



Adobe® Primetime Live Packager Getting Started

Contents

Primetime Live Packager Getting Started Guide.....	4
What's new in Live Packager, Version 1.3.....	4
What's new in Live Packager, Version 1.2.....	4
Overview of Live Packager.....	4
Support for HLS.....	4
Support for HDS.....	5
Support for both HLS and HDS.....	5
Prerequisites for using the Live Packager.....	5
Enhanced SCTE 35 cues support.....	5
Query parameters in http push.....	6
Getting started with Live Packager.....	6
Starting the packager.....	6
Stopping the packager.....	6
Restarting the packager.....	7
Uninstalling the packager.....	7
Periodic time-bomb check.....	7
Configuration.....	7
Stream configuration.....	9
Packager configuration.....	10
Local origin HTTP server.....	15
Ingesting stream.....	17
The manifest file.....	17
Packager usage scenarios.....	18
Local origin scenario.....	18
Remote origin scenario.....	19
Multi-bitrate scenario.....	20
Audio-only scenario.....	23
Protecting content scenario.....	24
Ingestion of RTMP streams.....	30
server.xml definition for RTMP.....	30

stream.xml definition for RTMP.....	31
RTMP ingest for HLS.....	32
Authentication between RTMP Publisher and Primetime Live Packager.....	32
Enabling RTMP authentication.....	33
Enabling authentication at Live Packager.....	34
Enabling authentication at the container.....	34
Configure user credentials.....	34
Adding custom authentication.....	36
Performing administrator actions.....	38
Getting the packager's version.....	38
Managing the stream container.....	39
Managing the packager.....	43
Packager-specific APIs.....	44
Ad cues.....	44
Loading new containers and streams.....	45
Load on startup.....	45
Viewing stream statistics.....	45
Logging.....	47
Logging namespaces.....	47
Modifying the log level for a stream.....	48
Appendixes.....	49
Supported SCTE-35 splice_insert() messages.....	49
#EXT-X-CUE tag.....	49
#EXT-X-CUE-CONT tag.....	50
HLS manifest custom tag for synchronization.....	51
Copyright.....	51

Primetime Live Packager Getting Started Guide

What's new in Live Packager, Version 1.3

- Primetime streaming server
- *Authentication between RTMP Publisher and Primetime Live Packager*
- *Using HSM to store packager credentials*
- *RTMP ingest for HLS*
- *Load on startup*
- *Periodic time-bomb check*

What's new in Live Packager, Version 1.2

- *Enhanced SCTE 35 cues support*
- *Query parameters in http push*
- *HLS manifest custom tag for synchronization*
- *DRM/FAXS impact*

Overview of Live Packager

You can use Adobe Primetime Live Packager to capture encoded broadcast feed (MPEG-TS) directly and translate the feed into packaged HDS/HLS content. LivePackager also captures metadata from the broadcast feed, including SCTE-35 ad markers. The captured metadata is sent along with the stream for downstream processors to process the advertisements.

You should not deploy the LivePackager outside the firewall. Otherwise, a rogue encoder can pump RTMP into the packager.



Note: LivePackager can process the live stream to prepare the output for multiple formats supporting HTTP delivery protocols, such as HDS and HLS.

Support for HLS

The Live Packager supports:

- Ingesting a live MPEG-2 transport stream
- Generating the MPEG-2 transport stream fragments (as per the Apple HLS specifications)
- Generating an m3u8 file containing information about these fragments.

To process an HLS stream, you can directly fragment the incoming MPEG-TS stream in TS packets at the packager level. In this mode, the Live packager directly publishes the TS fragments and m3u8 on the origin server. The origin server serves out these fragments and m3u8 based on the HTTP GET requests.

Support for HDS

You can configure Live Packager to create fragments based on the Adobe HDS specifications with corresponding manifest (F4M) and bootstrap files.

You can enable the Live Packager to create F4F fragments for HDS streaming and then publish these fragments to the origin server.

You can configure the packager to push HDS manifest and fragment data to a remote server. The data is pushed to the origin server using standard HTTP PUT calls. The packager can push to multiple origin servers to support fail-over and load balancing.

Support for both HLS and HDS

Once the MPEG2-TS stream is packaged, the packager can write the content to disk and send it to another server using HTTP PUT.

This capability is intended to allow content to be pushed from the local broadcast center to a remote origin. This keeps the source packaging portions of the workflow distinct from the distribution portions of the workflow. For example, a broadcaster may run a local packager on premises, offloading the actual content distribution to a CDN or an Origin Services provider.

In addition to the content packaging functionality, the packager also captures the SCTE-35 ad markers from the broadcast stream and encodes them into the generated manifest file.

These ad markers can be used by the downstream processors (Primetime Player) to perform ad insertion. Also, read Supported SCTE-35 splice_insert() messages and #EXT-X-CUE tag.

The packager can run an internal HTTP server to deliver the HDS/HLS content. The configuration for this will be read from the <PACKAGER_ROOT>/conf/http/origin.xml file, and if this file is not found, the internal HTTP server will not be started. The packager is capable of writing fragmented content directly to the local file system.

Prerequisites for using the Live Packager

The Live Packager accepts standard multicast MPEG-2 transport stream as input, with some restrictions regarding supported codecs and the number of elementary streams.

The supported codecs are:

- Audio: AAC and MP3
- Video: H.264/AVC

Enhanced SCTE 35 cues support

Primetime LivePackagers performs enhanced parsing of SCTE 35 cues to support a wider part of the SCTE 35 specifications.

Primetime Live Packager has the capability to parse in-stream SCTE 35 cues and then add them as cue info in the generated manifest. The Primetime Live Packager captures SCTE 35 cues from the broadcast feed and passes them along so that they may be used for ad decisioning / insertion / blackouts by downstream processors. These cues are added in f4m for HDS output and m3u8 for HLS output.

In Primetime packagers, parsing of SCTE 35 cues has been enhanced to support a wider part of SCTE 35 spec. The cue formats specified in Digital program insertion specifications are supported.

Query parameters in http push

The target URL for HTTP Push in the LivePackager's configuration supports query parameters.

The following is a sample HTTP Push target url:

```
<HttpPush>
  <UseSecurityToken>true</UseSecurityToken>
  <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
  <TargetURL>http://host:port/module_path?p1=val1&p2=val2</TargetURL>
</HttpPush>
```

The following are some sample fragment/manifest URIs for this HTTP Push target:

- `http://host:port/module_path/livestream.f4m?p1=val1&p2=val2`
- `http://host:port/module_path/livestreamSeg1-Frag9?p1=val1&p2=val2`
- `http://host:port/module_path/livestream.2347823.ts?p1=val1&p2=val2`
- `http://host:port/module_path/livestream.m3u8?p1=val1&p2=val2`

Getting started with Live Packager

The Primetime platform archive file (zip) contains all the Primetime platform components.

To run the Packager you require Java 1.7 or higher. Download the software from the [Oracle site](#) and follow the installation instructions.

Extract the archive file to your disk.

Starting the packager

There are two ways to start the packager.

1. On Linux, run the following command:

```
$ nohup sh packager_start.sh &
```

2. Alternatively, run the following command using java:

```
java -Djava.util.logging.config.file=logging.properties -server -jar Packager.jar
```

Stopping the packager

On Linux, run the following command:

```
$ ./packager_stop.sh
```

Restarting the packager

1. Stop the packager.
2. Start the packager.



Note: When the packager starts, it tries to initialize the bootstrap information from the disk. If the bootstrap information is found at the fragment target, it implies that the packager has been restarted. In case of restart, the packager drops the fragments till the next fragment boundary and then starts packaging. Packager inserts a gap entry in the bootstrap to indicate that there are missing fragments.

For a clean restart (starting the new stream and not appending to the previous stream), before restarting the packager, manually delete the stream's temp directory from `<PACKAGER_ROOT>/temp/<output type>/<container name>/<stream name>`.

Uninstalling the packager

Remove the packager directory in the Primetime Platform directory.

Periodic time-bomb check

Time-bombed builds perform a periodic check every 24 hours and log the expiry date.

If the expiry date is within 30 days, it logs the message at SEVERE level. Otherwise, the message is logged at the INFO level. If the build is not updated before the expiry date, the periodic check causes the running instance of Packager to stop after the expiry date.

Configuration

The distribution file contains a sample configuration for both packager and origin server, which can be used for testing.

The default origin server configuration available in the `<PACKAGER_ROOT>/conf/http/origin.xml` file:

```
<Config>
<!-- Set this to false to disable the local HTTP server. -->
<LocalOriginEnabled>true</LocalOriginEnabled>
<!-- Interface and Port for the HTTP server to listen on. -->
<InterfaceAddress></InterfaceAddress>
<Port>8080</Port>
<!-- The packaged content will be under this directory. -->
<Webroot>./webroot</Webroot>
<!-- The MaxContentLength should be greater than the maximum fragment size, in bytes. -->
<MaxContentLength>8192000</MaxContentLength>
<!-- If a requested fragment is not found, this response will be returned.
-- The default is 404. -->
<MissingFragmentResponseCode>503</MissingFragmentResponseCode>
<!-- Uncomment to override default setting: WorkerThreads = 2 * NoOfProcessors. -->
<!-- <WorkerThreadCount>10</WorkerThreadCount> -->
<!-- Specifies TTL values in integral seconds for various file types. When TTL value is not
set for a file type, then following headers are set in Http Response of corresponding file
type:
1) max-age header - set to TTL value
2) Date header - set to response date
3) Expires header - set to response date + TTL The file types supported are F4F, F4M,
```

Bootstrap, CrossDomain, M3U8, HLSSegment, DRMMetadata, Timeline, DashSegment, DashManifest and DashManifestPart.

```
-->
<Headers>
<TTL>
</TTL>
<F4M>0</F4M>
<Bootstrap>0</Bootstrap>
<M3U8>0</M3U8>
<Timeline>0</Timeline>
<DashManifest>0</DashManifest>
<DashManifestPart>0</DashManifestPart>
<DRMMetadata>60</DRMMetadata>
<F4F>60</F4F>
<HLSegment>60</HLSegment>
</Headers>
</Config>
```

The default packager configuration available in the `packager.xml` file:

```
<Config>
<ContentProtection>
  <FAXS4>
    <LicenseServerURL>
      http://primetime.aaxs.adobe.com
    </LicenseServerURL>
    <LicenseServerCertificate>
      creds/static/phds_license_server.der
    </LicenseServerCertificate>
    <LicenseServerCredential>
      creds/static/phds_license_server.pfx
    </LicenseServerCredential>
    <LicenseServerCredentialPassword>xxxxxxxxxxxx</LicenseServerCredentialPassword>
    <PackagerCredential>creds/static/phds_production_packager.pfx</PackagerCredential>
    <PackagerCredentialPassword>xxxxxxxxxxxx</PackagerCredentialPassword>
    <TransportCertificate>
      creds/static/phds_production_transport.der
    </TransportCertificate>
    <CommonKey>creds/common-key.bin</CommonKey>
    <PolicyFile>creds/static/phds_policy.pol</PolicyFile>
    <RecipientCertificates>creds/sd</RecipientCertificates>
  </FAXS4>
</ContentProtection>
</Config>
```

After extracting the Primetime platform archive, change to the packager directory, and run the packager_start.bat (Windows) or packager_start.sh (Linux) script. The sample server listens on the multicast address 239.235.0.3:14000, and runs the local origin server on port 8080. The output is configured to be written to packager/webroot/_default/_default/_ directory.

The packager has a two-level hierarchy for grouping streams. At the top level, the Containers correspond to individual server tenants, such as users, and the next level, Streams correspond to programs. A Stream may have only one configured multicast input, but may have multiple output targets. New containers can be loaded dynamically at runtime, and individual containers may be refreshed at runtime in order to pick up new child Streams. These admin operations are exposed through the JMX management interface of the packager.

The packager configuration files are located in the conf directory. Under the conf directory, there are two main sub directories (containers and http). The containers directory can have one or more <containername> subdirectories. The name of the directory identifies the container. Each <ContainerName> directory must have a file named container.xml for container level configurations, although this file can be empty. Within a container directory there can be one or more <StreamName> sub directories. Each of these directories must have a file named stream.xml containing the configuration for the stream. The stream directory name identifies the stream.

The http directory is used for configuring the bundled HTTP server. It uses the origin.xml file to set the interface, address, port number, and the webroot directory.

For instance, if you need to configure the packager for two containers, user A with streams channel 1 and channel 2 and user B with a single stream channel 1, the directory structure will be:

```
/conf
/http
  /origin.xml
/containers
  /user A
    /container.xml
    /streams
      /channel 1
        /stream.xml
      /channel 2
        /stream.xml
  /user B
    /channel 1
      /stream.xml
```

Stream configuration

The stream configurations are organized in containers enabling logical grouping of streams.

The packager is multi-tenant, meaning it may have multiple separate ingest streams being packaged at once using multiple individual configurations. Each stream may be packaged as HDS, HLS, or both.

The stream configurations are organized in containers enabling logical grouping of streams. For example, if you have multiple ingest streams of the same media but in different bit-rates, you can logically group them under a common container.

The packager supports a simple configuration, which will have the following directory structure:

```
/packager
/conf
  /containers
    /_default_
      container.xml
    /streams
      /_default_
        stream.xml
```

From the above structure, you can infer that there is a single container, which contains a single stream configuration. Both the container and the stream are named “_default_”. There are no restrictions on the names apart from the file system’s disallowed characters. Each container must contain a container.xml file, which can be empty. Currently a container has no configuration settings. Each stream must contain a stream.xml file, which is the configuration for a single stream.

Each stream’s configuration must define a multicast socket to receive the ingest stream, plus one or more output for the packaged media files. The sample configuration from the build file uses the multicast socket 239.235.0.3:14000. Once the packager starts, it will create a stream module, which will listen on this multicast socket.

