



**Adobe® Primetime**  
**PSDK 1.3 for Android Programmer's Guide**

# Contents

<b>PSDK 1.3 for Android Programmer's Guide</b> .....	<b>5</b>
Overview and prerequisites.....	5
Overview of the Player SDK.....	5
Considerations.....	5
Requirements.....	6
Best practices.....	7
New features for 1.3.....	7
SDK changes for 1.3.....	8
Migrate from PSDK 1.2 to 1.3.....	8
Update ad-playback behavior for PSDK 1.3.....	9
Create a video player.....	9
Work with cookies.....	9
Set up the MediaPlayer.....	10
Listen for Primetime Player events.....	10
Set up error handling.....	14
Configure the player user interface.....	15
Initialize the MediaPlayer for a specific video.....	24
Monitor quality of service statistics.....	27
HTTP 302 redirect optimization.....	29
Work with MediaPlayer objects.....	29
About the MediaPlayerItem class.....	29
Lifecycle and states of the MediaPlayer object.....	30
MediaPlayer methods for accessing MediaResource information.....	32
Reuse or remove a MediaPlayer instance.....	33
Inspect the playback timeline.....	34
Configure buffering.....	35
Include advertising.....	36
Advertising requirements.....	36
About ad insertion.....	36
Playback behavior with ads: Default and customized.....	38

Customize playback with ads.....	39
Metadata for ad insertion.....	44
Clickable ads.....	47
Repackage third-party ads.....	50
Configure and use custom HLS tags.....	50
<b>Add custom ad markers.....</b>	<b>55</b>
TimeRange class.....	55
MediaPlayer and MediaResource classes.....	56
TimeRangeCollection class.....	56
Place TimeRange ad markers on the timeline.....	56
Control the seek-over-ads behavior.....	57
<b>Customize opportunity detectors and content resolvers.....</b>	<b>58</b>
About opportunity detectors and content resolvers.....	58
Implement a custom opportunity detector.....	59
Implement a custom content resolver.....	60
<b>Set up closed captioning.....</b>	<b>63</b>
Control closed-caption visibility.....	64
Control closed-caption styling.....	65
<b>Set up alternate audio.....</b>	<b>69</b>
Alternate audio tracks in the playlist.....	69
Access alternate audio tracks.....	70
<b>Content security using DRM.....</b>	<b>70</b>
DRM interface overview.....	70
DRM authentication before playback.....	71
DRM authentication during playback.....	74
<b>Use the notification system.....</b>	<b>75</b>
Notification content.....	75
Set up your notification system.....	76
Add real-time logging and debugging.....	76
Retrieve ID3 metadata.....	77
<b>Playback and failover.....</b>	<b>78</b>
Media playback and failover.....	78
Advertising insertion and failover for VOD.....	79
Get information about profile switch .....	81

Primestime player classes summary.....	81
Mediacore classes.....	81
Info classes.....	82
Logging classes.....	83
Metadata classes.....	83
Notification classes.....	83
QoS and QoS metrics classes.....	84
Timeline classes .....	85
Timeline advertising classes.....	85
Utility classes.....	86
Notification codes.....	87
ERROR notification codes.....	87
WARNING notification codes.....	90
INFO notification codes.....	91
Details for the NATIVE_ERROR notification.....	93
Glossary.....	101
A.....	101
C.....	101
D.....	101
F.....	102
H.....	102
I.....	102
K.....	102
L.....	103
M.....	103
N.....	103
P.....	103
Q.....	104
R.....	104
S.....	104
T.....	104
V.....	104
Copyright.....	104

# PSDK 1.3 for Android Programmer's Guide

## Overview and prerequisites

The Adobe Primetime Software Development Kit (PSDK) client for Android is a toolkit that provides you with the means to add advanced video playback functionality to your applications. This PSDK also supports ad insertion, content protection, and analytics.

The PSDK for Android is implemented entirely in Java. It includes an API and sample code to help you create an advanced media player and integrate Primetime features into the player.

This guide assumes that you understand developing applications using Java. It walks you through the steps required to create a video player in Java on Android devices using the Primetime Player SDK.

## Overview of the Player SDK

The Primetime Player SDK for Android includes a variety of features.

The following list describes some of the PSDK's functionality:

- VOD and live/linear streaming
- Support for full-event replay
- Seamless ad insertion and tracking through VAST/VMAP
- Access to digital rights management (DRM)-related services
- Playback of HLS streams unencrypted or with Protected HTTP Live Streaming (PHLS), AES128, or Adobe Access protection
- Client tracking and QoS measurement and reporting
- Notifications that enable the PSDK and your application to communicate asynchronously about the status of videos, advertisements, and other elements, and also that log activity
- Management of the playback window, including methods that play, stop, pause, seek, and retrieve the playhead position
- Closed captioning (608, 708, WebVTT) and alternate forms of audio for increased accessibility
- Control for text styling in captions
- DVR capability, fast forward and fast rewind (trick-play mode), and slow motion
- Bit-rate controls and adaptive bit-rate logic
- Custom cue tags for ads
- Failover
- Debug logging
- Adjustable playback buffers
- 302 redirect optimization
- Customizable content/ad insertion workflow including blackout signaling
- Customizable advertising playback policies
- Fragment duration, size and time-to-download tracking support

## Considerations

Keep this information in mind while using the PSDK for Android.

- Playback is supported only for HTTP Live Streaming (HLS) content.
- Main video content can be either multiplexed (video and audio streams are in the same rendition) or unmultiplexed (video and audio streams are in separate renditions).
- The PSDK API is implemented in Java .
- Video playback requires the Adobe Video Engine (AVE). This affects how and when media resources can be accessed:
  - Closed captioning is supported to the extent provided by AVE.
  - Depending on encoder precision, the actual encoded media duration may differ from the durations recorded in the stream resource manifest.

There is no reliable way to resynchronize between the ideal virtual timeline and actual playout timeline. Progress tracking of the stream playback for ad management and Video Analytics must use the actual playout time, so reporting and user interface behavior might not precisely track the media and advertisement content.

- The incoming user agent for all HTTP requests from the Android PSDK contains the following string pattern:

```
"Adobe Primetime/psdk_version + originalUserAgent"
```

## Requirements

The Android player requires specific properties for media content, manifest content, DRM, and software versions.

### System and software requirements

Confirm that your hardware, operating system, and application versions are sufficient for this version of the PSDK.

Operating system	Android 2.3 or later (3.1 is not officially supported)
CPU	1 GHz Single Core or equivalent
RAM	256 MB
GPU	Hardware GPU required for playback

### Content and manifest requirements

Check the restrictions and requirements for streams and playlist/manifests, including DRM encryption keys.

Content segment duration	The duration of any given segment must not exceed the target duration specified in the manifest file.
DRM - Adobe Access	If the DRM-protected stream is multiple bit rate (MBR), the DRM encryption key used for the MBR should be the same as the key used in all the bit-rate streams.
Ad variant manifests	Ad variant manifests must have the same bit-rate renditions as that of the main content.

### #EXT-X-VERSION requirements

The version of #EXT-X-VERSION in the .m3u8 file affects what features are available to your application and what EXT tags are valid in your playlist/manifest.

About the #EXT-X-VERSION tag, which specifies the HLS protocol version:

- The version must match the features and attributes contained within the HLS playlist; otherwise, playback errors might occur. Refer to the most recent version of the [HTTP Live Streaming specification](#).

- If the tag is not included in the master or media playlists or if no version is specified, then version 1 is used by default and any content that does not comply with version 1 will not play.
- We recommend the use of at least version 2 for playback in PSDK-based clients. Clients and servers must implement the versions as follows for various features:

Use at least this version	To use these features
EXT-X-VERSION:2	The IV attribute of the EXT-X-KEY tag
EXT-X-VERSION:3	<ul style="list-style-type: none"> <li>• Floating-point EXTINF duration values</li> </ul> <p>The duration tags (#EXTINF:&lt;duration&gt;,&lt;title&gt;) in version 2 were rounded to integer values. Version 3 and above require durations to be exact in floating point.</p> <ul style="list-style-type: none"> <li>• PSDK features such as Ad Insertion and seamless ABR</li> </ul>
EXT-X-VERSION:4	<ul style="list-style-type: none"> <li>• The EXT-X-BYTERANGE tag</li> <li>• The EXT-X-I-FRAME-STREAM-INF tag</li> <li>• The EXT-X-I-FRAMES-ONLY tag</li> <li>• The EXT-X-MEDIA tag</li> <li>• The AUDIO and VIDEO attributes of the EXT-X-STREAM-INF tag</li> <li>• PSDK alternate audio</li> </ul>

## Best practices

These are recommended practices for the Adobe PSDK for Android .

- Use HLS Version 3.0 or above for program content.

## New features for 1.3

Summary of product enhancements for the 1.3 Android PSDK.

### • Blackout signaling

The opportunity resolver workflow ( [PlacementOpportunityDetector](#) ) is a component of the PSDK used to detect custom tags in the stream and create placement opportunities. These placement opportunities are then sent to the content resolver ( [ContentResolver](#) ) to customize the content/ad insertion workflow based on the placement opportunity properties and metadata.

The PSDK includes a default opportunity resolver that understands default ad cues, and a default ad resolver that provides ads to be inserted based on the ad metadata provided in the player item. You can replace these default resolvers to customize the workflow of custom tag detection and ad/content insertion to implement different scenarios such as recognizing custom tags for ad insertion, implementing a custom ad provider, or providing blackouts. See [Customize opportunity and content resolvers](#).

### • Greater Ad Insertion Flexibility

You can now customize your viewers' advertising experience. Set custom ad policies for unique viewer situations (play, seek, fast forward, rewind, resume) and business rules. See [Playback behavior and optional customizations](#).

### • Enhanced video analytics with heartbeat support

Turn-key integration with the Adobe Analytics and Primetime player monitoring through the Video heartbeat library. Simply provide your SiteCatalyst tracking server, report suite ID, and configuration parameters to have a fully functional video implementation. Analytics about video viewing sessions can be viewed in real-time inside Primetime player monitoring and is fully integrated into your Adobe Analytics report suite.

See [Use Video Analytics in a PSDK-based player](#).

#### • Enhanced FF and RW (trick play)

Rewind and Fast Forward your video streams just like your favorite DVD player or DVR. Primetime players now support the ability to play back content at speeds 2x - 32x.



**Note:** Live DVR use cases are not fully supported at this time.

See [Use trick play mode \(FF and REW\) and slow motion](#).

#### • New font type sets for closed captioning

Adobe now bundles CVAA-compliant fonts with our Android PSDK. This change was made with accessibility in mind and will ensure that there is a consistent captioning experience across all Android devices. See [Control closed-caption styling](#).

## SDK changes for 1.3

Several API interfaces have changed for the 1.3 PSDK for Android .

