter` object by using its constructor.
- Set the user identifier value by invoking the `PrincipalSearchFilter` object's `setUserId` method. Pass a string value that represents the user identifier value.
- Invoke the `DirectoryManagerServiceClient` object's `findPrincipals` method and pass the `PrincipalSearchFilter` object. This method returns a `java.util.List` instance, where each element is a `User` object. Iterate through the `java.util.List` instance to locate the user.
- Add a user to the group by invoking the `DirectoryManagerServiceClient` object's `addPrincipalToLocalGroup` method. Pass the return value of the `User` object's `getOid` method. Pass the return value of the `Group` objects's `getOid` method (use the `Group` instance that represents the new group).

See also

"Summary of steps" on page 1021

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Managing Users and Groups

This topic describes how you can use (Java) to programmatically assign, remove, and query domains, users, and groups.

Note: *When configuring a domain, you must set the unique identifier for groups and users. The attribute that is chosen must not only be unique within the LDAP environment, but must also be immutable and will not change within the directory. This attribute must also be of a simple string data type (the only exception currently allowed for Active Directory 2000/2003 is `"objectsid"`, which is a binary value). The Novell eDirectory attribute `"GUID"`, for example, is not a simple string data type and therefore will not work.*

- For Active Directory, use `"objectsid"`.
- For SunOne, use `"nsuniqueid"`.

Note: *Creating multiple local users and groups while an LDAP directory synchronization is in progress is not supported. Attempting this process may result in errors.*

Summary of steps

To manage users and groups, perform the following steps:

1 Include project files.

2 Create a DirectoryManagerService client.

3 Invoke the appropriate user or group operations.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a DirectoryManagerService client

Before you can programmatically perform a Directory Manager service operation, you must create a Directory Manager service client. With the Java API this is accomplished by creating a `DirectoryManagerServiceClient` object. With the web service API this is accomplished by creating a `DirectoryManagerServiceService` object.

Invoke the appropriate user or group operations

Once you have created the service client, you can then invoke the user or group management operations. The service client allows you to assign, remove, and query domains, user, and groups. Note that it is possible to add either a directory principal or a local principal to a local group, but it is not possible to add a local principal to a directory group.

See also

"Managing users and groups using the Java API" on page 1024

"Managing users and groups using the web service API" on page 1025

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"User Manager Java API Quick Start(SOAP)" on page 419

Managing users and groups using the Java API

To programmatically manage users, groups, and domains using the (Java), perform the following tasks:

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path. For information about the location of these files, see "Including AEM Forms Java library files" on page 491.

2 Create a DirectoryManagerService client.

Create a `DirectoryManagerServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties. For information, see "Setting connection properties" on page 500.

3 Invoke the appropriate user or group operations.

To find a user or group, invoke one of the `DirectoryManagerServiceClient` object's methods for finding principals (since a principal can be a user or a group). In the example below, the `findPrincipals` method is called using a search filter (a `PrincipalSearchFilter` object).

Since the return value in this case is a `java.util.List` containing `Principal` objects, iterate through the result and cast the `Principal` objects to either `User` or `Group` objects.

Using the resultant `User` or `Group` object (which both inherit from the `Principal` interface), retrieve the information you need in your workflows. For example, the domain name and canonical name values, in combination, uniquely identify a principal. These are retrieved by invoking the `Principal` object's `getDomainName` and `getCanonicalName` methods, respectively.

To delete a local user, invoke the `DirectoryManagerServiceClient` object's `deleteLocalUser` method and pass the user's identifier.

To delete a local group, invoke the `DirectoryManagerServiceClient` object's `deleteLocalGroup` method and pass the group's identifier.

See also

"Summary of steps" on page 1024

Quick Start (SOAP): Managing users and groups using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Managing users and groups using the web service API

To programmatically manage users, groups, and domains using the Directory Manager Service API (web service), perform the following tasks:

1 Include project files.

- Create a Microsoft .NET client assembly that consumes the Directory Manager WSDL. (See "Invoking AEM Forms using Base64 encoding" on page 525.)
- Reference the Microsoft .NET client assembly. (See "Creating a .NET client assembly that uses Base64 encoding" on page 525.)

2 Create a DirectoryManagerService client.

Create a `DirectoryManagerServiceService` object by using your proxy class' constructor.

3 Invoke the appropriate user or group operations.

To find a user or group, invoke one of the `DirectoryManagerServiceService` object's methods for finding principals (since a principal can be a user or a group). In the example below, the `findPrincipalsWithFilter` method is called using a search filter (a `PrincipalSearchFilter` object). When using a `PrincipalSearchFilter` object, local principals are only returned if the `isLocal` property is set to `true`. This behavior is different than what would occur with the Java API.

Note: *If the maximum number of results is not specified in the search filter (through the `PrincipalSearchFilter.resultsMax` field), a maximum of 1000 results will be returned. This is different behavior than what occurs using the Java API, in which 10 results is the default maximum. Also, the search methods such as `findGroupMembers` will not yield any results unless the maximum number of results is specified in the search filter (for example, through the `GroupMembershipSearchFilter.resultsMax` field). This applies to all search filters that inherit from the `GenericSearchFilter` class. For more information, see AEM Forms API Reference.*

Since the return value in this case is an `object[]` containing `Principal` objects, iterate through the result and cast the `Principal` objects to either `User` or `Group` objects.

Using the resultant `User` or `Group` object (which both inherit from the `Principal` interface), retrieve the information you need in your workflows. For example, the domain name and canonical name values, in combination, uniquely identify a principal. These are retrieved by invoking the `Principal` object's `domainName` and `canonicalName` fields, respectively.

To delete a local user, invoke the `DirectoryManagerServiceService` object's `deleteLocalUser` method and pass the user's identifier.

To delete a local group, invoke the `DirectoryManagerServiceService` object's `deleteLocalGroup` method and pass the group's identifier.

See also

"Summary of steps" on page 1024

Quick Start (MTOM): Managing users and groups using the web service API

"Invoking AEM Forms using MTOM" on page 529

Managing Roles and Permissions

This topic describes how you can use the Authorization Manager Service API (Java) to programmatically assign, remove, and determine roles and permissions.

In AEM Forms, a *role* is a group of permissions for accessing one or more system-level resources. These permissions are created through User Management and are enforced by the service components. For example, an Administrator could assign the role of "Policy Set Author" to a group of users. Rights Management would then permit the users of that group with that role to create policy sets through administration console.

There are two types of roles: *default roles* and *custom roles*. Default roles (*system roles)* are already resident in AEM Forms. It is assumed that default roles may not be deleted or modified by the administrator, and are thus immutable. Custom roles created by the administrator, who may subsequently modify or delete them, are thus mutable.

Roles make it easier to manage permissions. When a role is assigned to a principal, a set of permissions is automatically assigned to that principal, and all the specific access-related decisions for the principal are based on that overall set of assigned permissions.

Summary of steps

To manage roles and permissions, perform the following steps:

1 Include project files.

2 Create an AuthorizationManagerService client.

3 Invoke the appropriate role or permission operations.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create an AuthorizationManagerService client

Before you can programmatically perform a User Management AuthorizationManagerService operation, you must create an AuthorizationManagerService client. With the Java API this is accomplished by creating an `AuthorizationManagerServiceClient` object.

Invoke the appropriate role or permission operations

Once you have created the service client, you can then invoke the role or permission operations. The service client allows you to assign, remove, and determine roles and permissions.

See also

"Managing roles and permissions using the Java API" on page 1027

"Managing roles and permissions using the web service API" on page 1027

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"User Manager Java API Quick Start(SOAP)" on page 419

Managing roles and permissions using the Java API

To manage roles and permissions using the Authorization Manager Service API (Java), perform the following tasks:

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path.

2 Create an AuthorizationManagerService client.

Create an `AuthorizationManagerServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the appropriate role or permission operations.

To assign a role to a principal, invoke the `AuthorizationManagerServiceClient` object's `assignRole` method and pass the following values:

- A `java.lang.String` object that contains the role identifier
- An array of `java.lang.String` objects containing the principal identifiers.

To remove a role from a principal, invoke the `AuthorizationManagerServiceClient` object's `unassignRole` method and pass the following values:

- A `java.lang.String` object that contains the role identifier.
- An array of `java.lang.String` objects containing the principal identifiers.

See also

"Summary of steps" on page 1026

Quick Start (SOAP mode): Managing roles and permissions using the Java API

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Managing roles and permissions using the web service API

Manage roles and permissions by using the Authorization Manager Service API (web service):

1 Include project files.

Create a Microsoft .NET project that uses MTOM. Ensure that you use the following WSDL definition: `http://localhost:8080/soap/services/AuthorizationManagerService?WSDL&lc_version=9.0.1`.

Note: *Replace* `localhost` *with the IP address of the server hosting AEM Forms.*

2 Create an AuthorizationManagerService client.

- Create an `AuthorizationManagerServiceClient` object by using its default constructor.

- Create an `AuthorizationManagerServiceClient.Endpoint.Address` object by using the `System.ServiceModel.EndpointAddress` constructor. Pass a string value that specifies the WSDL to the AEM Forms service (for example, `http://localhost:8080/soap/services/AuthorizationManagerService?blob=mtom`.) You do not need to use the `lc_version` attribute. This attribute is used when you create a service reference.
- Create a `System.ServiceModel.BasicHttpBinding` object by getting the value of the `AuthorizationManagerServiceClient.Endpoint.Binding` field. Cast the return value to `BasicHttpBinding`.
- Set the `System.ServiceModel.BasicHttpBinding` object's `MessageEncoding` field to `WSMessageEncoding.Mtom`. This value ensures that MTOM is used.
- Enable basic HTTP authentication by performing the following tasks:
  - Assign the AEM forms user name to the field `AuthorizationManagerServiceClient.ClientCredentials.UserName.UserName`.
  - Assign the corresponding password value to the field `AuthorizationManagerServiceClient.ClientCredentials.UserName.Password`.
  - Assign the constant value `HttpClientCredentialType.Basic` to the field `BasicHttpBindingSecurity.Transport.ClientCredentialType`.
  - Assign the constant value `BasicHttpSecurityMode.TransportCredentialOnly` to the field `BasicHttpBindingSecurity.Security.Mode`.

3 Invoke the appropriate role or permission operations.

To assign a role to a principal, invoke the `AuthorizationManagerServiceClient` object's `assignRole` method and pass the following values:

- A `string` object that contains the role identifier
- A `MyArrayOf_xsd_string` object that contains the principal identifiers.

To remove a role from a principal, invoke the `AuthorizationManagerServiceService` object's `unassignRole` method and pass the following values:

- A `string` object that contains the role identifier.
- An array of `string` objects containing the principal identifiers.

See also

"Summary of steps" on page 1026

Quick Start (MTOM): Managing roles and permissions using the web service API

"Invoking AEM Forms using MTOM" on page 529

Authenticating Users

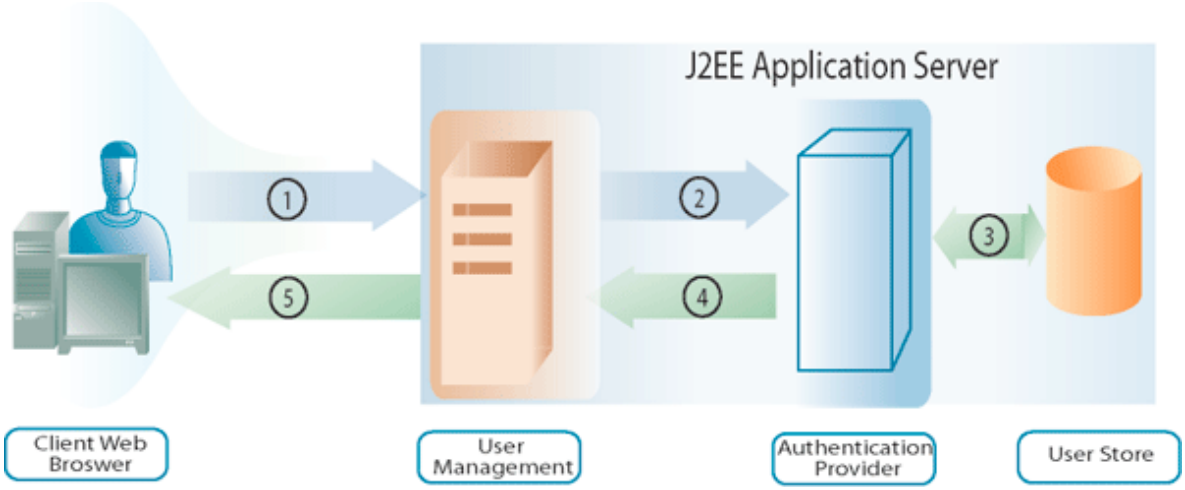
This topic describes how you can use the Authentication Manager Service API (Java) to enable your client applications to programmatically authenticate users.

User authentication may be required to interact with an enterprise database or other enterprise repositories that store secure data.

Consider, for example, a scenario where a user enters a user name and password into a web page and submits the values to a J2EE application server hosting Forms. A Forms custom application can authenticate the user with the Authentication Manager service.

If the authentication is successful, the application accesses a secured enterprise database. Otherwise, a message is sent to the user stating that the user is not an authorized user.

The following diagram shows the application's logic flow.



The following table describes the steps in this diagram

Step	Description
1	The user accesses a web site and specifies a user name and password. This information is submitted to a J2EE application server hosting AEM Forms.
2	The user credentials are authenticated with the Authentication Manager service. If the user credentials are valid, the workflow proceeds to step 3. Otherwise, a message is sent to the user stating that the user is not an authorized user.
3	User information and a form design are retrieved from a secured enterprise database.
4	User information is merged with a form design and the form is rendered to the user.

Summary of steps

To programmatically authenticate a user, perform the following steps:

1. Include project files.
2. Create an AuthenticationManagerService client.
3. Invoke the authentication operation.
4. If necessary, retrieve the context so that the client application can forward it to another AEM Forms service for authentication.

Include project files

Include necessary files in your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create an AuthenticationManagerService client

Before you can programmatically authenticate a user, you must create a AuthenticationManagerService client. When using the Java API, create an `AuthenticationManagerServiceClient` object.

Invoke the authentication operation

Once you have created the service client, you can then invoke the authentication operation. This operation will need information about the user, such as the user's name and password. If the user does not authenticate, an exception is thrown.

Retrieve the authentication context

Once you have authenticated the user, you can create a context based in the authenticated user. Then you can use the content to invoke another AEM Forms services. For example, you can use the context to create an `EncryptionServiceClient` and encrypt a PDF document with a password. Ensure that the user that was authenticated has the role named `Services User` that is required to invoke a AEM Forms service.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"User Manager Java API Quick Start(SOAP)" on page 419

"Encrypting PDF Documents with a Password" on page 806

Authenticate a user using the Java API

Authenticate a user using the Authentication Manager Service API (Java):

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar, in your Java project's class path.

2 Create an AuthenticationManagerServices client.

Create an `AuthenticationManagerServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Invoke the authentication operation.

Invoke the `AuthenticationManagerServiceClient` object's `authenticate` method and pass the following values:

- A `java.lang.String` object that contains the user's name.
- A byte array (a `byte[]` object) containing the user's password. You can obtain the `byte[]` object by invoking the `java.lang.String` object's `getBytes` method.

The authenticate method returns an `AuthResult` object, which contains information about the authenticated user.

4 Retrieve the authentication context.

Invoke the `ServiceClientFactory` object's `getContext` method, which will return a `Context` object.

Then invoke the `Context` object's `initPrincipal` method and pass the `AuthResult`.

Authenticate a user using the web service API

Authenticate a user using the Authentication Manager Service API (web service):

1 Include project files.

- Create a Microsoft .NET client assembly that consumes the Authentication Manager WSDL. (See "Invoking AEM Forms using Base64 encoding" on page 525.)

- Reference the Microsoft .NET client assembly. (See "Referencing the .NET client assembly" in "Invoking AEM Forms using Base64 encoding" on page 525.)

2 Create an AuthenticationManagerService client.

Create a `AuthenticationManagerServiceService` object by using your proxy class' constructor.

3 Invoke the authentication operation.

Invoke the `AuthenticationManagerServiceClient` object's `authenticate` method and pass the following values:

- A `string` object that contains the user's name
- A byte array (a `byte[]` object) containing the user's password. You can obtain the `byte[]` object by converting a `string` object containing the password to a `byte[]` array using the logic shown in the example below.
- The returned value will be an `AuthResult` object, which can be used to retrieve information about the user. In the example below, the user's information is retrieved by first obtaining the `AuthResult` object's `authenticatedUser` field and subsequently obtaining the resultant `User` object's `canonicalName` and `domainName` fields.

See also

Quick Start (MTOM): Removing a digital signature using the web service API

Quick Start (SwaRef): Removing a digital signature using the web service API

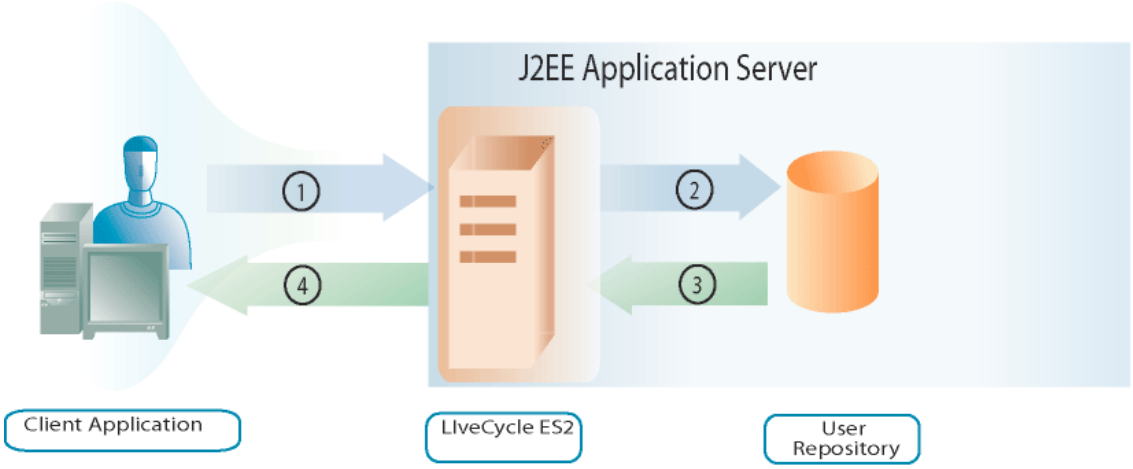
"Invoking AEM Forms using MTOM" on page 529

"Invoking AEM Forms using SwaRef" on page 531

Programmatically Synchronizing Users

You can programmatically synchronize users by using the User Management API. When you synchronize users, you are updating AEM Forms with user data that is located in your user repository. For example, assume that you add new users to your user repository. After you perform a synchronization operation, the new users become AEM forms users. As well, users no longer in your user respository are removed from AEM Forms.

The following diagram shows AEM Forms synchronizing with a user respository.



The following table describes the steps in this diagram

Step	Description
1	A client application requests that AEM Forms performs a synchronization operation.
2	AEM Forms performs a synchronization operation.
3	User information is updated.
4	A user is able to view the updated user information.

Summary of steps

To programmatically synchronize users, perform the following steps:

1. Include project files.
2. Create a UserManagerUtilServiceClient client.
3. Specify the enterprise domain.
4. Invoke the authentication operation.
5. Determine if the synchronization operation is complete

Include project files

Include necessary files in your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create a UserManagerUtilServiceClientclient

Before you can programmatically synchronize users, you must create a `UserManagerUtilServiceClient` object.

Specify the enterprise domain

Before you perform a synchronization operation by using the User Management API, you specify the enterprise domain to which users belong. You can specify one or many enterprise domains. Before you can programmatically perform a synchronization operation, you have to setup an enterprise domain using Administration Console. (See administration help.)

Invoke the synchronization operation

After you specify one or more enterprise domains, you can perform the synchronization operation. The time it takes to perform this operation depends upon the number of user records that are located in the user repository.

Determine if the synchronization operation is complete

After you programmatically perform a synchronization operation, you can detemine if the operation is complete.

See also

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"User Manager Java API Quick Start(SOAP)" on page 419

"Encrypting PDF Documents with a Password" on page 806

Programmatically synchronizing users using the Java API

Synchronize users by using the User Management API (Java):

1 Include project files.

Include client JAR files, such as adobe-usermanager-client.jar and adobe-usermanager-util-client.jar, in your Java project's class path.

2 Create a UserManagerUtilServiceClient client.

Create a `UserManagerUtilServiceClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the enterprise domain.

- Invoke the `UserManagerUtilServiceClient` object's `scheduleSynchronization` method to start the user synchronization operation.
- Create a `java.util.Set` instance by using a `HashSet` constructor. Ensure that you specify `String` as the data type. This `Java.util.Set` instance stores the domain names to which the synchronization operation applies.
- For each domain name to add, invoke the `java.util.Set` object's add method and pass the domain name.

4 Invoke the synchronization operation.

Invoke the `ServiceClientFactory` object's `getContext` method, which will return a `Context` object.

Then invoke the `Context` object's `initPrincipal` method and pass the `AuthResult`.

See also

"Programmatically Synchronizing Users" on page 1031

"Including AEM Forms Java library files" on page 491

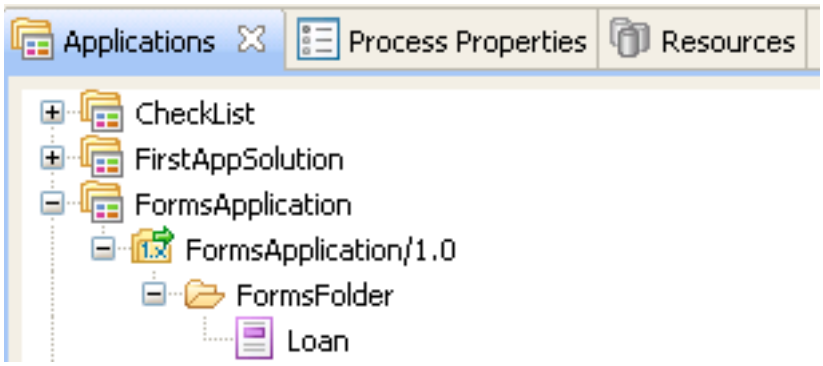
"Setting connection properties" on page 500

Working with AEM Forms Repository

About the Repository Service

The Repository service provides resource storage and management services to AEM Forms. When developers create an *AEM Forms* application, they can deploy the assets in the repository instead of the file system. The assets can include any type of collateral, including XML forms, PDF forms (including Acrobat forms), form fragments, images, profiles, policies, SWF files, DDX files, XML schemas, WSDL files, and test data.

For example, consider the following Forms application named *Applications/FormsApplication*:



Notice that there is a file named Loan.xdp located in the FormsFolder. To access this form design, you specify the complete path (including version): `Applications/FormsApplication/1.0/FormsFolder/Loan.xdp`.

Note: *For information about creating a Forms application using Workbench, see Workbench Help.*

The path to a resource located in the AEM Forms repository is:

```
Applications/Application-name/Application-version/Folder.../Filename
```

The following values show some examples of URI values:

- Applications/AppraisalReport/1.0/Forms/FullForm.xdp
- Applications/AnotherApp/1.1/Assets/picture.jpg
- Applications/SomeApp/2.0/Resources/Data/XSDs/MyData.xsd

You can browse the AEM Forms Repository by using a web browser. To browse the repository, enter the following URL into a web browser http://[server name]:[server port]/repository. You can verify quick start results that are associated with the Working with AEM Forms Repository section by using a web browser. For example, if you add content to the AEM Forms Repository, you can see the content in a web browser. (See "Quick Start (SOAP mode): Writing a resource using the Java API" on page 307.)

The repository API provides a number of operations that you can use to store and retrieve information from the repository. For example, you can obtain a list of resources or retrieve specific resources that are stored in the repository when a resource is needed as part of processing an application.

Note: *The repository API cannot be used to interact with Content Services (deprecated). To interact with Content Services (deprecated), you use the Document Management API. (See Performing Document Management Operations Using APIs)*

Using the Repository service API, you can accomplish the following tasks:

- Create folders. See "Creating Folders" on page 1035.
- Write resources and their properties. See "Writing Resources" on page 1037.
- List resources in a given collection or related to other resources. See "Listing Resources" on page 1041.
- Read resources and their properties. See "Reading Resources" on page 1043.
- Update resources and their properties. See "Updating Resources" on page 1045.
- Search for resources, including their history, related resources, and properties. See "Searching for Resources" on page 1048.
- Specify relationships between resources. See "Creating Resource Relationships" on page 1051.
- Manage resource access control, including locking and unlocking resources, and reading and writing access control lists (ACLs). See Controlling Access to Resources and "Locking Resources" on page 1054.
- Delete resources and their properties. See "Deleting Resources" on page 1058.

Note: *Using the repository API, you cannot manage resource access control, search for resources, or specify resource relationships by using an ECM repository.*

Important: *When an encrypted PDF is written to the repository, the automated relationship extraction feature cannot be used. Otherwise, an encrypted PDF can be stored in the repository and later retrieved. The retriever can choose to decrypt the PDF after it is retrieved from the repository.*

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Creating Folders

Folders (resource collections) are used to store objects (files or resources) in organized groupings. Folders can contain resources and other folders, also known as subfolders. Resources can only be stored in one folder at a time.

Files inherit access control lists (ACLs) from folders, and subfolders inherit ACLs from their parent folders. Therefore, the parent folders must exist before you can create child folders. The IDE lets you interact only on a folder-by-folder basis, not on a file-by-file basis. You cannot version folders and there is no need to do so; a folder does not contain data itself. Rather, it is only a container for resources that contain data. The default ACL is system-level permission, which means that users must have system-level permissions (read, write, traverse, managing ACLs) until someone gives them permissions for a particular folder. ACLs only work in the IDE.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To create a folder, follow these steps:

1 Include project files.

2 Create the service client.

3 Create the folder.

4 Write the folder to the repository.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically create a resource collection, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Create the folder

Invoke the Repository service method to create the resource collection and populate the resource collection with identifying information, including its UUID, folder name, and description.

Write the folder to the repository

Invoke the Repository service method to write the resource collection, specifying the target folder's URI.

See also

"Create folders using the Java API" on page 1036

"Create folders using the web service API" on page 1036

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Create folders using the Java API

Create a folder by using the Repository service API (Java):

1 Include project files

Include project files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Create the folder

To create a resource collection, you must first create a `com.adobe.repository.infomodel.bean.RepositoryInfomodelFactoryBean` object.

Invoke the `repositoryInfomodelFactoryBean` object's `newResourceCollection` method, and pass in the following parameters:

- A `com.adobe.repository.infomodel.Id` UUID identifier to be assigned to the resource.
- A `com.adobe.repository.infomodel.Lid` UUID identifier to be assigned to the resource.
- A `java.lang.String` containing the name of the resource collection. For example, `FormsFolder`.