Sample stream configuration

The following example describes a sample stream configuration:

```
<Stream>
<!-- Desired duration of each fragment (both HDS and HLS) in ms -->
<FragmentDuration>4000</FragmentDuration>
<!-- HLS target duration must be >= fragment duration + interval between I-frames in stream -->
<TargetDuration>6000</TargetDuration>
<Input>
  <TSMulticast>
    <MulticastAddress>239.235.0.3</MulticastAddress>
    <MulticastPort>14000</MulticastPort>
  </TSMulticast>
</Input>
<OutputPipeline>
  <OutputType>Hds</OutputType>
  <!-- duration of media to keep on disk in hours -->
  <DiskManagementDuration>3.0</DiskManagementDuration>
  <Output>
    <LocalOrigin>
      <ContentPath>webroot/live/sample</ContentPath>
    </LocalOrigin>
  </Output>
</OutputPipeline>
</Stream>
```

The HDS files are written to packager/webroot/live/sample with desired fragment duration of 4 seconds. Note that fragments older than 3 hours of content are removed automatically by the Live Packager. The HLS files are written to packager/webroot/live/sample_hls with desired fragment duration of 4 seconds and a target duration of 6 seconds. Fragments that are older than 3.0 hours of content are automatically removed from the disk. However, you can configure this value.

As the output is configured to write to the disk, the packager will serve the media content through its internal HTTP server. From the configuration provided above, you can use the following URLs to play back the streams:

- HDS - <http://<packager-hostname>:8080/live/sample/livestream.f4m>
- HLS - http://<packager-hostname>:8080/live/sample_hls/livestream.m3u8

Packager configuration

stream.xml

The stream.xml file has the following format:

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <!-- <TargetDuration>6000</TargetDuration> -->
  <!-- uncomment when <OutputType> element is set to HLS -->

  <Input>
    <!--
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <MulticastPort>14000</MulticastPort>
    </TSMulticast>
    -->
    <RTMP>
      <!-- If RTMP is enabled then only HDS output is supported -->
```

```

</RTMP>
</Input>
<OutputPipeline>
  <OutputType>HDS</OutputType> <!-- Supporting output type are HDS and HLS for input from
multicast and HDS for input over RTMP -->
  <DiskManagementDuration>3.0</DiskManagementDuration> <!-- Comment out this line when
<LocalOrigin> element is not use -->
  <!-- <KeyFrameOnly>true</KeyFrameOnly> --> <!-- uncomment to create a key frame only stream
for trick play -->
  <SkipBadFragments>true</SkipBadFragments>
  <!-- For FAXS3 setup, please refer to
http://www.adobe.com/support/flashaccess/pdfs/FAXS_3_0_QuickStart.pdf for detail. -->
  <!--
  <ContentProtection>
    <FAXS4>
      <LicenseServerURL>http://<IP or FQDN>:8080</LicenseServerURL>
      <LicenseServerCertificate> my.der</LicenseServerCertificate>
      <LicenseServerCredential> my.pfx </LicenseServerCredential>
      <LicenseServerCredentialPassword>xxxxxxxxxxxx</LicenseServerCredentialPassword>
      <PackagerCredential> my.pfx </PackagerCredential>
      <PackagerCredentialPassword>xxxxxxxxxxxx</PackagerCredentialPassword>
      <TransportCertificate> my.der </TransportCertificate>
      <CommonKey> Path_to_certs/commonkey.bin </CommonKey>
      <PolicyFile> Path_to_certs/rootPolicy2.pol </PolicyFile>
    </FAXS4>
  </ContentProtection>
-->
<Output>
  <LocalOrigin>
    <ContentPath>webroot/livepkg/livestream1</ContentPath>
    <!-- Write fragments to packager's webroot as default location . -->
  </LocalOrigin>
  <!-- Uncomment <HttpPush> element to enable the specified Origin to received pushed fragments
-->
  <!-- <UseSecurityToken> and <SecurityTokenKey> elements are required when the Origin is
configured with <Key> element enabled -->
  <!-- Note: The Origin must be configured with matching stream module path as specified in
<TargetURL> element. -->
  <!--
  <HttpPush>
    <UseSecurityToken>true</UseSecurityToken>
    <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abd</SecurityTokenKey>
    <TargetURL>http://localhost:8090/_default/_default_</TargetURL>
  </HttpPush>
-->
</Output>
</OutputPipeline>
</Stream>

```


stream.xml definition

Configuration	Details
//Stream/FragmentDuration	The fragment duration to use for packaging that is basically the length of each fragment. This is the duration of the output fragments in milliseconds. The default value is 4000.
//Stream/TargetDuration	The value to be set for #EXT-X-TARGETDURATION tag of the m3u8 file. The default value is 6000.
//Stream/DiskManagementDuration	The duration of the content window (in decimal fractions of hours). Fragments and cues will be removed as they roll out of the window. The default is 3.0 hours for HDS and 0.083 hours (5 minutes) for HLS; 0.0 disables disk management. This disk management duration also determines the size of the m3u8 file generated for HLS

Configuration	Details
	streaming. The value for <code>DiskManagementDuration</code> of HLS differs from that of HDS because the value for HLS will be much shorter as a large value for HLS can lead to the creation of large m3u8 files.
<code>//Stream/SkipBadFragments</code>	If this is set to true, and if an error such as a discontinuity is found in the input transport stream, the entire current fragment will be discarded, and a gap will be inserted in the bootstrap file.
<code>//Stream/ContainsVideo</code>	<p>It should be set to false if the stream doesn't contain video. The default is true. Note that <code>ContainsVideo=true</code> is required for keyframes only. <code>ContainsVideo=false</code> is required for audio-only ingestion. Also see <code>IsAudioOnly</code>.</p> <p>If this configuration is set to false and if the input stream contains video, then an error is logged and the video data is dropped. Only the audio data will be packaged.</p>
<code>//Stream/Input/TSMulticast/MulticastAddress</code>	The packager listens for the incoming multicast packets for this stream on this multicast address. The address can be either an IPV4 or IPV6 multicast address.
<code>//Stream/Input/TSMulticast/MulticastPort</code>	The packager listens for incoming multicast packets on this port.
<code>//Stream/Input/TSMulticast/InterfaceAddress</code>	An IP address that identifies the interface on which to receive the multicast packets (optional).
<code>//Stream/Input/TSMulticast/SocketReceiveBufferSize</code>	The size of the kernel multicast receive buffer for the stream. The default is 64K bytes.
<code>//Stream/Input/TSMulticast/ReceiveBufferSize</code>	The size of the user space buffer allocated to receive incoming multicast packets. This should be the same as the maximum packet size of the encoder. The default is 2048 bytes.
<code>//Stream/Input/TSMulticast/OutputQueueScoreThreshold</code>	The number of transport stream bytes that will be processed for this, before giving up its thread. The default is 16k bytes.
<code>//Stream/Input/TSMulticast/OutputQueueMaxSize</code>	The maximum number of transport stream bytes, that will be queued for processing, before older bytes will be dropped. The default is 256k bytes.
<code>//Stream/OutputPipeline</code>	Represents one output pipeline for incoming streams. Incoming streams can be fed to multiple output pipelines to support different types of clients.
<code>//Stream/OutputPipeline/CueMode</code>	(Optional) The Cue Info Mode used for cue information to be added to m3u8/f4m. Allowed values are <code>PT_1_0</code> , <code>DPISimple</code> , and <code>DPIScte35</code> . Default value: <code>DPIScte35</code> .
<code>//Stream/OutputPipeline/ID</code>	Unique ID for a given Output Pipeline. It can be any alphanumeric string. For backward compatibility, it is not a mandatory configuration parameter. However, if more than one Output Pipeline of the same Output Type is configured, ID is mandatory and each Output Pipeline should define a different ID.
<code>//Stream/OutputPipeline/OutputType</code>	Specifies the type of output pipeline. Supported types are Hds and Hls. Note that there can be multiple Output Type containers used to provide different output or to publish to multiple origin servers.

Configuration	Details
//Stream/OutputPipeline/TempDir	Specifies the path of the temp directory for this output pipeline. If not configured, the default temp directory path is specified as temp/<ContainerID>/<streamID>
//Stream/OutputPipeline/IsAudioOnly	Generates an audio only output stream. Requires an input stream that contains both video and audio. The default is false. Note that <code>ContainsVideo</code> should be set to true for this option to function properly. This option is mutually exclusive with the <code>KeyFrameOnly</code> option.
//Stream/OutputPipeline/FragmentName	(Optional) Specifies the custom fragment name prefix. By default, the fragment name is prefixed with livestream. The fragment name format is <fragmentNamePrefix>Seg1Frag<FragmentNumber>. Only alpha-numeric characters and underscores are allowed for the <code>FragmentName</code> configuration.
//Stream/OutputPipeline/KeyFrameOnly	Generates a keyframe-only output stream for the trick play support. This option is mutually exclusive with <code>IsAudioOnly</code> and requires <code>ContainsVideo</code> to be set to true. See Trick play scenario.
//Stream/OutputPipeline/ManifestBaseURL	This will be written into the manifest and is the location where clients can fetch the stream fragment files. This should normally be the same as the HTTP push/base URL or local file path.
//Stream/OutputPipeline/ContentProtection	Content protection configuration for HDS streams. This configuration is the same for both HDS and HLS.
//Stream/OutputPipeline/KeyFrameOnly (for OutputType=HDS)	(Optional). Set as true to generate a keyframe only output for trick play. By default, this option is false. This configuration is supported for HDS only.
//Stream/OutputPipeline/SMPTETimecodeMapInterval	Interval in milliseconds at which the current SMPTE Time code value is injected into the stream-level manifest. Default is 30s (30000 milliseconds).
//Stream/OutputPipeline/Output/HttpPush	Defines an output target where content will be pushed to an upstream server.
//Stream/OutputPipeline/Output/HttpPush/UseSecurityToken	<p>When active, adds an encrypted security token to the outgoing HTTP request. This can be validated by the upstream server to provide publish-time access control.</p> <p>While sending PUT requests to Primetime Origin, if the token security is enabled, Primetime Packager will use the <code>X-Adobe-HTTP-Token</code> header for the token. Because the token from Packager is sent in the new header, PUT request to security-enabled Primetime 1.0 Origins do not work. This parameter is disabled by default, meaning that its value is false. If <code>UseSecurityToken</code> is set to true, you must provide a valid AES 128 Key in the <code>SecurityTokenKey</code> tag. No default key is used to generate a token.</p>
//Stream/OutputPipeline/Output/HttpPush/SecurityTokenKey	The AES-128 bit key to be used for encrypting the PUT token. This value shall be a string of 32 hexadecimal digits.
//Stream/OutputPipeline/Output/HttpPush/TargetURL	URL for HTTP PUT. The file name is appended to this.

Configuration	Details
//Stream/OutputPipeline/Output/HttpPush/MaxConcurrentPushCount	Maximum number of fragments that can be dispatched in parallel. The default value is 1. This implies that the fragments will be pushed by the packager to remote origin serially by default. To push large fragments of high bitrate over high latency network concurrently, it can be set to a value greater than 1. This option supersedes the older configuration option SerializePush, which is no longer supported.
//Stream/OutputPipeline/Output/LocalOrigin	Defines a local origin output target, where content is written to the local filesystem where it can be pulled via HTTP GET.
//Stream/OutputPipeline/Output/LocalOrigin/ContentPath	Path for local file output.
//Stream/OutputPipeline/Output/ManifestPush	If configured, the manifest (.f4m) file is pushed (through POST) here, rather than the HTTP Push/baseURL (Optional).
//Stream/OutputPipeline/Output/ManifestPush/UseSecurityToken	When active, adds an encrypted security token to the outgoing HTTP request. This can be validated by the upstream server to provide publish-time access control.
//Stream/OutputPipeline/Output/ManifestPush/SecurityTokenKey	The AES-128 bit key to be used for encrypting the PUT token. This value shall be a string of 32 hexadecimal digits.
//Stream/OutputPipeline/Output/ManifestPush/BaseURL	This will be written into the manifest and is the location where clients can fetch the stream fragment files. This should normally be the same as the HTTP push/base URL or local file path.
//Stream/MediaBufferLength	<p>(Optional, defaults to 10000) Length of the timestamp ordering buffer in milliseconds. This buffer is used to handle the time lag between audio and video tracks and hence ensuring that the audio-video frames are PUT in fragments in monotonically increasing order of timestamp.</p> <p>This value should be set to maximum expected time lag between the audio-video tracks. This buffering is only required for multiple track streams. In case of single track input/output, this field should be set to 0 to avoid unnecessary lag.</p>
//Stream/Input/TSMulticast/OutputQueueScoreThreshold	The number of transport stream bytes which will be processed before giving up its thread. The default is 128 kbytes and 0 will drain the entire output queue.
//Stream/OutputPipeline/VideoStreamID	<p>The PID of the video stream to be packaged. Other video streams in the input transport stream will be discarded. If no video stream ID is specified then packager will start packaging the first video PID seen in the input stream.</p> <p>If no Video PID is configured, the first video PID encountered is picked for packaging.</p> <p>If the configured Video PID is not found, then an error is logged and the video is not packaged till the video data with the configured PID is received.</p>

Configuration	Details
//Stream/OutputPipeline/AACStreamID	<p>The PID of the audio stream to be packaged. Other audio streams in the input transport stream will be discarded. By default, the first audio PID detected by the packager will be selected.</p> <p> Note: In the case of invalid AACStreamID, an error will be logged in the console and the audio will not be packaged in the output till an audio stream with the configured AACStreamID is received by the packager.</p>

Local origin HTTP server

An optional local origin HTTP server can be configured using the `<PACKAGER_ROOT>/conf/http/origin.xml` file.

This runs an HTTP server in its own thread pool within the packager process. It can be used for serving HLS or HDS streams generated by the packager, as an alternative to using a remote origin.

: The local HTTP server supported by the packager is intended for development and debugging use only. For production use, set up a remote origin server. The local origin configuration to write files to disk is supported as a production workflow. However, the HTTP server to host those files to the client is not preferred in a production setup. For HTTP hosting, the packager should push the files to a remote Origin Server.

The local origin HTTP server has a static mapping for HTTP GET requests from the request path to the configured web root. The HTTP server will not allow access to folders when serving out a GET request. The HTTP server will set the “Content-type”, “Cache-Control”, and “last-Modified” HTTP headers as applicable. Additionally, it will also validate the “If-Modified-Since” header and will return an HTTP 304 response if the client has a valid copy of the content.

The local server allows GET requests of any file under the `webroot` folder. However, the packager uses a two-level configuration structure (streams and stream containers) to define input streams and output targets.

The directory structure for this is:

```
/conf
/http
  /origin.xml    <- Local origin HTTP server configuration (optional)
/containers
  /user A
    /container.xml <- Per container configuration
    /streams      <- Streams (i.e. channels) for this programmer (i.e. tenant)
      /channel 1
        /stream.xml <- Per stream configuration
      /channel 2
        /stream.xml
      ...
```

HTTP server configuration

When using the local origin configuration of the packager, the packager can serve the packaged media files over HTTP through GET requests.

By default, the packager’s HTTP server listens on port 8080 for client requests. You can change this port through the configuration file.

The following configuration is from the <PACKAGER_ROOT>/conf/http/origin.xml file:

```
<Config>
  <!-- IP address to set the HTTP server to listen on -->
  <InterfaceAddress></InterfaceAddress>
  <Webroot>./webroot</Webroot>
  <Port>8080</Port>
  <MaxContentLength>4096000</MaxContentLength>
  <WorkerThreadCount>10</WorkerThreadCount>
</Config>
```

origin.xml

If an origin.xml file exists, the packager runs an internal HTTP server to handle the GET requests, obtaining its global configuration from the origin.xml file.

The origin.xml file has the following format:

```
<Config>
  <!--
  Enable or disable the local origin
  -->
  <LocalOriginEnabled>true</LocalOriginEnabled>
  <!--
  IP address to set the HTTP server to listen on
  -->
  <InterfaceAddress></InterfaceAddress>
  <!--
  The folder from which content is read.
  -->
  <Webroot>./webroot</Webroot>
  <!--
  The HTTP server listens for incoming requests on this port
  -->
  <Port>8080</Port>
  <!-- To support failover, the server will return this HTTP response code if a requested fragment
  is missing. -->
  <MissingFragmentResponseCode>503</MissingFragmentResponseCode>
  <!-- Http cache control and expiration based upon the file type -->
  <Headers>
    <TTL>
      <F4M>0</F4M>
      <Bootstrap>0</Bootstrap>
      <M3U8>0</M3U8>
      <Timeline>0</Timeline>
      <DashManifest>0</DashManifest>
      <DashManifestPart>0</DashManifestPart>
    </TTL>
  </Headers>
</Config>
```

origin.xml definition

Configuration	Details
LocalOriginEnabled	Enable and disable the local HTTP server. The default value is true.
InterfaceAddress/Port	Interface and Port for the HTTP server to listen on. The default value for Port is 8080.
Webroot	The packaged content will be stored under this directory. The default value is ./webroot.

Configuration	Details
MaxContentLength	The value should be greater than the maximum fragment size, in bytes. The default value is 8192000.
MissingFragmentResponseCode	If a requested fragment is not found, this response code will be returned. The default is 404.
WorkingThreadCound	The total number of threads to be used by the server. Ideally, the number of worker threads = 2 * number of processors. The default value is 10.

container.xml

A container.xml is required for directories under `<PACKAGER_ROOT>/conf/containers` to be recognized as containers.

The container.xml file may be empty, or it may contain configuration information that applies to all streams under that container directory.

```
<Container>
  <Name>sample_set</Name>
  <ContentProtection>
  <Domain></Domain>
</ContentProtection>
</Container>
```

container.xml definition

Configuration	Details
//Container/Name	Name of the stream set.
//Container/ContentProtection	Contains the DRM settings for this stream set.
//Container/Domain	Name of the domain.

A container.xml is required for directories under `conf/containers` to be recognized as containers. The container.xml file may be empty, or it may contain configuration information that applies to all streams under the container directory.

The following is a sample container.xml file:

```
<Config>    </Config>
```

Ingesting stream

The packager requires an MPEG 2-TS formatted file streamed over a multicast socket for input.

For testing purposes, you can use tools like `ffmpeg` or a hardware encoder to ingest a stream.

The manifest file

The manifest file (`livestream.f4m`), contains the inline bootstrap file, which is base-64 encoded.

The bootstrap file contains bootstrap information for each segment. The manifest file format is same as the HDS manifest file, except for the following changes:

- A tag named “TargetDuration” is added under the document root which specifies the target duration of HLS stream in milliseconds. Its value is picked from //Stream/TargetDuration. The packager sets the value of TargetDuration in the m3u8 file.
- The cueInfo tag group represents the SCTE-35 splice-out cues taken from the ingest stream.

Packager usage scenarios

The following sections describe some common packager usage scenarios. There may be other valid usage scenarios not covered in this document.

Local origin scenario

Before a remote player (iOS/OSMF) can communicate with an origin server over HTTP for streaming content, an origin server must run as an HTTP server.

Changes in origin.xml for local origin

The <PACKAGER_ROOT>/conf/http/origin.xml is the configuration file used to set up an origin server as an HTTP Server. This configuration file lets administrators define the port to bind, set the web root entry point, and set the location for log files.

The example below shows the default settings supported by the origin server. The HTTP server is set to bind to port 8090 with the web root set to httporigin/webroot.

```
<Config>
  <!-- IP address to set the HTTP server to listen on -->
  <InterfaceAddress></InterfaceAddress>
  <Webroot>./webroot</Webroot>
  <Port>8090</Port>
  <AccessLogs>
    <LogFolder>./logs/http</LogFolder>
  </AccessLogs>
  <MaxContentLength>4096000</MaxContentLength>
  <MissingFragmentResponseCode>503</MissingFragmentResponseCode>
  <Headers>
    <TTL>
      <F4M>0</F4M>
      <Bootstrap>0</Bootstrap>
      <M3U8>0</M3U8>
      <Timeline>0</Timeline>
      <DashManifest>0</DashManifest>
      <DashManifestPart>0</DashManifestPart>
    </TTL>
  </Headers>
  <!-- uncomment to override default setting: WorkerThreads = 2 * NoOfProcessors -->
  <!-- <WorkerThreadCount>10</WorkerThreadCount> -->
</Config>
```

Writing HDS/HLS fragments to the local origin server

This is the basic packager setup that lets you quickly verify whether the Live Packager can ingest MPEG-2 transport streams through the specified multicast port and package them into HDS/HLS fragments.

In this simple setup, the configuration is set to write HLS content to disk (localhost). A difference between local origin and remote origin is how fragments are written out to the destinations. For remote origin, the Live Packager does an HTTP push (using the network pipe) to get fragments to the remote location. On the other hand, the local origin

does not use the network pipe but instead it uses disk I/O write to write out fragments to the designated directory under its web root directory.

This local origin setup does not include configuration for security or content protection. This capability is intended to allow content to be pushed from the local broadcast center to a remote origin. This keeps the source packaging portions of the workflow distinct from the distribution portions of the workflow. For example, a broadcaster may run a local packager on premises, and then offload the actual content distribution to a CDN or an Origin Services provider.

Changes in container.xml for local origin

The packager module loading system requires the file

`<PACKAGER_ROOT>/conf/containers/_default_/container.xml`. It can be an empty file if there is nothing to configure.

Changes in stream.xml for local origin

The following is a sample configuration for

`<PACKAGER_ROOT>/conf/containers/_default_/streams/_default_/stream.xml`.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <!-- Uncomment the line below if stream is configured for HLS output
  <TargetDuration>6000</TargetDuration>
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <MulticastPort>14000</MulticastPort>
    </TSMulticast>
  </Input>
  <OutputPipeline>
    <!-- Replace HLS with HDS in the line below if stream is configured for HDS output -->
    <OutputType>HLS</OutputType>
    <!-- the line below is only applicable when stream is configure to output to local origin
    -->
    <DiskManagementDuration>3.0</DiskManagementDuration>
    <Output>
      <LocalOrigin>
        <!-- Store HDS fragments to packager's webroot/live/sample directory -->
        <TargetURL>webroot/_default_/_default_/</TargetURL>
      </LocalOrigin>
    </Output>
  </OutputPipeline>
</Stream>
```

Viewing the HLS streaming content

You can view the HLS streaming content that the packager outputs.

Type the address of the m3u8

(`http://<your_origin_server>:8080/_default_/_default_/livestream.m3u8`) on your mobile device supporting HLS content.

Remote origin scenario

You can configure a packager to push HLS/HDS fragments to multiple remote origins to support fail-over and load balancing.

The following example shows a packager configured to upload HLS fragments to two Origin servers. The stream configuration below is set to listen to multicast MPEG 2 transport streams on 239:235.0.3 at port 14000 and to

package the content into HLS fragments. Then the fragments are sent to two remote origin servers. The token key setting is used to secure communication between the packager server and the remote origin servers. Only the packager with the proper token key can upload fragments to the origin server.

Changes in container.xml for remote origin

The packager module loading system requires the container.xml file (<PACKAGER_ROOT>/conf/containers/_default/container.xml). However, it can be empty.

Changes in stream.xml for remote origin

The following example is taken from

<PACKAGER_ROOT>/conf/containers/_default_/streams/_default_/stream.xml.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <!-- comment out the line below if stream is configured for HDS output -->
  <TargetDuration>6000</TargetDuration>
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <MulticastPort>14000</MulticastPort>
    </TSMulticast>
  </Input>
  <OutputPipeline>
    <!-- Replace HLS with HDS in the line below if -- stream is configured for HDS output -->
    <OutputType>HLS</OutputType>
    <Output>
      <!-- **Send HLS output to multiple remote origins' -- webroot/_default/_default_ directory -->
      <HttpPush>
        <!-- Secure communication between packager and httporigin -->
        <UseSecurityToken>true</UseSecurityToken>
        <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
        <TargetURL>http://host1:8090/_default/_default_</TargetURL>
      </HttpPush>
      <HttpPush>
        <UseSecurityToken>true</UseSecurityToken>
        <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
        <TargetURL>http://host2:8090/_default2/_default2_</TargetURL>
      </HttpPush>
    </Output>
  </OutputPipeline>
</Stream>
```

Multi-bitrate scenario

Setting for multi-bitrate allows a player to automatically switch to lower rendition for better performance when the network traffic is congested.

Multi-bitrate scenario for HDS

To configure a packager to support multi-bitrate, a user first needs to create a separate folder for each rendition under the packager's <PACKAGER_ROOT>/conf/containers/<programmer>/streams/directory. Then, create a stream.xml for each newly created rendition folder. The examples below shows stream settings for three different renditions (400kpbs, 700kpbs, 1000kpbs). Each rendition is broadcast on a separate port that the packager listening in.



Note: You must configure the remote HTTP origin to have a matching URI (for example, *stream_150kbps*). The value of the *SecurityTokenKey* element must match the corresponding one specified in the HTTP Origin's *stream.xml* file.

Changes in container.xml for multi-bitrate

The container.xml file is required by the packager module loading system. However, it may be empty.

Changes in stream.xml for multi-bitrate

The following example is taken from

`<PACKAGER_ROOT>conf/containers/_default_/streams/400kbps/stream.xml`.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <!-- rendition 400kbps MPEG-2 transport stream is broadcast on port 30004 -->
      <MulticastPort>30004</MulticastPort>
    </TSMulticast>
  </Input>
  <OutputPipeline>
    <OutputType>HDS</OutputType>
    <Output>
      <HttpPush>
        <UseSecurityToken>true</UseSecurityToken>
        <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
        <TargetURL>http://<your_origin_server>:8090/_default_/400kbps </TargetURL>
      </HttpPush>
    </Output>
  </OutputPipeline>
</Stream>
```

The following example is taken from

`<PACKAGER_ROOT>/conf/containers/_default_/streams/700kbps/stream.xml`.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <!-- rendition 700kbps MPEG-2 transport stream is broadcast on port 30007 -->
      <MulticastPort>30007</MulticastPort>
    </TSMulticast>
  </Input>
  <OutputPipeline>
    <OutputType>HLS</OutputType>
    <Output>
      <!-- **Store HLS fragments to a remote origin under the origin's webroot/_default_/700kbps
      directory -->
      <HttpPush>
        <UseSecurityToken>true</UseSecurityToken>
        <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
        <TargetURL>http://<your_origin_server>:8090/_default_/700kbps</TargetURL>
      </HttpPush>
    </Output>
  </OutputPipeline>
</Stream>
```

The following example is taken from

<PACKAGER_ROOT>/conf/containers/_default_/streams/1000kbps/stream.xml.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <!-- rendition 1000kbps MPEG-2 transport stream is broadcast on port 30010 -->
      <MulticastPort>30010</MulticastPort>
    </TSMulticast>
  </Input>
  <OutputPipeline>
    <OutputType>HLS</OutputType>
    <Output>
      <!-- **Store HLS fragments to a remote origin under the origin's webroot/_default_/1000kbps
      directory -->
      <HttpPush>
        <UseSecurityToken>true</UseSecurityToken>
        <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
        <TargetURL>http://<your_origin_server>:8090/_default_/1000kbps</TargetURL>
      </HttpPush>
    </Output>
  </OutputPipeline>
</Stream>
```

Changes for trick play in a multi-bitrate scenario

For each bitrate stream, specify its keyframe only stream path in the set level manifest.