Element	Description
Method changed: • <a href="#">MediaPlayerItemLoader</a>	The first parameter of MediaPlayerItemLoader now has the same context as that of the first parameter of the VideoEngineAdapter.
<a href="#">DefaultMediaPlayer</a> new methods: • <a href="#">seekToLocalTime</a> • <a href="#">convertToLocalTime</a> • <a href="#">getLocalTime</a>	
New classes: • <a href="#">DefaultAdPolicySelector</a> • <a href="#">AdvertisingMetadata</a> • <a href="#">AdPolicyInfo</a>	Supports customizable ad playback behavior.
Renamed classes: • AdBreakPolicySelector	• Renamed to <a href="#">AdPolicySelector</a>

## Migrate from PSDK 1.2 to 1.3

The 1.3 PSDK includes various API changes that might affect your existing applications.

## Update ad-playback behavior for PSDK 1.3

The way in which ad break behavior is controlled has changed. In addition, some API elements have changed or been renamed as a result.

Ad behavior changes include:

- The `AdBreakPolicySelector` interface is replaced by `AdPolicySelector`, which has the following methods:
  - `public AdBreakPolicy selectPolicyForAdBreak(AdPolicyInfo adPolicyInfo);`
  - `public List<AdBreakPlacement> selectAdBreaksToPlay(AdPolicyInfo adPolicyInfo);`
  - `public AdPolicy selectPolicyForSeekIntoAd(AdPolicyInfo adPolicyInfo);`
  - `public AdBreakAsWatched selectWatchedPolicyForAdBreak(AdPolicyInfo adPolicyInfo);`
- The `DefaultAdBreakPolicySelector` class is replaced by `DefaultAdPolicySelector`, which implements `AdPolicySelector`.

To reproduce the same ad behavior as in the PSDK 1.2:

Create a `CustomAdPolicySelector` class that either extends the `DefaultAdPolicySelector` class or implements the `AdPolicySelector` interface. In `CustomAdPolicySelector`:

- Override the `selectPolicyForAdBreak` method. For normal playback, make sure that it returns the same `AdBreakPolicy` as in the PSDK 1.2.  
For example:
  - Return `AdBreakPolicy.PLAY` if your old ad breaks are always kept on the timeline after play.
  - Return `AdBreakPolicy.REMOVE` if your old ad breaks are always removed from the timeline after play.
- Override `selectAdBreaksToPlay` method and implement:
  - Return the last `AdBreak` whenever a seek forward occurs. (Note: Playback resumes from the desired seek position instead of from the ad break end position.)
  - Return `null` whenever a seek backward occurs.
- Override `selectPolicyForSeekIntoAd` method and implement:
  - Return `AdPolicy.PLAY_FROM_AD_BREAK_BEGIN` whenever a seek forward occurs.
  - Return `AdPolicy.SKIP_AD_BREAK` whenever a seek backward occurs.

## Create a video player

The PSDK provides the tools that you need for creating your own advanced video player application, which you can integrate with other Primetime components.

Follow these instructions to create your own Primetime player:

### Work with cookies

You can use Android utilities to write arbitrary data to cookies for session management, gate access, and so on.

One possible use for this is in conjunction with some kind of authentication when making requests to the key server. For example:

1. Your customer logs into your website in a browser and their login shows that they are allowed to view content.

2. Your application generates an authentication token, based on what is expected by the license server. Pass that value to the PSDK.
3. The PSDK sets that value in the cookie header.
4. When the PSDK makes a request to the key server to get a key to decrypt the content, that request contains the authentication value in the cookie header, so the key server knows the request is valid.

Create a `cookieManager` and add your cookies for the URIs to your `cookieStore`. For example:

```
CookieManager cookieManager=new CookieManager();
CookieHandler.setDefault(cookieManager);
HttpCookie cookie=new HttpCookie("lang","fr");
cookie.setDomain("twitter.com");
cookie.setPath("/");
cookie.setVersion(0);
cookieManager.getCookieStore().add(newURI("http://twitter.com/"),cookie);
```

The PSDK queries this `cookieManager` at runtime and checks whether there are any cookies associated with the URL and uses those automatically.

## Set up the MediaPlayer

The `MediaPlayer` interface for Android encapsulates the functionality and behavior of a media player.

The PSDK library provides a single implementation of the `MediaPlayer` interface: the `DefaultMediaPlayer` class.

When you need video-playback functionality, instantiate `DefaultMediaPlayer`.



**Note:** Interact with the `DefaultMediaPlayer` instance only with the methods exposed by the `MediaPlayer` interface.

1. Instantiate a `MediaPlayer` using the public `DefaultMediaPlayer.create` factory method, passing a Java Android application context object:

```
public static MediaPlayer create(Context context)
```

2. Call `MediaPlayer.getView` to get a reference to the `MediaPlayerView` instance:

```
MediaPlayerView getView() throws IllegalStateException;
```

3. Place the `MediaPlayerView` instance inside a `FrameLayout` instance, effectively placing the video on the device's screen:

```
FrameLayout playerFrame = (FrameLayout) view.findViewById(R.id.playerFrame);
playerFrame.addView(mediaPlayer.getView());
```

The `MediaPlayer` instance is now available and properly configured to display video content on the device screen.

## Listen for Primetime Player events

Your player must listen for events (notifications) from the PSDK that indicate the state of the player or the completion of various actions.

The real-time nature of video playback requires asynchronous (nonblocking) activity for many PSDK operations. The PSDK communicates asynchronously with your application by dispatching events (notifications). You implement event listeners as callbacks that take appropriate actions when they receive events from the PSDK.

Your application generally initiates nonblocking operations, and the PSDK calls your registered callback event listeners when the associated operation is complete. In addition, the PSDK uses events to notify you of errors, state changes, and other information.

### Implement event listener callbacks

You should implement callback event listeners for most of the possible PSDK events (notifications) and register them with the `MediaPlayer`.

Listeners are defined as public internal interfaces inside the `MediaPlayer` interface.

1. Implement callbacks for each of the events that your application is likely to receive.

For example, if you incorporate advertising in your playback, implement the `AdPlaybackEventListener` callbacks.

Implement callbacks for playback events, which include notifications event listeners for the `MediaPlayerStatusChangeEvent.STATUS_CHANGE` event, which notifies your application that the player's `state` has changed. Changes in `state` can indicate errors and can also affect which actions your player can take.

that the player's status has changed.

2. Register your callback listeners with the `MediaPlayer` object using `MediaPlayer.addEventListener`.

### Playback events

The PSDK dispatches playback events in response to media playback operations such as when a video starts playing.

To be notified about all playback-related events, register an implementation of `MediaPlayer.PlaybackEventListener` including the following event callbacks:

Event	Meaning
<a href="#"><i>onPlayComplete</i></a>	The end of a media source has been reached.
<a href="#"><i>onPlayStart</i></a>	Playback of a media source has started.
<a href="#"><i>onPrepared</i></a>	The media player has successfully prepared the media.
<a href="#"><i>onSizeAvailable</i></a> (long height, long width)	The size of the media is available.
<a href="#"><i>onStateChanged</i></a> ( <i>MediaPlayer.PlayerState</i> state, <i>MediaPlayerNotification</i> notification)	The <code>state</code> of the media player has changed. Your application should handle errors in this callback.
<a href="#"><i>onTimedMetadata</i></a> ( <i>TimedMetadata</i> <code>timedMetadata</code> )	A new timed metadata is discovered in the manifest.
<a href="#"><i>onTimelineUpdated</i></a>	The media player has an updated timeline. The timeline might contain new ads or some ads might have been removed.
<a href="#"><i>onTimelineUpdated</i></a>	The manifest refreshed for a live asset and old ad breaks were removed from the timeline or new ad opportunities (cue points) were discovered. The media player tries to resolve and place any new ads on the timeline. Therefore, use this event to check whether the timeline has any updates ( <a href="#"><i>MediaPlayer.getTimeline</i></a> ). (VOD does not change during playback.)
<a href="#"><i>onUpdated</i></a>	The media player has successfully updated the media in either of these cases: <ul style="list-style-type: none"> <li>• For each manifest refresh for a live asset.</li> </ul>

Event	Meaning
	<ul style="list-style-type: none"> <li>When a VOD or live asset has closed captioning and activity is first discovered for a closed captioning track.</li> </ul>

Some examples of the order of some events that include playback events:

- When successfully loading a media resource through the `MediaPlayer.replaceCurrent` method, the order of events is:
  - `MediaPlayerStatusChangedEvent.INITIALIZING`
  - `MediaPlayerItemEvent.ITEM_CREATED`
  - `MediaPlayerStatusChangeEvent.INITIALIZED`
- When preparing for playback through `MediaPlayer.prepareToPlay`, the order of events is:
  - `MediaPlayerStatusChangedEvent.PREPARING`
  - `TimelineOperationEvent.TIMELINE_UPDATED`
  - `MediaPlayerStatusChangedEvent.PREPARED`
- For live/linear streams, during playback as playback window advances and additional opportunities are resolved, the order of events is:
  - `MediaPlayerItemEvent.ITEM_UPDATED`
  - `TimelineOperationEvent.TIMELINE_UPDATED`

The following example shows a typical progression of events:

```

mediaPlayer.addListener(MediaPlayer.Event.PLAYBACK, new MediaPlayer.PlaybackEventListener()
{
    @Override
    public void onPrepared() {...}
    @Override
    public void onUpdated() {...}
    @Override
    public void onPlayStart() {...}
    @Override
    public void onPlayComplete() {...}
    @Override
    public void onSizeAvailable(long height, long width) {...}
    @Override
    public void onStateChanged(MediaPlayer.PlayerState state,
        MediaPlayerNotification notification) {...}
});

```

## Ad playback events

The PSDK dispatches ad-playback events in response to ad-related operations such as when an ad starts playing.

To be notified about all ad-playback related events, register an implementation of `MediaPlayer.AdPlaybackEventListener` including the following callbacks:

Event	Meaning
<a href="#"><i>onAdBreakComplete</i></a> (AdBreak adBreak)	An ad break has played completely.
<a href="#"><i>onAdBreakStart</i></a> (AdBreak adBreak)	An ad break has started.
<a href="#"><i>onAdClick</i></a> (AdBreak adBreak, Ad ad, AdClick adClick)	The user has clicked the ad. Provides information to your application about the ad that the user clicked, in response to your application calling <code>notifyClick</code> on the <code>MediaPlayerView</code> .
<a href="#"><i>onAdComplete</i></a> (AdBreak adBreak, Ad ad)	An ad has played completely.

Event	Meaning
<a href="#">onAdProgress</a> (AdBreak adBreak, Ad ad, int percentage)	Ad playback has progressed.
<a href="#">onAdStart</a> (AdBreak adBreak, Ad ad)	An ad has started.