The method returns a `com.adobe.repository.infomodel.bean.ResourceCollection` object representing the new folder.

Set the folder's description by using the `setDescription` method and pass in the following parameter:

- A `String` that describes the resource collection. In this example, `"test Folder"` is used .

4 Write the folder to the repository

Invoke the `ResourceRepositoryClient` object's `writeResource` method and pass in the URI of the folder and the `ResourceCollection` object. For example, the URI to the folder can be the following value `/Applications/FormsApplication/1.0/`.

The method returns an instance of the newly created `com.adobe.repository.infomodel.bean.Resource` object. You can, for example, retrieve the identifier value of the new resource by invoking the `com.adobe.repository.infomodel.bean.Resource` object's `getId` method.

See also

"Creating Folders" on page 1035

"Quick Start (SOAP mode): Creating a folder using the Java API" on page 305

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Create folders using the web service API

Create a folder by using the Repository service API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL using base64.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object that contains the user name and password.

3 Create the folder

Create the folder by using the default constructor for the `ResourceCollection` class and pass in the following parameters:

- An `Id` object, which is created by invoking the default constructor for the `Id` class and assigned to the `Resource` object's `id` field.
- An `Lid` object, which is created by invoking the default constructor for the `Lid` class and assigned to the `Resource` object's `lid` field.
- A string containing the name of the resource collection, which is assigned to the `Resource` object's `name` field. The name used in this example is `"testfolder"`.
- A string containing the description of the resource collection, which is assigned to the `Resource` object's `description` field. The description used in this example is `"test folder"`.

4 Write the folder to the repository

Invoke the `RepositoryServiceService` object's `writeResource` method and pass in the following parameters:

- The path where the folder is to be created.
- The `ResourceCollection` object representing the folder.
- Pass `null` for the other two parameters.

See also

"Creating Folders" on page 1035

Quick Start (Base64): Creating a folder using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Writing Resources

You can create resources in a given location in the repository. The natural file size is subject to database limitations and session time-out. For the default configuration, files are limited to 25 MB. To raise or lower the maximum file size, you must change the database configuration.

Writing resources is equivalent to storing data in the repository. Once you write a resource to the repository, it becomes accessible to all clients in the repository ecosystem. When you write resources, such as XML schemas, XDP files, and XSD files, to the repository, the contents are parsed based on the MIME type. If the MIME type is supported, the parser determines whether there is an implied relationship to other content. For example, if a cascading style sheet (CSS) has a relative URL that references a common CSS, it is expected that you will submit the common CSS into the repository as well. The relationship between the two resources is stored as a pending relationship for a non-adjustable period of 30 days. When you submit the common CSS to the repository within the 30-day period, the relationship is formed.

When you create a resource, the access control list (ACL) is inherited from the parent folder. The root folder has system-level permissions until an initial resource or folder is created, at which point the resource or folder is given default ACL permissions.

You can programmatically write resources by using the Repository service Java API or web service API.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To write a resource, follow these steps:

1 Include project files.

2 Create a Repository service client.

3 Specify the URI of the resource to be read.

4 Read the resource.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the URI of the target folder for the resource

Create a string containing the URI of the resource to be read. The syntax includes forward slashes, as in this example: "*/path/folder*".

Create the resource

Invoke the Repository service method to create the resource, and populate the resource with identifying information, including its UUID, resource name, and description.

Specify the resource content

Invoke the Repository service method to create resource content, and store that content in the resource.

Write the resource to the target folder

Invoke the Repository service method to write the resource, specifying the target folder's URI.

See also

"Write resources using the Java API" on page 1038

"Write resources using the web service API" on page 1040

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Write resources using the Java API

Write a resource by using the Repository service API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the URI of the target folder for the resource

Specify the URI of the target folder for the resource. In this case, because the resource named `testResource` will be stored in the folder named `testFolder`, the folder's URI is `"/testFolder"`. The URI is stored as a `java.lang.String` object.

4 Create the resource

To create a resource, you must first create a `com.adobe.repository.infomodel.bean.RepositoryInfomodelFactoryBean` object.

Invoke the `RepositoryInfomodelFactoryBean` object's `newResource` method, which creates a `com.adobe.repository.infomodel.bean.Resource` object. In this example, the following parameters are provided:

- A `com.adobe.repository.infomodel.Id` object, which is created by invoking the default constructor for the `Id` class.
- A `com.adobe.repository.infomodel.Lid` object, which is created by invoking the default constructor for the `Lid` class.
- A `java.lang.String` containing the file name of the resource.

To specify the resource's description, invoke the `Resource` object's `setDescription` method and pass a string containing the description. In this example, the description is `"test resource"`.

5 Specify the resource content

To create content for the resource, invoke the `RepositoryInfomodelFactoryBean` object's `newResourceContent` method, which returns a `com.adobe.repository.infomodel.bean.ResourceContent` object. Add content to the `ResourceContent` object. In this example, this is accomplished by doing the following tasks:

- Invoking the `ResourceContent` object's `setDataDocument` method and passing in a `com.adobe.idp.Document` object
- Invoking the `ResourceContent` object's `setSize` method and passing in the size in bytes of the `Document` object

Add the content to the resource by invoking the `Resource` object's `setContent` method and passing in the `ResourceContent` object. For more information, see AEM Forms API Reference.

6 Write the resource to the target folder

Invoke the `ResourceRepositoryClient` object's `writeResource` method and pass in the URI of the folder, as well as the `Resource` object.

See also

"Writing Resources" on page 1037

"Quick Start (SOAP mode): Writing a resource using the Java API" on page 307

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Write resources using the web service API

Write a resource by using the Repository service API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL using base64.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Specify the URI of the target folder for the resource

Specify the URI of the target folder for the resource. In this case, because the resource named `testResource` will be stored in the folder named `testFolder`, the folder's URI is `"/testFolder"`. When using a language compliant with the Microsoft .NET Framework (for example, C#), store the URI in a `System.String` object.

4 Create the resource

To create a resource, invoke the default constructor for the `Resource` class. In this example, the following information is stored in the `Resource` object:

- A `com.adobe.repository.infomodel.Id` object, which is created by invoking the default constructor for the `Id` class and assigned to the `Resource` object's `id` field.
- A `com.adobe.repository.infomodel.Lid` object, which is created by invoking the default constructor for the `Lid` class and assigned to the `Resource` object's `lid` field.
- A string containing the file name of the resource, which is assigned to the `Resource` object's `name` field. The name used in this example is `"testResource"`.
- A string containing the description of the resource, which is assigned to the `Resource` object's `description` field. The description used in this example is `"test resource"`.

5 Specify the resource content

To create content for the resource, invoke the default constructor for the `ResourceContent` class. Then add content to the `ResourceContent` object. In this example, this is accomplished by doing the following tasks:

- Assigning a `BLOB` object containing a document to the `ResourceContent` object's `dataDocument` field.
- Assigning the size in bytes of the `BLOB` object to the `ResourceContent` object's `size` field.

Add the content to the resource by assigning the `ResourceContent` object to the `Resource` object's `content` field.

6 Write the resource to the target folder

Invoke the `RepositoryServiceService` object's `writeResource` method and pass in the URI of the folder, as well as the `Resource` object. Pass `null` for the other two parameters.

See also

"Writing Resources" on page 1037

Quick Start (Base64): Writing a resource using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Listing Resources

You can discover resources by listing resources. A query is performed against the repository to find all the resources that are related to a given resource collection.

Once you organize your resources, you can inspect the structure you created by seeing a particular branch of the structure, much like you would do in an operating system.

Listing resources operates by relationship: resources are members of folders. Membership is represented by a relationship of type "member of". When you list resources in a given folder, you are querying for resources that are related to a given folder by the relationship "member of". Relationships are directional: a member of a relationship has a source that is a member of the target. The source is the resource; the target is the parent folder.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To list resources, follow these steps:

1. Include project files.
2. Create the service client.
3. Specify the folder path.
4. Retrieve the list of resources.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically create a resource collection, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the folder path

Create a string containing the path of the folder containing the resources. The syntax includes forward slashes, as in this example: "/*path*/*folder*".

Retrieve the list of resources

Invoke the Repository service method to retrieve the list of resources, specifying the target folder's path.

See also

"List resources using the Java API" on page 1042

"List resources using the web service API" on page 1042

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

List resources using the Java API

List resources by using the Repository service API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the folder path

Specify the URI of the resource collection to be queried. In this case, its URI is `"/testFolder"`. The URI is stored as a `java.lang.String` object.

4 Retrieve the list of resources

Invoke the `ResourceRepositoryClient` object's `listMembers` method and pass in the URI of the folder.

The method returns a `java.util.List` of `com.adobe.repository.infomodel.bean.Resource` objects that are the source of a `com.adobe.repository.infomodel.bean.Relation` of type `Relation.TYPE_MEMBER_OF` and have the resource collection URI as the target. You can iterate through this `List` to retrieve each of the resources. In this example, the name and description of each resource is displayed.

See also

"Listing Resources" on page 1041.

"Quick Start (SOAP mode): Listing resources using the Java API" on page 309

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

List resources using the web service API

List resources by using the Repository service API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Specify the folder path

Specify a string containing the URI of the folder to be queried. In this case, its URI is `"/testFolder"`. When using a language that is compliant with the Microsoft .NET Framework (for example, C#), store the URI in a `System.String` object.

4 Retrieve the list of resources

Invoke the `RepositoryServiceService` object's `listMembers` method and pass in the URI of the folder as the first parameter. Pass `null` for the other two parameters.

The method returns an array of objects that can be cast to `Resource` objects. You can iterate through the object array to retrieve each of the related resources. In this example, the name and description of each resource is displayed.

See also

"Listing Resources" on page 1041.

Quick Start (Base64): Listing resources using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Reading Resources

You can retrieve resources from a given location in the repository in order to read their content and metadata. The workflow is front-ended by an initialization form. The process has all the permissions it needs to read the form. The system retrieves the data form and reads the content from the repository. The repository grants access to the content and the metadata (the ability to even know the resource exists).

The repository has the following four permission types:

- **traverse**: allows you to list resources; that is, to read resource metadata, but not resource content
- **read**: allows you to read resource content
- **write**: allows you to write resource content
- **managing access control lists (ACLs)**: allows you to manipulate ACLs on resources

Users can only run processes when they have permission to run the process. IDE users need traverse and read permissions to synchronize with the repository. ACLs apply only at design time because runtime occurs within the system context.

You can programmatically read resources by using the Repository service Java API or web service API.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To read a resource, follow these steps:

1. Include project files.
2. Create a Repository service client.
3. Specify the URI of the resource to be read.
4. Read the resource.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the URI of the resource to be read

Create a string containing the URI of the resource to be read. The syntax includes forward slashes, as in this example: "/*path*/*resource*".

Read the resource

Invoke the Repository service method to read the resource, specifying the URI.

See also

"Read resources using the Java API" on page 1044

"Reading resources using the web service API" on page 1044

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Read resources using the Java API

Read a resource by using the Repository service API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the URI of the resource to be read

Specify a string value that represents the URI of the resource to retrieve. For example, assuming the resource is named *testResource* which is located in a folder named *testFolder*, specify `/testFolder/testResource`.

4 Read the resource

Invoke the `ResourceRepositoryClient` object's `readResource` method and pass the URI of the resource as a parameter. This method returns a `Resource` instance that represents the resource.

See also

"Reading Resources" on page 1043

"Quick Start (SOAP mode): Reading a resource using the Java API" on page 311

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Reading resources using the web service API

Read a resource by using the Repository service API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL. (See "Creating a .NET client assembly that uses Base64 encoding" on page 525.)
- Reference the Microsoft .NET client assembly. (See "Creating a .NET client assembly that uses Base64 encoding" on page 525.)

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Specify the URI of the resource to be read

Specify a string containing the URI of the resource to be retrieved. In this case, because the resource named `testResource` is in the folder named `testFolder`, its URI is `"/testFolder/testResource"`. When using a language compliant with the Microsoft .NET Framework (for example, C#), store the URI in a `System.String` object.

4 Read the resource

Invoke the `RepositoryServiceService` object's `readResource` method and pass the URI of the resource as the first parameter. Pass `null` for the other two parameters.

See also

"Reading Resources" on page 1043

Quick Start (Base64): Reading a resource using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Updating Resources

You can retrieve and update the content of resources in the repository. When you update resources, access control to those resources remains unchanged between versions. When performing an update, you have the option of incrementing the major version. If you do not choose to increment the major version, the minor version is automatically updated.

When you update a resource, the new version is created based on the specified resource attributes. When you update a resource you specify two important parameters: the target URI and a resource instance containing all the updated metadata. It is important to note that if you are not changing a given attribute (for example, the name), the attribute is still required in the instance you pass in. The relationships that are created when parsing the content are added to the specific version and are not brought forward unless specified.

For example, if you update an XDP file and it contains references to other resources, those additional references will also be recorded. Suppose that form.xdp version 1.0 has two external references: a logo and a style sheet, and you subsequently update form.xdp so that it now has three references: a logo, a style sheet, and a schema file. During the update, the repository will add the third relationship (to the schema file) to its pending relation table. Once the schema file is present in the repository, the relationship will automatically be formed. However, if form.xdp version 2.0 no longer uses the logo, form.xdp version 2.0 will not have a relationship to the logo.

All update operations are atomic and transactional. For example, if two users read the same resource and both decide to update version 1.0 to version 2.0, one of them will succeed and one of them will fail, the integrity of the repository will be maintained, and both will get a message confirming success or failure. If the transaction does not commit, it will roll back in the case of database failure and will time out or roll back depending on the application server.

You can programmatically update resources by using the Repository service Java API or web service API.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To update a resource, follow these steps:

1 Include project files.

2 Create a Repository service client.

3 Retrieve the resource to be updated.

4 Update the resource.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Retrieve the resource to be updated

Read the resource. For more information, see "Reading Resources" on page 1043.

Update the resource

Set the new information in the resource and invoke the Repository service method to update the resource, specifying the URI, the updated resource, and how the version information should be updated.

See also

"Update resources using the Java API" on page 1046

"Update resources using the web service API" on page 1047

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Update resources using the Java API

Update a resource by using the Repository service API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Retrieve the resource to be updated

Specify the URI of the resource to retrieve and read the resource. In this example, the URI of the resource is `"/testFolder/testResource"`.

4 Update the resource

Update the `Resource` object's information. In this example, to update the description, invoke the `Resource` object's `setDescription` method and pass the new description string as a parameter.

Then invoke the `ServiceClientFactory` object's `updateResource` method, and pass in the following parameters:

- A `java.lang.String` object containing the resource's URI.
- The `Resource` object containing the updated resource information.
- A `boolean` value indicating whether to update the major or minor version. In this example, a value of `true` is passed in to indicate that the major version is to be incremented.

See also

"Updating Resources" on page 1045

"Quick Start (SOAP mode): Updating a resource using the Java API" on page 313

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Update resources using the web service API

Update a resource by using the Repository API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Retrieve the resource to be updated

Specify the URI of the resource to be retrieved and read the resource. In this example, the URI of the resource is `"/testFolder/testResource"`. For more information, see "Reading Resources" on page 1043.

4 Update the resource

Update the `Resource` object's information. In this example, to update the description, assign a new value to the `Resource` object's `description` field.

5 Invoke the `RepositoryServiceService` object's `updateResource` method, and pass in the following parameters:

- A `System.String` object containing the resource's URI.
- The `Resource` object containing the updated resource information.
- A `boolean` value indicating whether to update the major or minor version. In this example, a value of `true` is passed in to indicate that the major version is to be incremented.
- Pass `null` for the remaining two parameters.

See also

"Updating Resources" on page 1045

Quick Start (Base64): Updating a resource using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Searching for Resources

You can construct queries used to search for resources in the repository, including history, related resources, and properties.

You can retrieve related resources to determine dependencies between a form and its fragments. For example, if you have a form you can determine what fragments or external resources it uses. If you have an image, you can also find out what forms use the image. You can also search for related resources using filtering based on properties. For example, you can search for all forms that use an image with a specified name, or find any image used by a form with a specified name. You can also search using resource properties. For example, you can conduct a query to find all forms or resources whose name starts with a given string that may include '%' and '_' wildcards. Remember that searches based on properties are not based on relationships; such searches are based on the assumption that you have specific knowledge about a given resource.

Query statements

A *query* contains one or more statements that are logically joined with conditions. A *statement* consists of a left operand, an operator, and a right operand. In addition, you can specify the sort order to be used for the search results. The *sort order* contains information equivalent to an SQL `ORDER BY` clause and is comprised of elements that contain the attributes on which the search was based as well as a value indicating whether ascending or descending order is to be used.

You can programmatically search for resources by using the Repository service Java API. At this time, it is not possible to use the web service API to search for resources.

Sort behaviour

Sort order is not respected when invoking the `ResourceRepositoryClient` object's `searchProperties` method and specifying a sort order. For example, assume that you create a resource with three custom properties, where attribute names are `name`, `secondName`, and `asecondName`. Next you create a sort order element on the attribute name and set the `ascending` value to `true`.

Then you invoke the `ResourceRepositoryClient` object's `searchProperties` method and pass in the sort order. The search returns the right resource, with the three properties. However, the properties are not sorted by attribute name. They are returned in the order they were added: `name`, `secondName`, and `asecondName`.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To search for resources, follow these steps:

1. Include project files.
2. Create a Repository service client.
3. Specify the target folder for the search.
4. Specify the attributes used in the search.
5. Create the query used in the search.
6. Create the sort order for the search results.
7. Search for the resources.
8. Retrieve the resources from the search result.

Include project files
Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client
Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the target folder for the search
Create a string containing the base path from which to conduct the search. The syntax includes forward slashes, as in this example: "/*path*/*folder*".

Specify the attributes used in the search
You can base your search on the attributes contained within resources. Specify the values of the attributes on which to conduct the search.

Create the query used in the search
Construct a query by using statements and conditions. Each statement will specify the attribute on which to base the search, the condition to be used, and the attribute value to be used in the search.

Create the sort order for the search results
The sort order is comprised of elements, each of which contains one of the attributes used in the search and a value indicating whether ascending or descending order is to be used.

Search for the resources
Search for the resources using the folder, query, and sort order. In addition, indicate the depth of the search and an upper limit on the number of results to be returned.

Retrieve the resources from the search result
Iterate through the returned list of resources and extract the information for further processing.

See also
"Search for resources using the Java API" on page 1049

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Search for resources using the Java API

Search for a resource by using the Repository service API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the target folder for the search

Specify the URI of the base path from which to execute the search. In this example, the URI of the resource is `/testFolder`.

4 Specify the attributes used in the search

Specify the values for the attributes on which to conduct the search. The attributes exist within a `com.adobe.repository.infomodel.bean.Resource` object. In this example, the search will be conducted on the name attribute; therefore, a `java.lang.String` containing the `Resource` object's name is used, which is `testResource` in this case.

5 Create the query used in the search

To create a query, create a `com.adobe.repository.query.Query` object by invoking the default constructor for the `Query` class and add statements to the query.

To create a statement, invoke the constructor for the `com.adobe.repository.query.Query.Statement` class and pass in the following parameters:

- A left operand containing the resource attribute constant. In this example, because the resource's name is used as the basis for the search, the static value `Resource.ATTRIBUTE_NAME` is used.
- An operator containing the condition used in the search for the attribute. The operator must be one of the static constants in the `Query.Statement` class. In this example, the static value `Query.Statement.OPERATOR_BEGINS_WITH` is used.
- A right operand containing the attribute value on which to conduct the search. In this example, the name attribute, a `String` containing the value `"testResource"`, is used.

Specify the namespace of the left operand by invoking the `Query.Statement` object's `setNamespace` method and passing in one of the static values contained in the `com.adobe.repository.infomodel.bean.ResourceProperty` class. In this example, `ResourceProperty.RESERVED_NAMESPACE_REPOSITORY` is used.

Add each statement to the query by invoking the `Query` object's `addStatement` method and passing in the `Query.Statement` object.

6 Create the sort order for the search results

To specify the sort order used in the search results, create a `com.adobe.repository.query.sort.SortOrder` object by invoking the default constructor for the `SortOrder` class, and add elements to the sort order.

To create an element for the sort order, invoke one of the constructors for the `com.adobe.repository.query.sort.SortOrder.Element` class. In this example, because the resource's name is used as the basis for the search, the static value `Resource.ATTRIBUTE_NAME` is used as the first parameter, and ascending order (a `boolean` value of `true`) is specified as the second parameter.

Add each element to the sort order by invoking the `SortOrder` object's `addSortElement` method and passing in the `SortOrder.Element` object.

7 Search for the resources

To search for `resources` based on attribute properties, invoke the `ResourceRepositoryClient` object's `searchProperties` method and pass in the following parameters:

- A `String` containing the base path from which to execute the search. In this case, `"/testFolder"` is used.
- The query used in the search.

- The depth of the search. In this case, `com.adobe.repository.infomodel.bean.ResourceCollection.DEPTH_INFINITE` is used to indicate that the base path and all its folders are to be used.
- An `int` value indicating the first row from which to select the unpaged result set. In this example, `0` is specified.
- An `int` value indicating the maximum number of results to be returned. In this example, `10` is specified.
- The sort order used in the search.

The method returns a `java.util.List` of `Resource` objects in the specified sort order.

8 Retrieve the resources from the search result

To retrieve the resources contained in the search result, iterate through the `List` and cast each object to a `Resource` in order to extract its information. In this example, the name of each resource is displayed.

See also

"Searching for Resources" on page 1048

"Quick Start (SOAP mode): Searching for resources using the Java API" on page 316

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Creating Resource Relationships

You can specify relationships between resources in the repository. There are three kinds of relationships:

- **Dependence**: a relationship in which a resource depends on other resources, meaning that all related resources are needed in the repository.
- **Membership (file system)**: a relationship in which a resource is located within a given folder.
- **Custom**: a relationship you specify between resources. For example, if one resource has been deprecated and another resource introduced into the repository, you could specify your own replacement relationship.

You can create your own custom relationships. For example, if you store an HTML file in the repository and it uses an image, you could specify a custom relationship to relate the HTML file with the image (since normally only XML files are associated with images using a repository-defined dependence relationship). Another example of a custom relationship would be if you wanted to build a different view of the repository with a cyclical graph structure instead of a tree structure. You could define a circular graph along with a viewer to traverse those relationships. Finally, you could indicate that a resource replaces another resource even though the two resources are completely different. In that case you could define a relationship type outside of the reserved range and create a relationship between those two resources. Your application would be the only client that could detect and process the relationship, and it could be used to conduct searches on that relationship.

You can programmatically specify relationships between resources by using the Repository service Java API or web service API.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To specify a relationship between two resources, follow these steps:

1 Include project files.

2 Create a Repository service client.

3 Specify the URIs of the resources to be related.

4 Create the relationship.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the URIs of the resources to be related

Create strings containing the URIs of the resource to be related. The syntax includes forward slashes, as in this example: "/*path*/*resource*".

Create the relationship

Invoke the Repository service method to create and specify the type of relationship.

See also

"Create relationship resources using the Java API" on page 1052

"Create relationship resources using the web service API" on page 1053

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Create relationship resources using the Java API

Create relationship resources by using the Repository service Java API, perform the following tasks:

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the URIs of the resources to be related

Specify the URIs of the resources to be related. In this case, because the resources are named `testResource1` and `testResource2` and are located in the folder named `testFolder`, their URIs are `"/testFolder/testResource1"` and `"/testFolder/testResource2"`. The URIs are stored as a `java.lang.String` objects. In this example, the resources are first written to the repository, and their URIs are retrieved. For more information about writing a resource, see "Writing Resources" on page 1037.

4 Create the relationship

Invoke the `ResourceRepositoryClient` object's `createRelationship` method and pass in the following parameters:

- The URI of the source resource.
- The URI of the target resource.

- The type of relationship, which is one of the static constants in the `com.adobe.repository.infomodel.bean.Relation` class. In this example, a dependence relationship is established by specifying the value `Relation.TYPE_DEPENDANT_OF`.
- A `boolean` value indicating whether the target resource is automatically updated to the `com.adobe.repository.infomodel.Id`-based identifier of the new head resource. In this example, because of the dependence relationship, the value `true` is specified.

You can also retrieve a list of related resources for a given resource by invoking the `ResourceRepositoryClient` object's `getRelated` method and passing in the following parameters:

- The URI of the resource for which to retrieve related resources. In this example, the source resource (`"/testFolder/testResource1"`) is specified.
- A `boolean` value indicating whether the specified resource is the source resource in the relationship. In this example, the value `true` is specified because this is the case.
- The relationship type, which is one of the static constants in the `Relation` class. In this example, a dependence relationship is specified by using the same value used earlier: `Relation.TYPE_DEPENDANT_OF`.

The `getRelated` method returns a `java.util.List` of `Resource` objects through which you can iterate to retrieve each of the related resources, casting the objects contained in the `List` to `Resource` as you do so. In this example, `testResource2` is expected to be in the list of returned resources.

See also

"Creating Resource Relationships" on page 1051

"Quick Start (SOAP mode): Creating relationships between resources using the Java API" on page 318

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Create relationship resources using the web service API

Create relationship resources by using the Repository API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Specify the URIs of the resources to be related

Specify the URIs of the resources to be related. In this case, because the resources are named `testResource1` and `testResource2` and are located in the folder named `testFolder`, their URIs are `"/testFolder/testResource1"` and `"/testFolder/testResource2"`. When using a language compliant with the Microsoft .NET Framework (for example, C#), the URIs are stored as a `System.String` objects. In this example, the resources are first written to the repository, and their URIs are retrieved. For more information about writing a resource, see "Writing Resources" on page 1037.

4 Create the relationship

Invoke the `RepositoryServiceService` object's `createRelationship` method and pass in the following parameters:

- The URI of the source resource.
- The URI of the target resource.
- The type of relationship. In this example, a dependence relationship is established by specifying the value `3`.
- A `boolean` value indicating whether the relationship type was specified. In this example, the value `true` is specified.
- A `boolean` value indicating whether the target resource is automatically updated to the `Id`-based identifier of the new head resource. In this example, because of the dependence relationship, the value `true` is specified.
- A `boolean` value indicating whether the target head was specified. In this example, the value `true` is specified.
- Pass `null` for the last parameter.

You can also retrieve a list of related resources for a given resource by invoking the `RepositoryServiceService` object's `getRelated` method and passing in the following parameters:

- The URI of the resource for which to retrieve related resources. In this example, the source resource (`"/testFolder/testResource1"`) is specified.
- A `boolean` value indicating whether the specified resource is the source resource in the relationship. In this example, the value `true` is specified because this is the case.
- A `boolean` value indicating whether the source resource was specified. In this example, the value `true` is provided.
- An array of integers containing the relationship types. In this example, a dependence relationship is specified by using the same value in the array as was used earlier: `3`.
- Pass `null` for the remaining two parameters.