For example, in the set level manifest, specify the trick play or keyframe only streams using relative paths, like this:

```
<manifest xmlns="http://ns.adobe.com/f4m/2.0">

  <id>mbr_trickplay</id>
  <imeType/>
  <streamType>live</streamType>
  <dvrInfo windowDuration="7200"/>
  <baseUrl>http://<your_origin_server>:8090/_default_/</baseUrl>

  <media href="400kbps/livestream.f4m" type="video" bitrate="400"/>
  // keyframe only stream for '400' bit rate
  <media href="400kbpsTrickPlay/livestream.f4m" bitrate="400" type="video-keyframe-only"/>

  <media href="700kbps/livestream.f4m" type="video" bitrate="700"/>
  // keyframe only stream for '700' bit rate
  <media href="700kbpsTrickPlay/livestream.f4m" bitrate="700" type="video-keyframe-only"/>

  <media href="1000kbps/livestream.f4m" type="video" bitrate="1000"/>
  // keyframe only stream for '1000' bit rate
  <media href="1000kbpsTrickPlay/livestream.f4m" bitrate="1000" type="video-keyframe-only"/>

</manifest>
```

Or, you can specify the keyframe only streams as follows, using the URL for your origin server:

```
<manifest xmlns="http://ns.adobe.com/f4m/2.0">

  <media href="http://<your_origin_server>:8090/_default_/400kbps/livestream.f4m" bitrate="400"/>
  // keyframe only stream for '400' bit rate
  <media href="http://<your_origin_server>:8090/_default_/400kbps/kfonly/livestream.f4m"
  bitrate="400" type="video-keyframe-only"/>

  <media href="http://<your_origin_server>:8090/_default_/700kbps/livestream.f4m" bitrate="700"/>
  // keyframe only stream for '700' bit rate
  <media href="http://<your_origin_server>:8090/_default_/700kbps/kfonly/livestream.f4m"
  bitrate="700" type="video-keyframe-only"/>

  <media href="http://<your_origin_server>:8090/_default_/1000kbps/livestream.f4m" bitrate="1000"/>
```

```
// keyframe only stream for '1000' bit rate
<media href="http://<your_organ_server>:8090/_default_/1000kpbs/kfonly/livestream.f4m"
bitrate="1000" type="video-keyframe-only"/>

</manifest>
```



Note: When specifying trick play or keyframe streams for VOD, only the media type attribute changes. The rest of the format remains the same as for live streams.

Multi-bitrate scenario for HLS

Once the streams are generated, you need to create an MLM (multi-level manifest) file which references the available streams at a particular network bitrate.

The file extension must be F4M and can be stored at any HTTP server that a player can access over the network. Such HTTP sites also need to have a valid crossdomain.xml at the web root level. Therefore, the MLM example (mbr.f4m) below indicates that when the network traffic is at 1000 kpbs or above, the player switches to the 1000 kpbs stream. The 700 kpbs stream is used when the bandwidth drops between 700kpbs & 1000-1kpbs. The switching happens automatically based on the current network traffic.

An example configuration:

```
<manifest xmlns="http://ns.adobe.com/f4m/2.0">
  <media href="http://<your_organ_server>:8090/_default_/400kpbs/livestream.f4m" bitrate="400"/>

  <media href="http://<your_organ_server>:8090/_default_/700kpbs/livestream.f4m" bitrate="700"/>

  <media href="http://<your_organ_server>:8090/_default_/1000kpbs/livestream.f4m" bitrate="1000"/>
</manifest>
```

Variant playlist for HLS

Apple iOS devices can request an M3U8 variant playlist file that can contain the location, bitrate, and optionally the codec of each stream.

The following configuration depicts a variant playlist:

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1280000
http://<remote-origin>/hls/low/livestream.m3u8,
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=2560000
http://<remote-origin>/hls/mid/livestream.m3u8,
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=7680000
http://<remote-origin>/hls/high/livestream.m3u8
```

Audio-only scenario

The packager can create audio-only streams from an incoming MPEG 2 TS stream that contains both audio and video content to provide support for adaptive bitrate streaming over cellular networks and synchronized audio stream for Second Audio Program (SAP).

You must add IsAudioOnly parameter to the OutputPipeline in the stream.xml file:

```
<OutputPipeline>
...
  <IsAudioOnly>true</IsAudioOnly>
...
</OutputPipeline>
```

Note the following:

- This function requires an input source that contains both audio and video. If the input contains only audio, do not use this function; rather, set the ContainsVideo option to false.
- IsAudioOnly and KeyFrameOnly are mutually exclusive; you must define each in separate OutputPipeline containers.

Protecting content scenario

There are various ways of using the packager for packaging and streaming protected content.

The Live packager supports content protection through the following mechanisms:

- PHDS
- PHLS
- Adobe Access

Protecting content using PHDS/PHLS

You can use PHDS or PHLS to protect your content without using a license server. All the necessary credentials are stored locally in the server. The packager supports the default setup for PHDS and PHLS streams.

The packager bundle includes all the necessary certificate files to enable content protection. These files are located at `<PACKAGER_ROOT>/creds`.

To enable PHDS or PHLS streams, you need to make the necessary changes in the stream.xml file as shown below:

```
<OutputPipeline>
...
<ContentProtection>
  <FAXS4><!-- empty config to use default --></FAXS4>
</ContentProtection>
...
</OutputPipeline>
```

When the `<FAXS4>` tag is empty, the packager extracts the default credential specified in the packager.xml file located at `<PACKAGER_ROOT>/conf`.

Changes in container.xml for content protection

The container.xml file (`<PACKAGER_ROOT>/conf/containers/live/container.xml`) is required by the Packager module loading system, but can be empty.

PHDS example for content protection

The following example is taken from

`<PACKAGER_ROOT>/conf/containers/_default_/streams/_default_/stream.xml`.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <!-- <TargetDuration>6000</TargetDuration> -->
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <MulticastPort>14002</MulticastPort>
    </TSMulticast>
  </Input>
  <!-- Supporting output type are HDS and HLS for input from multicast and HDS for input over RTMP -->
  <OutputPipeline>
    <OutputType>HDS</OutputType>
    <DiskManagementDuration>3.0</DiskManagementDuration>
```



```

<ContentProtection>
  <FAXS4><!-- empty config to use default --></FAXS4>
</ContentProtection>
<Output>
  <LocalOrigin>
    <ContentPath>webroot/_default/_default_</ContentPath>
  </LocalOrigin>
</Output>
</OutputPipeline>
</Stream>

```

PHLS example for content protection

The following example is taken from

<PACKAGER_ROOT>/conf/containers/_default_/streams/_default_/stream.xml.

```

<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <!-- <TargetDuration>7000</TargetDuration> -->

  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <MulticastPort>14002</MulticastPort>
    </TSMulticast>
  </Input>

  <!-- Supporting output type are HDS and HLS for input from multicast and HDS for input over
  RTMP -->

  <OutputPipeline>
    <OutputType>HLS</OutputType>
    <DiskManagementDuration>3.0</DiskManagementDuration>

    <ContentProtection>
      <FAXS4><!-- empty config to use default --></FAXS4>
    </ContentProtection>

    <Output>
      <LocalOrigin>
        <ContentPath>webroot/_default/_default_</ContentPath>
      </LocalOrigin>
    </Output>

  </OutputPipeline>
</Stream>

```

App whitelisting for content protection

For the PHDS/PHLS protection mechanisms, you can also enable app whitelisting. App whitelisting restricts the playback of encrypted content only to the specified apps/SWFs.

The application information (SWF Hash/iOS App Information) is included in the embedded license. The Adobe Access client ensures that the application information provided in the embedded license matches that of the application used for playback. If you want to enable App whitelisting for PHDS/PHLS, you can specify the config parameter <WhitelistFolder> in the configuration file.

In the case of iOS applications, the application information includes:

- publisher-id
- app-id
- min-ver
- max-ver

If you specify only a publisher identifier, any application from that publisher is valid. If only a publisher and application identifier are specified, any version of that application is valid. Publisher and application identifier are required if minimum or maximum version is specified. If a minimum version is specified, applications with a version number greater than or equal to the specified version are allowed. If a maximum version is specified, applications with a version number less than or equal to the specified version are allowed.

For PHDS, you can specify the directory path containing the whitelist SWFs. The Packager will include a hash of these whitelist SWFs in the license which is embedded in the metadata.

For PHLS too, you can specify the directory path containing the whitelist SWFs. This directory must contain a text file named `app.whitelist`. This file can have multiple entries where each entry is a comma separated list of publisher certificate, app-id, min-ver, max-ver of the app. The publisher certificate is a mandatory field in each entry and rest are optional. The publisher certificate is the certificate that is used to sign the applications.



Note: The specified publisher certificates must be present in the same directory. The Primetime packager will extract the publisher ID from these certificates.

The following example shows a sample `app.whitelist` file:

```
pub1_cert.cer, FJ8SDN490SJSD9HA83JA073JD03KSA0, 1.0, 4.0
pub2_cert.cer, NJFS84NFW0942NFWJFW904FW904THFW
pub3_cert.cer
```

This file will enable in whitelisting:

- Versions 1.0 to 4.0 of applications with ID FJ8SDN490SJSD9HA83JA073JD03KSA0 signed by `pub1_cert.cer`.
- All versions of the app with ID NJFS84NFW0942NFWJFW904FW904THFW signed by `pub2_cert.cer`.
- All apps signed by `pub3_cert.cer`.

In the case of Smart Origin, when the same module is used to JIT-encrypt both HDS and HLS, the same `<WhitelistFolder>` can contain both the whitelist SWFs and the `app.whitelist` file with the appropriate publisher certificates.

Protecting content using Adobe Access

The packager supports the DRM capabilities for HLS and HDS streaming using the features provided by Adobe Access.

Implementation of DRM will include encryption along with policy/right management through an Adobe Access license server. The packager supports the most common DRM setup involving HDS and HLS delivery using Adobe Access.

The Primetime supports Adobe Access encryption in both chained and non-chained mode for PHDS/PHLS. With non-chained license mode, the Adobe Access License Server credentials are not required. Therefore, content protection configuration `LicenseServerCredential` is no longer a mandatory configuration.

The Adobe Access core certificates are supported in the case of PHDS, PHLS, and Non License chaining mode. The Adobe Access Professional certificates are required in the case of Enhanced License Chaining mode. For Primetime Packager 1.2, or higher, an expired leaf license is not created for chained license policy. Leaf license properties, such as validity, start date, and end date are determined from the policy. To create an expired leaf license to enforce the license validity from root license, specify the end data in the policy.

Changes in `container.xml` for DRM

The `container.xml` (`<PACKAGER_ROOT>/conf/containers/_default_/container.xml`) file is required by the packager module loading system. However, it can be empty.

Changes in stream.xml for DRM

The following example is taken from

<PACKAGER_ROOT>/conf/containers/_default_/streams/_default_/stream.xml.

```
<Stream>
  <FragmentDuration>4000</FragmentDuration>
  <Input>
    <TSMulticast>
      <MulticastAddress>239.235.0.3</MulticastAddress>
      <MulticastPort>14000</MulticastPort>
    </TSMulticast>
  </Input>
  <OutputPipeline>
    <OutputType>HDS</OutputType>
    <DiskManagementDuration>3.0</DiskManagementDuration>
    <ContentProtection>
      <AAXS4>
        <KeyRotation>true</KeyRotation>
        <LicenseServerURL>http://<IP or FQDN>:8080</LicenseServerURL>
        <KeyServerURL>http://<IP or FQDN>:8080</KeyServerURL>
        <LicenseServerCertificate> <your cert.der> </LicenseServerCertificate>
        <RecipientCertificates> Path_to_shared_domain_certs </RecipientCertificates>
        <LicenseServerCredential> <your cert.pfx> </LicenseServerCredential>
        <LicenseServerCredentialPassword>xxxxxxxxxxxx</LicenseServerCredentialPassword>
        <PackagerCredential> <your cert.pfx> </PackagerCredential>
        <PackagerCredentialPassword>xxxxxxxxxxxx</PackagerCredentialPassword>
        <TransportCertificate> <your cert.der> </TransportCertificate>
        <CommonKey> Path_to_certs/commonkey.bin </CommonKey>
        <PolicyFile> Path_to_certs/rootPolicy2.pol </PolicyFile>
      </AAXS4>
    </ContentProtection>
    <Output>
      <HttpPush>
        <UseSecurityToken>true</UseSecurityToken>
        <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
        <TargetURL>http://host1:8090/_default/_default_ </TargetURL>
      </HttpPush>
    </Output>
  </OutputPipeline>
</Stream>
```

Key rotation

Instead of encrypting the content directly with the Content Encryption Key (CEK), you can use a Rotation Key (RK) to encrypt the content.

The Rotation Key will change every *N* seconds. Each Rotation Key will be encrypted with the CEK.

Key rotation can be enabled or disabled using the configuration parameter <KeyRotation>.

Key rotation for HLS

For HLS, if the key rotation is enabled, Adobe Access will provide the encrypted rotation key used to encrypt the TS fragments. This encrypted rotation key must be specified as a value for parameter `EncryptedRK` in the key URI in the playlist file.