When playing ads, the order of events is:

- AdBreakPlaybackEvent.AD\_BREAK\_STARTED
- AdPlaybackEvent.AD\_STARTED
- AdPlaybackEvent.AD\_PROGRESS ( this will be dispatched multiple times during the ad playback)
- AdPlaybackEvent.AD\_COMPLETED
- AdBreakPlaybackEvent.AD\_BREAK\_COMPLETED

The ad-related events (AD\_STARTED, AD\_PROGRESS, and AD\_COMPLETED) are dispatched for every ad contained within an ad break.

The following example shows a typical progression of ad playback events:

```
mediaPlayer.addEventListener(MediaPlayer.Event.AD_PLAYBACK, new
    MediaPlayer.AdPlaybackEventListener(){
@Override
public void onAdBreakStart(AdBreak adBreak) { ... }
@Override
public void onAdStart(AdBreak adBreak, Ad ad) { ... }
@Override
public void onAdProgress(AdBreak adBreak, Ad ad, int percentage) { ... }
@Override
public void onAdComplete(AdBreak adBreak, Ad ad) { ... }
@Override
public void onAdBreakComplete(AdBreak adBreak) { ... }
@Override
public void onAdClick(AdBreak adBreak, Ad ad, AdClick adClick) { ... }
});
```

### QoS events

The PSDK dispatches quality of service (QoS) events to notify your application about events that could influence the computation of QoS statistics, such as the buffering, seeking, and loading events.

To be notified about all QoS-related events, register an implementation of `MediaPlayer.QOSEventListener` including the following callbacks:

Event	Meaning
<a href="#">onBufferComplete</a>	Buffering is complete.
<a href="#">onBufferStart</a>	Buffering has started.
<a href="#">onLoadInfo</a> (loadInfo)	A fragment has successfully downloaded.
<a href="#">onOperationFailed</a> (MediaPlayerNotification. <a href="#">Warning</a> warning)	A recoverable error has occurred.
<a href="#">onSeekComplete</a> (long adjustedTime)	Seeking is complete.
<a href="#">onSeekStart</a>	Seeking is starting.

When seeking, the order of events is as follows:

- SeekEvent.SEEK\_POSITION\_ADJUSTED (if the desired seek position is modified by the current advertising policies)
- SeekEvent.SEEK\_BEGIN

- `SeekEvent.SEEK_END`
- 

Any ad-related events are dispatched after the `SeekEvent.SEEK_END` event.

The following example shows a typical progression of these events:

```
mediaPlayer.addListener(MediaPlayer.Event.QOS, new MediaPlayer.QOSEventListener() {
    @Override
    public void onBufferStart()
    @Override
    public void onBufferComplete()
    @Override
    public void onSeekStart()
    @Override
    public void onSeekComplete(long adjustedTime)
    @Override
    public void onLoadInfo(LoadInfo loadInfo)
    @Override
    public void onOperationFailed(MediaPlayerNotification.Warning warning)
});
```

### DRM events

The PSDK dispatches digital rights management (DRM) events in response to DRM-related operations such as when new DRM metadata becomes available.

To be notified about all DRM-related events, register an implementation of `MediaPlayer.DRMEventListener` including the following callbacks:

Event	Meaning
<a href="#">onDRMMetadata</a> ( <code>DRMMetadataInfo drmMetadataInfo</code> ) <code>DRMMetadataEvent.DRM_METADATA_INFO_AVAILABLE</code>	New DRM metadata is available.

The following example shows a typical progression of these events:

```
mediaPlayer.addListener(MediaPlayer.Event.DRM, new MediaPlayer.DRMEventListener() {
    @Override
    public void onDRMMetadata(byte[] data, int length, long timestamp) {...}
});
```

### Loader events

The PSDK dispatches media player item events in response to loading a media item.

These events provide an alternative workflow. You are not required to implement this interface when creating a `MediaPlayer`. Use this when you want to have a `MediaPlayerItemLoader`.

To be notified about events related to loading a media player resource, register an implementation of `MediaPlayerItemLoader.LoaderListener` including the following callbacks:

Event	Meaning
<a href="#">onLoadComplete</a> ( <code>MediaPlayerItem</code> playerItem)	Media resource loading completed successfully.
<a href="#">onError</a>	A problem occurred with media resource loading.

### Set up error handling

Set up a single place in your application in which to perform error handling in response to the `ERROR` state .

1. Implement a callback for `PlaybackEventListener.onStateChanged`.
2. In your callback, when the state parameter is `PlayerState.ERROR`, provide logic to handle all errors.
3. After the error is handled, reset the `MediaPlayer` object or load a new media resource.

When the `MediaPlayer` object is in the `ERROR` state, it cannot exit this state until you either reset the `MediaPlayer` object (via the `MediaPlayer.reset` method) or load a new media resource (`MediaPlayer.replaceCurrentItem` or `MediaPlayer.ReplaceCurrentItem`).

For example:

```
mediaPlayer.addListener(MediaPlayer.Event.PLAYBACK, new MediaPlayer.PlaybackEventListener()
{
    @Override
    public void onStateChanged(MediaPlayer.PlayerState state, MediaPlayerNotification notification)
    {
        if (state == MediaPlayer.PlayerState.ERROR) {
            // handle PSDK error here
        }
    }
})
```

## Configure the player user interface

With the PSDK, you can control the basic playback experience for live and video on demand (VOD). The PSDK does not configure the player for you; instead, it provides methods and properties on the player instance that you can use to configure the player user interface.

### Implement a play/pause button

Using PSDK methods, you can set up pause and play buttons for the user.

1. Wait for the PSDK to call your `PlaybackEventListener.onPrepared` callback.

This ensures that the media resource has successfully loaded. If the player is not in the `PREPARED` state, attempting to call the following methods throws an `IllegalStateException`.

2. Create a pause/play button and have it call the PSDK:

- To start playback.

```
void play() throws IllegalStateException;
```

- To pause playback.

```
void pause() throws IllegalStateException;
```

3. Use the `MediaPlayer.PlaybackEventListener.onStateChanged` callback to check for errors or to take other appropriate actions.

The PSDK calls this callback when the pause or play methods are called. The PSDK passes information about the state change in the callback, including the new state, such as `PAUSED` or `PLAYING`.

### Identify whether the content is live or VOD

In some cases, you need to know whether the media content is live or VOD.

1. Wait for the PSDK to call your `PlaybackEventListener.onPrepared` callback.

This ensures that the media resource has successfully loaded. If the player is not in at least the PREPARED state, attempting to call the following methods throws an `IllegalStateException`.

2. Call `isLive` from the `MediaPlayerItem` interface.

```
boolean isLive();
```

### Provide a volume control

You can set up a user interface control for the player's volume.

1. Wait for the PSDK to call your `PlaybackEventListener.onPrepared` callback.

This ensures that the media resource has successfully loaded. If the player is not in the PREPARED state, attempting to call the following methods throws an `IllegalStateException`.

2. Call `setVolume` to set the audio volume.

The parameter for the `setVolume` method represents the requested actual volume level expressed as a percentage of the maximum level, where 0 is silent and 100 is full volume. The default is 1.

```
void setVolume(int volume) throws IllegalStateException;
```

Your player can be in any state except RELEASED or ERROR to use this method.

### Display the duration, current time, and remaining time of the video

You can use the PSDK to retrieve information about the media that you can then display on the seek bar.

1. Wait for the PSDK to call your `PlaybackEventListener.onPrepared` callback.

This ensures that the media resource has successfully loaded. If the player is not in at least the PREPARED state, attempting to call the following methods throws an `IllegalStateException`.

2. Retrieve the current playhead time using the `MediaPlayer.getCurrentTime` method.

This returns the current playhead position on the virtual timeline expressed in milliseconds. The time is calculated relative to the resolved stream, which could contain multiple instances of alternate content (multiple ads or ad breaks spliced into the main stream). For live/linear streams, the returned time is always inside the playback window range.

```
long getCurrentTime() throws IllegalStateException;
```

3. Retrieve the playback range of the stream and determine the duration.

- a) Use the `mediaPlayer.getPlaybackRange` method to get the virtual timeline time range.

```
TimeRange getPlaybackRange() throws IllegalStateException;
```

- b) To determine the duration, subtract the start from the end of the range.

This includes the duration of any additional content inserted into the stream (ads)

- c) Parse the time range using `mediacore.utils.TimeRange` `mediacore.utils.TimeRange`.

For VOD, the range always begins with zero and the end value equals the sum of the main content duration and the durations of any additional content inserted into the stream (ads).

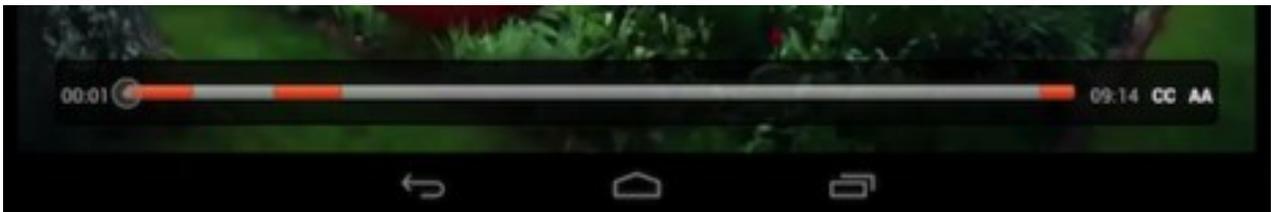
For a linear/live asset, the range represents the playback window range. This range changes during playback. The PSDK calls your `onUpdated` callback to indicate that the media item was refreshed and that its attributes (including the playback range) were updated.

- Use the methods available on the `MediaPlayer` and the `SeekBar` class that is publicly available in the Android SDK to set up the control-bar parameters.

For example, here is a possible layout that contains the `SeekBar` and two `TextView` elements.

```
<LinearLayout
  android:id="@+id/controlBarLayout"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_alignParentBottom="true"
  android:background="@android:color/black"
  android:orientation="horizontal" >
  <TextView
    android:id="@+id/playerCurrentTimeText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="7dp"
    android:text="00:00"
    android:textColor="@android:color/white" />
  <SeekBar
    android:id="@+id/playerSeekBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1" />
  <TextView
    android:id="@+id/playerTotalTimeText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="7dp"
    android:text="00:00"
    android:textColor="@android:color/white" />
</LinearLayout>
```

- Use a timer to periodically retrieve the current time and update the `SeekBar`, as shown in the following figure:



The following example uses the `Clock.java` helper class as the timer, which is available in the `PMPDemoApp`. This class sets an event listener and triggers an `onTick` event every second, or another timeout value that you can specify.

```
playbackClock = new Clock(PLAYBACK_CLOCK, CLOCK_TIMER);
playbackClockEventListener = new Clock.ClockEventListener() {
  @Override
  public void onTick(String name) {
    // Timer event is received. Update the SeekBar here.
  }
};
playbackClock.addClockEventListener(playbackClockEventListener);
```

On every clock tick, this example retrieves the media player's current position and updates the `SeekBar`. It uses the two `TextView` elements to mark the current time and the playback range end position as numeric values.

```
@Override
public void onTick(String name) {
  if (mediaPlayer != null && mediaPlayer.getStatus() == PlayerState.PLAYING) {
    handler.post(new Runnable() {
      @Override
      public void run() {
        seekBar.setProgress((int)
          mediaPlayer.getCurrentTime());
      }
    });
  }
}
```

```

        currentTimeText.setText(timestampToText(
            mediaPlayer.getCurrentTime()));
        totalTimeText.setText(timestampToText(
            mediaPlayer.getPlaybackRange().getEnd()));
    });
}

```

## Display a seek scrub bar with the current playback time position

The PSDK supports seeking to a specific position (time) in live streams where the stream is either a sliding-window playlist or an event type and in VOD.