The `getRelated` method returns an array of objects that can be cast to `Resource` objects through which you can iterate to retrieve each of the related resources. In this example, `testResource2` is expected to be in the list of returned resources.

See also

"Creating Resource Relationships" on page 1051

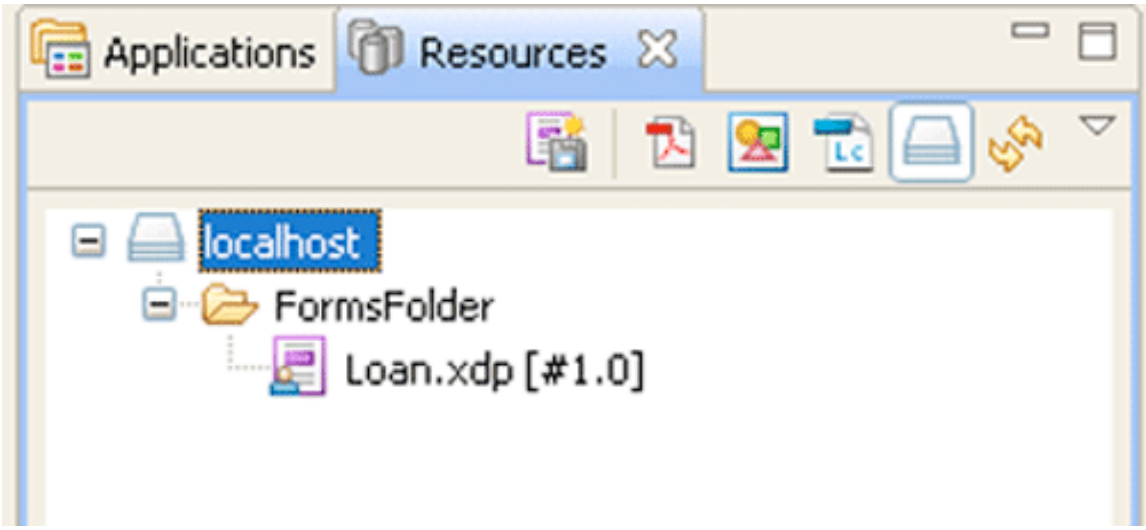
Quick Start (Base64): Creating relationships between resources using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Locking Resources

You can lock a resource or set of resources for exclusive use by a particular user or shared use among more than one user. A shared lock is an indication that something will happen with the resource, but it does not prevent anyone else from taking actions with that resource. A shared lock should be considered a signaling mechanism. An exclusive lock means that the user who locked the resource is going to change the resource, and the lock ensures that nobody else can do so until the user no longer needs access to the resource and has released the lock. If a repository administrator unlocks a resource, all exclusive and shared locks on that resource will automatically be removed. This type of action is meant for situations in which a user is no longer available and has not unlocked the resource.

When a resource is locked, a lock icon appears when you view the Resources tab located in Workbench, as shown in the following illustration.



You can programmatically control access to resources by using the Repository service Java API or web service API.

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To lock and unlock resources, follow these steps:

1. Include project files.
2. Create a Repository service client.
3. Specify the URI of the resource to be locked.
4. Lock the resource.
5. Retrieve the locks for the resource.
6. Unlock the resource

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the URI of the resource to be locked

Create a string containing the URI of the resource to be locked. The syntax includes forward slashes, as in this example: "/*path*/*resource*".

Lock the resource

Invoke the Repository service method to lock the resource, specifying the URI, the type of lock, and the locking depth.

Retrieve the locks for the resource
Invoke the Repository service method to retrieve the locks for the resource, specifying the URI.

Unlock the resource
Invoke the Repository service method to unlock the resource, specifying the URI.

See also
"Lock resources using the Java API" on page 1056

"Lock resources using the web service API" on page 1057

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Lock resources using the Java API

Lock resources by using the Repository service API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the URI of the resource to be locked

Specify the URI of the resource to be locked. In this case, because the resource named `testResource` is in the folder named `testFolder`, its URI is `"/testFolder/testResource"`. The URI is stored as a `java.lang.String` object.

4 Lock the resource

Invoke the `ResourceRepositoryClient` object's `lockResource` method and pass in the following parameters:

- The URI of the resource.
- The lock scope. In this example, because the resource will be locked for exclusive use, the lock scope is specified as `com.adobe.repository.infomodel.bean.Lock.SCOPE_EXCLUSIVE`.
- The lock depth. In this example, because the locking will apply only to the particular resource and none of its members or children, the lock depth is specified as `Lock.DEPTH_ZERO`.

Important: *The overloaded version of the `lockResource` method that requires four parameters throws an exception. Ensure to use the `lockResource` method that requires three parameters as shown in this walkthrough.*

5 Retrieve the locks for the resource

Invoke the `ResourceRepositoryClient` object's `getLocks` method and pass the URI of the resource as a parameter. The method returns a List of Lock objects through which you can iterate. In this example, the lock owner, depth, and scope are printed for each object by invoking each Lock object's `getOwnerUserId`, `getDepth`, and `getType` methods, respectively.

6 Unlock the resource

Invoke the `ResourceRepositoryClient` object's `unlockResource` method and pass the URI of the resource as a parameter. For more information, see the AEM Forms API Reference.

See also

"Locking Resources" on page 1054

"Quick Start (SOAP mode): Locking a resource using the Java API" on page 321

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Lock resources using the web service API

Lock resources by using the Repository service API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL using Base64.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Specify the URI of the resource to be locked

Specify a string containing the URI of the resource to be locked. In this case, because the resource named `testResource` is in the folder `testFolder`, its URI is `"/testFolder/testResource"`. When using a language compliant with the Microsoft .NET Framework (for example, C#), store the URI in a `System.String` object.

4 Lock the resource

Invoke the `RepositoryServiceService` object's `lockResource` method and pass in the following parameters:

- The URI of the resource.
- The lock scope. In this example, because the resource will be locked for exclusive use, the lock scope is specified as `1`.
- The lock depth. In this example, because the locking will apply only to the particular resource and none of its members or children, the lock depth is specified as `2`.
- An `int` value indicating the number of seconds until the lock expires. In this example, the value of `1000` is used.
- Pass `null` for the last parameter.

5 Retrieve the locks for the resource

Invoke the `RepositoryServiceService` object's `getLocks` method and pass the URI of the resource as the first parameter and `null` for the second parameter. The method returns an `object` array containing `Lock` objects through which you can iterate. In this example, the lock owner, depth, and scope are printed for each object by accessing each `Lock` object's `ownerUserId`, `depth`, and `type` fields, respectively.

6 Unlock the resource

Invoke the `RepositoryServiceService` object's `unlockResource` method and pass the URI of the resource as the first parameter and `null` for the second parameter.

See also

"Locking Resources" on page 1054

Quick Start (Base64): Locking a resource using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Deleting Resources

You can programmatically delete resources from a given location in the repository by using the Repository service Java API(SOAP).

When you delete a resource, the deletion is normally permanent, though in some cases ECM repositories may store the versions of the resource according to their history mechanisms. Therefore, when deleting a resource, it is important to be sure that you will never need that resource again. The common reasons for deleting a resource include the need to increase the available space in the database. You can delete a version of a resource, but if you do so you must specify the resource identifier, and not its logical identifier (LID) or path. If you delete a folder, everything in that folder, including subfolders and resources, will be automatically deleted.

Related resources are not deleted. For example, if you have a form that uses the logo.gif file, and you delete logo.gif, a relationship will be stored in the pending relationship table. As an alternative, for version deprecation, set the object status of the latest version to deprecated.

A deletion operation is not transaction-safe in ECM systems. For example, if you attempt to delete 100 resources and the operation fails on the 50th resource, the first 49 instances will be deleted but the rest will not be. Otherwise, the default behavior is rollback (non-commitment).

Note: *When using the* `com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient.deleteResources()` *method with ECM repository (EMC Documentum Content Server and IBM FileNet P8 Content Manager), the transaction will not be rolled back if the deletion fails for one of the specified resources, which means that those files that have been deleted cannot be undeleted.*

Note: *For more information about the Repository service, see Services Reference for AEM Forms.*

Summary of steps

To delete a resource, follow these steps:

1. Include project files.
2. Create a Repository service client.
3. Specify the URI of the resource to be deleted.
4. Delete the resource.

Include project files

Include the necessary files in your development project. If you are creating a client application using Java, include the necessary JAR files. If you are using web services, include the proxy files.

Create the service client

Before you can programmatically read a resource, you must establish a connection and provide credentials. This is accomplished by creating a service client.

Specify the URI of the resource to be deleted

Create a string containing the URI of the resource to be deleted. The syntax includes forward slashes, as in this example: "/*path*/*resource*". If the resource to be deleted is a folder, the deletion will be recursive.

Delete the resource

Invoke the Repository service method to delete the resource, specifying the URI.

See also

"Delete resources using the Java API(SOAP)" on page 1059

"Delete resources using the web service API" on page 1059

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Repository Service API Quick Starts" on page 304

Delete resources using the Java API(SOAP)

Delete a resource by using the Repository API (Java):

1 Include project files

Include client JAR files in your Java project's class path.

2 Create the service client

Create a `ResourceRepositoryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Specify the URI of the resource to be deleted

Specify the URI of the resource to be retrieved. In this case, because the resource named testResourceToBeDeleted is in the folder named testFolder, its URI is `/testFolder/testResourceToBeDeleted`. The URI is stored as a `java.lang.String` object. In this example, the resource is first written to the repository, and its URI is retrieved. For more information about writing a resource, see "Writing Resources" on page 1037.

4 Delete the resource

Invoke the `ResourceRepositoryClient` object's `deleteResource` method and pass the URI of the resource as a parameter.

See also

"Deleting Resources" on page 1058

"Quick Start (SOAP mode): Searching for resources using the Java API" on page 316

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Delete resources using the web service API

Delete a resource by using the Repository API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the Repository WSDL using Base64.
- Reference the Microsoft .NET client assembly.

2 Create the service client

Using the Microsoft .NET client assembly, create a `RepositoryServiceService` object by invoking its default constructor. Set its `Credentials` property using a `System.Net.NetworkCredential` object containing the user name and password.

3 Specify the URI of the resource to be deleted

Specify the URI of the resource to be retrieved. In this case, because the resource named `testResourceToBeDeleted` is in the folder named `testFolder`, its URI is `"/testFolder/testResourceToBeDeleted"`. In this example, the resource is first written to the repository, and its URI is retrieved. For more information about writing a resource, see "Writing Resources" on page 1037.

4 Delete the resource

Invoke the `RepositoryServiceService` object's `deleteResources` method and pass a `System.String` array containing the URI of the resource as the first parameter. Pass `null` for the second parameter.

See also

"Deleting Resources" on page 1058

Quick Start (Base64): Deleting a resource using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Working with Credentials

About the Credential Service

A credential contains your private key information needed for signing or identifying documents. A certificate is public key information that you configure for trust. AEM Forms uses certificates and credentials for several purposes:

- Acrobat Reader DC extensions uses a credential to enable Adobe Reader usage rights in PDF documents. (See "Applying Usage Rights to PDF Documents" on page 751.)
- The Signature service accesses certificates and credentials while performing operations such as digitally signing PDF documents. (See "Digitally Signing PDF Documents" on page 892.)

You can programmatically interact with the Credential service using the Trust Manager Java API. You can perform the following tasks:

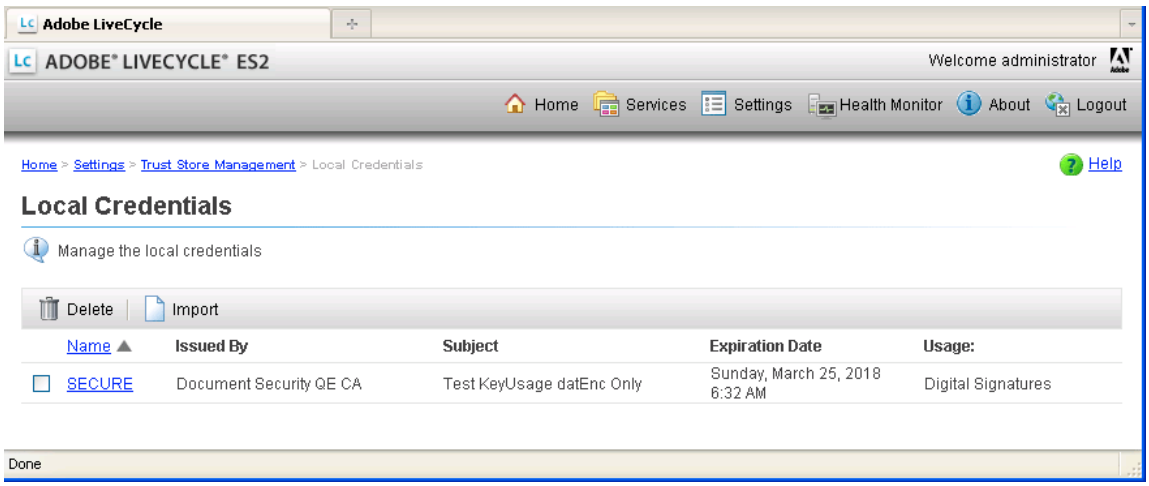
- "Importing Credentials by using the Trust Manager API" on page 1060
- "Deleting Credentials by using the Trust Manager API" on page 1063

Note: *You can also import and delete certificates by using administration console. (See administration help.)*

Importing Credentials by using the Trust Manager API

You can programmatically import a credential into AEM Forms by using the Trust Manager API. For example, you can import a credential used to sign a PDF document. (See "Digitally Signing PDF Documents" on page 892).

When importing a credential, you specify an alias for the credential. The alias is used to perform a Forms operation that requires a credential. Once imported, a credential can be viewed in administration console, as shown in the following illustration. Notice that the alias for the credential is *Secure*.



Note: *You cannot import a credential into AEM Forms using web services.*

Summary of steps

To import a credential into AEM Forms, perform the following steps:

1 Include project files.

2 Create a credential service client.

3 Reference the credential.

4 Perform the import operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-truststore-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a credential service client

Before you can programmatically import a credential into AEM Forms, create a credential service client. For information, see "Setting connection properties" on page 500.

Reference the credential

Reference a credential that you want to import into AEM Forms. The quick start associated with this section references a P12 file located in the file system.

Perform the import operation

After you reference the credential, import the credential into AEM Forms. If the credential is not successfully imported, an exception is thrown. When importing a credential, you specify an alias for the credential.

See also

"Import credentials using the Java API" on page 1062

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Credential Service Java API Quick Start(SOAP)" on page 85

"Deleting Credentials by using the Trust Manager API" on page 1063

Import credentials using the Java API

Import a credential into AEM Forms by using the Trust Manager API (Java):

1 Include project files

Include client JAR files, such as adobe-truststore-client.jar, in your Java project's class path.

2 Create a credential service client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `CredentialServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Reference the credential

- Create a `java.io.FileInputStream` object by using its constructor. Pass a string value that specifies the location of the credential.
- Create a `com.adobe.idp.Document` object that stores the credential by using the `com.adobe.idp.Document` constructor. Pass the `java.io.FileInputStream` object that contains the credential to the constructor.

4 Perform the import operation

- Create a string array that holds one element. Assign the value `truststore.usage.type.sign` to the element.
- Invoke the `CredentialServiceClient` object's `importCredential` method and pass the following values:
  - A string value that specifies the alias value for the credential.
  - The `com.adobe.idp.Document` instance that stores the credential.
  - A string value that specifies the password that is associated with the credential.
  - The string array that contains the usage value. For example, you can specify this value `truststore.usage.type.sign`. To import a Reader Extension credential, specify `truststore.usage.type.lcre`.

See also

"Importing Credentials by using the Trust Manager API" on page 1060

"Quick Start (SOAP mode): Importing credentials using the Java API" on page 85

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Deleting Credentials by using the Trust Manager API

You can programmatically delete a credential by using the Trust Manager API. When deleting a credential, you specify an alias that corresponds to the credential. Once deleted, a credential cannot be used to perform an operation.

Note: *You cannot delete a credential into AEM Forms using web services.*

Summary of steps

To delete a credential, perform the following steps:

1. Include project files.
2. Create a credential service client.
3. Perform the delete operation.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-truststore-client.jar
- adobe-utilities.jar (Required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (Required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a credential service client

Before you can programmatically delete a credential, create a Data Integration service client. When creating a service client, you define connection settings that are required to invoke a service. For information, see "Setting connection properties" on page 500.

Perform the delete operation

To delete a credential, specify the alias that corresponds to the credential. If you specify an alias that does not exist, an exception is thrown.

See also

"Import credentials using the Java API" on page 1062

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Import credentials using the Java API" on page 1062

Deleting credentials using the Java API

Delete a credential from AEM Forms by using the Trust Manager API (Java):

1 Include project files

Include client JAR files, such as adobe-truststore-client.jar, in your Java project's class path.

2 Create a credential service client

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `CredentialServiceClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Perform the delete operation

Invoke the `CredentialServiceClient` object's `deleteCredential` method and pass a string value that specifies the alias value.

See also

"Deleting Credentials by using the Trust Manager API" on page 1063

"Quick Start (SOAP mode): Deleting credentials using the Java API" on page 87

"Including AEM Forms Java library files" on page 491

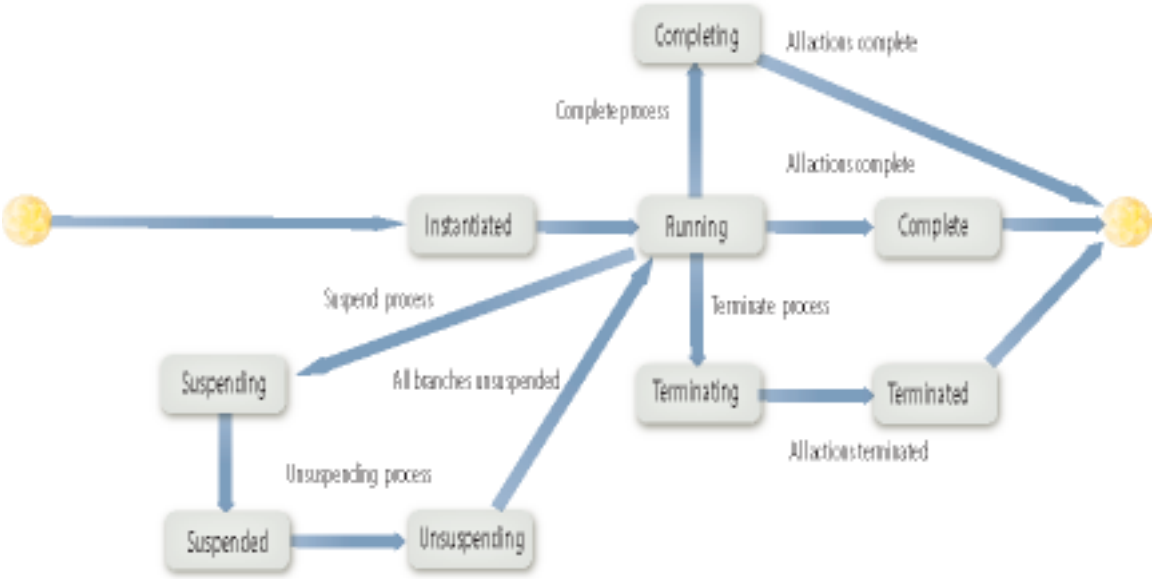
"Setting connection properties" on page 500

Managing Processes and Tasks

About the Process Manager Service

The Process Manager service enables you to interact with process instances at run time. Using the Process Manager service, you can perform task such as suspending running process instances. To interact with the Process Manager service, it is recommended that you have a solid understanding of processes.

Process instances are created to support business transactions that occur within and between organizations. AEM Forms drives process instances that interact with users and systems based on a deployed process that was created in Workbench. Process instances go through several states and state transitions. The following diagram depicts the life cycle of a process instance.



A process instance is instantiated by an explicit request made to AEM Forms to create a process instance of a specific process type. An explicit request to start the process instance places the process in a starting state before implicitly converting it into a running state. A process instance remains in a running state either until an explicit request is issued to complete or terminate the process instance or until AEM Forms implicitly completes the process instance after all the subordinate branches are completed.

A process instance enters a suspended state when a client application makes an explicit request. In this state, navigation is stopped, no more assignments are made, and no more steps are carried out. A process instance remains in a suspended state until an explicit request is made to resume it.

AEM Forms instantiates a branch instance that acts as the main branch instance and places it into a running state. The branch instance remains in a running state either until an explicit request is issued to complete or terminate the process or until AEM Forms implicitly completes the branch instance after the flow of action instances is completed.

About the Task Manager Service

The Task Manager service enables you to interact with tasks during run time. Tasks are units of work that are associated with people. Typically, tasks are presented to users as forms that they open in client software, such as Acrobat or Adobe Reader. Using the client software, users can fill and submit the form back to AEM Forms.

These items are associated with each task:

- **Task identification**: A long object that uniquely identifies tasks from all other tasks of any process type.
- **Form instance**: A `FormInstance` object that identifies the location of a form and the associated form data.
- **Instructions**: Text that explains what to do with the form that is associated with the task.
- **Attachments**: Files that are associated with tasks and are provided to users who are assigned tasks.
- Tasks can be in one of the following states:

- **Assigned**: Retrieve tasks that are assigned to a specific user.
- **Assigned_saved**: Retrieve tasks that are assigned to a user and saved.
- **Completed**: Retrieve tasks that are completed.
- **Created**: Retrieve tasks that are completed.
- **Created_saved**: Retrieve tasks that are created and saved.
- **Deadlined**: Retrieve tasks that are past the deadline.
- **Terminated**: Retrieve tasks that were terminated.

Searching for Process Instances

You can programmatically search for process instances by using the Java API and web services. A *process instance* is an occurrence of a specific process that was started by an invocation method such as the Invocation API or from within Workspace. For example, when you invoke a long-lived process by using the Invocation API, a process instance is created.

Searching for a process instance lets you track its information, such as its status. That is, a process instance can be in one of the following states:

- **Completed**: Indicates that the process instance is complete
- **Completing**: Indicates that the process instance is about to complete
- **Initiated**: Indicates that the process instance has been initiated
- **Running**: Indicates that the process instance is currently running
- **Suspended**: Indicates that the process instance is suspended
- **Suspending**: Indicates that the process instance is about to be suspended
- **Terminated**: Indicates that the process instance was terminated before all operations in the process were completed
- **Terminating**: Indicates that the process instance is about to be terminated
- **Unsuspending**: Indicates that the process instance is about to be unsuspended

When searching for processes, you can specify search criteria, such as the process name, that lets you select specific process instances.

Note: *The name of the service that is invoked when searching for process instances is TaskManagerQueryService.*

Summary of steps

To search for process instances, perform the following tasks:

1. Include project files.
2. Create a TaskManagerQueryService Client API object.
3. Specify search criteria.
4. Perform the search.
5. Iterate through the returned process instances.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a TaskManagerQueryService Client API object

Before you can programmatically search for process instances, you must create a `TaskManagerQueryService` object.

Specify search criteria

To search for process instances, define search criteria. You can, for example, search for all process instances that are based on a specific process. In this situation, all process instances that are based on the process are returned.

Perform the search

After you specify search criteria, you can perform the search. All process instances that conform to the search criteria are returned within a list.

Iterate through the returned process instances

Iterate through the returned list to retrieve each process instance. After you obtain a process instance, you can obtain information about it. For example, you can determine its status.

See also

"Search for process instances using the Java API" on page 1067

"Search for process instances using the web service API" on page 1068

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"LiveCycleProcess Java API(SOAP) Quick Start" on page 284

"Invoking Human-Centric Long-Lived Processes" on page 560.

Search for process instances using the Java API

Search for process instances by using the TaskManagerQuery service API (Java):

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManagerQueryService Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManagerQueryService` object by invoking the `TaskManagerClientFactory` object's static `getQueryManager` method and passing the `ServiceClientFactory` object.

3 Specify search criteria

- Create a `ProcessSearchFilter` object that is used to define search criteria by using its constructor.
- Define search criteria by invoking an appropriate method that belongs to the `ProcessSearchFilter` object. For example, to define the process on which a process instance is based, invoke the `ProcessSearchFilter` object's `setServiceName` method, and pass a string value that specifies the process name.

4 Perform the search

Search for process instances by invoking the `TaskManagerQueryService` object's `processSearch` method and passing the `ProcessSearchFilter` object. This method returns a `java.util.List` instance where each element is a `ProcessInstanceRow` instance that represents a process instance that conforms to the specified search criteria.

5 Iterate through the returned process instances

- Create a `java.util.Iterator` object by invoking the `java.util.List` object's `iterator` method. This object lets you iterate through the `java.util.List` instance to retrieve process instances.
- Iterate through the `java.util.List` object to determine where process instances are present. If they are present, each element is a `ProcessInstanceRow` instance.
- Retrieve information about a process instance by invoking an appropriate method that belongs to the `ProcessInstanceRow` object. For example, to get the status of the process instance, invoke the `ProcessInstanceRow` object's `getProcessInstanceStatus` method. If the process instance is completed, this method returns `ProcessInstanceRow.STATUS_COMPLETED`.

See also

"Searching for Process Instances" on page 1066

"Quick Start (SOAP mode): Searching for Process Instances using the Java API" on page 284

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Search for process instances using the web service API

Search for process instances by using the TaskManagerQuery service API (Java):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the TaskManagerQueryService WSDL using Base64 encoding. To create a proxy object that lets you invoke its operations by using a web service, specify the following WSDL definition:

  `http://localhost:8080/soap/services/TaskManagerQueryService?WSDL`

- Reference the Microsoft .NET client assembly in your client project.

2 Create a TaskManagerQueryService Client API object

- Using the Microsoft .NET client assembly, create a `TaskManagerQueryServiceService` object by invoking its default constructor.
- Set the `TaskManagerQueryServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Specify search criteria

- Create a `ProcessSearchFilter` object that is used to define search criteria by using its constructor.
- Define search criteria by assigning a value to the appropriate data member that belongs to the `ProcessSearchFilter` object. For example, to specify the process on which a process instance is based, assign a string value that specifies the process name to the `serviceName` data member.

4 Perform the search

Search for process instances by invoking the `TaskManagerQueryServiceService` object's `processSearch` method and passing the `ProcessSearchFilter` object. This method returns anarray of `Objects` where each element is a `ProcessInstanceRow` object that represents a process instance that conforms to the specified search criteria.