The following example shows a sample M3U8 file containing the rotation key:

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-FAXS-CM:URI="livestream.drm"
#EXT-X-KEY:METHOD=AES-128,
URI="faxes://faxes.adobe.com?EncryptedRK=0x00112233445566778899aabbccddeeff",IV=0x15161718191A1B1C1D1E1F2021222324
#EXTINF:10,
http://orgserver/live/sample/livestream23585170.ts
#EXTINF:10,
```

```
http://orgserver/live/sample/livestream23595170.ts
#EXTINF:10,
http://orgserver/live/sample/livestream23605170.ts
#EXT-X-KEY:METHOD=AES-128,
URI="faxes://faxes.adobe.com?EncryptedRK=0x0068899a112233bb677ccdde4455aff",IV=0x15161718191A1B1C1D1E1F2021222324
#EXTINF:10,
http://orgserver/live/sample/livestream23615170.ts
#EXTINF:10,
http://orgserver/live/sample/livestream23625170.ts
#EXTINF:10,
http://orgserver/live/sample/livestream23635170.ts
```

Key rotation for HDS

For HDS, in order to support random access, the encrypted rotation key will be put at the beginning of each encrypted audio or video packet.

To decrypt the packet, the player will first use CEK to decrypt the rotation key and then use the rotation key to decrypt the content.

Using HSM to store packager credentials

Primestime Live Packager supports access to packaging credential certificates and common keys stored in a hardware security module (HSM) for enhanced security. To perform an encryption operation, Live Packager must fetch the private key from the HSM.

To be able to access the HSM Module, Live Packager must be in the same virtual private cloud (VPC) as that of the HSM Module. Alternatively, you can configure an OpenVPN server to authenticate and route traffic between Live Packager and the HSM.

Consequently, Primestime Platform components, including Live Packager, Offline Packager, and Origin Server that support the DRM packaging service must be within a VPC.

You can manage the HSM using the LunaSA cmu and vtl tools from any client that HSM trusts. If the HSM is not on the same VPC as the Live Packager instance, create a trust relationship between the HSM and the client machine.

To enable HSM access from Live Packager, configure the <HSMModule> element in the server.xml file within the <ContentProtection>/<FAXS4> element.

The following is a sample configuration for the <ContentProtection> element with HSM access enabled:

```
<ContentProtection>
  <FAXS4>
    ...
    <HSMModule>
      <!-- Provide Config information for the HSM Module access -->
      <SunPKCS11ConfigFile></SunPKCS11ConfigFile>
      <!-- Config file used with the SunPKCS11 Provider to access HSM -->
      <Password></Password>
      <!-- base64 encoded format -->
      </HSMModule>
    ...
  </FAXS4>
</ContentProtection>
```

The useHSM attribute for the PackagerCredential element indicates if the credential should be accessed using HSM or the file system. The credential is an alias if it is accessed using HSM.

```
<PackagerCredential useHSM="true">...</PackagerCredential>
```

The useHSM attribute for the CommonKey element indicates whether the key should be accessed using HSM or the file system.

```
<CommonKey useHSM="true">...</CommonKey>
```

The following is the complete configuration for the Content Protection element:

```
<Config>
  <ContentProtection>
    <FAXS4>
      <HSModule>
        <SunPKCS11ConfigFile>hsm_conf/pkcs11.cfg</SunPKCS11ConfigFile>
        <Password>VxfVTnUVjMn=</Password>
      </HSModule>
      <LicenseServerURL>http://primetype.aaxs.adobe.com</LicenseServerURL>
    </FAXS4>
    <LicenseServerCertificate>creds/static/phds_license_server.der</LicenseServerCertificate>
    <LicenseServerCredential>creds/static/phds_license_server.pfx</LicenseServerCredential>
    <LicenseServerCredentialPassword>v/lB4EoNQqQ=</LicenseServerCredentialPassword>
    <PackagerCredential useHSM="true">packager-credential-alias</PackagerCredential>
  </ContentProtection>
</Config>
```

Verifying protected streams

You can verify that the stream protection is functional and that the DRM metadata is created at the target-end along with fragments.

Check the console output or <PACKAGER_ROOT>/logs/packager.log.0 for messages similar to the ones shown below:

```
[root@example]# ./packager_start.sh
Oct 30, 2012 1:39:26 AM com.adobe.fms.packager.Packager main
INFO: Adobe Primetype Packager version: 1-0-1-bXXXX
Oct 30, 2012 1:39:27 AM com.adobe.fms.packager.Packager start
INFO: Adobe Primetype Packager starting up...
Oct 30, 2012 1:39:27 AM com.adobe.fms.module.ClassSandboxStreamContainerModule setRootDir
INFO: StreamContainer: [_default_] loads with 0 jar file(s):
Oct 30, 2012 1:39:28 AM com.adobe.fms.encryption.ContentEncryptor
INFO: [_default_,_default_] Packager Certificate is valid till: Sat Nov 03 16:59:59 PDT 2012
Oct 30, 2012 1:39:28 AM com.adobe.fms.encryption.ContentEncryptor
INFO: [_default_,_default_] Transport Certificate is valid till: Sat Nov 03 16:59:59 PDT 2012
Oct 30, 2012 1:39:28 AM com.adobe.fms.encryption.ContentEncryptor
INFO: [_default_,_default_] License Server Certificate is valid till: Sat Nov 03 16:59:59 PDT 2012
Oct 30, 2012 1:39:29 AM com.adobe.fms.m2tspackager.M2TSInput inputFromMulticast
INFO: [_default_,_default_] Joined multicast group 239.235.0.3:14002
Oct 30, 2012 1:39:29 AM com.adobe.fms.http.HttpServer start
INFO: HTTP File Server on /0:0:0:0:0:0:0:0:9880 is started.
Oct 30, 2012 1:39:29 AM com.adobe.fms.packager.Packager start
INFO: Adobe Primetype Packager has successfully started.
```

Ingestion of RTMP streams

This feature extends the functionality of the Live Packager to allow ingestion of RTMP streams, in addition to the TS streams over multicast.

This feature also allows the Live Packager to act as an RTMP server, to which the RTMP streams can be published. The HDS output functionalities available to streams ingested over TS-Multicast will also be available to streams ingested over RTMP. Conversion to HLS at the packager level is not supported. Only RTMP publish workflows are supported by Primetype.

You can find an RTMP server per packager instance having a valid `/conf/rtmp/server.xml` file. The RTMP server will cater to all the incoming connections and it will also dispatch audio, video, and data messages to stream modules described by the `stream.xml` files allowing RTMP input. The stream modules in turn can be configured for HDS output in the same way as they are configured for TS multicast input. All the output options including content protection and dispatch of generated fragments to local/remote origin are supported.

All valid RTMP publish streams are supported. The publishing endpoint correctly handles the error responses to the following RTMP commands:

- **Connect** - A valid RTMP connect URL must be of the form `rtmp://<server>:<port>/<config>` where `<config>` must correspond to a valid container in the packager configuration. Other query parameters on the URL will be ignored.
- **Publish** - A valid RTMP publish command must contain a stream name that corresponds to a packager stream configuration that has RTMP input enabled. Such a stream must be available within the container referred to while establishing the corresponding RTMP connections through the connect command. Additional parameters in the publish command will be ignored. The publish command will be allowed to the packager stream only once.

To publish to a packager stream configuration of the form `/container1/stream1`, your RTMP publish client must:

- Issue a valid connect command with the URL `rtmp://<server>:<port>/container1`
- Issue a valid publish command with the stream name = "container1"

server.xml definition for RTMP

The `server.xml` file, available in the `./conf/rtmp/` folder, enables the RTMP server in the packager.

The following fields are available for configuration in this XML file:

Configuration	Optional	Default	Description
<code>//Config/Port</code>	Yes	1935	The port on which the RTMP server will listen for incoming connections.
<code>//Config/InterfaceAddress</code>	Yes	0.0.0.0	The IP address of the interface on which the RTMP server will listen.
<code>//Config/WorkerThreadCount</code>	Yes	2 processors	2*Number_of_processors and 1MB in bytes
<code>//Config/SocketReceiveBuffer</code>	Yes	1048576	Size in bytes of the TCP Socket receive buffer.
<code>//Config/AuthPlugin</code>	Yes		Tag to be added to the <code>server.xml</code> file to enable authentication
<code>//Config/ClassName</code>	Yes		Name of the main class of the plugin
<code>//Config/Init</code>	Yes		Custom element where initialization configuration for the auth plugin can be specified

The following example depicts a valid server.xml configuration:

```
<Config>
  <!--
  IP address to set the RTMP server to listen on
  -->
  <InterfaceAddress></InterfaceAddress>
  <!--
  The RTMP server listens for incoming requests on this port
  -->
  <Port>1935</Port>
</Config>
```

stream.xml definition for RTMP

The stream.xml file for ingesting an RTMP stream must have an RTMP tag under //Stream/Input. All the other fields available in the packager stream.xml file as described in stream.xml are applicable.

The following example depicts a valid stream.xml configuration:

```
<Stream>
  <FragmentDuration>2000</FragmentDuration>
  <DiskManagementDuration>3.0</DiskManagementDuration>
  <!-- Length of timestamp ordering buffer in milliseconds. This buffer is used to handle
  time lag between audio and video tracks and hence ensuring that audio-video frames
  are put in fragments in monotonically increasing order of timestamp.
  This should be set to maximum expected time lag between audio-video tracks.-->
  <MediaBufferLength>3000</MediaBufferLength>
  <ContentProtection>
    <FAXS3>
      <LicenseServerURL><IP or FQDN>:8080</LicenseServerURL>
      <LicenseServerCertificate>
Path_to_certs/PRERELEASE-TRIAL-PRO-20110629.der</LicenseServerCertificate>
      <LicenseServerCredential> Path_to_certs/PRERELEASE-TRIAL-PRO-20110629.pfx
</LicenseServerCredential>
      <licenseServerCredentialPassword>abc01DeFghs=</licenseServerCredentialPassword>
      <PackagerCredential> Path_to_certs/PRERELEASE-TRIAL-PRO-20110629.pfx </packagerCredential>
      <PackagerCredentialPassword>abc01DeFghs=</ackagerCredentialPassword>
      <TransportCertificate> Path_to_certs/PRERELEASE-TRIAL-PRO-20110629.der </TransportCertificate>

      <CommonKey> Path_to_certs/commonkey.bin </CommonKey>
      <PolicyFile> Path_to_certs/rootPolicy2.pol </PolicyFile>
    <!--
      The time interval in milliseconds after which the DRM metadata will be re-sent to targets
      (such as the Origin Server)
    -->
      <RefreshInterval>15000</RefreshInterval>
    </FAXS3>
  </ContentProtection>
  <Input>
    <RTMP>
    </RTMP>
  </Input>
  <OutputPipeline>
    <Output>
    <!--
      The entries in this section represent output targets. Multiple output targets can be
      configured, and the output
      will be sent to all of these simultaneously. All output targets are optional, but if no output
      target is configured,
      a warning will be logged.
    -->
    <LocalOrigin>
    <!--
      Optional, defaults to {webroot}/containerID/streamID/. If specified, this is either an
      absolute path, or
```

```

    a path relative to the directory in which the Packager is started.
    Only one LocalOrigin may be specified, if more than one is configured, only the first
    will be used.
    -->
    <ContentPath>webroot/live/sample</ContentPath>
  </LocalOrigin>
  <ManifestPush>
    <UseSecurityToken>true</UseSecurityToken>
    <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
    <TargetURL></TargetURL>
    <MaxQueueSize>5</MaxQueueSize>
  </ManifestPush>
  <HttpPush>
    <UseSecurityToken>true</UseSecurityToken>
    <SecurityTokenKey>4ff4756ed68239d34d482dbc88819abc</SecurityTokenKey>
    <TargetURL></TargetURL>
    <MaxQueueSize>5</MaxQueueSize>
  </HttpPush>
</Output>
</OutputPipeline>
</Stream>

```

RTMP ingest for HLS

Primestime Live Packager supports RTMP 2 HLS workflow, which hitherto was achieved through JIT HLS at Origin. You can directly publish HLS streams from the packager.

This feature extends the functionality of the Primestime Live Packager to allow ingestion of RTMP streams, in addition to TS streams over multicast. It allows Live Packager to act as an RTMP server, to which RTMP streams can be published. All HDS output functionality available to streams ingested over TS-Multicast are available to streams ingested over RTMP. Conversion to HLS at the packager is not supported. The scope of this feature is restricted to RTMP publish workflows only.

Authentication between RTMP Publisher and Primestime Live Packager

Real-time messaging protocol (RTMP) supports high-performance transmission of audio, video, and data messages between an RTMP publisher/encoder and Primestime Live Packager. The RTMP publisher or encoder sends unencrypted data, including authentication information (such as name and password) over TCP connections to Live Packager.

Primestime Live Packager uses the following authentication plan to ingest RTMP streams from an RTMP publisher/encoder. Encoders should implement this plan for RTMP to be compatible with Primestime.



Note:

The plan fails if you use a different authentication mechanism.

1. Connect to Live Packager using the URL `rtmp://<packagerAddress>/<ContainerName>`
2. Live Packager, with authentication enabled, rejects the publisher's connection request and sends the following information in `rejectInfo`:

```
[code=403 need auth; authmod=adobe]
```
3. Code=403 indicates that Live Packager requires authentication. The RTMP publisher should use the value of the `authmod` parameter sent by Live Packager to establish a connection.

In this case, `authmod=adobe` so set `authmod=adobe` in the publisher

- Using the user-provided credentials, set the username parameter. Connect to Live Packager with the modified URL that contains the string

```
"?authmod={authmod_val}&user={username}"
[rtmp://<packagerAddress>/<ContainerName>?authmod=adobe&user=billjoy]
```

- Live Packager rejects the connection again and response within the `rejectinfo` parameter indicates that authentication information is required. The response has the format

```
[[authmod=adobe]:?reason=needauth&user=billjoy&salt=3e5a16bd&challenge=e0f1b9c5&opaque=5c9b1f0e]
```

- The RTMP publisher should regenerate the response using the parameters that Live Packager sent:
 - MD5 hash should be generated from a combination of username (user-provided credential), salt (present in the value of `rejectinfo` sent in the previous step) and password (user-provided credential)

```
[base64(md5(username + salt + password))]
```

- Generate a random number (encoded in base64) and store it as challenge.

- Create New MD5 hash using hash generated in first step, challenge sent by Live Packager and challenge generated in the previous step:

```
Response= [base64(md5(hashCalculatedInFirstStep + challengeSentByPackager +
challengeGenerated)))] (the strings should be appended in the same order)
```

RTMP Publisher should now connect to Live Packager with the new modified URL generated as shown:

```
rtmp://<packagerAddress>/<ContainerName>?authmod=adobe&user=billjoy&challenge=a2f1456b5&response=7e1f80342
aa19d567e1f80342aa19d56&opaque=5c9b1f0e
```

Opaque parameter has the same value as sent by Live Packager. Challenge is the string that was generated in step 2.

- Live Packager accepts the connection if the credentials are valid. If the connection is rejected, Live Packager sends one of these reason strings:
 - `[authmod=adobe]:?reason=authfailed` (Connection attempt failed due to invalid password)
 - `[authmod=adobe]:?reason=nosuchuser` (Connection attempt failed because the specified user doesn't exist)
 - `[authmod=adobe]:?reason=cannot_load_password_file` (Connection attempt failed because the password file could not be loaded)

Enabling RTMP authentication

To support RTMP authentication in Live Packager, first enable RTMP authentication in the Live Packager configuration. Thereafter, you can enable/disable authentication at the container level.

Primestime Live Packager implements a default authentication scheme. However, you can implement your custom logic and place the JAR file for the implementation in the root directory. For details, see [Adding custom authentication](#).

Enabling authentication at Live Packager

Authentication is enabled by default on the Live Packager. The following is a sample configuration within the `server.xml` file required to enable RTMP authentication in Live Packager.