You can seek to a particular location in the stream.



### Note:

*Seeking in a live stream is allowed only for DVR.*

1. Wait for the PSDK to be in a valid state for seeking.

Valid states are PREPARED, COMPLETE, PAUSED, and PLAYING. Being in a valid state ensures that the media resource has successfully loaded. If the player is not in a valid seekable state, attempting to call the following methods throws an `IllegalStateException`.

For example, wait for the PSDK to call your `PlaybackEventListener.onPrepared` callback.

2. Pass the requested seek position to the `MediaPlayer.seek` method as a parameter expressed in milliseconds.

This moves the play head to a different position in the stream. Note that the requested seek position might not coincide with the actual computed position.

```
void seek(long position) throws IllegalStateException;
```

3. Wait for the PSDK to call the `QOSEventListener.onSeekComplete` callback, which returns the adjusted position in the callback's position parameter.

This is important because the actual start position after the seek could be different than the requested position.

Various rules might apply, including:

- Playback behavior is affected if a seek or other repositioning ends in the middle of an ad break or skips ad breaks.
- You can seek only in the asset's seekable duration. For video on demand, that is from 0 through the asset's duration.

4. Use the native `SeekBar` to set an `OnSeekBarChangeListener` to see when the user is scrubbing.

In the following example, the user scrubs the seek bar to seek to the desired position.

```

seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress, boolean isFromUser) {
        if (isFromUser) {
            // Update the seek bar thumb, with the position provided by the user.
            setPosition(progress);
        }
    }
    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {
        isSeeking = true;
    }
    @Override

```

```

public void onStopTrackingTouch(SeekBar seekBar) {
    isSeeking = false;
    // Retrieve the playback range.
    TimeRange playbackRange = mediaPlayer.getPlaybackRange();
    // Make sure to seek inside the playback range.
    long seekPosition = Math.min(Math.round(seekBar.getProgress()), playbackRange.getDuration());
    // Perform seek.
    seek(playbackRange.getBegin() + seekPosition);
}
});

```

##### 5. Set up event listener callbacks for changes in the user's seek activity.

The seek operation is asynchronous, so the PSDK dispatches these events related to seeking:

- `QOSEventListener.onSeekStart` - Called to indicate that seek is starting.
- `QOSEventListener.onSeekComplete` - Called to indicate that seeking was successful.
- `QOSEventListener.onOperationFailed` - Called to indicate that seeking was unsuccessful.

The media player might readjust the seek position provided by the user. Therefore, check the `QOSEventListener.onSeekComplete` callback, which returns the adjusted seek position.

##### 6. Listen for the `QOSEventListener.onOperationFailed` event, which indicates that the seek operation failed for various reasons, and take appropriate actions.

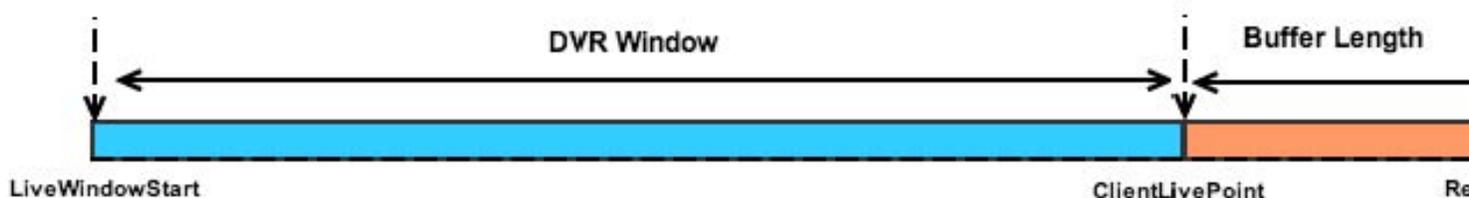
`QOSEventListener.onOperationFailed` passes the appropriate warning. Your application needs to decide how to proceed in this case. For instance, it could try to seek again or it could continue playback from the previous position.

### Construct a control bar enhanced for DVR

You can implement a control bar with DVR support for VOD and live streaming. DVR support includes the concept of a seekable window and the client live point.

- For VOD, the length of the seekable window is the duration of the entire asset.
- For live streaming, the length of the DVR (seekable) window is defined as the time range starting at the live playback window and ending at the client live point.

The client live point is calculated by subtracting the buffered length from the live window end. The target duration is a value bigger than or equal to the maximum duration of a fragment in the manifest; the default value is 10000 ms. The control bar for live playback supports DVR by first positioning the thumb at the client live point when starting playback and by displaying a region that marks the area where seek is not allowed.



To implement a control bar with DVR support, follow the steps for displaying a seek scrub bar, with a few minor differences:

- You can choose to implement a control bar that is mapped only for the seekable range instead of for the playback range. Any user interaction for seek can be considered safe in the seekable range.

- You can choose to implement a control bar that is mapped for the playback range but that also displays the seekable range. To do this:
  1. Add an overlay over the control bar that represents the playback range.
  2. When the user starts to seek, check whether the desired seek position is within the seekable range using `getSeekableRange`.

```
TimeRange seekableRange = _mediaPlayer.getSeekableRange();
if (seekableRange.contains(desiredSeekPosition)) {
    _mediaPlayer.seek(desiredPosition);
}
```

You can also choose to seek to the client live point using the `LIVE_POINT` constant.

```
mediaPlayer.seek(MediaPlayer.LIVE_POINT);
```

### Use trick play mode (FF and REW) and slow motion

You can set up your player to allow the user to fast forward, rewind, play in slow motion, and resume playing at a normal speed.

This feature allows your video player to switch from normal playback (1x) to fast forward or fast rewind mode; these features are collectively known as *trick play*. The transition from normal play mode to trick play is performed by setting the rate on the `MediaPlayer` to a trick-play value.

The PSDK dispatches events related to trick play:

- `MediaPlayerEvent.RATE_SELECTED` when the `rate` value changes to a different value.
- `MediaPlayerEvent.RATE_PLAYING` when playback resumes at the selected rate
- Both of these events when returning from trick-play mode to normal play mode because the PSDK internally sets the rate to normal play

The `MediaPlayerItem` class exposes the allowed playback rates. The PSDK automatically selects the closest allowed rate if the set rate is not in the list of allowed rates.

### Limitations and behavior at various rates of playback

Using different rates of playback has an effect on what features are enabled and what actions are allowed.

The following apply during trick-play mode (fast forward and fast rewind):

- The master playlist must contain I-frame-only segments.
- Only the key frames from the I-frame track are displayed on the screen.
- Play and pause are enabled.
- Seek is disallowed. If you need to seek, call `pause` to get out of trick play mode and then call `seek`.
- The adaptive bit rate (ABR) logic is disabled. A single bit rate is chosen and used during the entire trick-play session. The chosen bit rate is between the lowest provided one and 800 kbps.
- The audio track and closed captions are disabled.
- When ads are incorporated into the stream:
  - Ad breaks are ignored and no ad events are fired.
  - The `MediaPlayerEvent.AD_BREAK_SKIPPED` event is raised right before an ad break is about to be skipped. The `reason` property indicates that trick-play has caused the skip operation (seek can also cause skips). Your player application can use this event to implement custom logic related to the skipped ad breaks.

- When returning from trick play, a seek operation is performed internally on the return position. This means that the selected ad playback policy related to seek over ads applies also when returning from trick play. The default behavior would be that the last skipped ad break is played each time the player returns from trick-play mode, forcing the playback to resume to the selected return position only after the last skipped ad break plays. This behavior depends on your application's selected ad policy and can change when the policy changes.
- The timeline exposed by PSDK to your player app is not modified even if ad breaks are skipped. The PSDK informs your application with the `AD_BREAK_SKIPPED` event that an ad break is about to be skipped, even if it is present on the timeline.
- The current time value jumps forward (on fast forward) or backward (on fast rewind) with the duration of the skipped ad break. This jump behavior for the current time allows the stream duration to remain unmodified during trick play. Your player application can get the local time value to track the time relative only to the main content. No time jumps are performed on the values returned for the local time when skipping an ad.
- Your player application can transition to trick play only while playing the main content. An error is dispatched if the application tries to switch to trick play while playing an ad break.
- You cannot change the rate to trick play mode when you are in the allowed playback rate range and within an ad break.
- You can drop out of trick play mode into play, pause, or slow motion (that is, the allowed playback rates) and all the removed ads are restored.
- You can enter trick play mode from any of the allowed playback rates, and only at that point ads are removed from the timeline.

The following apply during slow-motion playback:

- All HLS streams support slow-motion playback (the I-frame playlist is not used in this case).
- The audio track is disabled; closed captions continue to render.
- ABR logic continues to work.
- Ad-related workflows are the same as for normal playback.
- Play, pause, and seek behavior is the same as for trick play.
- When ads are incorporated into the stream:
  - Ads are not removed from the timeline and ad breaks are not skipped.

### Trick-play API elements

The PSDK includes methods, properties, and events for implementing fast forward and rewind.

- `MediaPlayer.setRate`

Rate value	Effect on playback
2, 4, 8, 16	Switches to fast-forward mode with the specified multiplier faster than normal (for example, 4 is 4 times faster than normal)
-2, -4, -8, -16	Switches to fast-rewind mode
1	Switches to normal play mode (calling <code>play</code> is the same as setting the rate property to 1.0)
0	Pauses (calling <code>pause</code> is the same as setting the rate property to 0.0)
0.25, 0.5, 0.75	Switches to slow motion

- `MediaPlayer.getRate`

- `PlaybackRateEvent.getRate`
- `MediaPlayer.getLocalTime`
- `MediaPlayerEvent.RATE_SELECTED`
- `MediaPlayerEvent.RATE_PLAYING`
- `MediaPlayerItem.isTrickPlaySupported`
- `MediaPlayerItem.getAvailablePlaybackRates`
- `MediaPlayerEvent.AD_BREAK_SKIPPED`

### Implement fast forward and rewind (trick-play mode)

Changing the playback rate is a single step.