5 Iterate through the returned process instances

- Iterate through the `Object` array by creating a loop structure and, for each element, cast the element value to a `TaskRow` instance.
- Retrieve information about a task by getting the value of an appropriate data member that belongs to the `TaskRow` object. For example, to get the task identifier value, get the value of the `TaskRow` object's `taskId` data member.

See also

"Searching for Process Instances" on page 1066

Quick Start (Base64): Searching for Process Instances using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Suspending Process Instances

You can programmatically suspend a running process instance by using the Java API and web services. When suspended, a process instance is placed in a suspending state and then a suspended state. Suspending a process instance is typically performed to reduce load on AEM Forms. That is, if a long-lived process is running and requires input from an employee who is unavailable for a week, the process can be suspended for that time period. When the employee is available, the process can be put back into a running state.

Summary of steps

To suspend a process instance, perform the following tasks:

1 Include project files.

2 Create a ProcessManager Client API object.

3 Suspend the process instance.

4 Verify that the process instance is in a suspended state.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a ProcessManager Client API object

Before you can programmatically search for process instances, you must create a `TaskManagerQueryService` object.

Suspend the process instance

To suspend a process, ensure that it is a running state. If you attempt to suspend a process that is not in a running state, you will create an exception. To successfully suspend a process instance, you require the process invocation identifier that can be obtained when invoking a long-lived process by using the Invocation API. (See "Invoking Human-Centric Long-Lived Processes" on page 560.)

See also

"Suspend process instances using the Java API" on page 1070

"Suspend process instances using the web service API" on page 1070

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"LiveCycleProcess Java API(SOAP) Quick Start" on page 284

"Invoking Human-Centric Long-Lived Processes" on page 560.

Suspend process instances using the Java API

Suspend a process instance by using the ProcessManager API (Java):

1 Include project files

Include client JAR files, such as adobe-workflow-client-sdk.jar, in your Java project's class path.

2 Create a ProcessManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ProcessManager` object by using its constructor and passing a `ServiceClientFactory` object.

3 Suspend the process instance

Suspend the process instance by invoking the `ProcessManager` object's `suspendProcess` method and passing a string value that specifies the process invocation identifier value. This value can be obtained when invoking a process by using the Invocation API.

See also

"Suspending Process Instances" on page 1069

"Quick Start (SOAP mode): Suspending process instances using the Java API" on page 287

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Invoking Human-Centric Long-Lived Processes" on page 560

Suspend process instances using the web service API

Suspend a process instance by using the ProcessManager API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the ProcessManager service WSDL. To create a proxy object that lets you invoke its operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/ProcessManager?WSDL`

- Reference the Microsoft .NET client assembly.

2 Create a ProcessManager Client API object

- Using the Microsoft .NET client assembly, create a `ProcessManagerService` object by invoking its default constructor.
- Set the `ProcessManagerService` object's `Credentials` data member with a `System.Net.NetworkCredential` value.

3 Suspend the process instance

Suspend the process instance by invoking the `ProcessManagerService` object's `suspendProcess` method and passing a string value that specifies the process invocation identifier value. This value can be obtained when invoking a process by using the Invocation API.

See also

"Suspending Process Instances" on page 1069

Quick Start (Base64): Suspending process instances using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Terminating Process Instances

You can terminate a process instance by using the Java API and web service API. When terminated, a process instance no longer performs actions. One reason to terminate a process instance is that it is no longer applicable. Assume, for example, that a long-lived process requires a bank manager's input. However, because the bank manager is not available and cannot respond to the process instance, the process instance keeps running while waiting for the bank manager's input. During that time period, a potential bank customer withdraws their mortgage application, resulting in the bank manager's input being unnecessary. As a result, the process instance can be terminated.

Summary of steps

To terminate a process instance, perform the following tasks:

1 Include project files.

2 Create a ProcessManager Client API object.

3 Terminate the process instance.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a ProcessManager Client API object

Before you can programmatically terminate a process instance, you must create a `ProcessManager` object.

Terminate the process instance

To terminate a process instance, ensure that it is a valid process instance. If you attempt to terminate a process instance that does not exist or was previously terminated, a run-time exception is generated. To successfully terminate a process instance, you require the process invocation identifier that can be obtained when invoking a long-lived process by using the Invocation API.

See also

"Terminate process instances using the Java API" on page 1072

"Terminating process instances using the web service API" on page 1072

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"LiveCycleProcess Java API(SOAP) Quick Start" on page 284

"Invoking Human-Centric Long-Lived Processes" on page 560.

Terminate process instances using the Java API

Terminate a process instance by using the ProcessManager API (Java):

1 Include project files

Include client JAR files, such as adobe-workflow-client-sdk.jar, in your Java project's class path.

2 Create a ProcessManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ProcessManager` object by using its constructor and passing a `ServiceClientFactory` object.

3 Terminate the process instance

Terminate a process instance by invoking the `ProcessManager` object's `terminateProcess` method and passing a string value that specifies the process invocation identifier value. This value can be obtained when invoking the process by using the Invocation API.

See also

"Terminating Process Instances" on page 1071

"Quick Start (SOAP mode): Terminating process instances using the Java API" on page 291

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

"Invoking Human-Centric Long-Lived Processes" on page 560

Terminating process instances using the web service API

Terminate a process instance by using the ProcessManager API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the ProcessManager service WSDL. To create a proxy object that lets you invoke its operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/ProcessManager?WSDL`

- Reference the Microsoft .NET client assembly.

2 Create a ProcessManager Client API object

- Using the Microsoft .NET client assembly, create a `ProcessManagerService` object by invoking its default constructor.
- Set the `ProcessManagerService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Terminate the process instance

Terminate a process instance by invoking the `ProcessManager` object's `terminateProcess` method and passing a string value that specifies the process invocation identifier value. This value can be obtained when invoking a process by using the Invocation API.

See also

"Terminating Process Instances" on page 1071

Quick Start (Base64): Terminating process instances using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Purging Process Data

AEM Forms provides a means to purge process data by using the AEM Forms Java API and web service API. Process data that is generated when a long-lived process is invoked can become too large, resulting in lower AEM Forms performance and the use of unnecessary disk space. It is good practice to purge process data when records are no longer necessary. For information about long-lived processes, see "Understanding AEM Forms Processes" on page 441.

When purging process data, you can purge a specific process or all processes that belong to a category defined in Workbench. For example, consider the processes shown in the following illustration.

If you purge process data that corresponds to processes under the Samples - AEM Forms category, then all process data that belongs to the MortgageLoan - Prebuilt and SecureDocument processes is deleted. To purge data that belongs to a specific process, specify the name of the process. For example, you can purge process data that corresponds to the SecureDocument process (shown in the previous illustration). Optionally, when purging process data, you can define a filter expression that only purges data that conforms to the filter.

When purging data, it is possible to purge just that process or the process and its child processes. A child process is a process that was instantiated as part of another process execution (for example, the parent process).

Note: *This topic discusses how to purge a specific process using a filter as opposed to purging processes that belong to a category.*

Summary of steps

To purge process data, perform the following tasks:

1. Include project files.
2. Create a ProcessManager Client API object.
3. Specify the filter that determines the data to purge.
4. Perform the purge operation.

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-workflow-client-sdk.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a ProcessManager Client API object

To programmatically purge process data, you create a forms workflow client object to use the Process Manager Service API.

Specify the filter that determines the data to purge

To create a filter that purges process data, specify three values:

- The name of the process variable on which the filter is based.
- The condition that you want to use (use a condition operator).
- The value of the process variable.

The following condition operators can be used:

- equal
- not equal
- less-than
- less than or equal
- greater than
- greater than or equal
- SQL LIKE
- begins with text
- ends with text
- contains text

These operators are expressed as a ConditionEnum enumeration value. For example, to specify greater than, you use the following value `ConditionEnum.GREATER_THAN`.

Condition filters can be combined using logical filters (`AndFilter` and `OrFilter`) to build complex expressions. For example:

```
(approved=true | approved=yes) & date<'January 10, 2008' & customer ~ 'John Doe'
```

Using complex expressions is the only way to specify multiple process variables in one filter expression. The logical filters are created by using the `AndFilter` (PurgeFilter left, PurgeFilter right) or `OrFilter` (PurgeFilter left, PurgeFilter right) constructors, where left and right are other instances of a condition or logical filter.

Note: *A filter is only used when purging data that corresponds to a specific process. If you are purging data that corresponds to processes belonging to a category, then a filter is not used.*

Perform the purge operation

After you include required library files, create a ProcessManager Client API object, and optionally define a filter, you can purge process data. When purging process data, you can specify the minor and major version of the process and specify whether child process data should also purged.

See also

"Purge process data using the Java API" on page 1075

"Purge process data using the web service API" on page 1076

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"LiveCycleProcess Java API(SOAP) Quick Start" on page 284

"Invoking Human-Centric Long-Lived Processes" on page 560.

Purge process data using the Java API

Purge process data by using the ProcessManager API (Java):

1 Include project files

Include client JAR files, such as adobe-workflow-client-sdk.jar, in your Java project's class path.

2 Create a ProcessManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ProcessManager` object by using its constructor and passing a `ServiceClientFactory` object.

3 Specify the filter that determines the data to purge

Create a `ConditionFilter` object by using its constructor and passing the following values:

- A string value that specifies the process variable on which the condition is based.
- A `ConditionEnum` value that specifies the condition operator. For example, specify `ConditionEnum.GREATER_THAN` to specify greater than.
- A string value that specifies the value of the process variable.

4 Perform the purge operation

Perform the purge operation by invoking the `ProcessManager` object's `purgeProcess` method and passing the following values:

- A string value that specifies the name of the process to purge.
- A short value that specifies the major version of the process.
- A short value that specifies the minor version of the process.
- An integer value that specifies the status of the process. Valid values are `1` which specifies to purge completed processes only; `2` which specifies to purge terminated processes only; `3` which specifies to purge both completed and terminated processes.
- A long value that specifies the seconds defining the age of the process to purge. The process is purged if its completion time is less than or equal to the time calculated by subtracting the number of seconds specified by this value from the time when the purge started. For example, to purge processes completed a day ago, set this field to 86400 (number of seconds in a day).
- A `ConditionFilter` object that represents the filter that is used to purge the process.
- A Boolean value that specifies whether to delete child processes. Specify `false` to only delete just the parent process. In this situation, process data related to child processes are not deleted.

Note: *You can purge all processes that belong to a category by invoking the `ProcessManager` object's `purgeProcesses` method.*

See also

"Purging Process Data" on page 1073

"Quick Start (SOAP mode): Purging process data using the Java API" on page 293

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

"Invoking Human-Centric Long-Lived Processes" on page 560

Purge process data using the web service API

Purge process data by using the ProcessManager API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the ProcessManagerService WSDL. To create a proxy object that lets you invoke purge data operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/ProcessManager?WSDL&lc_version=8.2.1.`

  You must specify `&lc_version=8.2.1` (or later) because the purge operation was added in LiveCycle 8.2.

- Reference the Microsoft .NET client assembly.

2 Create a ProcessManager Client API object

- Using the Microsoft .NET client assembly, create a `ProcessManagerService` object by invoking its default constructor.
- Set the `ProcessManagerService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Specify the filter that determines the data to purge

- Create a `ConditionFilter` object by using its constructor.
- Specify the process variable on which the condition is based by assigning a string value to the `ConditionFilter` object's `variable` data member.
- Specify the condition operator by assigning a ConditionEnum value to the ConditionFilter object's condition data member. For example, specify `ConditionEnum.GREATER_THAN` to specify greater than.
- Specify the value of the process variable by assigning a string value to the `ConditionFilter` object's `value` data member.

4 Perform the purge operation

Perform the purge operation by invoking the `ProcessManagerService` object's `purgeProcess` method and passing the following values:

- A string value that specifies the name of the process to purge.
- A short value that specifies the major version of the process.
- A Boolean value that specifies whether to use the previous value. Specify `true`.
- A short value that specifies the minor version of the process.
- A Boolean value that specifies whether to use the previous value. Specify `true`.
- An integer value that specifies the status of the process. Valid values are `1` which specifies to purge completed processes only; `2` which specifies to purge terminated processes only; `3` which specifies to purge both completed and terminated processes.
- A Boolean value that specifies whether to use the previous value. Specify `true`.
- A long value that specifies the seconds defining the age of the process to purge. The process is purged if its completion time is less than or equal to the time calculated by subtracting the number of seconds specified by this value from the time when the purge started. For example, to purge processes completed a day ago, set this field to 86400 (number of seconds in a day).
- A Boolean value that specifies whether to use the previous value. Specify `true`.
- A `ConditionFilter` object that represents the filter that is used to purge the process.
- A Boolean value that specifies whether to delete child processes. Specify `false` to only delete the parent process.

- A Boolean value that specifies whether to use the previous value. Specify `true`.

See also

"Purging Process Data" on page 1073

Quick Start (Base64): Purging process data using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Retrieving the Status of an AEM Forms Job

You can programmatically retrieve the status of a Forms job by using the Job Manager API. A job occurs when a long-lived process is invoked. For example, when you invoke a long-lived process by using the Invocation API, a job is created.

The following list specifies job status values:

- **0 (JOB_STATUS_UNKNOWN)**: Indicates that the job status is unknown
- **1 (JOB_STATUS_QUEUED)**: Indicates that the job is queued.
- **2 (JOB_STATUS_RUNNING)**: Indicates that the job is currently running
- **3 (JOB_STATUS_COMPLETED)**: Indicates that the job is completed
- **4 (JOB_STATUS_FAILED)**: Indicates that the job failed
- **5 (JOB_STATUS_TERMINATED)**: Indicates that the job is terminated
- **6 (JOB_STATUS_SUSPENDED)**: Indicates that the job is suspended
- **7 (JOB_STATUS_COMPLETE_REQUESTED)**: Indicates that a job complete request was made
- **8 (JOB_STATUS_TERMINATE_REQUESTED)**: Indicates that a job terminate request was made
- **9 (JOB_STATUS_SUSPEND_REQUESTED)**: Indicates that a job suspend request was made
- **10 (JOB_STATUS_RESUME_REQUESTED)**: Indicates that a job resume request was made

Summary of steps

To retrieve the status of a Forms job, perform the following tasks:

1. Include project files.
2. Create a JobManager Client API object.
3. Specify search criteria.
4. Perform the search.
5. Iterate through the returned job instances.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a JobManager Client API object

Before you can programmatically retrieve AEM Forms job instances, you must create a `JobManager` object.

Specify filter criteria

To search for job status values, define filter criteria. You can, for example, limit the number of jobs that are returned. To define filter criteria, you use a `JobInstanceFilter` instance.

Perform the search

After you specify filter criteria, you can perform the search. All job instances are returned within a list. That is, each element in the list is a `JobInstance` object.

Iterate through the returned job instances

Iterate through the returned list to retrieve each process instance. After you obtain a process instance, you can obtain information about it. For example, you can determine its status.

Note: *For information on how to retrieve the status of a job using web sevices, see "Creating an ASP.NET web application that invokes a human-centric long-lived process" on page 569.*

See also

"Retrieve the status of a job using the Java API" on page 1078

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"LiveCycleProcess Java API(SOAP) Quick Start" on page 284

"Invoking Human-Centric Long-Lived Processes" on page 560.

Retrieve the status of a job using the Java API

Retrieve the status of a job by using the Job Manager API (Java):

1. Include project files

   Include client JAR files, such as adobe-jobmanager-client-sdk.jar, in your Java project's class path.

2. Create a JobManager Client API object
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `JobManager` object by invoking its constructor and passing the `ServiceClientFactory` object.
3. Specify filter criteria
   - Create a `JobInstanceFilter` object that is used to define search criteria by using its constructor.
   - Define filter criteria by invoking an appropriate method that belongs to the `JobInstanceFilter` object. For example, to define the maximum number of jobs to return, invoke the `JobInstanceFilter` object's `setMaxObjects` method, and pass an integer value that specifies value.
4. Perform the search

   Search for jobs by invoking the `JobManager` object's `getJobInstances` method and passing the `JobInstanceFilter` object. This method returns a `java.util.List` instance, where each element is a `JobInstance` instance that represents a Forms job.
5. Iterate through the returned job instances
   - Create a `java.util.Iterator` object by invoking the `java.util.List` object's `iterator` method. This object lets you iterate through the `java.util.List` instance to retrieve job instances.
   - Iterate through the `java.util.List` object. If they are jpb instances, each element is a `JobInstance` instance.

- Retrieve information about a job by invoking an appropriate method that belongs to the `JobInstance` object. For example, to get the status of the job, invoke the `JobInstance` object's `getStatus` method.

See also

"Retrieving the Status of an AEM Forms Job" on page 1077

"Quick Start (SOAP Mode): Retrieving the status of a job using the Java API" on page 295

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Invoking Human-Centric Long-Lived Processes" on page 560

Programmatically Deploying Components

You can programmatically deploy components to the service container by using the Java API. After you deploy a component to AEM Forms, you can use it within Workbench to build processes. By programmatically deploying components, you can automate the process of deploying components after a component is created or when you want to upgrade to a newer component version.

When the component is deployed, the services located within the component become AEM Forms services. As a result, you can invoke the component's services by using an invocation method. (See "Service container" on page 443.)

Note: *You cannot deploy a component by using web services.*

Summary of steps

To deploy a component to the service container, perform the following tasks:

1. Include project files.
2. Create an Component Manager Client API object.
3. Retrieve the component JAR file.
4. Install the component.
5. Start the component.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create an ComponentRegistry Client API object

Before you can programmatically perform an component manager service operation, you must create a `ComponentRegistryClient` object.

Retrieve the component JAR file

To programmatically deploy a component, reference the component JAR file. When you reference the JAR file, create a `com.adobe.idp.Document` object that stores the JAR file.

Install the component

To deploy a component, first install it. When installed, the component is in a stopped state.

Start the component

Start a component to invoke its services. After the component is started, its services can be started. (See .)

See also

"Deploy components using the Java API" on page 1080

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Components and Services Java API Quick Start(SOAP)" on page 68

Creating Your First Component

Starting Services

Deploy components using the Java API

Deploy a component by using the Java API:

1. Include project files

   Include client JAR files, such as adobe-workflow-client-sdk.jar, in your Java project's class path.

2. Create an ComponentRegistry Client API object

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `ComponentRegistryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3. Retrieve the component JAR file

   - Create a `java.io.FileInputStream` object that represents the component JAR file by using its constructor and passing a string value that specifies the location of the JAR file.
   - Create a `com.adobe.idp.Document` object by using its constructor and passing the `java.io.FileInputStream` object.

4. Install the component

   Install the component by invoking the `ComponentRegistryClient` object's `install` method and passing the `com.adobe.idp.Document` object that contains the component. This method returns a `com.adobe.idp.dsc.registry.infomodel.Component` object that represents the component that exists in a stopped state.

5. Start the component

   Start the component by invoking the `ComponentRegistryClient` object's `start` method and passing the `com.adobe.idp.dsc.registry.infomodel.Component` object that represents the component that exists in a stopped state.

See also

"Programmatically Deploying Components" on page 1079

"Quick Start (SOAP mode): Deploying a component using the Java API" on page 68

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Setting the Execution Context of a Service

The security of a service can be either enabled or disabled. When disabled, a service operation can be invoked without specifying a user name and password. That is, when setting connection properties that are required to invoke AEM Forms operations, you do not need to specify a user name and corresponding password. (See "Setting connection properties" on page 500).

When the security of a service is enabled, the user who is invoking a service operation must be authenticated and authorized to invoke the service. Otherwise, the service container denies the invocation request. (See "Disabling Service Security" on page 1083.)

A service can originate from a process created by using Workbench. By default, the way that each service (that is part of the process) is invoked depends on whether the process is a short-lived process or a long-lived process. For a short-lived process, the user's context is used to invoke each service that is part of the process (this is known as the *execution context*). For a long-lived process, a system context is used to invoke each service that is part of the process. For information about long-lived and short-lived processes, see "Understanding AEM Forms Processes" on page 441.

Using the AEM Forms Java API, you can specify the execution context in which each service (that is part of a process) is invoked. This execution context is used regardless of whether the process is a long-lived process or a short-lived process. Three types of execution context exists:

- **Run-As Invoker:** The context of the invoker is used as the execution context for each service that is part of a process.
- **Run-As System:** The system context is used as the execution context for each service that is part of a process. This setting is the current default setting for long-lived processes.
- **Run-As Named User:** A specific AEM forms user is specified in the configuration. When invoked, all actions performed by the process are performed as if by that user rather than by the user that initially invoked the process.

AEM Forms checks the execution context for authorization on a service operation before an invocation request proceeds. AEM Forms also checks the execution context of a service and changes it to the execution context that is set before proceeding with an invocation request.

AEM Forms performs the following tasks in response to an invocation request:

1. Checks whether security is disabled for the service, sets the execution context (if set), and lets the invocation request proceed.
2. Checks whether the service operation has anonymous access enabled (from the component.xml file).
3. Checks whether the execution context is authorized to invoke the service.

If the execution context for a service is not set, the default behavior is used. For a short-lived process, the identifier of the user who invokes the process is used. For a long-lived process, the system context is used.

Summary of steps

To set the execution context of a service, perform the following tasks:

1. Include project files.
2. Create a `ServiceRegistryClient` API object.
3. Reference a service.
4. Set the execution context.

Note: *You cannot set the execution context of a service by using web services.*

Include project files

Include necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- commons-codec-1.3.jar
- commons-collections-3.1.jar
- commons-discovery.jar
- commons-logging.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a ServiceRegistryClient API object

Before you can programmatically set the execution context of a service, create a `ServiceRegistryClient` object.

Reference a service

To set the execution context of a service, the service must be referenced. For example, assume that you want to set the execution content of a service named *EncryptDocument*. In this situation, you must reference the EncryptDocument service.

Set the execution context

You can set the execution context to `Run-As Invoker`, `Run-As system`, or `Run-as named user`. After the specified execution context is set, it is used when the process is invoked. For example, assume that you set the execution context for the EncryptDocument service to `Run-As Invoker`. In this situation, the context of the invoker is used as the invocation context for each service that is part of the EncryptDocument process.

See also

"Set the execution context of a service using the Java API" on page 1082

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Starting Services

Set the execution context of a service using the Java API

Set the execution context of a service by using the Java API:

1 Include project files.

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create a ServiceRegistryClient API object.

   - Create a `ServiceClientFactory` object that contains connection properties.

- Create a `ServiceRegistryClient` object by using its constructor and passing a `ServiceClientFactory` object.

3 Reference a service.

Reference the service for which an execution context is set by invoking the `ServiceRegistryClient` object's `getHeadActiveConfiguration` method and passing a string value that specifies the name of the service. If multiple service versions exist, the latest version (referred to as the *head version*) is returned. This method returns a `ServiceConfiguration` object that represents the service.

4 Set the execution context.

- Create a `ModifyServiceConfigurationInfo` object by using its constructor.
- Set the service identifier value by invoking the `ModifyServiceConfigurationInfo` object's `setServiceId` method and passing the service identifier value (pass the return value of the `ServiceConfiguration` object's `getServiceId` method).
- Set the major version of the service by invoking the `ModifyServiceConfigurationInfo` object's `setMajorVersion` method and passing the major version value (pass the return value of the `ServiceConfiguration` object's `getMajorVersion` method).
- Set the minor version of the service by invoking the `ModifyServiceConfigurationInfo` object's `setMinorVersion` method and passing the minor version value (pass the return value of the `ServiceConfiguration` object's `getMinorVersion` method).
- Set the execution context of the service by invoking the `ModifyServiceConfigurationInfo` object's `setRunAsConfiguration` method and passing a `ServiceConfiguration` enumeration value that specifies the execution context type. For example, to set `Run-As Invoker`, pass `ServiceConfiguration.RUN_AS_INVOKER`.
- Invoke the `ServiceRegistryClient` object's `modifyConfiguration` method and pass the `ModifyServiceConfigurationInfo` object.

See also

"Setting the Execution Context of a Service" on page 1081

"Quick Start (SOAP mode): Setting the execution context of a service using the Java API" on page 70

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Disabling Service Security

You can disable service security by using the AEM Forms Java API. By default, service security is enabled. However, when service security is disabled, service operations can be invoked without specifying a user name and password. That is, when setting connection properties that are required to invoke service operations, you do not need to specify a user name and corresponding password. (See "Setting connection properties" on page 500).

To invoke a service that originated from a process created by using Workbench without specifying a user name or corresponding password, you must disable security of the service and all of the services that are invoked from within the process. Consider, for example, the following short-lived process named *EncryptDocument*. (For information about a short-lived process, see "Understanding AEM Forms Processes" on page 441.)

To invoke this service without specifying a user name or password, disable security from three services:

- **EncryptDocument**: The name of process (which is a service once activated within Workbench)
- **SetValue**: The name of the first service within this process
- **Encryption service**: The name of the second service within this process

Note: *You cannot disable service security by using web services.*

Summary of steps

To disable service security, perform the following tasks:

1. Include project files.
2. Create a `ServiceRegistryClient` API object.
3. Reference services that belong to the process.
4. Disable security.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- commons-codec-1.3.jar
- commons-collections-3.1.jar
- commons-discovery.jar
- commons-logging.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create a ServiceRegistryClient API object

Before you can programmatically set the execution context of a service, create a `ServiceRegistryClient` object.

Reference services that belong to the process

To set the execution context of a service, the service must be referenced. For example, assume that you want to set the execution content of a service named *EncryptDocument*. In this situation, you must reference the EncryptDocument service.

Disable security

Security must be disabled from each service that is located within a process in order to invoke the process without specifying a user name or password. For example, consider the process introduced in this section. To invoke this process without specifying a user name or password, security must be disabled from the EncryptDocument service, the SetValue service, and the Encryption service.

See also

"Disabling service security using the Java API" on page 1085

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Starting Services

Disabling service security using the Java API

To disable service security by using the Java API, perform the following steps:

1. Include project files.

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2. Create a ServiceRegistryClient API object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `ServiceRegistryClient` object by using its constructor and passing a `ServiceClientFactory` object.

3. Reference services that belong to the process.

   Reference each service that belongs to the process by invoking the `ServiceRegistryClient` object's `getHeadActiveConfiguration` method and passing a string value that specifies the name of the service. If multiple service versions exist, the latest version (referred to as the *head version*) is returned. This method returns a `ServiceConfiguration` object that represents the service (a `ServiceConfiguration` object must exist for each service from which security is disabled).

4. Disable security.

   - Create a `ModifyServiceInfo` object by using its constructor.
   - Set the service identifier value by invoking the `ModifyServiceInfo` object's `setId` method and passing the service identifier value (pass the return value of the `ServiceConfiguration` object's `getServiceId` method).
   - Disable security from the service by invoking the `ModifyServiceInfo` object's `setSecurityEnabled` method and passing the Boolean value `false`.
   - Invoke the `ServiceRegistryClient` object's `modifyConfiguration` method and pass the `ModifyServiceInfo` object.