```
LivePackager: <ROOT DIR>/conf/rtmp/server.xml:<Root Dir>
<Config>...
  <AuthPlugin>
    <ClassName>com.adobe.fms.util.authadaptor.AuthAdaptorPlugin</ClassName>
    <Init>
      <ConfRoot>./packager_conf</ConfRoot>
      <UserFile>users.dat</UserFile>
    </Init>
  </AuthPlugin>
</Config>

Primestime Streaming Services:
<ROOT DIR>/packager_conf/rtmp/server.xml:<Root Dir>
<Config>...
  <AuthPluginClassName>com.adobe.fms.util.authadaptor.AuthAdaptorPlugin</ClassName>
  <Init>
    <ConfRoot>./conf</ConfRoot>
    <UserFile>users.dat</UserFile>
  </Init>
</AuthPlugin>
</Config>
```

The following are the configuration elements for RTMP authentication:

Name	Description
AuthPlugin	Tag to be added to the <code>server.xml</code> file to enable authentication.
ClassName	Name of the main class of the plugin.
Init	Custom element where initialization configuration for the auth plugin can be specified. For the default implementation: <ul style="list-style-type: none"> • ConfRoot: Path to the configuration directory • UserFile: Name of the users file where user names and passwords are saved



Note: To disable authentication at the server level, remove the `<AuthPlugin>` tag in the `server.xml` file.

Enabling authentication at the container

To enable authentication for a particular container, call the `createRootModule` API with non-empty credentials. See Using JMX API for more details.

Authentication is disabled for the container if null/empty credentials are sent in the `createRootModule` API.

Configure user credentials

You can either use the standalone tool `usermgmt.jar` or JMX APIs to configure user credentials in Live Packager.

Using usermgmnt.jar to configure user credentials

If you use the default authentication scheme in LivePackager, add valid users for each new container in Live Packager to authenticate connections.

Use the configuration tool `usermgmnt.jar` in Live Packager to manage users. You can specify configuration parameters as command line options.

```
java -jar usermgmnt.jar -param_name param_value
```

The following table describes the parameters you can use:

Parameter Name	Description	Data Type	Type
add	Option to add a new user.	NA	Optional
remove	Option to remove a user.	NA	Optional
check	Option to check if a user exists or is valid.	NA	Optional
r	Option to replace an existing user (change password), valid with -add option.	NA	Optional
u	User name.	String	Mandatory
p	Password.	String	Mandatory parameter for all options other than -remove and -help
container_path	Absolute file-path to the container directory.	String	Mandatory
help	Prints usage and version information.	NA	Optional

The following code block depicts a sample usage of the tool:

```
Add user fms with password test:
>java -jar usermgmnt.jar -add -u fms -p test -container_path "./conf/containers/livepkgr"

Check user fms for password test:
>java -jar usermgmnt.jar -check -u fms -p test -container_path "./conf/containers/livepkgr"

Replace user fms (change password to test1):
>java -jar usermgmnt.jar -add -r -u fms -p test1 -container_path "./conf/containers/livepkgr"

Remove user fms:
>java -jar usermgmnt.jar -remove -u fms -container_path "./conf/containers/livepkgr"

Print usage:
>java -jar usermgmnt.jar -help
```

The tool creates/modifies the file `users.dat` at the location specified by the `container_path` param parameter.

The output file has the following format:

```
<userName>:<a 4 byte random salt>:<base64(MD5(userName+4 byte random salt+password))>
```



Note:

The symbol `+` denotes concatenation.

Using JMX API

You can configure users through the `createRootModule` JMX API for hosted use-cases.

The following is the signature of the API for Live Packager:

```
String createRootModule(String moduleID, String config, String credentials, boolean isEncoded)
```

The `isEncoded` parameter decides whether the credentials sent in the third parameter are already encoded.

If the value of the `isEncoded` parameter is true, credentials are expected to be encoded and sent in the following format:

```
<userName>:<a 4 byte random salt>:<base64(MD5(userName+4 byte random salt+password))>.
```

Multiple such user credentials are separated by a semicolon. Live Packager saves the credentials as is in the `users.dat` file.

The `createRootModule` function calls the `saveEncCredentials` API in the auth plugin.

If the value of the `isEncoded` parameter is false, the credentials parameter takes the credentials in the form:

```
<username1>:<password1>;<username2>:<password2>
```

All the credentials valid for the container are passed to the API. The user name and the corresponding passwords are separated by a colon. Multiple credentials are separated by semicolons. The `createRootModule` function calls the `saveRawCredentials` API in the auth plugin.

Adding custom authentication

Here are the steps to add a custom authentication plugin:

1. Implement the following interface with the custom authentication scheme (in function `authenticateRequest`):

```
public interface AuthAdaptor{
    public interface Status    {
        public boolean getStatus();
        public String getReason();
    }
}

/**
 * Hook to initialize the plug-in, passing the <init> element within
 * the AuthAdaptor config element or <tt>null</tt> if no initialization
 * configuration was provided.
 * This method is invoked during module initialization and is guaranteed
 * to have finished execution before any calls to {@link #authenticateRequest} are made.
 *
 * @param configXML The <init> child element of the plug-in config
 * element.
 * @param logger Logger to be used for logging
 */
public void initialize(Element configXML, Logger logger)
    throws Exception;

/**
 * Hook to check if authentication is to be done for the request, passing the <CmdType>
```

```

* This method is invoked during CONNECT.
*
* @param cmdType The <CmdType>. At present only authentication on CONNECT is supported.
*
* element.
* @param url Input URL
*/

public Status isAuthEnabled(CmdType cmdType, String url)
    throws Exception;

/**
* Hook to authenticate the request, passing the <CmdType>;
* This method is invoked during CONNECT.
*
* @param cmdType The <CmdType>. At present only authentication on CONNECT is supported.
*
* element.
* @param url Input URL
*/

public Status authenticateRequest(CmdType cmdType, String url)
    throws Exception;

/**
* Hook to save raw credentials.
* This method is invoked during container creation.
*
* @param filePath The absolute <directory path> of the out file. FileName is to be appended
by the implementor
* @param credentials Map of username and passwords to be saved.
*/

public Status saveRawCredentials(String filePath, Map<String, String> credentials)
    throws Exception;

/**
* Hook to save encoded credentials.
* This method is invoked during container creation.
*
* @param filePath The absolute <directory path> of the out file. FileName is to be appended
by the implementor
* @param credentials list of encoded credentials to be saved.
* The string would be assumed to be encoded and would be saved as it is.
*/

public Status saveEncCredentials(String filePath, List<String> encodedCredentials)
    throws Exception;
    enum CmdType    {
        CONNECT
    }
}

```

2. Place the JAR file that you build in <ROOT DIR>/conf/rtmp/lib for Live Packager and <ROOT DIR>/packager_conf/rtmp/lib for Streaming Server.
3. Add the followinwhereg configuration to the RTMP configuration file:

```

LivePackager:

<ROOT DIR>/conf/rtmp/server.xml:<Root Dir>
<Config>...
  <AuthPlugin>Primetime>
    <ClassName>com.adobe.fms.util.authadaptor.AuthAdaptorPlugin</ClassName>
  <Init>
    <ConfRoot>../packager_conf</ConfRoot>
    <UserFile>users.dat</UserFile>
  </InitAuthPlugin>

```

```

</Config>

Primetype Streaming Services:

<ROOT DIR>/packager_conf/rtmp/server.xml:<Root Dir>
<Config>...
  <AuthPlugin>
    <ClassName>com.adobe.fms.util.authadaptor.AuthAdaptorPlugin</ClassName>
    <Init>
      <ConfRoot>./conf</ConfRoot>
      <UserFile>users.dat</UserFile>
    </Init>
  </AuthPlugin>
</Config>

```

The following table describes the configurations elements for RTMP authentication:

Name	Description
AuthPlugin	Tag to be added to enable authentication.
ClassName	Name of the main class of the plugin.
Init	Custom element where you specify initialization configuration for the auth plugin. The following is the initialization configuration for the default implementation: <ul style="list-style-type: none"> • ConfRoot: Path to the configuration directory • UserFile: Name of the users file (where user names and passwords are saved)

4. Restart Live Packager.



Note:

If the custom authentication plugin doesn't use the default `users.dat` format (for on-premise cases), use a custom tool to save the credentials. In a hosted scenario, provide an implementation of `saveCredentials` in the auth plugin.

Performing administrator actions

You can use the JMX management functionalities provided by the packager to manage and monitor the packager.

You can also perform many administrator actions including:

- Loading newly added containers dynamically at runtime.
- Existing containers can be refreshed at runtime in order to account for the newly added child streams.

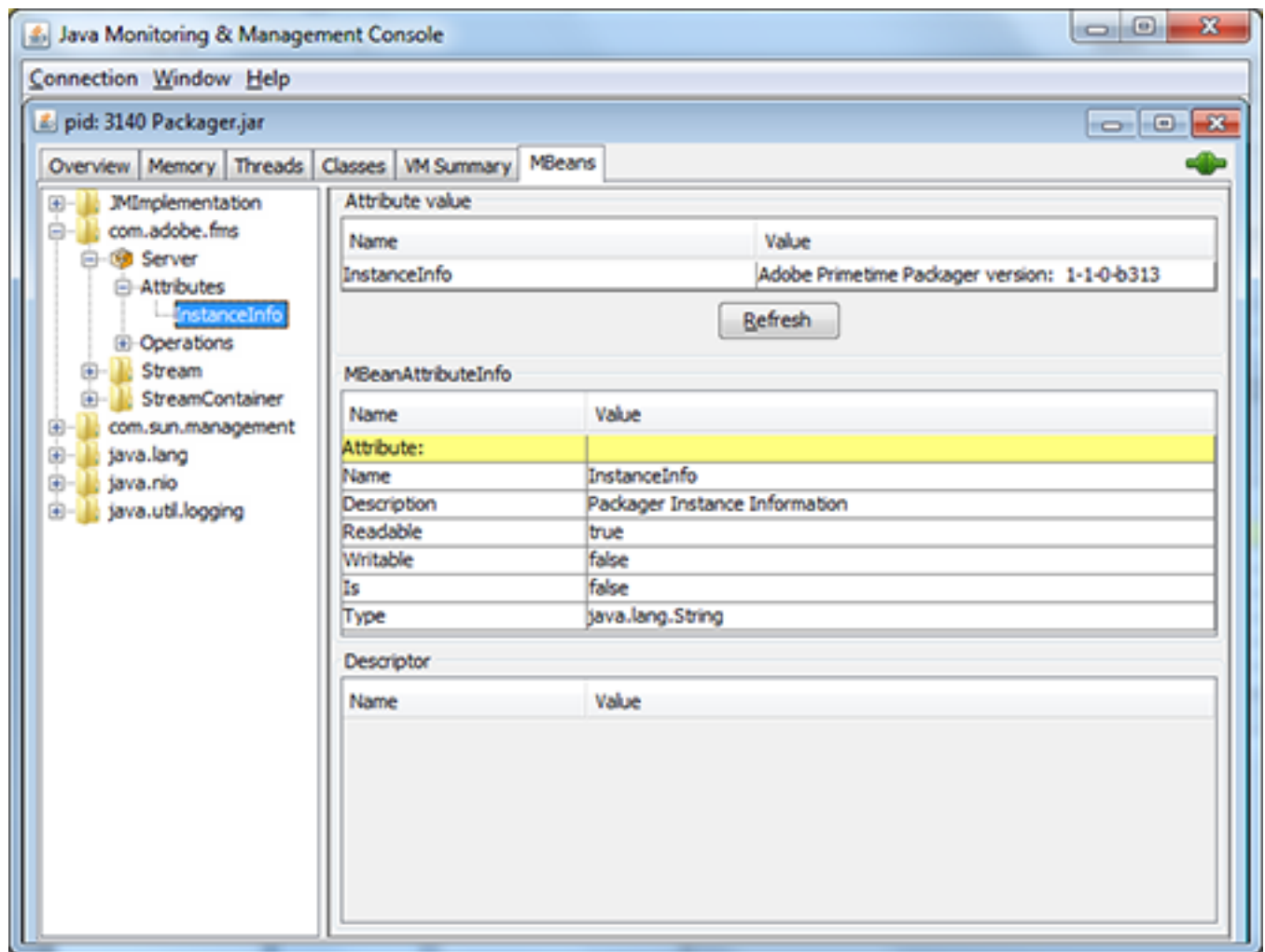
These administrator actions are exposed through the JMX management interfaces of the packager.



Note: The JMX port used is `Dcom.sun.management.jmxremote.port=56435`.

Getting the packager's version

In the JMX console, go to **com.adobe.fms > Server > Attributes > InstanceInfo** to view the packager's version.



Managing the stream container

Using the JMX Management console, you can perform a variety of operations.

Most of the JMX methods return a string that is a combination of a numeric status code and a readable message in the following format:

`StatusCode:Message`

The message part is optional and can be empty. However, the status code is always delimited by a colon. These codes represent various success or error cases. Here is a list of the Status Codes returned by various JMX methods:

Status Code	Description
0	Success
1	Unknown error
2	Method not implemented
3	Cannot create a module by given name as module by that name already exists

Status Code	Description
4	Cannot delete/update module by given name as it does not exist
5	Cannot delete module as it's running
6	Operation failed due to invalid config
7	Invalid module name provided as parameter
8	Returned when refresh is called on a container and some child modules fail to load

1. In the JMX console, go to **StreamContainer > Lifecycle > Operations** and perform the following operations:

Option	Description
Refresh()	<p>Refreshes the live packager thus resulting in loading any new descendant Stream Container modules that have been added to the system after this module was initialized.</p> <p>This module, and any descendants that are already loaded and initialized are not affected apart from new child modules being optionally connected to their parent module. For a Stream Module, this operation does nothing, as there are no child modules configured.</p> <ul style="list-style-type: none"> • Valid operation level: Server • Parameters: NIL • Returns: StatusCode:Message format
Shutdown()	<p>Shuts down the module and all descendant modules.</p> <ul style="list-style-type: none"> • Valid operation level: Server • Parameters: NIL • Returns: StatusCode:Message format
Reset()	<p>This operation resets a stream, removes all content that was generated, and then reloads itself. Note that the reset action can be performed at both container-level and stream-level.</p> <ul style="list-style-type: none"> • Valid operation level: Stream module • Parameters: NIL • Returns: StatusCode:Message format
setConfig(String xmlConfig)	<p>This operation is exposed to update the existing XML configuration for the stream module. It takes the new XML config as the parameter. This operation does not reload the module. Configuration changes are only reflected when module is reset using the reset() operation. Before replacing the configuration file on the disk, the given xml is validated to ensure that it is a well-formed xml.</p>

Option**Description**

Note: Attribute validation is not performed on the XML file.

- Valid operation level: Stream module
- Parameters: XML configuration file
- Returns: IModule.StatusCode as String
- Message on successful operation: Combination of a numeric status code from the list of status codes mentioned above and the corresponding readable message (optional) in the format StatusCode:Message.
- Errors
 - String "error.failed" for an unsuccessful operation
 - String "error.not_implemented" if the operation is invoked on a module which does not support it.

String createChildModule(String moduleName, String moduleType, String moduleConfig)

This operation is exposed to add an XML configuration for a new child module to the container module. If this module does not support child modules, or if another child module with the same name exists, an error is thrown. This method does not load the new module. To load the new module, the refresh() method must be called.

- Valid operation level: StreamContainerModule
- Parameters:
 - Module name: name of the child module.
 - Module type: Type of the module. For Origin's StreamContainer, it can be stream or vod. For Packager's StreamContainer, this value is ignored.
 - Module configuration file
- Returns: IModule.StatusCode as String
- Message on successful operation: Combination of a numeric status code from the list of status codes mentioned above and the corresponding readable message (optional) in the format StatusCode:Message.
- Error:
 - In case of an error, string "error.failed."
 - If it is invoked on a module which does not support this operation (for example, Stream Module), then string, "error.not_implemented."

String removeChildModule(String moduleName, String moduleType)

This operation is exposed to remove the configuration of a child module.

- Valid operation level: StreamContainerModule
- Parameters:
 - Module name: Name of the module that you want to remove.

Option**Description**

- **Module Type:** Type of the module. For Origin's StreamContainer, it can be stream or vod. For Packager's StreamContainer, this value is ignored.
- **Returns:** IModule.StatusCode as a string
- **Message on successful operation:** Combination of a numeric status code from the list of status codes mentioned above and the corresponding readable message (optional) in the format StatusCode:Message.
- **ErrorS**
 - In case of an error, string "error.failed."
 - If it is invoked on a module which does not support this operation (for example, Stream Module), then string, "error.not_implemented."

startingest

Signals the packaging stream container module to start ingesting. Returns the response status code and status message as a String. If the stream is already ingesting, no change is done to the state of the stream and an error message is returned stating that stream is already ingesting. If the previous state of stream has end of presentation marker then ingest is started only if resumIngestForEndedStream argument is true. If ingest is not started and an error message is returned indicating that stream has ended.

Parameters:

- **resumIngestForEndedStream:** If set as true, stream will start ingesting even if previous state contains end of presentation marker. If set as false, then the stream will not start ingestion if previous state has end of presentation marker.

signalEndOfStream

Operation exposed to signal end of live stream. The module stops receiving input and an end of stream marker is inserted in the manifest(f4m/m3u8). If stream is not ingesting an error message is returned.

- **Returns:** String in the status code: message format.

insertSCTE35CueNow

Injects a Splice Out SCTE35 cue into the stream. Used for debugging purposes.

- **Returns:** String in the status code: message format.

The following table lists the MBean attributes:

Name	Description	Type
ModuleID	Stream Module Name	String
StreamID	StreamID	String

Ingesting	A boolean denoting whether the stream is currently ingesting	Boolean
-----------	--	---------

2. In the JMX console, go to **StreamContainerModule > Lifecycle > Operations** and perform the following operations:

Option

Description

startIngest

Signals the packaging stream container module to start ingesting. Returns the response status code and status message as a String. If the stream is already ingesting, no change is done to the state of the stream and an error message is returned stating that stream is already ingesting. If the previous state of stream has end of presentation marker then ingest is started only if `resumeIngestForEndedStream` argument is true. If ingest is not started and an error message is returned indicating that stream has ended.

Parameters:

- `resumeIngestForEndedStream`: If set as true, stream will start ingesting even if previous state contains end of presentation marker. If set as false, then the stream will not start ingestion if previous state has end of presentation marker.

signalEndOfStream

Operation exposed to signal end of live stream. The module stops receiving input and an end of stream marker is inserted in the manifest(f4m/m3u8). If stream is not ingesting an error message is returned.

- Returns: String in the status code: message format.

insertSCTE35CueNow

Injects a Splice Out SCTE35 cue into the stream. Used for debugging purposes.

- Returns: String in the status code: message format.

Managing the packager

In the JMX console, go to **Server > Operations** and perform the following operations:

Option

Description

**String
createRootModule(String
moduleID, String config, String
credentials, boolean
isEncoded)**

This operation is exposed for the PackagerMBean to let you create the configuration files for new Root Containers for the Packager. Specify an ID and XML configuration file for the new container. This method checks for active container modules. If an active container has the ID that is passed into this function, the operation returns an error string. The `isEncoded` parameter decides whether the credentials sent in the third parameter are already encoded. This JMX operation only creates the configuration file for the container module. However, it does not initialize the container. The container is initialized when the Packager is refreshed using the `refresh()` API.

Option**Description**

- Returns: `IModule.StatusCode` as a string
- Message on successful operation: Combination of a numeric status code from the list of status codes mentioned above and the corresponding readable message (optional) in the format `StatusCode:Message`
- Error:
 - In case of an error, string "error.failed."
 - If it is invoked on a module which does not support this operation (for example, Stream Module), then string, "error.not_implemented."

**String
removeRootModule(String
moduleID)**

This operation is exposed for the `PackagerMBean` to enable you to remove the config files for the root container modules of the Packager. You need to specify the module ID for the container to be removed. If the container to be removed is running or does not exist, this operation returns an error string. The container must be stopped using the container's `shutdown()` operation before it can be removed. This operation does not perform any cleanup. To remove a container containing multiple child modules, it is advisable to invoke `cleanup/shutdown/remove` on the child modules (in that particular order) before removing the container.

- Returns: `IModule.StatusCode` as a string
- Message on successful operation: Combination of a numeric status code from the list of status codes mentioned above and the corresponding readable message (optional) in the format `StatusCode:Message`
- Error:
 - In case of an error, string "error.failed."
 - If it is invoked on a module which does not support this operation (for example, Stream Module), then string, "error.not_implemented."

Refresh()

Refreshes the packager, which results in loading any new descendant Stream Container modules that have been added to the system after this module was initialized. This module, and any descendants that are already loaded and initialized are not effected apart from the new child modules being optionally wired to their parent module.

Packager-specific APIs

The following APIs are applicable for the Packager only:

API	Description
<code>insertSCTE35CueNow</code>	This operation injects an SCTE35 cue into the stream.
<code>signalEndOfStream</code>	This operation is exposed to signal the end of live stream. The module stops receiving input and an end of stream marker is inserted in the bootstrap.

Ad cues

Simple mode cues from The Primetime DPI specification Primetime Digital Program Insertion Signaling Specification are supported in RTMP streams.

Refer to the DPI specification Primetime Digital Program Insertion Signaling Specification for details regarding the structure of simple mode cues in AMF messages.

Loading new containers and streams

1. Add new containers or new streams to existing container in the packager's `conf` folder.
2. Run `<JAVA_HOME>/bin/jconsole` from the command line (need Oracle JDK 5.0+).
A Connection dialog box appears.
3. Double click the packager.jar from the **Local Process** pane.
Optionally, if you are running the Live Packager on a headless machine, click **Remote Process** and enter the hostname and login details to connect to the remote machine.
4. Select the **MBeans** tab from the **Java Monitoring & Management Console** applet.
5. Expand **com.adobe.fms** and select **Operations** under **Server**.
6. Click **refresh** to display the newly added containers and streams.
7. Expand **StreamContainer** to see all the available containers.
8. Dynamically load each desired module ID by clicking on its **refresh** button.
In addition to dynamically loading the newly added containers and streams, jconsole also allows you to shut down a specific container or stream.
9. Select the shutdown operation from the Operations node and click the shutdown button.
Information about each loaded stream is exposed through an MBean interface using the namespace **com.adobe.fms**. All MBeans found in the namespace **com.adobe.fms** supports logging.

Load on startup

The Stream Module provides the configuration parameter LoadOnStartup, which can be set to true or false in stream.xml.

If this configuration parameter is set to false, the Stream Module does not ingest the input immediately when the module is started due to any reason, for example Packager start, parent Container start/refresh, and module reset. The Stream Module ingests input only when startIngest() JMX API is explicitly invoked on the module. If this configuration is set to True, the Stream Modules ingest input as soon as they are started. By default this configuration parameter is set to True.

Viewing stream statistics

Information about each loaded stream is exposed through an MBean interface.

In JConsole, click **com.adobe.fms > Stream > <Stream_Name> > com.adobe.fms > StreamContainer > <Container_Name> > stats**.

For example, click **com.adobe.fms > Stream > _default_ > com.adobe.fms > StreamContainer > _default_ > stats** to show the statistics for the stream named `_default_` available under the container named `_default_`.

Stream statistics for an output pipeline

The following table lists some of the statistics available for a given output pipeline:

Name	Description
mcastBytesReceived	The Number of bytes received from the multicast socket.

Name	Description
mcastReadQueue	The current size in bytes of the queue of packets received from the multicast socket.
bytesDropped	The Number of bytes which have been dropped when the OutputQueueMaxSize has been exceeded.
bytesReceived	Number of bytes processed by the transport stream parser.
bytesSkipped	Number of bytes skipped while seeking the transport stream sync byte.
tsPacketsReceived	The number of transport stream packets received.
pesPacketsReceived	The number of Packetized Elementary Stream packets received by HDS pipeline.
pesPacketsDropped	The number of Packetized Elementary Stream packets dropped by the transport stream parser, due to Continuity Counter field anomalies.
h264NALUsReceived	The number of H.264 NALUs received by HDS pipeline.
aacFramesReceived	The number of AAC audio frames received by HDS pipeline.
mp3FramesReceived	The number of mp3 audio frames received by HDS pipeline.
avcPacketsReceived	The number of AVC PES packets received by HLS pipeline.
fragmentsDropped	The number of fragments that were lost or unprocessed and resulted in a gap.
mcastSourceAddressChanges	Increments each time a multicast TS packet is received from a source address (multicast address and port) different to that of the previous packet.
avBufferDelta	The instantaneous maximum time-stamp difference (in ms) between the oldest and newest packets in the output AV buffer.
avBufferDropped	The number of packets dropped from the output AV buffer due to being older than the most recent in the buffer.
lastPTSInput	The most recently received transport stream 33-bit time-stamp.
rolloverBase	The current rollover base used for extending the transport stream of 33-bit time-stamps to 64-bits.
lastHLSPTSOutput	The time-stamp of the most recently received elementary stream packet for HLS.
lastFragmentName	The numeric part of the most recent fragment file name.
fragmentBytes	The number of bytes transferred over an output pipeline for fragment data.
fragmentPushCount	The number of fragments transferred over an output pipeline.
manifestBytes	The number of bytes transferred over an output pipeline for manifest data.
manifestPushCount	The number of manifests transferred over an output pipeline.
bootstrapBytes	The number of bytes transferred over an output pipeline for the bootstrap data.
m3u8Bytes	The number of bytes transferred over an output pipeline for the m3u8 data.
drmMetadataBytes	The number of bytes transferred over an output pipeline for DRM metadata.
fragmentDispatchDropped	The number of failed fragment dispatches for the configured output pipelines. This number is updated when the output queue exceeds the maxQueueSize configuration and when the job was subsequently dropped.

Name	Description
droppedOutOfOrderFrames	The number of audio or video frames dropped because of the backward timestamps. Note that this count is over and above the count represented by <code>avBufferDropped</code> . A positive value here is an indication that the time lag between audio-video tracks in the input is more than the <code>MediaBufferLength</code> configuration.
fragmentPushFailCount	The number of failed fragment writes to the remote or local Origin.
videoBytesReceived	The number of video bytes received for this stream over the RTMP connection.
videoPacketsReceived	The number of video packets (TCMessages) received for this stream over the RTMP connection.
audioBytesReceived	The number of audio bytes received for this stream over the RTMP connection.
audioPacketsReceived	The number of audio packets (TCMessages) received for this stream over the RTMP connection.
dataBytesReceived	The number of AMF bytes received for this stream over the RTMP connection.
dataPacketsReceived	The number of AMF packets (TCMessages) received for this stream over the RTMP connection.
otherBytesReceived	The number of bytes other than Audio/Video/Data received for this stream over the RTMP connection.
otherPacketsReceived	The number of packets (TCMessages) other than Audio/Video/Data received for this stream over the RTMP connection.

Logging

The packager uses the `java.util.logging` API for logging.

You can configure logging using the `logging.properties` located in the install directory. The packager scripts `packager_start.bat` (Windows) and `packager_start.sh` (Linux) loads the `logging.properties` file from the install directory. The default log file output will be in the `logs` directory under the install directory.

Logging namespaces

Name	Description
<code>com.adobe.fms</code>	The root namespace for all packager logging.
<code>com.adobe.fms.http</code>	Logging for the built in http server. Logs all requests and responses.
<code>com.adobe.fms.Stream</code>	Logging for configured streams. The stream name is appended to the namespace, allowing log levels and filtering to be set per stream.

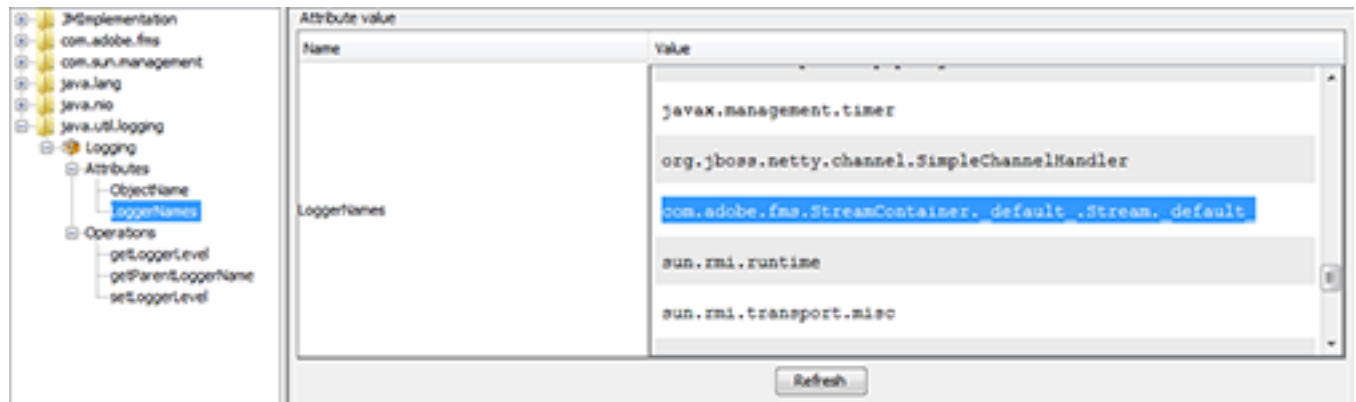
Modifying the log level for a stream

The default packager log level will be INFO. This means INFO, WARNING, and SEVERE are all captured in the packager.log file. It is a two-step process to modify the log level for a stream.

On a per-stream basis, log levels may be modified at runtime through JMX. The default log level is INFO, and it's fileHandler log level is FINEST. The output is placed in the same "logs" directory as the packager.log file, but in a separate log file based upon the name of the stream context. This is useful to diagnose issues on a per-stream basis, but it is not recommended for a production system as it may adversely affect performance.

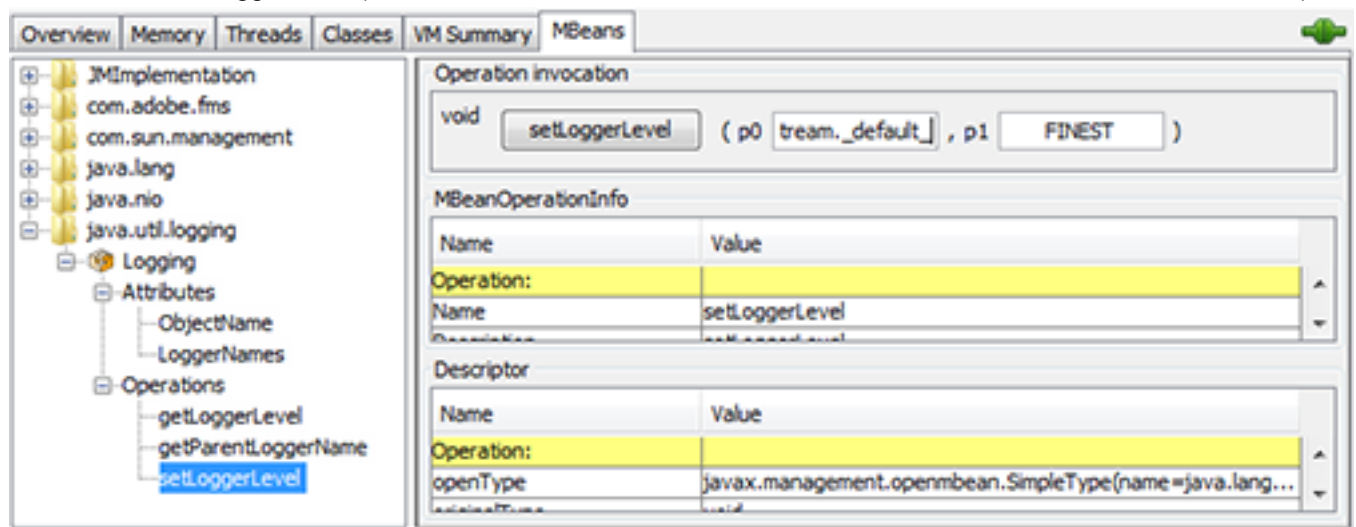
1. Go to **java.util.logging > Logging > Attributes > LoggerNames**.
2. Identify the stream for which you want to modify the log level.

For instance, `com.adobe.fms.StreamContainer._default_.Stream._default_`



3. Go to **java.util.logging > Logging > Operations > setLoggerLevel**.
4. Click the **setLoggerLevel** button on the java.util.logging MBean, with the logger name identified from step 1, and specify which log level you want to set.

For instance, `setLoggerLevel ("com.adobe.fms.StreamContainer._default_.Stream._default_", FINEST)`



Each individual stream will log to its own particular log file when a debug log level is set. The location of the log file will be found in the logs/stream folder. The naming convention for each individual stream will be the `<container_name>.<stream_name>.log`. The log file rolls over after 128 Mb has been logged.

Appendixes

Supported SCTE-35 splice_insert() messages

Primetime has the following assumptions/restrictions when it comes to supported SCTE-35 `splice_insert()` messages:

- The SCTE35 message must be `splice_insert()` - The value of `splice_command_type` in `splice_info_section()` message will be 0x5.
- The `out_of_network_indicator` - The Primetime Packager will only process messages where this flag is 1, which indicates a Splice Out point. Any SCTE-35 splice-in messages (`out-of-network = 0`) are ignored.
- The `duration_flag` must be set. If a `break_duration()` is not defined, the message will be ignored.
- The value of `auto_return` flag in a `break_duration()` is ignored. Primetime assumes that an auto-return from the insertion stream content to the network feed should happen at the end of the break duration.
- Messages having the `splice_immediate_flag = 1` and no defined `splice_time()` are ignored. Primetime currently requires an explicit `splice_time()` to be defined. The value of `program_splice_flag` must be 1.
- `splice_event_cancel_indicator` - The `splice_insert()` messages with `splice_event_cancel_indicator` set to true are currently ignored.

#EXT-X-CUE tag

Ad avails are signaled in the M3U8 file so that the Primetime HLS client can use this information to insert ad content.

For every SCTE-35 SpliceOut message, a custom #EXT-X-CUE tag is added to the media segment entry that starts with the Splice-Out point. The HLS segments are aligned with each Splice-Out and Splice-In point. This facilitates ad insertion just by replacing the segment entries of the main content in the M3U8 file. The following table describes the #EXT-X-CUE tag:

Attribute	Type	Required	Description
TYPE	enumerated-string	Required	The event type which must be SpliceOut
ID	enumerated-string	Required	A unique identifier for this event within the context of the program stream.
TIME	Number	Required	The stream's presentation time in fractional seconds at which point the splice out should occur.
DURATION	Number	optional	The splice duration in fractional seconds. If duration is not specified then this attribute is not present.

The following is a sample M3U8 file containing the #EXT-X-CUE tag with a single ad break of known duration:

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:44
#EXTINF:9.9,http://server-host/path/file44.ts
#EXTINF:4.2,http://server-host/path/file45.ts
#EXT-X-CUE:TYPE=SpliceOut,ID=1,DURATION=60.0,TIME=266.198
#EXTINF:5.8,http://server-host/path/file46.ts
#EXT-X-CUE-CONT:ID=1,AVAIL-DUR-ELAPSED=5.8
#EXTINF:9.9,http://server-host/path/file46.ts...
```