Initiate trick play mode by setting the rate property for the `MediaPlayer` class to an allowed value.

Just to show how to set the rate, this example sets it using a dialog.

```
private void selectPlaybackRate() {
    PMPDemoApp.logger.i(LOG_TAG + "#selectPlaybackRate", "Displaying playback rate chooser
    dialog.");

    final String items[] = getAvailabeRatesArray();
    final int selectedRateIndex = getSelectedRateIndex();
    new AlertDialog.Builder(getActivity())
        .setTitle(R.string.PlayerControlRTDialogTitle)
        .setSingleChoiceItems(items, selectedRateIndex, new
    DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
            MediaPlayerItem currentItem = _mediaPlayer.getCurrentItem();
            Float desiredRate =
    currentItem.getAvailablePlaybackRates().get(whichButton);

            // set rate
            _mediaPlayer.setRate(desiredRate.floatValue());

            // Dismiss dialog.
            dialog.cancel();
        }
    })
    .setNegativeButton(R.string.PlayerControlRTDialogCancel, new
    DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
            // Just cancel the dialog.
        }
    })
    .show();
}
```

### Enter a stream at a specific time

By default, when starting playback, VOD media starts at 0 and live media starts at the client live point (`MediaPlayer.LIVE_POINT`). You can choose to override the default behavior.

Pass a position to `MediaPlayer.prepareToPlay`.

The PSDK then considers the position given as the starting point for the asset. No seek operation is required.

If the position is not inside the seekable range, the default positions are used.

For example:

```
long desiredPostion = //TODO : choose a value;
@Override
public void onStateChanged(MediaPlayer.PlayerState state, MediaPlayerNotification notification)
{
```

```

switch (state) {
    case INITIALIZED:
        _mediaPlayer.prepareToPlay(desiredPosition);
        break;
    case PREPARING:
        showBufferingSpinner();
        break;
}

```

## Control the quality with adaptive bit rates (ABR)

The PSDK can play video assets that have multiple bit rates to provide more than one quality level.

Based on the bandwidth conditions and the quality of playback (frame rate), the video engine automatically switches the quality level to provide the best playback experience. The PSDK performs transitions between the various quality levels seamlessly and automatically. PSDK exposes the list of renditions and enables you to control some aspects of the adaptive streaming experience.

You can specify multiple profiles that have different bit rates. You can change your ABR settings at any time and the player switches to use the profile that most closely matches the most recent settings.

To configure the adaptive bit-rate (ABR) engine, choose values for the ABR parameters.

1. Decide on the minimum and maximum bit rates.

These define a range of bit rates from which the ABR engine can choose. The ABR engine uses profiles only from this range when downloading fragments and rendering images on screen.

2. Decide on the initial bit rate, in bits per second, to use when beginning playback.

For preload of the first downloaded fragment, the PSDK selects the profile with the bit-rate value that is closest to (equal to or less than) this initial value. For the first fragment, the minimum/maximum range is ignored.

3. Decide on the ABR policy, which determines how quickly the engine switches between available quality profiles.

When set to the aggressive policy, the video engine moves much more quickly to a higher-resolution profile than when the conservative policy is selected. The `ABRControlParameters` class exposes the `ABRPolicy` enumeration, which defines the following policy values:

- `ABR_CONSERVATIVE`
- `ABR_MODERATE`
- `ABR_AGGRESSIVE`

4. Set the ABR parameter values.

You can set these values only in the `ABRControlParameters` constructor, but you can construct a new one at any time.

```

public ABRControlParameters(int initialBitRate,
                           int minBitRate,
                           int maxBitRate,
                           ABRControlParameters.ABRPolicy abrPolicy)

```

5. Assign the new ABR parameter values to the media player.

```

MediaPlayer.setABRControlParameters(ABRControlParameters params)

```

### **Note:**

The `PMPDemoApp` scrub bar does not include settings for controlling the ABR parameters, but it makes the controls available from the options menu that is accessible with the `ActionBar`.

## Initialize the MediaPlayer for a specific video

Each time that you play different video content, you must initialize a `MediaResource` instance with information about the video content, then load the media resource.

### Create a media resource

The `MediaResource` class represents the content that is about to be loaded by the `MediaPlayer` class.

1. Create a `MediaResource` by passing information about the media to `MediaResource.createFromUrl`.

Parameter	Description
<b>URL</b>	Required. The location of the manifest/playlist for the content to be loaded.
<b>Metadata</b>	Required when using the <code>MediaResource</code> with the <code>createFromUrl</code> method. Otherwise, pass null. An instance of the <code>Metadata</code> class (a dictionary-like structure). This structure might contain additional information about the content that is about to be loaded, such as information about alternate or ad content to place inside the main content.

```
try {
    // create a MediaResource instance pointing to some HLS content
    Metadata metadata = //TODO: create metadata
    MediaResource mediaResource =
        MediaResource.createFromUrl("http://www.example.com/video/some-video.m3u8", metadata);
} catch(IllegalArgumentException ex) {
    // this exception is thrown if the URL does not point to a valid url.
}
```

2. Check the media type and optionally retrieve other information about the media:



**Important:** Currently, the Android PSDK supports only HLS content. If you attempt to load content of type other than HLS, such as HDS, the PSDK dispatches an error event.

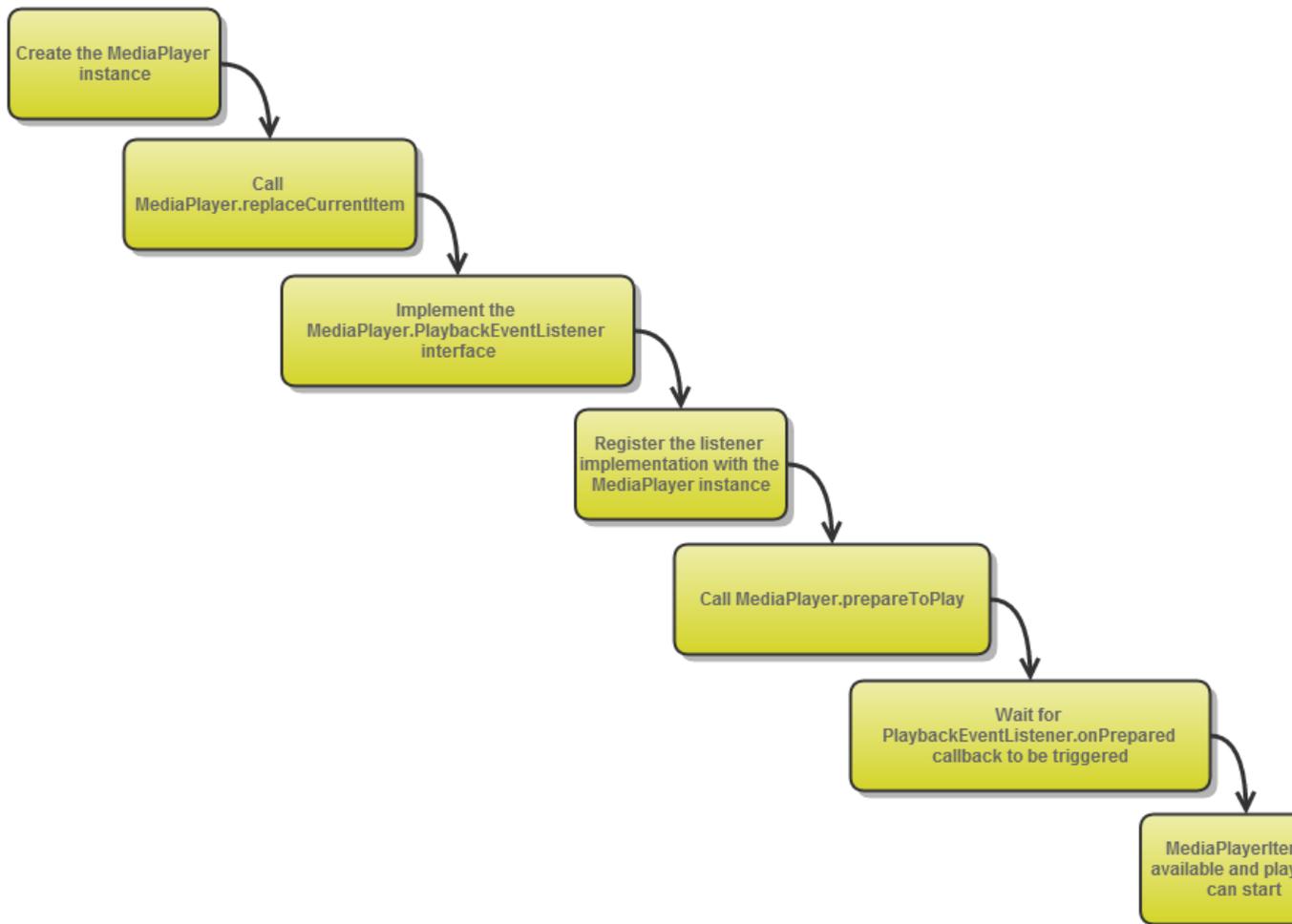
Method	Returns
<b>getType</b>	The type of the content to be loaded. Usually you can detect the type of the resource from the URL, but sometimes when custom naming schemes are involved, this is not possible. This is a simple enumeration, <code>MediaResource.Type</code> , which defines the types of content that can be loaded by the <code>MediaPlayer</code> . This enumeration defines two values: HLS and HDS. Each value is associated with the string representing the file extensions commonly used: "m3u8" for HLS and "f4m" for HDS, respectively.
<b>getUrl</b>	The URL of the location of the manifest/playlist for the content to be loaded.
<b>getMetadata</b>	An instance of the <code>Metadata</code> class (a dictionary-like structure). This structure contains additional optional information about the content that is about to be loaded.

3. Load the media resource in either way:
  - Load it directly inside the `MediaPlayer`
  - Load it using `MediaPlayerItemLoader`

### Load a media resource inside the MediaPlayer

One way of loading a media resource is to directly instantiate a `MediaResource` and load the video content to be played.

Loading a `MediaResource` is an asynchronous multiple-step process.



1. Replace the `MediaPlayer`'s currently playable item by calling `MediaPlayer.replaceCurrentItem`, passing an existing `MediaResource` instance.

This starts the resource loading process.

2. Register an implementation of the `MediaPlayer.PlaybackEventListener` interface with the `MediaPlayer` instance. Check for at least the following callbacks:

- `onPrepared`
- `onStateChanged`, and check for `INITIALIZED` and `ERROR`

Through this event listener, the `MediaPlayer` object can notify your application when the media resource is successfully loaded.

3. When the media player state changes to `INITIALIZED`, you can call `MediaPlayer.prepareToPlay`.

This `state` indicates that the media has successfully loaded. Calling `prepareToPlay` starts the advertising resolution and placement process.