   Note: *This step must be performed for each service from which security is disabled.*

See also

"Disabling Service Security" on page 1083

"Quick Start (SOAP mode): Disabling service security using the Java API" on page 72

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Modifying Service Configuration Values

You can programmatically modify a service's configuration values. Configuration values typically do not change during the duration of the service; that is, the value is a constant. For example, consider the sample email component that contains three configuration values:

- `smtpHost`: The IP address of the SMTP server that sends email messages
- `smtpUser`: The user name that is used to connect to the SMTP server
- `smtpPassword`: The corresponding password of the user

This topic discusses how to programmatically modify configuration values that belong to the sample email component. For information about creating the sample email component, see Creating Your First Component.

Note: *You cannot modify service configuration values by using web services.*

Summary of steps

To modify a service's configuration values, perform the following tasks:

1. Include project files.
2. Create a ServiceRegistryClient API object
3. Reference a service to modify.
4. Modify the services configuration values.
5. Verify the configuration values.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a ServiceRegistryClient API object

Before you can programmatically perform a service registry operation, you must create a `ServiceRegistryClient` object.

Reference a service to modify

To programmatically deploy a component, reference the component JAR file. When you reference the JAR file, create a `com.adobe.idp.Document` object that stores the JAR file. (See Passing data to AEM Forms services using the Java API.)

Modify configuration values

To deploy a component, install it first. When it is installed, the component is in a stopped state.

Verify the configuration values

Start a component to invoke its services. After the component is started, its services must be started as well. (See Starting Services.)

See also

"Modify a services configuration values using the Java API" on page 1087

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Components and Services Java API Quick Start(SOAP)" on page 68

Modify a services configuration values using the Java API

Modify a service's configuration values by using the Java API:

1 Include project files

Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create a ServiceRegistryClient API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ServiceRegistryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Reference a service to modify

Reference the service (whose configuration values to modify) by invoking the `ServiceRegistryClient` object's `getHeadActiveConfiguration` method and passing a string value that specifies the name of the service. If multiple service versions exist, the latest version (referred to as the head version) is returned. This method returns a `ServiceConfiguration` object that represents the service.

4 Modify configuration values

- Create a `ModifyServiceConfigurationInfo` object by using its constructor.
- Specify the service (whose configuration values are modified) by invoking the `ModifyServiceConfigurationInfo` object's `setServiceId` method and passing a string value that specifies the service identifier value. You can obtain this value by invoking the `ServiceConfiguration` object's `getServiceId` method.
- Set the major version of the service by invoking the `ModifyServiceConfigurationInfo` object's `setMajorVersion` method and passing an integer value that specifies the major version.
- For each configuration value to set, invoke the `ModifyServiceConfigurationInfo` object's `setConfigParameterAsText` method and pass the following string values:
  - A string value that specifies the name of the configuration value
  - A string value that specifies the value of the configuration value
- Invoke the `ServiceRegistryClient` object's `modifyConfiguration` method and pass the `ModifyServiceConfigurationInfo` object.

5 Verify the configuration values

- Invoke the `ServiceRegistryClient` object's `getServiceConfiguration` method and pass the following parameter values:
  - A string value that specifies the name of the service
  - An integer value that specifies the major version of the service
  - An integer value that specifies the minor version of the service

  The `getServiceConfiguration` method returns a `ServiceConfiguration` object.
- Reference the configuration value whose value you want to verify by invoking the `ServiceConfiguration` object's `getConfigParameter` method and passing a string value that specifies the name of the configuration value. This method returns a `ConfigParameter` object.
- Get the configuration value by invoking the `ConfigParameter` object's `getTextValue` method. This method returns the configuration value.

See also

"Modifying Service Configuration Values" on page 1086

"Quick Start (SOAP mode): Modifying a services configuration values using the Java API" on page 76

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Removing Components

You can remove a component by using the Java API. When updating a component, you should remove the outdated component before deploying the updated component. For example, if you develop a component and then you want to deploy a newer version, remove the older component prior to deploying the updated component. For information about creating components, see Creating Your First Component.

Note: *You cannot remove a component by using web services.*

Summary of steps

To remove a component, perform the following tasks:

1. Include project files.
2. Create an Component Manager Client API object.
3. Retrieve the component to remove.
4. Determine the components state.
5. Uninstall the component.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create an Component Manager Client API object

Before you can programmatically perform an Component Manager service operation, you must create a `ComponentRegistryClient` object.

Retrieve the component to remove

To programmatically deploy a component, reference the component JAR file. When you reference the JAR file, create a `com.adobe.idp.Document` object that stores the JAR file. (See "Passing data to AEM Forms services using the Java API" on page 505.)

Determine the state of the component

To remove a component, first ensure that it is not in a running state. If a component is running, put the component into a stopped state.

Uninstall the component

After you stop the component and remove its services, you can remove the component.

See also

"Remove components using the Java API" on page 1089

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Components and Services Java API Quick Start(SOAP)" on page 68

Remove components using the Java API

Remove a component by using the Java API:

1 Include project files

Include client JAR files, such as adobe-workflow-client-sdk.jar, in your Java project's class path.

2 Create an Component Manager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `ComponentRegistryClient` object by using its constructor and passing a `ServiceClientFactory` object that contains connection properties.

3 Retrieve the component to remove

Retrieve the component to remove by invoking the `ComponentRegistryClient` object's `getComponent` method and passing a string value that specifies the component identifier. This method returns a `Component` object.

4 Determine the state of the component

- Determine whether the component is running by invoking the `Component` object's `getState` method. This method returns a static member of `Component`. If the component is in a running state, this method returns `Component.RUNNING`.
- If the component is in a running state, stop the component by invoking the `ComponentRegistryClient` object's `stop` method and passing the `Component` object.

5 Uninstall the component

Uninstall the component by invoking the `ComponentRegistryClient` object's `uninstall` method and passing the `Component` object.

See also

"Removing Components" on page 1088

"Quick Start (SOAP mode): Removing components using the Java API" on page 78

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Assigning Tasks

You can assign an existing task to a user by using the Java API and web service API. When a task is assigned to a user, the task is placed in the user's queue and the user can view and complete the task in Workspace. If the task was previously assigned to another user, it is moved from one user queue and placed into the user queue to whom the task is assigned.

Note: *The name of the service that is invoked when assigning tasks is TaskManagerService. To create a proxy object that lets you invoke its operations using a web service, specify this WSDL definition:* `http://localhost:8080/soap/services/TaskManagerService?WSDL`. *(See "Invoking AEM Forms using Web Services" on page 514.)*

Summary of steps

To assign a task to a user, perform the following tasks:

1 Include project files.

2 Create a TaskManager Client API object.

3 Get the identifier of the user to whom the task is sent.

4 Forward the task to the user.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a TaskManager Client API object

Before you can programmatically assign a task to a user, you must create a `TaskManager` object.

Get the identifier of the user to whom the task is assigned

To assign a task to a user, you require the identifier of the user to whom the task is assigned. To obtain the user identifier, use the User Manager API.

Forward the task to a user

After you obtain the user identifier, you can assign the task to the user.

See also

"Assign tasks using the Java API" on page 1090

"Assigning tasks using the web service API" on page 1091

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Managing Users and Groups" on page 1023

Assign tasks using the Java API

Assign tasks by using the Java API:

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManager` object by invoking the `TaskManagerClientFactory` object's static `getTaskManager` method and passing the `ServiceClientFactory` object.

3 Get the identifier of the user to whom the task is assigned

- Create a `DirectoryManagerServiceClient` object by using its constructor and passing the `ServiceClientFactory` object that contains connection properties (the `DirectoryManagerServiceClient` belongs to the User Manager API).
- Create a `PrincipalSearchFilter` object by using its constructor.

- Set the user name by invoking the `PrincipalSearchFilter` object's `setUserId` method and passing a string value that specifies the user name.
- Find the user that corresponds to the user name by invoking the `DirectoryManagerServiceClient` object's `findPrincipals` method and passing the `PrincipalSearchFilter` object. This method returns a `java.util.List` object where each element is a `User` object.
- Create a `java.util.Iterator` object by invoking the `java.util.List` object's `iterator` method. This object lets you iterate through the `java.util.List` instance to retrieve `User` objects (in this situation, there is only one `User` object).
- Iterate through the `java.util.List` object to determine where there are users. If so, each element is a `User` object.
- Get the user identifier by invoking the `User` object's `getOid` method. This method returns a string value that represents the user identifier. The format of a user identifier is a GUID.

4 Forward the task to a user

Forward the task to a user by invoking the `TaskManager` object's `forwardTask` method and passing the task identifier and the user identifier value. You can determine the task identifier by retrieving tasks that are assigned to a specific user.

See also

"Assigning Tasks" on page 1089

"Quick Start (SOAP mode): Assigning tasks using the Java API" on page 398

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

"Retrieving Tasks Assigned to Users" on page 1094

Assigning tasks using the web service API

To assign a task by using the web service API, perform the following steps:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the TaskManager service WSDL. To create a proxy object that lets you invoke its operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/TaskManagerService?WSDL`

- Reference the Microsoft .NET client assembly.

2 Create a TaskManager Client API object

- Using the Microsoft .NET client assembly, create a `TaskManagerServiceService` object by invoking its default constructor.
- Set the `TaskManagerServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Get the identifier of the user to whom the task is assigned

- Using the Microsoft .NET client assembly, create a `DirectoryManagerServiceService` object by invoking its default constructor.
- Set the `DirectoryManagerServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

- Create a `PrincipalSearchFilter` object by using its constructor.
- Set the user name by assigning a string value to the `PrincipalSearchFilter` object's `userId` data member.
- Find the user that corresponds to the user name by invoking the `DirectoryManagerServiceClient` object's `findPrincipalsWithFilter` method and passing the `PrincipalSearchFilter` object. This method returns an `Object` array where each element is a `User` object (in this situation, only one `User` object corresponds to the specified `userId` value).
- Iterate through the `Object` array to determine whether there are users. If so, each element is a `User` object.
- Get the user identifier by getting the value of the `User` object's `oid` data member. The format of a user identifier is a GUID.

4 Forward the task to a user

Forward the task to a user by invoking the `TaskManagerServiceService` object's `forwardTask` method and passing the task identifier and the user identifier value. You can determine the task identifier by retrieving tasks that are assigned to a specific user. (See "Retrieving Tasks Assigned to Users" on page 1094.)

See also

"Assigning Tasks" on page 1089

Quick Start (Base64): Assigning tasks using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Locking Tasks

You can lock tasks by using the Java API and web services. Locking a task prevents other users from working on it. When a task is locked, a lock icon appears on the task card in Workspace.

Summary of steps

To lock a task, perform the following tasks:

1 Include project files.

2 Create a TaskManager Client API object.

3 Lock a task.

Include project files

Include necessary files into your development project. Because you are creating a client application by using Java, include the necessary JAR files.

Create a TaskManager Client API object

Before you can lock a task, you must create a `TaskManager` object.

Locking a task

To lock a task, reference the task by using its identifier. You can determine the task identifier by retrieving tasks that are assigned to a specific user.

See also

"Lock tasks using the Java API" on page 1093

"Lock tasks using the web service API" on page 1093

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Retrieving Tasks Assigned to Users" on page 1094

Lock tasks using the Java API

Lock a task by using the Java API:

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManager` object by invoking the `TaskManagerClientFactory` object's static `getTaskManager` method and passing the `ServiceClientFactory` object.

3 Locking a task

Lock a task by invoking the `TaskManager` object's `lockTask` method and passing the task identifier value. You can determine the task identifier by retrieving tasks that are assigned to a specific user.

Note: *You can unlock a task by invoking the `TaskManager` object's `unlockTask` method.*

See also

"Locking Tasks" on page 1092

"Quick Start (SOAP mode): Locking tasks using the Java API" on page 400

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

"Retrieving Tasks Assigned to Users" on page 1094

Lock tasks using the web service API

Lock a task by using the web service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the TaskManager service WSDL. To create a proxy object that lets you invoke its operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/TaskManagerService?WSDL`

- Reference the Microsoft .NET client assembly.

2 Create a TaskManager Client API object

- Using the Microsoft .NET client assembly, create a `TaskManagerServiceService` object by invoking its default constructor.
- Set the `TaskManagerServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Locking a task

Lock a task by invoking the `TaskManagerServiceService` object's `lockTask` method and passing the task identifier value and a `System.Boolean` value that specifies `true` (this argument value informs the Task Manager service to use the task identifier value). You can determine the task identifier by retrieving tasks that are assigned to a specific user. (See "Retrieving Tasks Assigned to Users" on page 1094.)

Note: *You can unlock a task by invoking the `TaskManagerServiceService` object's `unlockTask` method.*

See also

"Locking Tasks" on page 1092

Quick Start (Base64): Assigning tasks using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Retrieving Tasks Assigned to Users

You can retrieve tasks that are assigned to users using the Java API. That is, you can determine what tasks are located in a user's queue. By retrieving tasks assigned to users, you can determine information related to the task such as the task's identifier, the task's description, the task's start date, the task's status, and so on.

Note: *You cannot search for tasks assigned to users by using the web service API. The reason is because you cannot invoke the `taskList` method, which is a necessary method call to perform this task.*

Important: *When invoking the `TaskManagerQueryService` object's `TaskSearch` and the `TaskManager` object's `TerminateTask` method in the same transaction and the database that AEM Forms uses is DB2, you will cause a run-time error. Instead, make these calls in different transactions. The issue is related to how DB2 handles transactions.*

Summary of steps

To retrieve tasks assigned to users, perform the following tasks:

1. Include project files.
2. Create a TaskManagerQueryService Client API object.
3. Specify search criteria.
4. Perform the search.
5. Iterate through the returned tasks.

Include project files

Include necessary files into your development project. Because you are creating a client application using Java, include the necessary JAR files.

Create a TaskManagerQueryService Client API object

Before you can programmatically search for process instances, you must create a `TaskManagerQueryService` object.

To determine the tasks that are assigned to a specific user, specify the user name and the password value of the user for whom tasks are retrieved when setting connection properties. For example, to retrieve all tasks assigned to tony blue, use tony blue's user name and password.

Specify search criteria

You can search for tasks based on the following states:

- **Assigned**: Retrieve tasks that are assigned to a specific user.
- **Assigned_saved**: Retrieve tasks that are assigned to a user and saved.

- **Completed**: Retrieve tasks that are completed.
- **Created**: Retrieve tasks that are completed.
- **Created_saved**: Retrieve tasks that are created and saved.
- **Deadlined**: Retrieve tasks that are past the deadline.
- **Terminated**: Retrieve tasks that were terminated.

To retrieve tasks that are assigned to a specific user, perform your search based on the `Assigned` state.

Perform the search

After you define search criteria, you can search for tasks. All tasks that conform to the search criteria are returned in a list.

Iterate through the returned tasks

Iterate through the list of tasks to determine information about each task. You can, for example, determine a task's identifier value.

See also

"Retrieve tasks assigned to users using the Java API" on page 1095

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve tasks assigned to users using the Java API

Retrieve tasks assigned to a user by using the Java API:

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManagerQueryService Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManagerQueryService` object by invoking the `TaskManagerClientFactory` object's static `getQueryManager` method and passing the `ServiceClientFactory` object.

3 Specify search criteria

- Create a `TaskFilter` object by invoking the `TaskManagerQueryService` object's `newTaskFilter` method.
- Create a `StatusFilter` object by invoking the `TaskFilter` object's `newStatusFilter` method.
- Specify the status of the tasks to search for by invoking the `StatusFilter` object's `addStatus` method and passing a static data member that belongs to `StatusFilter`. For example, pass `StatusFilter.assigned` to retrieve tasks that are assigned to a specific user.
- Invoke the `TaskFilter` object's `setStatusFiltering` method and pass the `StatusFilter` object.

4 Perform the search

Search for tasks by invoking the `TaskManagerQueryServiceService` object's `taskList` method and passing the `TaskFilter` object. This method returns a `java.util.List` object where each element is a `TaskRow` object that represents a task that conforms to the specified search criteria.

5 Iterate through the returned tasks

- Create a `java.util.Iterator` object by invoking the `java.util.List` object's `iterator` method. This object enables you to iterate through the `java.util.List` instance to retrieve tasks.
- Iterate through the `java.util.List` object to determine if there are tasks. If so, each element is a `TaskRow` instance.
- Retrieve information about a task by invoking an appropriate method that belongs to the `TaskRow` object. For example, to get the task identifier value, invoke the `TaskRow` object's `getTaskId` method.

See also

"Retrieving Tasks Assigned to Users" on page 1094

"Quick Start (SOAP mode): Retrieving tasks assigned to users using the Java API" on page 402

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieving Task Information

You can dynamically retrieve information about tasks such the user who completed the task, the time and date that it was completed, and its identifier. By obtaining task information, you can track its details. For example, you can create a log file that specifies the user who completes a task and the time at which it was completed. This topic discusses how to retrieve information about completed tasks.

Summary of steps

To retrieve task information, perform the following tasks:

1 Include project files.

2 Create a TaskManagerQueryService and a TaskManager client API object.

3 Specify search criteria

4 Save file attachments.

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a TaskManagerQueryService and a TaskManager Client API object

To search for task information, create a `TaskManagerQueryService` and a `TaskManager` client API object.

Specify search criteria

To obtain information about a completed task, specify the user whom completed the task when setting connection properties required to invoke AEM Forms operations. That is, if you want to know what tasks tony blue completed, specify tonyb when defining connection settings.

You can also obtain information related to all tasks. For example, you can retrieve all completed tasks and determine the user who completed them (as opposed to retrieving tasks assigned to the user specified in the connection settings). To obtain all tasks, use an administrator account when defining connection settings. In addition, you can specify other search criteria such as the process name on which the task is based. For example, you can retrieve all tasks that are based on a specific process and then determine the users who completed them.

Note: *This topic discusses retrieving all completed tasks and obtaining information such as the user who completed the task.*

When retrieving all tasks, the result set may be very large. You can limit the result set using the Java and web service API.

Perform the search

After you define search criteria, you can search for tasks. All tasks that conform to the search criteria are returned in a list. You can get the status of each task when deciding whether you want to retrieve its information. For example, if you are only interested in obtaining information on completed tasks, ensure that the status of each task is 100 (which indicates that the task is completed).

Iterate through the returned tasks

Iterate through the list that contains the tasks to determine information about each task. You can, for example, determine the identifier value of each task.

See also

"Retrieve task information using the Java API" on page 1097

"Retrieve task information using the web service API" on page 1098

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Retrieving Tasks Assigned to Users" on page 1094

Retrieve task information using the Java API

Retrieve task information by using the Java API:

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManagerQueryService and a TaskManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManagerQueryService` object by invoking the `TaskManagerClientFactory` object's static `getQueryManager` method and passing the `ServiceClientFactory` object.
- Create a `TaskManager` object by invoking the `TaskManagerClientFactory` object's static `getTaskManager` method and passing the `ServiceClientFactory` object.

3 Specify search criteria

- Create a `TaskSearchFilter` object by using its constructor.
- Specify search criteria by invoking an appropriate method that belongs to the `TaskSearchFilter` object. For example, to specify the process on which a task is based, invoke the `TaskSearchFilter` object's `setServiceName` method and pass a string value that specifies the process name.
- Invoke the `TaskSearchFilter` object's `setAdminIgnoreAllAcls` method and pass `true` to enable all tasks to be returned (not just the user who is specified in the connection settings).

4 Perform the search

Search for tasks by invoking the `TaskManagerQueryServiceService` object's `taskSearch` method and passing the `TaskSearchFilter` object. This method returns a `java.util.List` object where each element is a `TaskRow` object that represents a task that conforms to the specified search criteria.

Note: *The `TaskSearch` method is a generic way to search for tasks. In contrast, the `TaskList` method is meant as a specialized way to retrieve tasks. For example, using the `TaskList` method, you can get all tasks that are assigned to a specific user. (See "Retrieving Tasks Assigned to Users" on page 1094.)*

5 Iterate through the returned tasks

- Create a `java.util.Iterator` object by invoking the `java.util.List` object's `iterator` method. This object lets you iterate through the `java.util.List` instance to retrieve tasks.
- Iterate through the `java.util.List` object to determine if there are tasks. If so, each element is a `TaskRow` instance.
- Get the task identifier value by invoking the `TaskRow` object's `getTaskId` method. This method returns a long value that specifies the task identifier value.
- Retrieve task information by invoking the `TaskManager` object's `getTaskInfo` method and passing the task identifier value.

See also

"Retrieving Task Information" on page 1096

"Quick Start (SOAP mode): Retrieving task information using the Java API" on page 411

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve task information using the web service API

Retrieve task information by using the web service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the TaskManager service WSDL. To create a proxy object that lets you invoke its operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/TaskManagerService?WSDL`

- Reference the Microsoft .NET client assembly.

2 Create a TaskManagerQueryService and a TaskManager Client API object

- Using the Microsoft .NET client assembly, create a `TaskManagerQueryServiceService` object by invoking its default constructor.
- Set the `TaskManagerQueryServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.
- Using the Microsoft .NET client assembly, create a `TaskManagerServiceService` object by invoking its default constructor.
- Set the `TaskManagerServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Specify search criteria

- Create a `TaskSearchFilter` object by using its constructor.

- Specify search criteria by assigning a value to an appropriate data member that belongs to the `TaskSearchFilter` object. For example, to specify the process on which a task is based, assign a string value that specifies the process name to the `TaskSearchFilter` object's `serviceName` data member.
- Assign the value `true` to the `TaskSearchFilter` object's `adminIgnoreAllAcls` data member to enable all tasks to be returned (not just the user who is specified in the connection settings).

4 Perform the search

Search for tasks by invoking the `TaskManagerQueryServiceService` object's `taskList` method and passing the `TaskFilter` object. This method returns an array of `Objects` where each element is a `TaskRow` object that represents a task that conforms to the specified search criteria.

5 Iterate through the returned tasks

- Iterate through the Object array by creating a loop structure and for each element, cast the element value to a `TaskRow` instance.
- Get the task identifier value by retrieving the value of the `TaskRow` object's `taskId` data member.
- Retrieve task information by invoking the `TaskManager` object's `getTaskInfo` method and passing the task identifier value.

See also

"Retrieving Task Information" on page 1096

Quick Start (Base64): Retrieving task information using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Modifying Form Data

You can programmatically modify form data that is associated with a specific task. For example, a user has an assigned task, which is either to approve or decline a mortgage application. While the task is sitting in the user queue waiting for the user to perform the task, you can programmatically modify form data. As a result, when the user opens the form in Workspace, the user can view the modified form data.

To modify form data, reference a valid XDP XML data source. Consider the following example mortgage application form.