The following is a sample M3U8 file containing the #EXT-X-CUE tag with a single ad break, where the duration is not known when the break begins:

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:44
#EXTINF:9.9,http://server-host/path/file44.ts
#EXTINF:4.2,http://server-host/path/file45.ts
#EXT-X-CUE:TYPE=SpliceOut,ID=1,TIME=266.198
#EXTINF:5.8,
#EXT-X-CUE-CONT:ID=1,AVAIL-DUR-ELAPSED=5.8
http://server-host/path/file46.ts
#EXTINF:9.9,
#EXT-X-CUE-CONT:ID=1,AVAIL-DUR-ELAPSED=15.7
http://server-host/path/file47.ts...
#EXTINF:9.9,http://server-host/path/file56.ts
#EXTINF:4.2,http://server-host/path/file57.ts
#EXTINF:9.9,http://server-host/path/file58.ts
```

#EXT-X-CUE-CONT tag

Media segments after the first segment in the AD avail are marked with the EXT-X-CUE-CONT tag.

The media segments that lie between the Splice Out point and Splice In point represent an AD avail. Splice In point is determined by adding splice duration to the Splice Out point or by Splice Time attribute of the SCTE Splice In Message.

The following are details of the #EXT-X-CUE-CONT tag:

Attribute	Type	Required	Description
ID	enumerated-string	Yes	A unique identifier for this event within the context of the program stream.
AVAIL-DUR-ELAPSED	Number	Yes	Number of seconds elapsed since the start of AD Avail(Splice Out Point)

All the segments in a given AD avail have the same value for ID attribute of #EXT-X-CUE-CONT tag as that of ID attribute of #EXT-X-CUE tag.

The following is an excerpt of a sample M3U8 containing the #EXT-X-CUE tag:

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:44
#EXTINF:9.9,http://server-host/path/file44.ts
#EXTINF:4.2,http://server-host/path/file45.ts
#EXT-X-CUE:TYPE=SpliceOut,ID=1,DURATION=30.0,TIME=266.198
#EXTINF:10,http://server-host/path/file46.ts
#EXT-X-CUE-CONT:ID=1,AVAIL-DUR-ELAPSED=10
#EXTINF:10,http://server-host/path/file47.ts
#EXT-X-CUE-CONT:ID=1,AVAIL-DUR-ELAPSED=20
#EXTINF:10,http://server-host/path/file48.ts
#EXTINF:10,http://server-host/path/file49.ts...
```

In the sample file, segments file46.ts, file47.ts and file48.ts are part of a 30 second AD avail. The first segment file46.ts is marked with EXT-X-CUE tag and rest of the two segments are marked with EXT-X-CUE-CONT tags.

HLS manifest custom tag for synchronization

The HLS Manifest (M3U8) contains a custom tag `#EXT-X-MEDIA-TIME`. The value of the tag is set to the stream time at the start of the TS segment. This time is the same as the presentation time of the first sample in the segment in fractional seconds.

`#EXT-X-MEDIA-TIME` occurs in M3U8 at:

- First segment in the M3U8 sliding window
- Each segment following the `#EXT-X-DISCONTINUITY` tag

This tag fills the gap in the HLS specification that it is not possible to derive timestamps just from M3U8 and without parsing the TS segments. It can be used to synchronize across different renditions during bitrate switch-over or failover. Synchronization is needed for example to insert pre-roll advertisements in a live stream having multiple renditions.

The following is a sample m3u8 file that contains an `#EXT-X-MEDIA-TIME` tag:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:6
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-MEDIA-TIME:0.0
#EXTINF:4.004,h.mp4..0_4004.ts?trackid=1,2
#EXTINF:4.004,h.mp4..4004_8008.ts?trackid=1,2....
```

Copyright

© 2014 Adobe Systems Incorporated. All rights reserved.

Adobe Primetime Offline Packager Getting Started Guide, Version 1.3 Adobe Primetime Live Packager Getting Started Guide, Version 1.3

Adobe and the Adobe logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

All other trademarks are the property of their respective owners.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.