4. When the PSDK calls the `onPrepared` callback, the media stream has successfully loaded and is prepared for playback.

In case of failure, the `MediaPlayer` switches to the `ERROR` state and notifies your application by calling your `PlaybackEventListener.onStateChanged` callback. The PSDK passes several parameters, including these:

- A state parameter of type `MediaPlayer.PlayerState` with the value of `MediaPlayer.PlayerState.ERROR`.
- A notification parameter of type `MediaPlayerNotification` that contains diagnostic information about the error event.

The following simplified sample code illustrates the process of loading a media resource:

```
// mediaResource is a properly configured MediaResource instance
// mediaPlayer is a MediaPlayer instance
// register a PlaybackEventListener implementation with the MediaPlayer instance
mediaPlayer.addListener(MediaPlayer.Event.PLAYBACK,
    new MediaPlayer.PlaybackEventListener() {
    @Override
    public void onPrepared() {
        // at this point, the resource is successfully loaded and available
        // and the MediaPlayer is ready to start the playback
        // once the resource is loaded, the MediaPlayer is able to
        // provide a reference to the current "playable item"
        MediaPlayerItem playerItem = mediaPlayer.getCurrentItem();
        if (playerItem != null) {
            // here we can take a look at the properties of the
            // loaded stream
        }
    }
    @Override
    public void onStateChanged(MediaPlayer.PlayerState state,
        MediaPlayerNotification notification) {
        if (state == MediaPlayer.PlayerState.ERROR) {
            // something bad happened - the resource cannot be loaded
            // details about the problem are provided via the MediaPlayerNotification instance
        } else
        if (state == MediaPlayer.PlayerState.INITIALIZED) {
            mediaPlayer.prepareToPlay();
        }
    }
    // implementation of the other methods in the PlaybackEventListener interface
    ...
}
```

### Load a media resource using `MediaPlayerItemLoader`

Another way to resolve a media resource is with `MediaPlayerItemLoader`. This is useful when you want to obtain information about a particular media stream without the full instantiation of a `MediaPlayer` instance.

Through the `MediaPlayerItemLoader` class, you can exchange a media resource for the corresponding `MediaPlayerItem` without attaching a view to a `MediaPlayer` instance, which would lead to the allocation of the video decoding hardware resources. The process of obtaining the `MediaPlayerItem` instance is asynchronous.

1. Implement the `MediaPlayerItemLoader.LoaderListener` callback interface. This interface defines two methods:
  - `LoaderListener.onError` callback function: The PSDK uses this to inform your application that an error has occurred. The PSDK provides as parameters an error code and a description string that contains diagnostic information.

- `LoaderListener.onLoadComplete` callback function: The PSDK uses this to inform your application that the desired information is available in the form of a `MediaPlayerItem` instance that is passed on as a parameter to the callback.
2. Register this instance to the PSDK by passing it as a parameter to the constructor of the `MediaPlayerItemLoader`.
  3. Call `MediaPlayerItemLoader.load`, passing an instance of a `MediaResource` object.

The URL of the `MediaResource` object must point to the stream for which you want to obtain information.

Here is an example:

```
// instantiate the listener interface
MediaPlayerItemLoader.LoaderListener _itemLoaderListener =
    new MediaPlayerItemLoader.LoaderListener() {
@Override
public void onError(MediaErrorCode mediaErrorCode, String description) {
    // something went wrong - look at the error code and description
}
@Override
public void onLoadComplete(MediaPlayerItem playerItem) {
    // information is available - look at the data in the "playerItem" object
}
}
// instantiate the MediaPlayerItemLoader object (pass the listener as parameter)
MediaPlayerItemLoader itemLoader = new MediaPlayerItemLoader(_itemLoaderListener);
// create the MediaResource instance and set the URL to point to the actual media stream
MediaResource mediaResource =
    MediaResource.createFromUrl("http://example.com/media/test_media.m3u8", null);
// load the media resource
itemLoader.load(mediaResource);
```

## Monitor quality of service statistics

Quality of service (QoS) offers a detailed view into how the video engine is performing, without a lot of integration work.

Some QoS information that you can use:

### Track at the fragment level using load information

The PSDK provides QoS information about downloaded fragments and tracks. All the QoS information related to resources that are downloaded is exposed through the `LoadInfo` class.

The PSDK provides download information about downloads of

- Playlist/manifest files
- Fragments files
- Tracks files

1. Implement the `onLoadInfo` callback event listener.
2. Register the event listener, which the PSDK calls every time a fragment has downloaded.
3. Read the data of interest from the `LoadInfo` parameter passed to the callback.

Property	Type	Description
<code>downloadDuration</code>	<code>long</code>	The duration of the download in milliseconds.
<code>mediaDuration</code>	<code>long</code>	The media duration of the downloaded fragments in milliseconds.
<code>periodIndex</code>	<code>int</code>	The timeline period index associated with the downloaded resource.

Property	Type	Description
size	long	The size of the downloaded resource in bytes.
trackIndex	int	The index of the corresponding track, if known; otherwise, 0.
trackName	String	The name of the corresponding track, if known; otherwise, null.
trackType	String	The type of the corresponding track, if known; otherwise, null.
type		The type of the download : <ul style="list-style-type: none"> <li>• MANIFEST - indicates that the PSDK downloaded a playlist/manifest</li> <li>• FRAGMENT - indicates that the PSDK downloaded a fragment</li> <li>• TRACK - indicates that the PSDK downloaded a fragment associated with a specific track</li> </ul>
url	String	The URL pointing to the downloaded resource.

**Note:**

The PSDK does not differentiate between the time it took the client to connect to the server and the time it took to download the full fragment. For example, the PSDK provides information that a 10 MB segment took 8 seconds to download, but does not identify that it took 4 seconds until the first byte and another 4 seconds to download the entire fragment.

## Read QOS playback and device statistics

You can read playback and device statistics from the `QOSProvider` as often as needed.

The `QOSProvider` class provides various statistics, including information related to buffering, bit rates, frame rates, time data, and various other topics.

You can also get information about the device, such as manufacturer, model, operating system, SDK version, manufacturer's device ID, and screen size/density manufacturer, model, operating system, SDK version, and screen size/density.

1. Instantiate a media player.
2. Create a `QOSProvider` object and attach it to the media player.

```
// Create Media Player.
_mediaQosProvider = new QOSProvider(getActivity().getApplicationContext());
_mediaQosProvider.attachMediaPlayer(_mediaPlayer);
```

The `QOSProvider` constructor takes a player context, because it will need it to retrieve device-specific information such as the model, manufacturer, and operating system.

3. Optionally read playback statistics.

One solution for reading playback statistics would be to have a timer that periodically fetches the new QoS values from the `QOSProvider`. For example:

```
_playbackClock = new Clock(PLAYBACK_CLOCK, 1000); // every 1 second
_playbackClockEventListener = new Clock.ClockEventListener() {
@Override
public void onTick(String name) {
getActivity().runOnUiThread(new Runnable() {
@Override
public void run() {
PlaybackInformation playbackInformation = _mediaQosProvider.getPlaybackInformation();
```



## Lifecycle and states of the MediaPlayer object

From the moment when the `MediaPlayer` instance is created to the moment it is terminated (reused or removed), the `MediaPlayer` object completes a series of transitions from one state to another.

The list of states is defined in `MediaPlayer.PlayerState`. You can retrieve the current state of the `MediaPlayer` object with `MediaPlayer.getStatus`.

```
PlayerState getStatus() throws IllegalStateException;
```

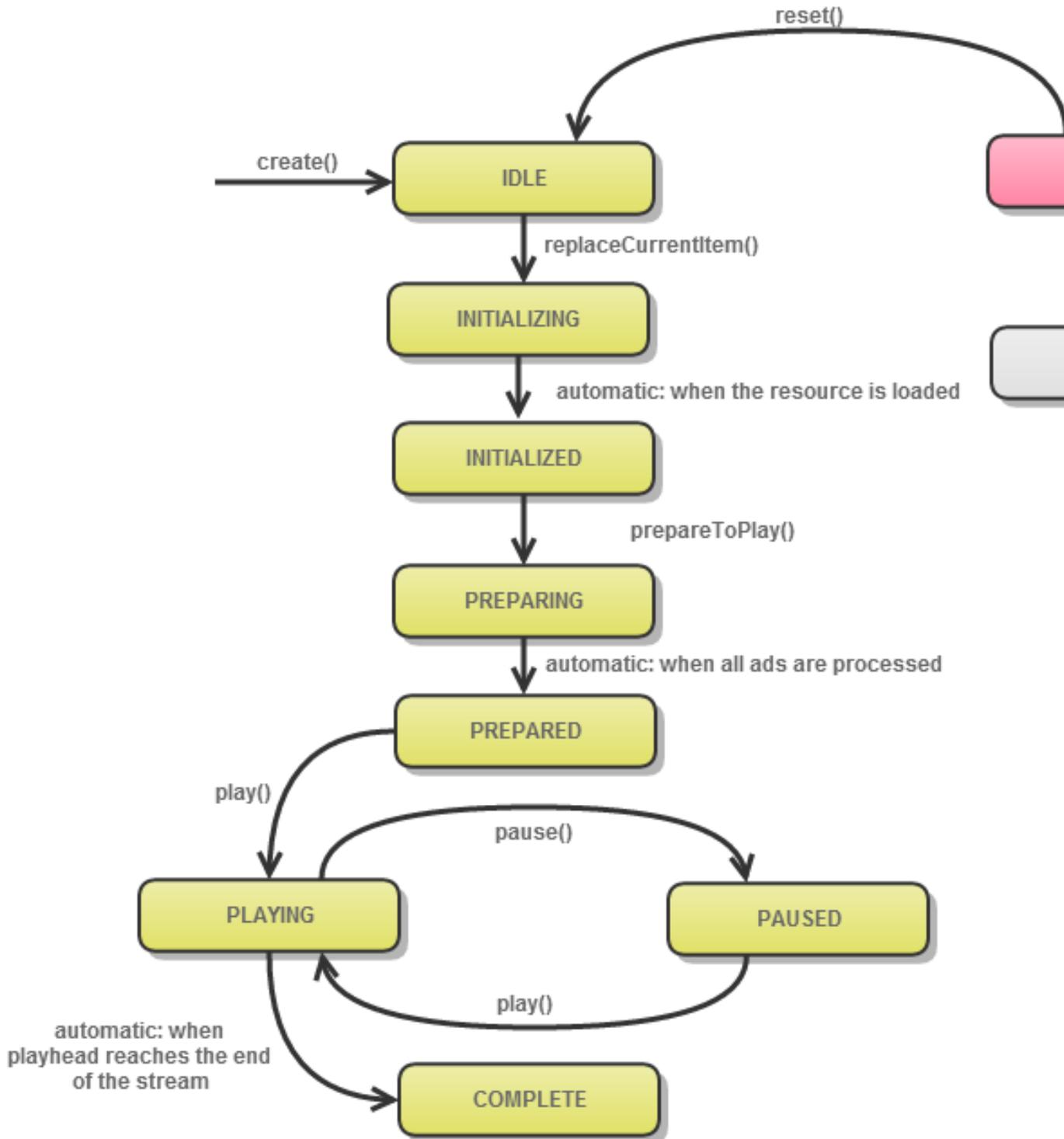
Knowing the player's state is useful because some operations are permitted only while the player is in a particular state. For example, `play` cannot be called while in `IDLE`. It must be called after reaching the `PREPARED` state.