To modify form data, you require a valid XDP XML data source that corresponds to the form. The following XML represents an XDP XML data source that corresponds to the example mortgage application form.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
- <xfa:data>
- <data>
    - <Layer>
        <closeDate>1/26/2007</closeDate>
        <lastName>Johnson</lastName>
        <firstName>Jerry</firstName>
        <mailingAddress>JJohnson@NoMailServer.com</mailingAddress>
        <city>New York</city>
        <zipCode>00501</zipCode>
        <state>NY</state>
        <dateBirth>26/08/1973</dateBirth>
        <middleInitials>D</middleInitials>
        <socialSecurityNumber>(555) 555-5555</socialSecurityNumber>
        <phoneNumber>5555550000</phoneNumber>
    </Layer>
    - <Mortgage>
        <mortgageAmount>295000.00</mortgageAmount>
        <monthlyMortgagePayment>1724.54</monthlyMortgagePayment>
        <purchasePrice>300000</purchasePrice>
        <downPayment>5000</downPayment>
        <term>25</term>
        <interestRate>5.00</interestRate>
    </Mortgage>
</data>
</xfa:data>
</xfa:datasets>
```

Summary of steps

To modify form data, perform the following tasks:

1. Include project files.
2. Create a TaskManager Client API object.
3. Specify form data.
4. Specify the form design.
5. Save the task.

Include project files

Include necessary files into your development project. Because you are creating a client application using Java, then include the necessary JAR files.

Create a TaskManagerService Client API object

Before you can programmatically modify form data, you must create a `TaskManager` object.

To modify form data associated with a task, specify the user name and the password value of the user to whom the task is assigned when setting connection properties. For example, to retrieve form data from a task assigned to tony blue, use tony blue's user name and password. If you specify a user that is not assigned the task, an exception is thrown.

Specify form data

To modify form data, reference a valid XML data source that contains form data. If an element located in the XML data source does not correspond to a field in the form, the element is ignored.

Specify the form design

To modify form data, obtain a form instance that is associated with a task. After you obtain the form instance, specify the XML data source location.

Save the task

Modify the form data by saving the task. When you save the task, specify the task identifier and the form instance that represents the form that contains the modified form data.

See also

"Modify form data using the Java API" on page 1101

"Modify form data using the web service API" on page 1102

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Retrieving Tasks Assigned to Users" on page 1094

Modify form data using the Java API

Modify form data by using the Java API:

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManagerService Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManagerService` object by invoking the `TaskManagerClientFactory` object's static `getTaskManager` method and passing the `ServiceClientFactory` object.

3 Specify form data

- Create a `java.io.FileInputStream` object by using its constructor and passing a string value that specifies the location of the XML file that contains form data.
- Create `com.adobe.idp.Document` object to store form data by using its constructor and passing the `java.io.FileInputStream` object.
- Create a `java.io.InputStream` object that contains form data by invoking the `com.adobe.idp.Document` object's `getInputStream` method.
- Create a byte array that is used to store form data. Allocate the size of the `java.io.InputStream` object to the byte array.
- Populate the byte array by invoking the `java.io.InputStream` object's `read` method and passing the byte array.

4 Specify the form design

- Create a new `FormInstance` object by invoking the `TaskManagerService` object's `getEmptyForm` method. This represents the form that will be populated with form data and used in the task.
- Specify the form design to use by invoking the `FormInstance` object's `setTemplatePath` and pass a string value that specifies the location of the XDP file (because the form instance is based on an empty form, the form design must be specified by invoking the `setTemplatePath` method).

- Specify the form data by invoking the `FormInstance` object's `setXFAData` method and passing the byte array that contains the form data.
- Invoke the `FormInstance` object's `setDocument` and pass the `com.adobe.idp.Document` object that contains form data.

5 Save the task

Save the task so that the form that contains the modified data is displayed by invoking the `TaskManagerService` object's `save` method and passing the following values:

- A long value that specifies the task identifier.
- The `FormInstance` object that represents the form.

This method returns a `SaveTaskResult` object.

See also

"Modifying Form Data" on page 1099

"Quick Start (SOAP mode): Modifying form data using the Java API" on page 407

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500.

Modify form data using the web service API

Modify form data by using the web service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the TaskManager service WSDL. To create a proxy object that lets you invoke its operations by using Base64 encoding, specify this WSDL definition:

  `http://localhost:8080/soap/services/TaskManagerService?WSDL`

- Reference the Microsoft .NET client assembly.

2 Create a TaskManagerService Client API object

- Using the Microsoft .NET client assembly, create a `TaskManagerServiceService` object by invoking its default constructor.
- Set the `TaskManagerServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Specify form data

- Create a `BLOB` object by using its constructor. The `BLOB` object is used to store form data.
- Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that specifies the location of the XML file that contains form data and the mode in which to open the file.
- Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
- Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
- Create a byte array that is used to store form data. Allocate the size of the `java.io.InputStream` object to the byte array.

- Populate the byte array by invoking the `java.io.InputStream` object's `read` method and passing the byte array.
- Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.

4 Specify the form design

- Get the form used in the task by invoking the `TaskManager` object's `getFormInstanceForTask` method and passing the following values:
  - The task identifier value that specifies the task.
  - A `System.Boolean` value that specifies whether the task identifier was specified.
  - The form identifier value. Typically the value is the same as the task.
  - A `System.Boolean` value that specifies whether the form identifier was specified.
  - A `System.Boolean` value that specifies whether to retrieve form data (for this task, specify `true`).
- A `System.Boolean` value that specifies whether the previous parameter was specified (for this task, specify `true`).

The `getFormInstanceForTask` method returns a `FormInstance` object.

- Specify the form data by assigning the `FormInstance` object's `XFAData` method with the byte array that contains the form data.
- Assign the `FormInstance` object's `document` data member with the `BLOB` object that contains form data.

5 Save the task

Save the task so that the form that contains the modified data is displayed by invoking the `TaskManagerService` object's `save` method and passing the following values:

- A long value that specifies the task identifier.
- A `System.Boolean` value that specifies whether the task identifier is specified.
- The `FormInstance` object that represents the form.

This method returns a `SaveTaskResult` object.

See also

"Modifying Form Data" on page 1099

Quick Start (Base64): Modifying form data using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Retrieving File Attachments from Tasks

When a process instance is started from within Workspace, a user can attach files, such as image files, which lets the user view the files while completing a task. You can retrieve file attachments from tasks by using the Java API and web service API. After you retrieve file attachments, you can process them to meet your business requirements. For example, you can retrieve a file attachment and save it as a local file.

Note: *The name of the service that is invoked when retrieving file attachments from a task is TaskManagerService. To create a proxy object that lets you invoke its operations by using a web service, specify the following WSDL definition:* `http://localhost:8080/soap/services/TaskManagerService?WSDL`. *(See "Invoking AEM Forms using Web Services" on page 514.)*

Summary of steps

To retrieve file attachments from tasks, perform the following tasks:

1 Include project files.

2 Create a TaskManager Client API object.

3 Retrieve the file attachments.

Include project files

Include necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

Create a TaskManager Client API object

Before you can programmatically retrieve file attachments, you must create a `TaskManager` object.

Retrieve the file attachments

To retrieve form data from a task, reference the task that contains the form by using the task identifier. You can determine the task identifier by retrieving tasks that are assigned to a specific user. (See "Retrieving Tasks Assigned to Users" on page 1094.)

After you obtain a file attachment, you can save it as a local file (or perform another task that meets your business requirements).

See also

"Retrieve file attachments from tasks using the Java API" on page 1104

"Retrieve file attachments from tasks using the web service API" on page 1105

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Retrieving Tasks Assigned to Users" on page 1094

Retrieve file attachments from tasks using the Java API

Retrieve file attachments from tasks by using the Java API:

1 Include project files

Include client JAR files, such as adobe-taskmanager-client-sdk.jar, in your Java project's class path.

2 Create a TaskManager Client API object

- Create a `ServiceClientFactory` object that contains connection properties.
- Create a `TaskManager` object by invoking the `TaskManagerClientFactory` object's static `getTaskManager` method and passing the `ServiceClientFactory` object.

3 Retrieve the file attachments

- Retrieve file attachments by invoking the `TaskManager` object's `getAttachmentListForTask` method and passing the task identifier value. This method returns a `java.util.List` object where each element is a `com.adobe.idp.Document` object that contains a file attachment. You can determine the task identifier by retrieving tasks that are assigned to a specific user.
- Iterate through the `java.util.List` object and for each element, cast the element value to a `com.adobe.idp.Document` instance.

See also

"Retrieving File Attachments from Tasks" on page 1103

"Quick Start (SOAP mode): Retrieving file attachments from tasks using the Java API" on page 409

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

"Retrieving Tasks Assigned to Users" on page 1094

Retrieve file attachments from tasks using the web service API

Retrieve file attachments from tasks by using the the web service API:

1 Include project files

- Create a Microsoft .NET client assembly that consumes the TaskManagerService WSDL.
- Reference the Microsoft .NET client assembly.

2 Create a TaskManager Client API object

- Using the Microsoft .NET client assembly, create a `TaskManagerServiceService` object by invoking its default constructor.
- Set the `TaskManagerServiceService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Retrieve the file attachments

- Retrieve file attachments by invoking the `TaskManagerServiceService` object's `getAttachmentListForTask` method and passing the task identifier value and a `Boolean` value that specifies `true`. This method returns an `Object` array where each element is a `BLOB` object that contains a file attachment. You can determine the task identifier by retrieving tasks that are assigned to a specific user. (See "Retrieving Tasks Assigned to Users" on page 1094.)
- Iterate through the `Object` array by creating a loop structure and, for each element, cast the element value to a `BLOB` instance.

See also

"Retrieving File Attachments from Tasks" on page 1103

Quick Start (Base64): Retrieving file attachments from tasks using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

Deploying applications

Using the Java and web service API, you can programmatically deploy a Forms application (an LCA file) to AEM Forms. Programmatically deploying a Forms application results in an application being deployed to AEM Forms as though you imported the application using Applications and Services, which is accessed by logging in to administration console. When programmatically deploying an application, specify an Administrator or Super Administrator when setting connection settings.

To deploy a Forms application, reference an existing AEM Forms archive (LCA) file. For information about creating an LCA file, see Workbench Help.

Note: *You can invoke the Application Manager service using web services. To demonstrate how to deploy a AEM Forms application using web services, the Deploying applications web service quick start uses SwaRef. (See "Invoking AEM Forms using SwaRef" on page 531.)*

Summary of steps

To programmatically deploy a Forms application, perform the following tasks:

1. Include project files.
2. Create required AEM Forms Client API objects.
3. Retrieve an existing archive file.
4. Import the Forms application.
5. Check the status of the AEM Forms application.

Include project files

Include necessary files into your development project. Because you are creating a client application by using Java, include the necessary JAR files.

Create required AEM Forms Client API objects

Before you can programmatically import an application, create an `ApplicationManager` object. Ensure that you specify an administrator user name when setting connection properties. (See "Setting connection properties" on page 500.)

Retrieve an existing archive file

To import an application, reference a valid LCA file. A LCA file is created using Workbench. (See workbench Help .)

Import the AEM Forms application

After you reference an LCA file, you can import the application. After you import the application, you can check its status to ensure that the application was successfully imported. If the status code is 1, the application was successfully imported.

Check the status

After you import an AEM Forms application, you can check its status to ensure that the application was successfully deployed. The following list specfies possible values:

- 0 - The status of the application is unknown.
- 1 - The application is successfully deployed.
- 2 - An import error occurred.
- 3 - The application was successfully exported.
- 4 - An error occurred when the application was exported.
- 5 - The application was removed successfully.
- 6 - An error occurred when the application was removed.
- 7 - The application was successfully previewed .
- 8- A status preview error occurred.
- 9 - Information about the appliation was successfully retrieved.
- 10 - An error occurred when information was retrieved.

Note: *When importing an application, values 0, 1 or 2 are returned.*

See also

"Deploy a Forms application using the Java API" on page 1107

"Deploy an AEM Forms application using the web service API" on page 1108

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Deploy a Forms application using the Java API

Deploy a Forms application by using the Application Manager API (Java):

1 Include project files

Include client JAR files, such as adobe-applicationmanager-client-sdk.jar, in your Java project's class path.

2 Create required AEM Forms Client API objects

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `ApplicationManager` object by using its constructor and passing the `ServiceClientFactory` object.
- Create a `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.
- Create a `ServiceRegistryClient` object by using its constructor and passing a `ServiceClientFactory` object.

3 Retrieve an existing archive file

- Reference an LCA file that represents the application to import by creating a `FileInputStream` object by using its constructor. Pass a string value that specifies the location of the LCA file.
- Create a `com.adobe.idp.Document` object by using its constructor and passing the `FileInputStream` object.

4 Import the Forms application

Import the application by invoking the `ApplicationManager` object's `importApplicationArchive` method. Pass the `com.adobe.idp.Document` object that contains the LCA file. This method returns an `ApplicationStatus` object that specifies whether the application was successfully imported.

5 Check the status

Check the status of the application by invoking the `ApplicationStatus` object's `getStatusCode` method. If this method returns the value 1, the application is successfully imported.

See also

"Deploying applications" on page 1105

"Quick Start (SOAP mode): Deploying Applications using the Java API" on page 4

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Deploy an AEM Forms application using the web service API

Deploy a Forms application by using the Application Manager API (web services):

1 Include project files

Create Java proxy classes using JAX-WS. Ensure that the Java proxy classes support SwaRef.

Ensure that you use the following WSDL defination:
`http://localhost:8080/soap/services/ApplicationManager?WSDL&lc_version=9.0.1`.

Note: *Replace `localhost` with the IP address of the server hosting AEM Forms.*

2 Create required AEM Forms Client API objects

- Create an `ApplicationManagerService` object by using its constructor.
- Create an `ApplicationManager` object by invoking the `ApplicationManagerService` object's `getApplicationManager` method.
- Set authentication values by invoking the `ApplicationManager` object's `getRequestContext` method.

3 Retrieve an existing archive file

- Retrieve the LCA file by creating a `java.io.File` object by using its constructor. Pass a string value that specifies the location of the PDF document.
- Create a `javax.activation.DataSource` object by using the `FileDataSource` constructor. Pass the `java.io.File` object.
- Create a `javax.activation.DataHandler` object by using its constructor and passing the `javax.activation.DataSource` object.
- Create a `BLOB` object by using its constructor.
- Populate the `BLOB` object by invoking its `setSwaRef` method and passing the `javax.activation.DataHandler` object.

4 Import the AEM Forms application

Import the application by invoking the `ApplicationManager` object's `importApplicationArchiveDocument` method. Pass the `BLOB` object that contains the LCA file. This method returns an `ApplicationStatus` object that specifies whether the application was successfully imported.

5 Check the status

Check the status of the application by invoking the `ApplicationStatus` object's `getStatusCode` method. If this method returns the value 1, the application is successfully imported.

See also

"Deploying applications" on page 1105

Quick Start (SwaRef): Deploying applications using the web service API

"Invoking AEM Forms using SwaRef" on page 531

Removing Applications

You can programmatically remove an application by using the Java and web service API. Because all applications are visible to others who use the forms server, you should exercise caution before you remove an application.

Note: *The name of the service that is invoked when removing an application is ApplicationManager. Therefore, to create a proxy object that lets you invoke its operations by using a web service, specify the following WSDL definition:* `http://localhost:8080/soap/services/ApplicationManager?WSDL`. *(See "Invoking AEM Forms using Web Services" on page 514.)*

Summary of steps

To remove an application, perform the following tasks:

1 Include project files.

2 Create an ApplicationManager Client API object.

3 Locate the application to remove.

4 Remove the application.

Include project files

Include the necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure you include the proxy files.

Create an ApplicationManager Client API object.

To remove an application, create an `ApplicationManager` object.

Locate the application to remove

Obtain the identifier of the application to remove. If you do not know the identifier value, you can retrieve all applications and search by name. Using the application name, you can obtain the identifier value that corresponds to the application you want to remove.

Remove the application

You can remove an application by using its identifier value.

See also

"Remove an application using the Java API" on page 1109

"Remove an application using the web service API" on page 1110

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Remove an application using the Java API

Remove a AEM Forms application by using the Application Manager API (Java):

1 Include project files

Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an ApplicationManager Client API object.

- Create a `ServiceClientFactory` object that contains connection properties. (See "Setting connection properties" on page 500.)
- Create an `ApplicationManager` object by invoking its constructor and passing the `ServiceClientFactory` object that contains connection properties.

3 Locate the application to remove

- Retrieve all applications by invoking the `ApplicationManager` object's `getApplications` method. This method returns a `java.util.List` object where each element is an `Application` object.
- Iterate through the list by casting each element to an `Application` object.
- Create an `ApplicationId` object by invoking the `Application` object's `getApplicationId` method.
- Get the application name by invoking the `ApplicationId` object's `getApplicationName` method. This method returns a string value that specifies the application name. If the application name corresponds to the application that you want to remove, you can use the `ApplicationId` object to remove the application.

4 Remove the application

After you locate the application to remove, invoke the `ApplicationManager` object's `removeApplication` method and pass the `ApplicationId` object that represents the application to remove.

See also

"Removing Applications" on page 1108

"Quick Start (SOAP mode): Removing an application using the Java API" on page 6

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Remove an application using the web service API

Remove a Forms application by using the Application Manager API (web service):

1 Include project files

- Create a Microsoft .NET client assembly that consumes the ApplicationManagerService WSDL.
- Reference the Microsoft .NET client assembly.

2 Create an ApplicationManager Client API object.

- Using the Microsoft .NET client assembly, create an `ApplicationManagerService` object by invoking its default constructor.
- Set the `ApplicationManagerService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3 Locate the application to remove

- Retrieve all applications by invoking the `ApplicationManagerService` object's `getApplications` method. This method returns an array of `Objects` where each element is a `Application` object that represents an application.
- Iterate through the array by casting each element to an `Application` object.
- Create an `ApplicationId` object by getting the value of the `Application` object's `applicationId` data member.
- Get the application name by getting the value of the `ApplicationId` object's `applicationName` data member. If the application name corresponds to the application that you want to remove, you can use the `ApplicationId` object to remove the application.

4 Remove the application

After you locate the application to remove, invoke the `ApplicationManager` object's `removeApplication` method and pass the `ApplicationId` object that represents the application to remove.

See also

"Removing Applications" on page 1108

Quick Start (Base64): Removing an application using the web service API

"Invoking AEM Forms using Base64 encoding" on page 525

"Creating a .NET client assembly that uses Base64 encoding" on page 525

Programmatically Managing Endpoints

About Endpoint Registry Service

The Endpoint Registry service provides the ability to programmatically manage endpoints. You can, for example, add the following types of endpoints to a service:

- EJB
- SOAP
- Watched Folder
- Email
- (Deprecated for AEM forms) Remoting
- Task Manager

  Note: *SOAP, EJB, and (Deprecated for AEM forms on JEE) Remoting endpoints are automatically created for each activated service. The SOAP and EJB endpoints enable SOAP and EJB for all service operations.*

  A Remoting endpoint enables Flex clients to invoke operations on the AEM Forms service that the endpoint is added to. A Flex destination with the same name as the endpoint is created and Flex clients can create RemoteObjects that point to this destination to invoke operations on the relevant service.

  The Email, Task Manager, and Watched Folder endpoints expose only a specific operation of the service. Adding these endpoints requires a second configuration step to select a method to invoke, set configuration parameters, and specify input and output parameter mappings.

  You can organize TaskManager endpoints into groups called *categories*. These categories are then exposed to Workspace through TaskManager, with end users seeing the TaskManager endpoints as they are categorized. Within Workspace, end users see these categories in the navigation pane. The endpoints within each category are displayed as process cards on the Start Processes page in Workspace.

  You can accomplish these tasks using the Endpoint Registry service:

- Add EJB endpoints. (See "Adding EJB Endpoints" on page 1112.)
- Add SOAP endpoints. (See "Adding SOAP Endpoints" on page 1114.)
- Add Watched Folder endpoints (See "Adding Watched Folder Endpoints" on page 1116.)
- Add Email endpoints. (See "Adding Email Endpoints" on page 1122.)
- Add Remoting endpoints. (See "Adding Remoting Endpoints" on page 1129.)
- Add TaskManager endpoints (See "Adding TaskManager Endpoints" on page 1132.)
- Modify endpoints (See "Modifying Endpoints" on page 1135.)
- Remove endpoints (See "Removing Endpoints" on page 1137.)
- Retrieve endpoint connector information (See "Retrieving Endpoint Connector Information" on page 1139.)

Adding EJB Endpoints

You can programmatically add an EJB endpoint to a service by using the AEM Forms Java API. By adding an EJB endpoint to a service, you are enabling a client application to invoke the service by using the EJB mode. That is, when setting connection properties that are required to invoke AEM Forms, you can select the EJB mode. (See "Setting connection properties" on page 500.)

Note: *You cannot add an EJB endpoint by using web services.*

Note: *Typically, an EJB endpoint is added to a service by default, However, an EJB endpoint can be added to a process that is programmatically deployed or when an EJB endpoint was removed and has to be added again.*

Summary of steps

To add an EJB endpoint to a service, perform the following tasks:

1. Include project files.
2. Create an `EndpointRegistry Client` object.
3. Set EJB endpoint attributes.
4. Create an EJB endpoint.
5. Enable the endpoint.

Include project files

Include necessary files in your development project. The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

Before you can programmatically add an EJB endpoint, you must create an `EndpointRegistryClient` object.

Set EJB endpoint attributes

To create an EJB endpoint for a service, specify the following values:

- **Connector identifier**: Specifies the type of endpoint to create. To create an EJB endpoint, specify `EJB`.
- **Description**: Specifies the endpoint description.
- **Name**: Specifies the name of the endpoint.
- **Service identifier**: Specifies the service to which the endpoint belongs.
- **Operation name**: Specifies the name of the operation that is invoked by using the endpoint. When creating an EJB endpoint, specify a wildcard character (`*`). However, if you want to specify a specific operation as opposed to invoking all service operations, specify the name of the operation as opposed to using the wildcard character (`*`).

Create an EJB endpoint

After you set EJB endpoint attributes, you can create an EJB endpoint for a service.

Enable the endpoint

After you create a new endpoint, you must enable it. After you enable the endpoint, it can be used to invoke the service. After you enable the endpoint, you can view it within administration console.

See also

"Adding an EJB endpoint using the Java API" on page 1113

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Adding an EJB endpoint using the Java API

Add an EJB endpoint by using the Java API:

1 Include project files.

Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path. (

2 Create an EndpointRegistry Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set EJB endpoint attributes.

- Create a `CreateEndpointInfo` object by using its constructor.
- Specify the connector identifier value by invoking the `CreateEndpointInfo` object's `setConnectorId` method and passing the string value `EJB`.
- Specify the description of the endpoint by invoking the `CreateEndpointInfo` object's `setDescription` method and passing a string value that describes the endpoint.
- Specify the name of the endpoint by invoking the `CreateEndpointInfo` object's `setName` method and passing a string value that specifies the name.
- Specify the service to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setServiceId` method and passing a string value that specifies the service name.
- Specify the operation that is invoked by invoking the `CreateEndpointInfo` object's `setOperationName` method and pass a string value that specifies the operation name. For SOAP and EJB endpoints, specify a wildcard character (`*`), which implies all operations.

4 Create an EJB endpoint.

Create the endpoint by invoking the `EndpointRegistryClient` object's `createEndpoint` method and passing the `CreateEndpointInfo` object. This method returns an `Endpoint` object that represents the new EJB endpoint.

5 Enable the endpoint.

Enable the endpoint by invoking the `EndpointRegistryClient` object's enable method and passing the `Endpoint` object that was returned by the `createEndpoint` method.

See also

"Summary of steps" on page 1112

"QuickStart: Adding an EJB endpoint using the Java API" on page 131

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Adding SOAP Endpoints

You can programmatically add a SOAP endpoint to a service by using the AEM Forms Java API. By adding a SOAP endpoint, you enable a client application to invoke the service by using the SOAP mode. That is, when setting connection properties required to invoke AEM Forms, you can select the SOAP mode.

Note: *You cannot add a SOAP endpoint by using web services.*

Note: *Typically, a SOAP endpoint is added to a service by default, However, a SOAP endpoint can be added to a process that is programmatically deployed or when a SOAP endpoint was removed and has to be added again.*

Summary of steps

To add a SOAP endpoint to a service, perform the following tasks:

1. Include project files.
2. Create an `EndpointRegistryClient` object.
3. Set SOAP endpoint attributes.
4. Create a SOAP endpoint.
5. Enable the endpoint.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

These JAR files are required to create a SOAP endpoint. However, you require addition JAR files if you use the SOAP endpoint to invoke the service. For information about AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

To programmatically add a SOAP endpoint to a service, you must create an `EndpointRegistryClient` object.

Set SOAP endpoint attributes

To add a SOAP endpoint to a service, specify the following values:

- **Connector identifier value**: Specifies the type of endpoint to create. To create a SOAP endpoint, specify `SOAP`.
- **Description**: Specifies the endpoint description.
- **Name**: Specifies the endpoint name.
- **Service identifier value**: Specifies the service to which the endpoint belongs.

- **Operation name**: Specifies the name of the operation that is invoked by using the endpoint. When creating a SOAP endpoint, specify a wildcard character (`*`). However, if you want to specify a specific operation as opposed to invoking all service operations, specify the name of the operation as opposed to using the wildcard character (`*`).

Create a SOAP endpoint

After you set SOAP endpoint attributes, you can create a SOAP endpoint.

Enable the endpoint

After you create a new endpoint, you must enable it. When the endpoint is enabled, it can be used to invoke the service. After you enable the endpoint, you can view see it within administration console.

See also

"Add a SOAP endpoint using the Java API" on page 1115

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Add a SOAP endpoint using the Java API

Add a SOAP endpoint to a service by using the Java API:

1 Include project files.

Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an EndpointRegistry Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set SOAP endpoint attributes.

- Create a `CreateEndpointInfo` object by using its constructor.
- Specify the connector identifier value by invoking the `CreateEndpointInfo` object's `setConnectorId` method and passing the string value `SOAP`.
- Specify the description of the endpoint by invoking the `CreateEndpointInfo` object's `setDescription` method and passing a string value that describes the endpoint.
- Specify the name of the endpoint by invoking the `CreateEndpointInfo` object's `setName` method and passing a string value that specifies the name.
- Specify the service to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setServiceId` method and passing a string value that specifies the service name.
- Specify the operation that is invoked by invoking the `CreateEndpointInfo` object's `setOperationName` method and passing a string value that specifies the operation name. For SOAP and EJB endpoints, specify a wildcard character (`*`), which implies all operations.

4 Create a SOAP endpoint.

Create the endpoint by invoking the `EndpointRegistryClient` object's `createEndpoint` method and passing the `CreateEndpointInfo` object. This method returns an `Endpoint` object that represents the new SOAP endpoint.

5 Enable the endpoint.

Enable the endpoint by invoking the `EndpointRegistryClient` object's enable method and pass the `Endpoint` object that was returned by the `createEndpoint` method.

See also

"Summary of steps" on page 1114

"QuickStart: Adding a SOAP endpoint using the Java API" on page 133

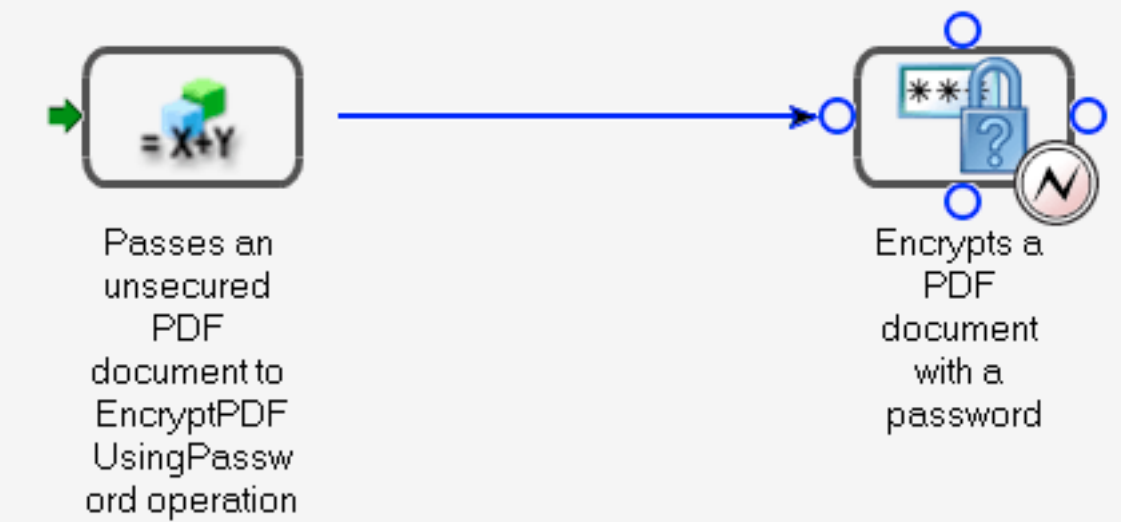
"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Adding Watched Folder Endpoints

You can programmatically add a Watched Folder endpoint to a service by using the AEM Forms Java API. By adding a Watched Folder endpoint, you enable users to place a file (such as a PDF file) in a folder. When the file is placed in the folder, the configured service is then invoked and manipulates the file. After the service performs the specified operation, it saves the modified file in a specified output folder. A watched folder is configured to be scanned at a fixed rate interval or with a cron schedule, such as every Monday, Wednesday, and Friday at noon.

For the purposes of programmatically adding a Watched Folder endpoint to a service, consider the following short-lived process named *EncryptDocument*. (See "Understanding AEM Forms Processes" on page 441.)



This process accepts an unsecured PDF document as an input value and then passes the unsecured PDF document to the Encryption service's `EncryptPDFUsingPassword` operation. The PDF document is encrypted with a password, and the password-encrypted PDF document is the output value of this process. The name of the input value (the unsecured PDF document) is `InDoc` and the data type is `com.adobe.idp.Document`. The name of the output value (the password-encrypted PDF document) is `SecuredDoc` and the data type is `com.adobe.idp.Document`.

Note: *You cannot add a Watched Folder endpoint by using web services.*

Summary of steps

To add a Watched Folder endpoint to a service, perform the following tasks:

1 Include project files.

2 Create an `EndpointRegistryClient` object.

3 Set Watched Folder endpoint attributes.

4 Specify configuration values.

5 Define input parameter values.

6 Define an output parameter value.

7 Create a Watched Folder endpoint.

8 Enable the endpoint.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

To programmatically add a Watched Folder endpoint, you must create an `EndpointRegistryClient` object.

Set Watched Folder endpoint attributes

To create a Watched Folder endpoint for a service, specify the following values:

- **Connector identifier**: Specifies the type of endpoint that is created. To create a Watched Folder endpoint, specify `WatchedFolder`.
- **Description**: Specifies the description of the endpoint.
- **Name**: Specifies the name of the endpoint.
- **Service identifier**: Specifies the service to which the endpoint belongs. For example, to add a Watched Folder endpoint to the process that is introduced in this section (a process becomes a service when activated using Workbench), specify `EncryptDocument`.
- **Operation name**: Specifies the name of the operation that is invoked by using the endpoint. Typically, when creating a Watched Folder endpoint for a service that originated from a process created in Workbench, the name of the operation is `invoke`.

Specify configuration values

You must specify configuration values for a Watched Folder endpoint when programmatically adding a Watched Folder endpoint to a service. These configuration values are specified by an administrator if a Watched Folder endpoint is added by using administration console.

The following list specifies configuration values that are set when programmatically adding a Watched Folder endpoint to a service:

- **url**: Specifies the watched folder location. In a clustered environment, this value must point to a shared network folder that is accessible from every computer in the cluster.

- **asynchronous**: Identifies the invocation type as asynchronous or synchronous. Transient and synchronous processes can only be invoked synchronously. The default value is true. Asynchronous is recommended.
- **cronExpression**: Used by quartz to schedule the polling of the input directory. For details about configuring the cron expression, see http://quartz.sourceforge.net/javadoc/org/quartz/CronTrigger.html.
- **purgeDuration**: This is a mandatory attribute. Files and folders in the result folder are purged when they are older than this value. This value is measured in days. This attribute is useful in ensuring the result folder does not become full. A value of -1 days indicates to never delete the results folder. The default value is -1.
- **repeatInterval**: The interval, in seconds, for scanning the Watched Folder for input. Unless throttling is enabled, this value should be longer than the time to process an average job; otherwise, the system may become overloaded. The default value is 5.
- **repeatCount**: The number of times a Watched Folder scans the folder or directory. A value of -1 indicates indefinite scanning. The default value is -1.
- **throttleOn**: Limits the number of Watched Folder jobs that can be processed at any given time. The maximum number of jobs is determined by the batchSize value.
- **userName**: The user name used when invoking a target service from the Watched Folder. This value is mandatory. The default value is SuperAdmin.
- **domainName**: The user's domain. This value is mandatory. The default value is DefaultDom.
- **batchSize**: The number of files or folders to be picked up per scan. Use this value to prevent an overload on the system; scanning too many files at one time can result in a crash. The default value is 2.
- **waitTime**: The time, in milliseconds, to wait before scanning a folder or file after creation. For example, if wait time is 36,000,000 milliseconds (one hour) and the file was created one minute ago, this file is picked up after 59 or more minutes have passed. This attribute is useful to ensure that a file or folder is completely copied to the input folder. For example, if you have a large file to process and the file takes ten minutes to download, set the wait time to 10*60 *1000 milliseconds. This setting prevents the watched folder from scanning the file if it has not been waiting for ten minutes. The default value is 0.
- **excludeFilePattern**: The pattern that a watched folder uses to determine which files and folders to scan and pick up. Any file or folder that has this pattern will not be scanned for processing. This setting is useful when the input is a folder that contains multiple files. The contents of the folder can be copied into a folder that has a name that will be picked up by the watched folder. This step prevents the watched folder from picking up a folder for processing before the folder is completely copied into the input folder. For example, if the excludeFilePattern value is `data*`, all files and folders that match `data*` are not picked up. This includes files and folders named `data1`, `data2`, and so on. Additionally, the pattern can be supplemented with wildcard patterns to specify file patterns. The watched folder modifies the regular expression to support wildcard patterns such as `*.*` and `*.pdf`. These wildcard patterns are not supported by regular expressions.
- **includeFilePattern**: The pattern that the watched folder uses to determine which folders and files to scan and pick up. For example, if this value is `*`, all files and folders that match `input*` are picked up. This includes files and folders named `input1`, `input2`, and so on. The default value is `*`. This value indicates all files and folders. Additionally, the pattern can be supplemented with wildcard patterns to specify file patterns. The watched folder modifies the regular expression to support wildcard patterns such as `*.*` and `*.pdf`. These wildcard patterns are not supported by regular expressions. This value is a mandatory.
- **resultFolderName**: The folder where the saved results are stored. This location can be an absolute or a relative directory path. If the results do not appear in this folder, check the failure folder. Read-only files are not processed and will be saved in the failure folder. The default value is `result/%Y/%M/%D/`. This is the results folder inside the watched folder.

- **preserveFolderName**: The location where files are stored after successful scanning and pickup. This location can be an absolute, a relative, or a null directory path. The default value is `preserve/%Y/%M/%D/`.
- **failureFolderName**: The folder where failure files are saved. This location is always relative to the watched folder. Read-only files are not processed and will be saved in the failure folder. The default value is `failure/%Y/%M/%D/`.
- **preserveOnFailure**: Preserve input files in case of failure to execute the operation on a service. The default value is true.
- **overwriteDuplicateFilename**: When set to true, files in the results folder and preserve folder are overwritten. When set to false, files and folders that have a numeric index suffix are used for the name. The default value is false.

Define input parameter values

When creating a Watched Folder endpoint, you must define input parameter values. That is, you must describe the input values that are passed to the operation that is invoked by the watched folder. For example, consider the process introduced in this topic. It has one input value named `InDoc` and its data type is `com.adobe.idp.Document`. When creating a Watched Folder endpoint for this process (after a process is activated, it becomes a service), you must define the input parameter value.

To define input parameter values required for a Watched Folder endpoint, specify the following values:

Input parameter name: The name of the input parameter. The name of an input value is specified in Workbench for a process. If the input value belongs to a service operation (a service that is not a process created in Workbench), the input name is specified in the component.xml file. For example, the name of the input parameter for the process introduced in this section is `InDoc`.

Mapping type: Used to configure the input values required to invoke the service operation. There are two types of mapping types:

- `Literal`: The Watched Folder endpoint uses the value entered in the field as it is displayed. All basic Java types are supported. For example, if an API uses input such as String, long, int, and Boolean, the string is converted into the proper type and the service is invoked.
- `Variable`: The value entered is a file pattern that the watched folder uses to pick the input. For example, if you select Variable for the mapping type and the input document must be a PDF file, you can specify `*.pdf` as the mapping value.

Mapping value: Specifies the value of the mapping type. For example, if you select a `Variable` mapping type, you can specify `*.pdf` as the file pattern.

Data type: Specifies the data type of the input value(s). For example, the data type of the input value of the process introduced in this section is `com.adobe.idp.Document`.

Define an output parameter value

When creating a Watched Folder endpoint, you must define an output parameter value. That is, you must describe the output value that is returned by the service that is invoked by the Watched Folder endpoint. For example, consider the process introduced in this topic. It has an output value named `SecuredDoc` and its data type is `com.adobe.idp.Document`. When creating a Watched Folder endpoint for this process (after a process is activated, it becomes a service), you must define the output parameter value.

To define an output parameter value required for a Watched Folder endpoint, specify the following values:

Output parameter name: The name of the output parameter. The name of a process output value is specified in Workbench. If the output value belongs to a service operation (a service that is not a process created in Workbench), the output name is specified in the component.xml file. For example, the name of the output parameter for the process introduced in this section is `SecuredDoc`.

Mapping type: Used to configure the output of the service and operation. The following options are available:

- If the service returns a single object (a single document), the pattern is `%F.pdf` and the source destination is sourcefilename.pdf. For example, the process introduced in this section returns a single document. As a result, the mapping type can be defined as `%F.pdf` (`%F` means use the given file name). The pattern `%E` specifies the extension of the input document.
- If the service returns a list, the pattern is `Result\%F\`, and the source destination is Result\sourcefilename\source1 (output 1) and Result\sourcefilename\source2 (output 2).
- If the service returns a map, the pattern is `Result\%F\`, and the source destination is Result\sourcefilename\file1 and Result\sourcefilename\file2. If the map has more than one object, the pattern is `Result\%F.pdf` and the source destination is Result\sourcefilename1.pdf (output 1), Result\sourcefilenam2.pdf (output 2), and so on.

Data type: Specifies the data type of the return value. For example, the data type of the return value of the process introduced in this section is `com.adobe.idp.Document`.

Create a Watched Folder endpoint

After you set the endpoint's attributes, configuration values, and define input and output parameter values, you must create the Watched Folder endpoint.

Enable the endpoint

After you create a Watched Folder endpoint, you must enable it. When the endpoint is enabled, it can be used to invoke the service. After you enable the endpoint, you can view it within administration console.

See also

"Add a Watched Folder endpoint using the Java API" on page 1120

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Add a Watched Folder endpoint using the Java API

Add a Watched Folder endpoint by using the AEM Forms Java API:

1 Include project files.

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an EndpointRegistry Client object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set Watched Folder endpoint attributes.

   - Create a `CreateEndpointInfo` object by using its constructor.
   - Specify the connector identifier value by invoking the `CreateEndpointInfo` object's `setConnectorId` method and passing the string value `WatchedFolder`.
   - Specify the description of the endpoint by invoking the `CreateEndpointInfo` object's `setDescription` method and passing a string value that describes the endpoint.
   - Specify the name of the endpoint by invoking the `CreateEndpointInfo` object's `setName` method and passing a string value that specifies the name.

- Specify the service to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setServiceId` method and passing a string value that specifies the service name.
- Specify the operation that is invoked by invoking the `CreateEndpointInfo` object's `setOperationName` method and passing a string value that specifies the operation name. Typically, when creating a Watched Folder endpoint for a service that originated from a process created in Workbench, the name of the operation is invoke.

4 Specify configuration values.

For each configuration value to set for the Watched Folder endpoint, you must invoke the `CreateEndpointInfo` object's `setConfigParameterAsText` method. For example, to set the `url` configuration value, invoke the `CreateEndpointInfo` object's `setConfigParameterAsText` method and pass the following string values:

- A string value that specifies the name of the configuration value. When setting the `url` configuration value, specify `url`.
- A string value that specifies the value of the configuration value. When setting the `url` configuration value, specify the watched folder location.

Note: *To see all the configuration values set for the EncryptDocument service, see the Java code example located at* *"QuickStart: Adding a Watched Folder endpoint using the Java API" on page 135.*

5 Define input parameter values.

Define an input parameter value by invoking the `CreateEndpointInfo` object's `setInputParameterMapping` method and pass the following values:

- A string value that specifies the name of the input parameter. For example, the name of the input parameter for the EncryptDocument service is `InDoc`.
- A string value that specifies the data type of the input parameter. For example, the data type of the `InDoc` input parameter is `com.adobe.idp.Document`.
- A string value that specifies the mapping type. For example, you can specify `variable`.
- A string value that specifies the mapping type value. For example, you can specify *.pdf as the file pattern.

Note: *Invoke the `setInputParameterMapping` method for each input parameter value to define. Because the EncryptDocument process has only one input parameter, you need to invoke this method once.*

6 Define an output parameter value.

Define an output parameter value by invoking the `CreateEndpointInfo` object's `setOutputParameterMapping` method and pass the following values:

- A string value that specifies the name of the output parameter. For example, the name of the output parameter for the EncryptDocument service is `SecuredDoc`.
- A string value that specifies the data type of the output parameter. For example, the data type of the `SecuredDoc` output parameter is `com.adobe.idp.Document`.
- A string value that specifies the mapping type. For example, you can specify `%F.pdf`.

7 Create a Watched Folder endpoint.

Create the endpoint by invoking the `EndpointRegistryClient` object's `createEndpoint` method and passing the `CreateEndpointInfo` object. This method returns an `Endpoint` object that represents the Watched Folder endpoint.

8 Enable the endpoint.

Enable the endpoint by invoking the `EndpointRegistryClient` object's `enable` method and passing the `Endpoint` object that was returned by the `createEndpoint` method.

See also

"Summary of steps" on page 1116

"QuickStart: Adding a Watched Folder endpoint using the Java API" on page 135

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Watched folder configuration values constant file

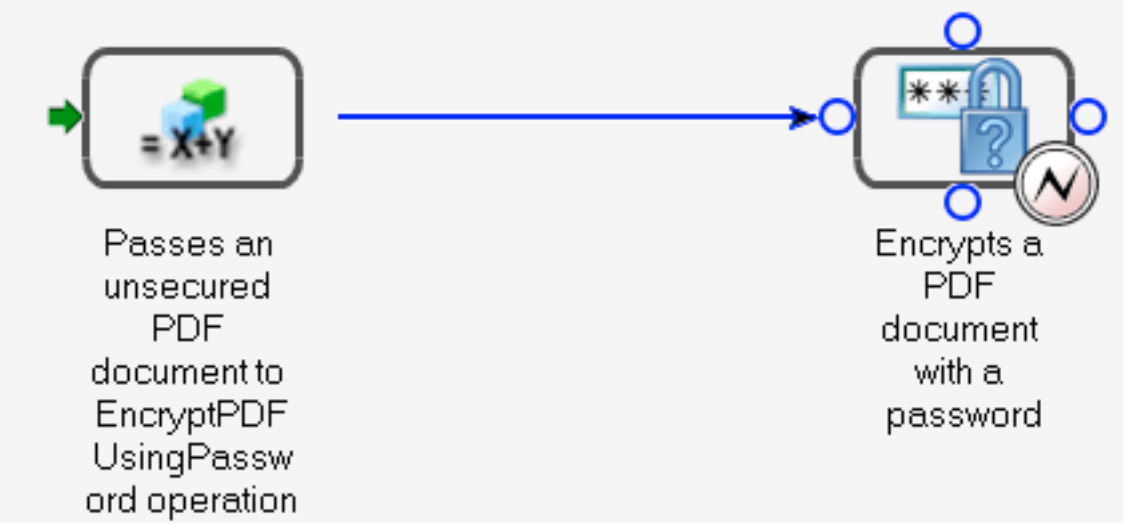
The "QuickStart: Adding a Watched Folder endpoint using the Java API" on page 135 uses a constant file that must be part of your Java project in order to compile the quick start. This constant file represents configuration values that must be set when adding a Watched Folder endpoint. The following Java code represents the constant file. For information about these configuration values, see .