The `ERROR` state also changes what can happen next.

Here is how the basic procedure for loading a media resource inside the `MediaPlayer` corresponds to state transitions:

1. The initial state is `IDLE`.
2. Your application calls `MediaPlayer.replaceCurrentItem`, which moves the player to `INITIALIZING`.
3. The PSDK loads the resource. If successful, the state becomes `INITIALIZED`.
4. Your application calls `MediaPlayer.prepareToPlay`. The state becomes `PREPARING`.
5. The PSDK prepares the media stream and starts the ad resolving and ad insertion (if enabled).
6. When this finishes (either ads are inserted into the timeline or the ad procedure has failed), the player state becomes `PREPARED`.
7. As your application plays and pauses the media, the state moves among `PLAYING` and `PAUSED` \*\*\*Android n/a\*\*\*.
8. When the player reaches the end of the stream, the state becomes `COMPLETE`.
9. When your application releases the media player, the state becomes `RELEASED`.
10. If an error occurs at any time during the process, the state becomes `ERROR`.

The following state-transition diagram shows the lifecycle of a `MediaPlayer` instance.



You can use the `state` to provide feedback to the user on the process (for example, a spinner while waiting for the next `state` change) or to take the next steps in playing the media, such as waiting for the appropriate `state` before calling the next method.

For example:

```
@Override
public void onStateChanged(MediaPlayer.PlayerState state, MediaPlayerNotification notification)
{
    switch (state) {
        // It is recommended that you call prepareToPlay() after receiving the INITIALIZED state.
        case INITIALIZED:
            _mediaPlayer.prepareToPlay();
            break;
        case PREPARING:
            showBufferingSpinner();
            break;
        case PREPARED:
            hideBufferingSpinner();
            . . . . .
    }
}
```

## MediaPlayer methods for accessing MediaResource information

The methods in the `MediaPlayerItem` class allow you to obtain information about the content stream represented by a loaded `MediaResource`.

Purpose	Method	Description
Ad tags	<code>List&lt;String&gt; getAdTags()</code>	Provides the list of ad tags that are used for the ad placement process.
Live stream	<code>boolean isLive();</code>	True if the stream is live; false if it is VOD.
DRM protected	<code>boolean isProtected();</code>	True if the stream is DRM protected.
	<code>List&lt;DRMMetadataInfo&gt; getDRMMetadataInfos();</code>	Lists all the DRM metadata objects discovered in the manifest.
Closed captions	<code>boolean hasClosedCaptions();</code>	True if closed-caption tracks are available.
	<code>List&lt;ClosedCaptionsTrack&gt; getClosedCaptionsTracks();</code>	Provides a list of available closed-caption tracks.
	<code>ClosedCaptionsTrack get SelectedClosedCaptionsTrack();</code>	Retrieves the closed caption track selected with <code>SelectClosedCaptionsTrack</code> .
Alternate audio tracks	<code>boolean hasAlternateAudio();</code>	<p>True if the stream has alternate audio tracks.</p> <p> <b>Note:</b> The main (default) audio track is also part of the alternate audio track list.</p> <p>The PSDK for Android considers the main audio track to be one of the items in the alternate audio track list. Because of this, the only case where <code>MediaPlayerItem.hasAlternateAudio</code> returns false is when the stream has no audio at all. If the content has only one audio track, this method returns true, and <code>MediaPlayerItem.getAudioTracks</code> returns a list with a single element (the default audio track).</p>
	<code>List&lt;AudioTrack&gt; getAudioTracks();</code>	Provides a list of available alternate audio tracks.

Purpose	Method	Description
	<code>AudioTrack.getSelectedAudioTrack();</code>	Retrieves the currently selected audio track selected with <code>SelectAudioTrack</code> .
Timed metadata	<code>boolean hasTimedMetadata();</code>	True if the stream has associated timed metadata.
	<code>List&lt;TimedMetadata&gt; getTimedMetadata();</code>	Provides a list of the timed metadata objects associated with the stream.
Multiple profiles (bit rates)	<code>boolean isDynamic();</code>	True if the stream is a multiple bit-rate (MBR) stream.
	<code>List&lt;Profile&gt; getProfiles();</code>	Provides a list of the associated profiles.
Trick play	<code>boolean isTrickPlaySupported();</code>	True if the player supports fast forward, rewind, and resume.
	<code>List&lt; Float&gt; getAvailablePlaybackRates ()</code>	Provides the list of available playback rates in the context of the trick-play feature.
Media resource	<code>MediaResource getResource();</code>	Returns the media resource associated with this item.

## Reuse or remove a MediaPlayer instance

You can terminate a `MediaPlayer` instance in two ways.

### Reset or reuse a MediaPlayer instance

Resetting a `MediaPlayer` instance returns it to its uninitialized state, which is called `IDLE` in `MediaPlayer.PlayerState`.

Resetting a `MediaPlayer` instance is useful in the following cases:

- When you want to reuse a `MediaPlayer` instance but need to load a different `MediaResource` (video content). Resetting allows you to reuse the `MediaPlayer` instance without the overhead of releasing resources, recreating the `MediaPlayer`, and reallocating resources. The `replaceCurrentItem` method automatically does these steps for you, without having to call the reset method.
- When the `MediaPlayer` is in an `ERROR` state and needs to be cleared. This is the only way to recover from the `ERROR` state.

1. Call `reset` to return the `MediaPlayer` instance to its uninitialized state:

```
void reset() throws IllegalStateException;
```

2. Use `MediaPlayer.replaceCurrentItem` to load another `MediaResource` (or, in the case of clearing an error, the same `MediaResource`).
3. When you receive the `PlaybackEventListener.onPrepared` callback, you can start the playback.

### Release a MediaPlayer instance and resources

Release a `MediaPlayer` instance and resources when you no longer need the `MediaResource`.

Releasing a `MediaPlayer` object deallocates the underlying hardware resources associated with this `MediaPlayer` object. Release a `MediaPlayer` when it is no longer useful. For several reasons, it is good practice to do this:

- Holding unnecessary resources can affect performance.

- Leaving an unneeded `MediaPlayer` object can lead to continuous battery consumption for mobile devices.
- Playback failure might occur for other applications if multiple instances of the same video-codec are not supported on a device.

Call the `release` method:

```
void release() throws IllegalStateException;
```

After the `MediaPlayer` instance is released, you can no longer use it. Calling any method of the `MediaPlayer` interface after it is released results in an `IllegalStateException` being thrown.

## Inspect the playback timeline

You can obtain a description of the timeline associated with the currently selected item that is being played by the PSDK. This is most useful when your application displays a custom scrub-bar control in which the content sections that correspond to ad content are identified.

The `PMPDemoApp` reference implementation for the Android PSDK library implements this as follows and as seen in the following screen shot.



1. Access the `Timeline` object in the `MediaPlayer` using the `getTimeline` method.

The `Timeline` class encapsulates the information related to the contents of the timeline associated with the media item that is currently loaded by the `MediaPlayer` instance. The `Timeline` class gives access to a read-only view of the underlying timeline. The `Timeline` class provides a getter method that provides an iterator through a list of `TimelineMarker` objects.

2. Iterate through the list of `TimelineMarkers` and use the returned information to implement your timeline.

A `TimelineMarker` object contains two pieces of information:

- The position of the marker on the timeline (in milliseconds)
- The duration of the marker on the timeline (in milliseconds)

3. Implement the listener callback interface `MediaPlayer.PlaybackEventListener.onTimelineUpdated` and register it with the `Timeline` object.

The `Timeline` object can then inform your application about any changes that may occur in the playback timeline by calling your `OnTimelineUpdated` listener.

```
// access the timeline object
Timeline timeline = mediaPlayer.getTimeline();
// iterate through the list of TimelineMarkers
Iterator<TimelineMarker> iterator = timeline.timelineMarkers();
while (iterator.hasNext()) {
    TimelineMarker marker = iterator.next();
    // the start position of the marker
    long startPos = marker.getTime();
    // the duration of the marker
    long duration = marker.getDuration();
}
```

## Configure buffering

You can set different buffering policies in the PSDK, consisting of the initial buffer time and the buffer time, depending on your needs, such as the device, the operating system, or the network conditions.

When the media player buffer reaches the initial buffer time, playback begins.

This improves the start-up time because the player does not wait for the entire buffer to fill before starting playback. Instead, after a few seconds are buffered, playback begins. While the video is being rendered, the player continues buffering new fragments. There is a risk, however, when network conditions are poor, that playback could be interrupted by periods of buffering, instead of having a long initial buffering time before starting playback (when the initial buffer time has a high value).

The player buffers fragments until it reaches the buffer time value. This is actually the minimum buffer length, so the actual buffer length can exceed this value.

A maximum buffer length, above which the buffer will not go, is calculated from the minimum length.

### Set buffering times

The `MediaPlayer` provides methods to set and get the buffer control parameters.

If you do not set the buffer control parameters before beginning playback, the media player defaults to 2 seconds for the initial buffer and 30 seconds for the buffer time.

1. Set up the `BufferControlParameters` object, which encapsulates the control parameters (the initial buffer time and play buffer time). This class provides two factory methods:

- `public static BufferControlParameters createSimple(long bufferTime)` (in this case the initial buffer time is equal to the buffer time)
- `public static BufferControlParameters createDual(long initialBuffer, long bufferTime)`

These methods throw an `IllegalArgumentException` if the parameters are not valid, such as when:

- The initial buffer time is lower than zero
- The initial buffer time is bigger than the buffer time

2. To set the buffer parameters:

```
void setBufferControlParameters(BufferControlParameters params)
```

3. To get the current buffer parameter values:

```
BufferControlParameters getBufferControlParameters()
```

If the AVE cannot set the specified values, the media player enters the ERROR state. The error code is SET\_BUFFER\_PARAMETERS\_ERROR.

For example, to set the initial buffer to 5 seconds and the buffer time to 30 seconds:

```
mediaPlayer.setBufferControlParameters(BufferControlParameters.createDual(5000, 30000));
```

The PMPDemoApp demonstrates this feature; use the application's settings to set the buffer values.