```
/**
    * This class contains constants that can be used when setting Watched Folder
    * configuration values
    */

public final class WatchedFolderEndpointConfigConstants {

        public static final String PROPERTY_FILEPROVIDER_URL = "url";
        public static final String PROPERTY_PROPERTY_ASYNCHRONOUS = "asynchronous";
        public static final String PROPERTY_CRON_EXPRESSION = "cronExpression";
        public static final String PROPERTY_PURGE_DURATION = "purgeDuration";
        public static final String PROPERTY_REPEAT_INTERVAL = "repeatInterval";
        public static final String PROPERTY_REPEAT_COUNT = "repeatCount";
        public static final String PROPERTY_THROTTLE = "throttleOn";
        public static final String PROPERTY_USERNAMER = "userName";
        public static final String PROPERTY_DOMAINNAME = "domainName";
        public static final String PROPERTY_FILEPROVIDER_BATCH_SIZE = "batchSize";
        public static final String PROPERTY_FILEPROVIDER_WAIT_TIME = "waitTime";
        public static final String PROPERTY_EXCLUDE_FILE_PATTERN = "excludeFilePattern";
        public static final String PROPERTY_INCLUDE_FILE_PATTERN = "excludeFilePattern";
        public static final String PROPERTY_FILEPROVIDER_RESULT_FOLDER_NAME =
"resultFolderName";
        public static final String PROPERTY_FILEPROVIDER_PRESERVE_FOLDER_NAME =
"preserveFolderName";
        public static final String PROPERTY_FILEPROVIDER_FAILURE_FOLDER_NAME =
"failureFolderName";
        public static final String PROPERTY_FILEPROVIDER_PRESERVE_ON_FAILURE =
"preserveOnFailure";
        public static final String PROPERTY_FILEPROVIDER_OVERWRITE_DUPLICATE_FILENAME =
"overwriteDuplicateFilename";
       }
```

Adding Email Endpoints

You can programmatically add an Email endpoint to a service by using the AEM Forms Java API. By adding an Email endpoint, you enable users to send an email message with one or more file attachments to a specified email account. Then the configure service operation is invoked and manipulates the files. After the service performs the specified operation, it sends an email message to the sender with the modified files as file attachments.

For the purposes of programmatically adding an Email endpoint to a service, consider the following short-lived process named *MyApplication\EncryptDocument*. For information about short-lived processes, see "Understanding AEM Forms Processes" on page 441.



This process accepts an unsecured PDF document as an input value and then passes the unsecured PDF document to the Encryption service's `EncryptPDFUsingPassword` operation. This process encrypts the PDF document with a password and returns the password-encrypted PDF document as the output value. The name of the input value (the unsecured PDF document) is `InDoc` and the data type is `com.adobe.idp.Document`. The name of the output value (the password-encrypted PDF document) is `SecuredDoc` and the data type is `com.adobe.idp.Document`.

Note: *You cannot add an Email endpoint by using web services.*

Summary of steps

To add an Email endpoint to a service, perform the following tasks:

1. Include project files.
2. Create an `EndpointRegistryClient` object.
3. Set Email endpoint attributes.
4. Specify configuration values.
5. Define input parameter values.
6. Define an output parameter value.
7. Create the Email endpoint.
8. Enable the endpoint.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar

- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

Before you can programmatically add an Email endpoint, you must create an `EndpointRegistryClient` object.

Set Email endpoint attributes

To create an Email endpoint for a service, specify the following values:

- **Connector identifier value**: Specifies the type of endpoint that is created. To create an Email endpoint, specify `Email`.
- **Description**: Specifies a description for the endpoint.
- **Name**: Specifies the name of the endpoint.
- **Service identifier value**: Specifies the service to which the endpoint belongs. For example, to add an Email endpoint to the process that is introduced in this section (a process becomes a service when activated using Workbench), specify `EncryptDocument`.
- **Operation name**: Specifies the name of the operation that is invoked by using the endpoint. Typically, when creating an Email endpoint for a service that originated from a process created in Workbench, the name of the operation is `invoke`.

Specify configuration values

You must specify configuration values for an Email endpoint when programmatically adding an Email endpoint to a service. These configuration values are specified by an administrator if an Email endpoint is added using administration console.

Important: *The email account that is monitored is a special account that is used for the Email endpoint only. This account is not a regular user's email account. A regular user's email account must not be configured as the account that the Email Provider uses because the Email Provider deletes email messages from the inbox after it is finished with the messages.*

The following configuration values are set when programmatically adding an Email endpoint to a service:

- **cronExpression**: A cron expression if the email must be scheduled by using a cron expression.
- **repeatCount**: Number of times the email endpoint scans the folder or directory. A value of -1 indicates indefinite scanning. The default value is -1.
- **repeatInterval**: The scanning rate in seconds that the receiver uses for checking for incoming mail. The default value is 10.
- **startDelay**: The time to wait to scan after the scheduler starts. The default time is 0.
- **batchSize**: The number of email messages the receiver processes per scan for optimum performance. A value of -1 indicates all emails. The default value is 2.
- **userName**: The user name used when invoking a target service from email. The default value is `SuperAdmin`.
- **domainName**: A mandatory configuration value. The default value is `DefaultDom`.
- **domainPattern**: Specifies the domain patterns of incoming email that the provider accepts. For example, if `adobe.com` is used, only email from adobe.com is processed, email from other domains is ignored.

- **filePattern**: Specifies the incoming file attachment patterns that the provider accepts. This includes files that have specific file name extensions (*.dat, *.xml), files that have specific names (data), and files that have composite expressions in the name and extension (*.[dD][aA][Tt]). The default value is `*`.
- **recipientSuccessfulJob**: An email address to which messages are sent to indicate successful jobs. By default, a successful job message is always sent to the sender. If you type `sender`, email results are sent to the sender. Up to 100 recipients are supported. Specify additional recipients with email addresses, each one separated by a comma. To turn off this option, leave this value blank. In some cases, you may want to trigger a process and do not want an email notification of the result. The default value is `sender`.
- **recipientFailedJob**: An email address to which messages are sent to indicate failed jobs. By default, a failed job message is always sent to the sender. If you type `sender`, email results are sent to the sender. Up to 100 recipients are supported. Specify additional recipients with email addresses, each one separated by a comma. To turn off this option, leave this value blank. The default value is `sender`.
- **inboxHost**: The inbox host name or IP address for the email provider to scan.
- **inboxPort**: The port that the email server uses. The default value for POP3 is 110 and the default value for IMAP is 143. If SSL is enabled, the default value for POP3 is 995 and the default value for IMAP is 993.
- **inboxProtocol**: The email protocol for the email endpoint to use to scan the inbox. The options are `IMAP` or `POP3`. The inbox host mail server must support these protocols.
- **inboxTimeOut**: Time-out in seconds for the email provider to wait for inbox responses. The default value is 60.
- **inboxUser**: The user name required to log in to the email account. Depending on the email server and configuration, this may only be the user name portion of the email or it may be the full email address.
- **inboxPassword**: The password for the inbox user.
- **inboxSSLEnabled**: Set this value to force the email provider to use SSL when sending notification messages of results or errors. Ensure the IMAP or POP3 host supports SSL.
- **smtpHost**: The host name of the mail server that the email provider sends results and error messages to.
- **smtpPort**: The default value for the SMTP port is 25.
- **smtpUser**: The user account for the email provider to use when it sends out email notifications of results and errors.
- **smtpPassword**: The password for the SMTP account. Some mail servers do not require an SMTP password.
- **charSet**: The character set used by the email provider. The default value is `UTF-8`.
- **smtpSSLEnabled**: Set this value to force the email provider to use SSL when sending notification messages of results or errors. Ensure that the SMTP Host supports SSL.
- **failedJobFolder**: Specifies a directory in which to store results when the SMTP mail server is not operational.
- **asynchronous**: When set to synchronous, all input documents are processed and a single response is returned. When set to asynchronous, a response is sent for each input document that is processed. For example, an Email endpoint is created for the process introduced in this topic, and an email message is sent to the endpoint's inbox that contains multiple unsecured PDF documents. When all PDF documents are encrypted with a password, and if the endpoint is configured as synchronous, a single response email message is sent with all secured PDF documents attached. If the endpoint is configured as asynchronous, a separate response email message is sent for each secured PDF document. Each email message contains a single PDF document as an attachment. The default value is asynchronous.

Define input parameter values

When creating an Email endpoint, you must define input parameter values. That is, you must describe the input values that are passed to the operation that is invoked by the Email endpoint. For example, consider the process introduced in this topic. It has one input value named `InDoc` and its data type is `com.adobe.idp.Document`. When creating an Email endpoint for this process (after a process is activated, it becomes a service), you must define the input parameter value.

To define input parameter values required for an Email endpoint, specify the following values:

Input parameter name: The name of the input parameter. The name of an input value is specified in Workbench for a process. If the input value belongs to a service operation (a Forms service that is not a process created in Workbench), the input name is specified in the component.xml file. For example, the name of the input parameter for the process introduced in this section is `InDoc`.

Mapping type: Used to configure the input values required to invoke the service operation. Two types of mapping types are as follows:

- `Literal`: The Email endpoint uses the value entered in the field as it is displayed. All basic Java types are supported. For example, if an API uses input such as String, long, int, and Boolean, the string is converted to the proper type and the service is invoked.
- `Variable`: The value entered is a file pattern that the Email endpoint uses to pick the input. For example, if you select Variable for the mapping type and the input document must be a PDF file, you can specify `*.pdf` as the mapping value.

Mapping value: Specifies the value of the mapping type. For example, if you select a Variable mapping type, you can specify `*.pdf` as the file pattern.

Data type: Specifies the data type of the input values. For example, the data type of the input value of the process introduced in this section is com.adobe.idp.Document.

Define an output parameter value

When creating an Email endpoint, you must define an output parameter value. That is, you must describe the output value that is returned by the service that is invoked by the Email endpoint. For example, consider the process introduced in this topic. It has an output value named `SecuredDoc` and its data type is `com.adobe.idp.Document`. When creating an Email endpoint for this process (after a process is activated, it becomes a service), you must define the output parameter value.

To define an output parameter value required for an Email endpoint, specify the following values:

Output parameter name: The name of the output parameter. The name of a process output value is specified in Workbench. If the output value belongs to a service operation (a service that is not a process created in Workbench), the output name is specified in the component.xml file. For example, the name of the output parameter for the process introduced in this section is `SecuredDoc`.

Mapping type: Used to configure the output of the service and operation. The following options are available:

- If the service returns a single object (a single document), the pattern is `%F.pdf` and the source destination is sourcefilename.pdf. For example, the process introduced in this section returns a single document. As a result, the mapping type can be defined as `%F.pdf` (`%F` means use the given file name). The pattern `%E` specifies the extension of the input document.
- If the service returns a list, the pattern is `Result\%F\`, and the source destination is Result\sourcefilename\source1 (output 1) and Result\sourcefilename\source2 (output 2).

- If the service returns a map, the pattern is `Result\%F\`, and the source destination is Result\sourcefilename\file1 and Result\sourcefilename\file2. If the map has more than one object, the pattern is `Result\%F.pdf` and the source destination is Result\sourcefilename1.pdf (output 1), Result\sourcefilenam2.pdf (output 2), and so on.

Data type: Specifies the data type of the return value. For example, the data type of the return value of the process introduced in this section is `com.adobe.idp.Document`.

Create the Email endpoint

After you set the Email endpoint attributes and configuration values, and define input and output parameter values, you must create the Email endpoint.

Enable the endpoint

After you create an Email endpoint, you must enable it. When the endpoint is enabled, it can be used to invoke the service. After you enable the endpoint, you can view it within administration console.

See also

"Add an Email endpoint using the Java API" on page 1127

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Add an Email endpoint using the Java API

Add an Email endpoint by using the Java API:

1 Include project files.

Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an EndpointRegistry Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set Email endpoint attributes.

- Create a `CreateEndpointInfo` object by using its constructor.
- Specify the connector identifier value by invoking the `CreateEndpointInfo` object's `setConnectorId` method and passing the string value `Email`.
- Specify the description of the endpoint by invoking the `CreateEndpointInfo` object's `setDescription` method and passing a string value that describes the endpoint.
- Specify the name of the endpoint by invoking the `CreateEndpointInfo` object's `setName` method and passing a string value that specifies the name.
- Specify the service to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setServiceId` method and passing a string value that specifies the service name.
- Specify the operation that is invoked by invoking the `CreateEndpointInfo` object's `setOperationName` method and passing a string value that specifies the operation name. Typically, when creating an Email endpoint for a service that originated from a process created in Workbench, the name of the operation is invoke.

4 Specify configuration values.

For each configuration value to set for the Email endpoint, you must invoke the `CreateEndpointInfo` object's `setConfigParameterAsText` method. For example, to set the `smtpHost` configuration value, invoke the `CreateEndpointInfo` object's `setConfigParameterAsText` method and pass the following values:

- A string value that specifies the name of the configuration value. When setting the `smtpHost` configuration value, specify `smtpHost`.
- A string value that specifies the value of the configuration value. When setting the `smtpHost` configuration value, specify a string value that specifies the name of the SMTP server.

Note: *To see all the configuration values set for the EncryptDocument service introduced in this section, see the Java code example located at "QuickStart: Adding an Email endpoint using the Java API" on page 138.*

5 Define input parameter values.

Define an input parameter value by invoking the `CreateEndpointInfo` object's `setInputParameterMapping` method and pass the following values:

- A string value that specifies the name of the input parameter. For example, the name of the input parameter for the EncryptDocument service is `InDoc`.
- A string value that specifies the data type of the input parameter. For example, the data type of the `InDoc` input parameter is `com.adobe.idp.Document`.
- A string value that specifies the mapping type. For example, you can specify `variable`.
- A string value that specifies the mapping type value. For example, you can specify *.pdf as the file pattern.

Note: *Invoke the `setInputParameterMapping` method for each input parameter value to define. Because the EncryptDocument process has only one input parameter, you need to invoke this method once.*

6 Define an output parameter value.

Define an output parameter value by invoking the `CreateEndpointInfo` object's `setOutputParameterMapping` method and passing the following values:

- A string value that specifies the name of the output parameter. For example, the name of the output parameter for the EncryptDocument service is `SecuredDoc`.
- A string value that specifies the data type of the output parameter. For example, the data type of the `SecuredDoc` output parameter is `com.adobe.idp.Document`.
- A string value that specifies the mapping type. For example, you can specify `%F.pdf`.

7 Create the Email endpoint.

Create the endpoint by invoking the `EndpointRegistryClient` object's `createEndpoint` method and passing the `CreateEndpointInfo` object. This method returns an `Endpoint` object that represents the Email endpoint.

8 Enable the endpoint.

Enable the endpoint by invoking the `EndpointRegistryClient` object's `enable` method and passing the `Endpoint` object that was returned by the `createEndpoint` method.

See also

"Summary of steps" on page 1123

"QuickStart: Adding a Watched Folder endpoint using the Java API" on page 135

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Email configuration values constant file

The "QuickStart: Adding an Email endpoint using the Java API" on page 138 uses a constant file that must be part of your Java project in order to compile the quick start. This constant file represents configuration values that must be set when adding an email endpoint. The following Java code represents the constant file. For information about these configuration values, see .

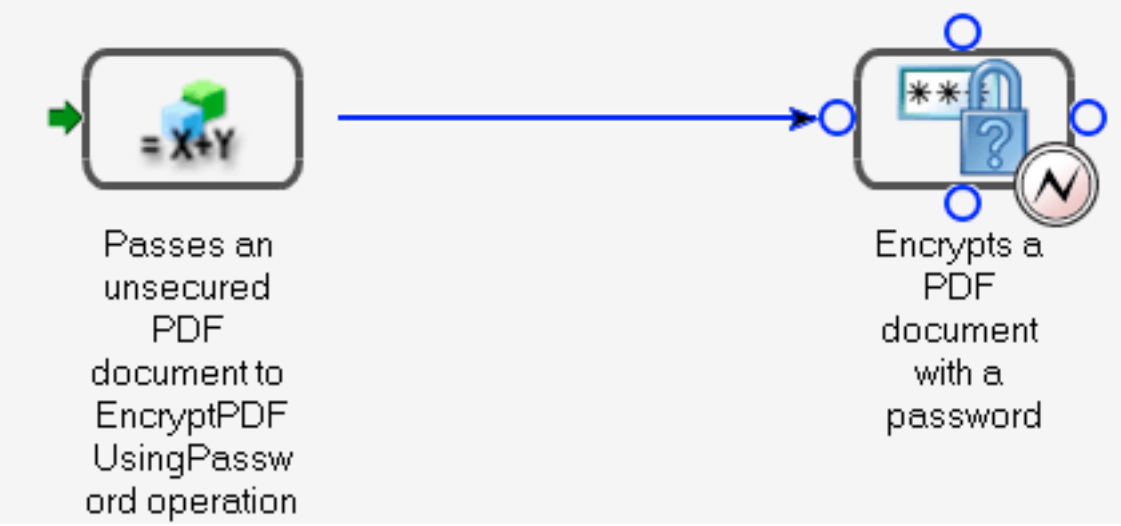
```
/**
    * This class contains constants that can be used when setting email endpoint
    * configuration values
    */
public class EmailEndpointConfigConstants {

    public static final String PROPERTY_EMAILPROVIDER_CRON_EXPRESSION = "cronExpression";
    public static final String PROPERTY_EMAILPROVIDER_REPREAT_COUNT = "repeatCount";
    public static final String PROPERTY_EMAILPROVIDER_REPREAT_INTERVAL = "repeatInterval";
    public static final String PROPERTY_EMAILPROVIDER_START_DELAY = "startDelay";
    public static final String PROPERTY_EMAILPROVIDER_BATCH_SIZE = "batchSize";
    public static final String PROPERTY_EMAILPROVIDER_USERNAME = "userName";
    public static final String PROPERTY_EMAILPROVIDER_DOMAINNAME = "domainName";
    public static final String PROPERTY_EMAILPROVIDER_DOMAINPATTERN = "domainPattern";
    public static final String PROPERTY_EMAILPROVIDER_FILEPATTERN = "filePattern";
    public static final String PROPERTY_EMAILPROVIDER_RECIPIENT_SUCCESSFUL_JOB =
"recipientSuccessfulJob";
    public static final String PROPERTY_EMAILPROVIDER_RECIPIENT_FAILED_JOB = "recipientFailedJob";
    public static final String PROPERTY_EMAILPROVIDER_INBOX_HOST = "inboxHost";
    public static final String PROPERTY_EMAILPROVIDER_INBOX_PORT = "inboxPort";
    public static final String PROPERTY_EMAILPROVIDER_PROTOCOL = "inboxProtocol";
    public static final String PROPERTY_EMAILPROVIDER_INBOX_TIMEOUT = "inboxTimeOut";
    public static final String PROPERTY_EMAILPROVIDER_INBOX_USER = "inboxUser";
    public static final String PROPERTY_EMAILPROVIDER_INBOX_PASSWORD = "inboxPassword";
    public static final String PROPERTY_EMAILPROVIDER_INBOX_SSL = "inboxSSLEnabled";
    public static final String PROPERTY_EMAILPROVIDER_SMTP_HOST = "smtpHost";
    public static final String PROPERTY_EMAILPROVIDER_SMTP_PORT = "smtpPort";
    public static final String PROPERTY_EMAILPROVIDER_SMTP_USER = "smtpUser";
    public static final String PROPERTY_EMAILPROVIDER_SMTP_PASSWORD = "smtpPassword";
    public static final String PROPERTY_EMAILPROVIDER_CHARSET = "charSet";
    public static final String PROPERTY_EMAILPROVIDER_SMTP_SSL = "smtpSSLEnabled";
    public static final String PROPERTY_EMAILPROVIDER_FAILED_FOLDER = "failedJobFolder";
    public static final String PROPERTY_EMAILPROVIDER_ASYNCHRONOUS = "asynchronous";
}
```

Adding Remoting Endpoints

Note: *LiveCycle Remoting APIs deprecated for AEM forms on JEE.*

You can programmatically add a Remoting endpoint to a service by using the AEM Forms Java API. By adding a Remoting endpoint, you are enabling a Flex application to invoke the service by using remoting. (See "Invoking AEM Forms using Remoting" on page 444.)

For the purposes of programmatically adding a Remoting endpoint to a service, consider the following short-lived process named *EncryptDocument*.



This process accepts an unsecured PDF document as an input value and then passes the unsecured PDF document to the Encryption service's `EncryptPDFUsingPassword` operation. The PDF document is encrypted with a password, and the password-encrypted PDF document is the output value of this process. The name of the input value (the unsecured PDF document) is `InDoc` and the data type is `com.adobe.idp.Document`. The name of the output value (the password-encrypted PDF document) is `SecuredDoc` and the data type is `com.adobe.idp.Document`.

To demonstrate how to add a Remoting endpoint to a service, this section adds a Remoting endpoint to a service named EncryptDocument.

Note: *You cannot add a Remoting endpoint by using web services.*

Summary of steps

To remove an endpoint from a service, perform the following tasks:

1. Include project files.
2. Create an `EndpointRegistryClient` object.
3. Set Remoting endpoint attributes.
4. Create a Remoting endpoint.
5. Enable the endpoint.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

To programmatically add a Remoting endpoint, you must create an `EndpointRegistryClient` object.

Set Remoting endpoint attributes

To create a Remoting endpoint for a service, specify the following values:

- **Connector identifier value**: Specifies the type of endpoint that is created. To create a Remoting endpoint, specify `Remoting`.
- **Description**: Specifies the description of the endpoint.
- **Name**: Specifies the name of the endpoint.
- **Service identifier value**: Specifies the service to which the endpoint belongs. For example, to add a Remoting endpoint to the process that is introduced in this section (a process becomes a service when it is activated within Workbench), specify `EncryptDocument`.
- **Operation name**: Specifies the name of the operation that is invoked by using the endpoint. When creating a Remoting endpoint, specify a wildcard character (*).

Create a Remoting endpoint

After you set Remoting endpoint attributes, you can create a Remoting endpoint for a service.

Enable the endpoint

After you create a new endpoint, you must enable it. When a Remoting endpoint is enabled, it enables a Flex client to invoke the service.

See also

"Add a Remoting endpoint using the Java API" on page 1131

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Add a Remoting endpoint using the Java API

Add a Remoting endpoint by using the Java API:

1 Include project files.

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an EndpointRegistry Client object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Set Remoting endpoint attributes.

   - Create a `CreateEndpointInfo` object by using its constructor.
   - Specify the connector identifier value by invoking the `CreateEndpointInfo` object's `setConnectorId` method and passing the string value `Remoting`.

- Specify the description of the endpoint by invoking the `CreateEndpointInfo` object's `setDescription` method and passing a string value that describes the endpoint.
- Specify the name of the endpoint by invoking the `CreateEndpointInfo` object's `setName` method and passing a string value that specifies the name.
- Specify the service to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setServiceId` method and passing a string value that specifies the service name.
- Specify the operation that is invoked by the `CreateEndpointInfo` object's `setOperationName` method and passing a string value that specifies the operation name. For a Remoting endpoint, specify a wildcard character (*).

4 Create a Remoting endpoint.

Create the endpoint by invoking the `EndpointRegistryClient` object's `createEndpoint` method and passing the `CreateEndpointInfo` object. This method returns an `Endpoint` object that represents the new Remoting endpoint.

5 Enable the endpoint.

Enable the endpoint by invoking the `EndpointRegistryClient` object's `enable` method and passing the `Endpoint` object that was returned by the `createEndpoint` method.

See also

"Summary of steps" on page 1130

"QuickStart: Adding a Remoting endpoint using the Java API" on page 141

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Adding TaskManager Endpoints

You can programmatically add a TaskManager endpoint to a service by using the AEM Forms Java API. By adding a TaskManager endpoint to a service, you enable a Workspace user to invoke the service. That is, a user working in Workspace can invoke a process that has a corresponding TaskManager endpoint.

Note: *You cannot add a TaskManager endpoint by using web services.*

Summary of steps

To add a TaskManager endpoint to a service, perform the following tasks:

1 Include project files.

2 Create an `EndpointRegistryClient` object.

3 Create a category for the endpoint.

4 Set TaskManager endpoint attributes.

5 Create a TaskManager endpoint.

6 Enable the endpoint.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

Before you can programmatically add a TaskManager endpoint, you must create an `EndpointRegistryClient` object.

Create a category for the endpoint

Categories are used to organize services within Workspace. That is, a Workspace user can invoke a service that has a TaskManager endpoint by selecting a category within Workspace. When creating a TaskManager endpoint, you can either reference an existing category or programmatically create a new category.

Note: *This section creates a new category as part of adding a TaskManager endpoint to a service.*

Set TaskManager endpoint attributes

To create a TaskManager endpoint for a service, specify the following values:

- **Connector identifier**: Specifies the type of endpoint that is created. To create a TaskManager endpoint, specify `TaskManagerConnector`.
- **Description**: Specifies the description of the endpoint.
- **Name**: Specifies the name of the endpoint.
- **Service identifier**: Specifies the service to which the endpoint belongs.
- **Category**: Specifies a category identifier value that is associated with the TaskManager endpoint.
- **Operation name**: Typically, when creating a TaskManager endpoint for a service that originated from a process created in Workbench, the name of the operation is `invoke`.

Create a TaskManager endpoint

After you set a TaskManager endpoint attributes, you can create a TaskManager endpoint for a service.

Enable the endpoint

After you create a new endpoint, you must enable it. When the endpoint is enabled, it can be used to invoke the service from within Workspace. After you enable the endpoint, you can view it within administration console.

See also

"Add a TaskManager endpoint using the Java API" on page 1134

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Add a TaskManager endpoint using the Java API

Add a TaskManager endpoint by using the Java API:

1 Include project files.

Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an EndpointRegistry Client object.

- Create a `ServiceClientFactory` object that contains connection properties.
- Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Create a category for the endpoint.

- Create a `CreateEndpointCategoryInfo` object by using its constructor and passing the following values:
  - A string value that specifies the identifier value of the category
  - A string value that specifies the description of the category
- Create the category by invoking the `EndpointRegistryClient` object's `createEndpointCategory` method and passing the `CreateEndpointCategoryInfo` object. This method returns an `EndpointCategory` object that represents the new category.

4 Set TaskManager endpoint attributes.

- Create a `CreateEndpointInfo` object by using its constructor.
- Specify the connector identifier value by invoking the `CreateEndpointInfo` object's `setConnectorId` method and passing the string value `TaskManagerConnector`.
- Specify the description of the endpoint by invoking the `CreateEndpointInfo` object's `setDescription` method and passing a string value that describes the endpoint.
- Specify the name of the endpoint by invoking the `CreateEndpointInfo` object's `setName` method and passing a string value that specifies the name.
- Specify the service to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setServiceId` method and passing a string value that specifies the service name.
- Specify the category to which the endpoint belongs by invoking the `CreateEndpointInfo` object's `setCategoryId` method and passing a string value that specifies the category identifier value. You can invoke the `EndpointCategory` object's `getId` method to get the identifier value of this category.
- Specify the operation that is invoked by invoking the `CreateEndpointInfo` object's `setOperationName` method and passing a string value that specifies the operation name. Typically, when creating a `TaskManager` endpoint for a service that originated from a process created in Workbench, the name of the operation is `invoke`.

5 Create a TaskManager endpoint.

Create the endpoint by invoking the `EndpointRegistryClient` object's `createEndpoint` method and passing the `CreateEndpointInfo` object. This method returns an `Endpoint` object that represents the new TaskManager endpoint.

6 Enable the endpoint.

Enable the endpoint by invoking the `EndpointRegistryClient` object's `enable` method and passing the `Endpoint` object that was returned by the `createEndpoint` method.

See also

"Summary of steps" on page 1132

"QuickStart: Adding a TaskManager endpoint using the Java API" on page 143

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Modifying Endpoints

You can programmatically modify an existing endpoint by using the AEM Forms Java API. By modifying an endpoint, you can change the behaviour of the endpoint. Consider, for example, a Watched Folder endpoint that specifies a folder that is used as the watched folder. You can programmatically modify configuration values that belong to the Watched Folder endpoint, resulting in another folder functioning as the watched folder. For information about configuration values that belong to a Watched Folder endpoint, see "Adding Watched Folder Endpoints" on page 1116.

To demonstrate how to modify an endpoint, this section modifies a Watched Folder endpoint by changing the folder that behaves as the watched folder.

Note: *You cannot modify an endpoint by using web services.*

Summary of steps

To modify an endpoint, perform the following tasks:

1. Include project files.
2. Create an `EndpointRegistryClient` object.
3. Retrieve the endpoint.
4. Specify new configuration values.

Include project files

Include the necessary files in your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

To programmatically modify an endpoint, you must create an `EndpointRegistryClient` object.

Retrieve the endpoint to modify

Before you can modify an endpoint, you must retrieve it. To retrieve an endpoint, you must connect as a user who can access an endpoint. It is recommended that you connect as an administrator. (See "Setting connection properties" on page 500).

You can retrieve an endpoint by retrieving a list of endpoints. You can then iterate through the list, searching for the specific endpoint to remove. For example, you can locate an endpoint by determining the service that corresponds to the endpoint and the type of endpoint. When you locate the endpoint, you can modify it.

Specify new configuration values

When modifying an endpoint, specify new configuration values. For example, to modify a Watched Folder endpoint, reset all Watched Folder endpoint configuration values, not just the ones that you want to modify. For information about configuration values that belong to a Watched Folder endpoint, see "Adding Watched Folder Endpoints" on page 1116.

Note: *For information about configuration values that belong to an Email endpoint, see "Adding Email Endpoints" on page 1122.*

Important: *You cannot modify the service that is invoked by the endpoint. If you attempt to modify the service, an exception is thrown. To modify the service associated with a given endpoint, remove the endpoint and create a new one. (See "Removing Endpoints" on page 1137.)*

See also

"Modifying an endpoint using the Java API" on page 1136

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Modifying an endpoint using the Java API

Modify an endpoint by using the Java API:

1. Include project files.

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2. Create an EndpointRegistry Client object.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Retrieve the endpoint to modify.
   - Retrieve a list of all endpoints to which the current user (specified in the connection properties) can access by invoking the `EndpointRegistryClient` object's `getEndpoints` method and passing a `PagingFilter` object that acts as a filter. You can pass a `(PagingFilter)null` value to return all endpoints. This method returns a `java.util.List` object where each element is an `Endpoint` object. For information about a `PagingFilter` object, see AEM Forms API Reference.
   - Iterate through the `java.util.List` object to determine whether it has endpoints. If endpoints exist, each element is an `EndPoint` instance.
   - Determine the service that corresponds to an endpoint by invoking the `EndPoint` object's `getServiceId` method. This method returns a string value that specifies the service name.

- Determine the type of endpoint by invoking the `EndPoint` object's `getConnectorId` method. This method returns a string value that specifies the type of endpoint. For example, if the endpoint is a Watched Folder endpoint, this method returns `WatchedFolder`.

4 Specify new configuration values.

- Create a `ModifyEndpointInfo` object by invoking its constructor.
- For each configuration value to set, invoke the `ModifyEndpointInfo` object's `setConfigParameterAsText` method. For example, to set the url configuration value, invoke the `ModifyEndpointInfo` object's `setConfigParameterAsText` method and pass the following values:
  - A string value that specifies the name of the configuration value. For example, to set the `url` configuration value, specify `url`.
  - A string value that specifies the value of the configuration value. To define a value for the `url` configuration value, specify the watched folder location.
- Invoke the `EndpointRegistryClient` object's `modifyEndpoint` method and pass the `ModifyEndpointInfo` object.

See also

"Summary of steps" on page 1135

"QuickStart: Modifying an endpoint using the Java API" on page 146

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Removing Endpoints

You can programmatically remove an endpoint from a service by using the AEM Forms Java API. After you remove an endpoint, the service cannot be invoked by using the invocation method that the endpoint enabled. For example, if you remove an SOAP endpoint from a service, you cannot invoke the service by using the SOAP mode.

To demonstrate how to remove an endpoint from a service, this section removes an EJB endpoint from a service named *EncryptDocument*.

Note: *You cannot remove an endpoint by using web services.*

Summary of steps

To remove an endpoint from a service, perform the following tasks:

1 Include project files.

2 Create an `EndpointRegistryClient` object.

3 Retrieve the endpoint.

4 Remove the endpoint.

Include project files

Include the necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar

- adobe-usermanager-client.jar
- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

For information about the location of these JAR files, see "Including AEM Forms Java library files" on page 491.

Create an EndpointRegistry Client object

To programmatically remove an endpoint, you must create an `EndpointRegistryClient` object.

Retrieve the endpoint to remove

Before you can remove an endpoint, you must retrieve it. To retrieve an endpoint, you must connect as a user who can access an endpoint. It is recommended that you connect as an administrator. (See "Setting connection properties" on page 500).

You can retrieve an endpoint by retrieving a list of endpoints. You can then iterate through the list, searching for the specific endpoint to remove. For example, you can locate an endpoint by determining the service that corresponds to the endpoint and the type of endpoint. When you locate the endpoint, you can remove it.

Remove the endpoint

After you create a new endpoint, you must enable it. When the endpoint is enabled, it can be used to invoke the service. After you enable the endpoint, you can view it within administration console.

See also

"Removing an endpoint using the Java API" on page 1138

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Removing an endpoint using the Java API

Remove an endpoint by using the Java API:

1 Include project files.

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2 Create an EndpointRegistry Client object.

   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create an `EndpointRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.

3 Retrieve the endpoint to remove.

   - Retrieve a list of all endpoints to which the current user (specified in the connection properties) has access by invoking the `EndpointRegistryClient` object's `getEndpoints` method and passing a `PagingFilter` object that acts as a filter. You can pass `(PagingFilter)null` to return all endpoints. This method returns a `java.util.List` object where each element is an `Endpoint` object.
   - Iterate through the `java.util.List` object to determine whether it has endpoints. If endpoints exist, each element is a `EndPoint` instance.

- Determine the service that corresponds to an endpoint by invoking the `EndPoint` object's `getServiceId` method. This method returns a string value that specifies the service name.
- Determine the type of endpoint by invoking the `EndPoint` object's `getConnectorId` method. This method returns a string value that specifies the type of endpoint. For example, if the endpoint is an EJB endpoint, this method returns `EJB`.

4 Remove the endpoint.

Remove the endpoint by invoking the `EndpointRegistryClient` object's `remove` method and passing the `EndPoint` object that represents the endpoint to remove.

See also

"Summary of steps" on page 1137

"QuickStart: Removing an endpoint using the Java API" on page 148

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieving Endpoint Connector Information

You can programmatically retrieve information about endpoint connectors using the AEM Forms API. A connector enables an endpoint to invoke a service using various invocation methods. For example, a Watched Folder connector enables an endpoint to invoke a service using watched folders. By programmatically retrieving information about endpoint connectors, you can retrieve configuration values associated with a connector such as which configuration values are required and which ones are optional.

To demonstrate how to retrieve information about endpoint connectors, this section retrieves information about a Watched Folder connector. (See "Adding Watched Folder Endpoints" on page 1116.)

Note: *You cannot retrieve information about endpoints by using web services.*

Note: *This topic uses the* `ConnectorRegistryClient` *API to retrieve information about endpoint connectors. (See AEM Forms API Reference.)*

Summary of steps

To retrieve endpoint connector information, perform the following tasks:

1 Include project files.

2 Create an `ConnectorRegistryClient` object.

3 Specify the connector type.

4 Retrieve configuration values.

Include project files

Include the necessary files into your development project. If you are creating a client application by using Java, include the necessary JAR files. If you are using web services, make sure that you include the proxy files.

The following JAR files must be added to your project's class path:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar

- adobe-utilities.jar (required if AEM Forms is deployed on JBoss Application Server)
- jbossall-client.jar (required if AEM Forms is deployed on JBoss Application Server)

If AEM Forms is deployed on a supported J2EE application server that is not JBoss, then replace adobe-utilities.jar and jbossall-client.jar with JAR files that are specific to the J2EE application server on which AEM Forms is deployed. For information about the location of all AEM Forms JAR files, see "Including AEM Forms Java library files" on page 491.

Create an ConnectorRegistry Client object

To programmatically retrieve endpoint connector information, create a `ConnectorRegistryClient` object.

Specify the connector type

Specify the type of connector from which to retrieve information. The following types of connectors exist:

- **EJB**: Enables a client application to invoke a service using the EJB mode.
- **SOAP**: Enables a client application to invoke a service using the SOAP mode.
- **Watched Folder**: Enables watched folders to invoke a service.
- **Email**: Enables email messages to invoke a service.
- **Remoting**: Enables a Flex client application to invoke a service.
- **TaskManagerConnector**: Enables a Workspace user to invoke a service from within Workspace.

Retrieve configuration values

After you specify the connector type, you can retrieve information about the connector such as supported configuration value. For example, for any connector, you can determine which configuration values are required and which ones are optional.

See also

"Retrieve endpoint connector information using the Java API" on page 1140

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Retrieve endpoint connector information using the Java API

Retrieve endpoint connector information by using the Java API:

1. Include project files. .

   Include client JAR files, such as adobe-livecycle-client.jar, in your Java project's class path.

2. Create a ConnectorRegistry Client object.
   - Create a `ServiceClientFactory` object that contains connection properties.
   - Create a `ConnectorRegistryClient` object by using its constructor and passing the `ServiceClientFactory` object.
3. Specify the connector type.

   Specify the connector type by invoking the `ConnectorRegistryClient` object's `getEndpointDefinition` method and passing a string value that specifies the connector type. For example, to specify the Watched Folder connector type, pass the string value `WatchedFolder`. This method returns an `Endpoint` object that corresponds to the connector type.

4 Retrieve configuration values.

- Retrieve configuration values that are associated within this endpoint by invoking the `Endpoint` object's `getConfigParameters` method. This method returns an array of `ConfigParameter` objects.
- Retrieve information about each configuration value by retrieving each element within the array. Each element is a `ConfigParameter` object. You can, for example, determine whether the configuration value is required or optional by invoking the `ConfigParameter` object's `isRequired` method. If the configuration value is required, then this method returns `true`.

See also

"Summary of steps" on page 1139

"QuickStart: Retrieving endpoint connector information using the Java API" on page 151

"Including AEM Forms Java library files" on page 491

"Setting connection properties" on page 500

Programmatically managing the Preferences Nodes

This topic describes how you can use the Preferences Manager Service API (Java) to programmatically manage the Preferences Nodes.

You can manually change configuration settings from Administrator UI. To change the options, navigate to `Home>Settings>User Management> Configuration>Manual Configuration`. Import `config.xml` after making the changes, you would notice that all the changes except changes made at node `/Adobe/Adobe Experience Manager Forms/Config/UM persist` are lost. The preview of User Management Import and export does not support changing configuration settings for other components. Now, these changes can be made using `PreferencesManagerServiceClient` APIs.

Summary of steps To programmatically manage the Preferences Nodes, perform the following steps:

1 Include project files.

2 Create an PreferencesManagerService client

3 Invoke the appropriate role or permission operations

Include project files

Include necessary files in your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

Create an PreferencesManagerService client

Before you can programmatically perform a User Management PreferencesManagerService operation, you must create a PreferencesManagerService client. With the Java API this is accomplished by creating an PreferencesManagerServiceClient object.

Invoke the appropriate role or permission operations

Once you have created the service client, you can then invoke the Preferences Manager operations. The service client allows you to read and set permissions.