## Include advertising

The Primetime SDK for Android allows you to request ads for your live/linear and VOD content through its Primetime Ad Decisioning interface.

Ad Decisioning works with the PSDK to identify ad opportunities, resolve ads, and insert resolved ads into your video streams.

The following sections describe how to incorporate ads in your video content.

### Advertising requirements

Advertising and main video content must meet certain requirements when you incorporate ads into your video content.

- The HLS version of the advertising content must be no later than the HLS version of the main content.
- Target duration and any individual fragment duration of the ad must not exceed the target duration of the main content.
- Ads can be either multiplexed or not, with no restrictions, whether or not the main content is multiplexed.

### About ad insertion

The PSDK advertising workflow has three phases, and information about ad placement comes from two possible sources.

PSDK ad insertion includes ad resolving for VOD, live streaming, and linear streaming, along with ad tracking and ad playback. The PSDK makes the required requests to the ad server, receives information about ads for the specified content, and places the ads into the content.

The PSDK groups ads into *ad breaks*, each of which contains one or more ads that play in sequence. The PSDK inserts ads into the main content as members of an ad break.

The PSDK process for placing ads within ad breaks into your main content has three phases:

- Opportunity detection: The PSDK uses stream information to detect possible and desired locations for ads.
- Ad resolution: The PSDK communicates with an advertisement server to retrieve the ads to splice into the content.
- Ad placement: The PSDK loads the specified ads and places them in ad breaks on the content timeline at the specified locations and recomputes the virtual timeline if needed.

The PSDK can obtain the possible locations for ad placement in two ways:

- Use manifest metadata/cues

This is common for live/linear streams. The PSDK is responsible for detecting such metadata/cues, extracting the necessary information from them, and communicating with an advertising server to obtain the corresponding ads.

The PSDK usually splices in the resolved ads by replacing main content at the location indicated by the metadata/cues; otherwise, the client would drop further and further behind the actual live point.

- Use advertising server map

This is common for video-on-demand (VOD) streams. Usually, metadata about these streams are registered into the advertising server before playback. The PSDK retrieves the ad timeline and corresponding ads from the server. The PSDK usually splices the resolved ads by insertion into the main content as indicated by the server map.

By default, the PSDK uses manifest cues for live/linear streams and advertising server maps for VOD streams. However, to support full-event replay for live events, your application must take extra steps.

### VOD client ad resolving and insertion

For VOD content, the PSDK inserts ad breaks by splicing into the main content so that the timeline duration increases. Before playback, the PSDK resolves known ads, inserts ad breaks into the main content as described by a timeline returned from Ad Decisioning, and recomputes the virtual timeline if needed.

In VOD, the PSDK inserts ads as follows:

- Pre-roll: The PSDK inserts an ad break at the beginning of the content.
- Mid-roll: The PSDK inserts one or more ad breaks in the middle of the content.
- Post-roll: The PSDK appends an ad break to the end of the content.

After playback starts, no further changes can occur in the content. For example:

- No additional ad insertion occurs.
- Ad deletion is not supported, so you cannot delete built-in ads from the content to offer an ad-free experience.
- Ad replacement is not supported, so you cannot replace built-in ads with targeted ads.

### Live and linear client ad resolving and insertion

For live/linear content, the PSDK replaces part of the stream by substituting a chunk of main content with an ad break of the same duration so that the timeline duration remains the same.

Before and during playback, the PSDK resolves known ads, replaces parts of the main content with ad breaks of equal duration at positions specified by cue points in the stream defined by the manifest, and recomputes the virtual timeline if needed.

For live and linear video streaming, the PSDK inserts ads as follows:

- Pre-roll: The PSDK inserts an ad break at the beginning of the content.
- Mid-roll: The PSDK inserts one or more ad breaks in the middle of the content.

The PSDK accepts the ad break even if the duration is longer or shorter than the cue point replacement duration.

By default, the PSDK supports the following cue as a valid ad marker when resolving and placing ads:

- #EXT-X-CUE

This marker requires the metadata field DURATION in seconds and the cue's unique ID. For example:

```
#EXT-X-CUE:DURATION=27, ID= "..."
```

You can define and subscribe to additional cues (tags).

After playback starts, the video engine periodically refreshes the manifest file. The PSDK resolves any new ads and inserts them when a cue point is encountered in the live or linear stream defined in the manifest.

Each time after ads are resolved and inserted, the PSDK recomputes the virtual timeline and dispatches a `MediaPlayer.PlaybackEventListener.onUpdated` event to your application.

## Client ad tracking

The PSDK automatically tracks ads for VOD and live/linear streaming.

The PSDK uses notifications to inform your application about an ad's progress, such as when an ad begins and when it ends.

## Playback behavior with ads: Default and customized

The behavior of media playback is affected by seeking, pausing, DVR fast forward or rewind (trick-play mode), and the inclusion of advertising. The PSDK provides ways for you to override the default behavior.



**Note:** *The PSDK does not provide a way to disable seeking during ads. Adobe recommends that your application disable seeking during ads.*

### For live/linear:

- Resuming playback after a pause results in continuing playback with the buffered content at the time of the pause operation. If the resuming position is still within the playback range, the playback should be continuous. Otherwise, the PSDK jumps to the new live point. Also, you can choose a different playback point by performing a seek operation.
- The PSDK resolves ads between cues after the position the application enters live playback. Playback begins after the first cue is resolved. The default value for entering live playback is the client live point, but you can choose a different position. All cues before the initial position are resolved after the application performs a seek in the DVR window.

The following table describes how the PSDK handles ads and ad breaks during playback.

Description	Default PSDK behavior policy	Customization available through <code>AdBreakPolicySelector</code>
During normal play, an ad break is encountered.	<ul style="list-style-type: none"> <li>• For live/linear, plays the ad break, whether or not it has already been watched.</li> <li>• For VOD, plays the ad break and then removes it.</li> </ul>	Specify a different policy for the ad break using <code>selectPolicyForAdBreak</code> .
Your application seeks forward over ad break(s) into main content.	Plays the last unwatched ad break that was skipped over, then resumes playback at the desired seek position upon break(s) playback completion.	Choose which of the skipped breaks to play using <code>selectAdBreaksToPlay</code> .
Your application seeks backward over ad break(s) into main content.	No ad break plays and skips to the desired seek position.	Choose which of the skipped breaks to play using <code>selectAdBreaksToPlay</code> .
Your application seeks forward into an ad break.	Plays from the beginning of the ad in which the seek ended.	Specify a different ad policy for the ad break and for the specific ad where the seek ended using <code>selectPolicyForSeekIntoAd</code> .

Your application seeks backward into an ad break.	Plays from the beginning of the ad in which the seek ended.	Specify a different ad policy for the ad break and for the specific ad in which the seek ended using <code>selectPolicyForSeekIntoAd</code> .
Your application seeks forward or backward over watched ad break(s) into main content.	If the last ad break skipped has already been watched, skips to the user-selected seek position.	Select which of the skipped breaks to play using <code>selectAdBreaksToPlay</code> , and determine which have already been watched using the <code>isWatched</code> property.
Your application seeks forward or backward over one or more ad breaks and drops into a watched ad break.	Skips the ad break and seeks to the position immediately following the ad break.	Specify a different ad policy for the ad break (with the watched status set to true) and for the specific ad where the seek ended using <code>selectPolicyForSeekIntoAd</code> .
Your application goes into trick-play (DVR mode). Play rate can be negative (rewind), between 0 and 1 (slow forward) or greater than 1 (fast forward).	Skips all ads during fast forward or rewind, plays the last break skipped after trick play ends, and skips to the user-selected trick play position upon break(s) playback completion.	Select which of the skipped breaks to play after trick play ends using <code>selectAdBreaksToPlay</code> .



**Note:** By default, the PSDK marks an ad break as watched immediately upon entering the first ad in the ad break.

## Customize playback with ads

When playback reaches or passes an ad break or ends within an ad break, the PSDK defines certain default behavior for the positioning of the current playhead. You can override the default behavior through the `AdBreakPolicySelector` class.

The default behavior might vary depending on whether the user passed the ad break during normal playback or by seeking in a video or repositioning it with fast forward or rewind (trick play).

Here are some ideas for possible ad-playback behavior customization.

- Save the position in a video where the user stopped watching and resume the user at the same position in a future session.
- If an ad break is presented to the user, display no more ads for some number of minutes even if the user seeks to a new position.
- If the content fails to play (live or VOD) after a few minutes, restart the stream or fail over to a different source for the same content.

On the fail-over playback session, you might want to disable pre-roll and/or mid-roll ads to allow the user to skip ads and resume to the previous failed position. The PSDK provides methods to enable skipping pre-roll and mid-roll ads.

## Advertising API elements for playback

The PSDK provides classes and methods with which you can customize the playback behavior of content that contains advertising.

The following API elements are useful for customizing playback:

API element	Provides
AdvertisingMetadata	adBreakAsWatched property. Specifies the policy for marking an ad break as having been watched.
MediaPlayer	Get the local time of the playback; seek to that time.   <b>Note:</b> <i>getLocalTime</i> in <i>MediaPlayer</i> returns the current time relative to the original content (without dynamically spliced ads). <i>getLocalTime</i> in <i>AdBreak</i> returns the start time of the break relative to the original content.
AdBreakPolicy	Enumerates possible playback policies for ad breaks.
AdPolicy	Enumerates possible playback policies for ads.
AdPolicySelector	Interface that allows customization of the PSDK ad behavior.
DefaultAdPolicySelector	Class that implements the default PSDK behavior. Your application can override this class to customize one or more of the default behaviors without having to implement the full interface.
MediaPlayer	<ul style="list-style-type: none"> <li>• <i>getLocalTime</i>. The local time of the playback excluding the placed ad breaks.</li> <li>• <i>seekToLocalTime</i>. Seeks to a local time in stream.</li> <li>• <i>convertToLocalTime</i>. Converts a virtual position on the timeline to the local position.</li> </ul>

## Use the default playback behavior

You can choose to use default ad behaviors.

To use default behaviors:

- If you have implemented your own *AdvertisingFactory* class, then return null for *createAdPolicySelector*.
- If you do not have custom implementation for the *AdvertisingFactory* class, then the PSDK by default uses a new instance of *DefaultAdPolicySelector*.

## Set up for customized playback

You can customize or override ad behaviors supported by PSDK in two ways. In either case, you must register the ad policy instance with the PSDK.

To customize ad behaviors, your application can either:

- Implement the `AdPolicySelector` interface and implement all its methods.

This is recommended if you need to override all the default ad behaviors.

- Extend the `DefaultAdPolicySelector` class and provide implementations for only those behaviors that require customization.

This is recommended if you need to override only some of the default behaviors.

In both cases, do the following.

1. Assign the policy instance to be used by the PSDK through the advertising factory.



**Note:** Any custom ad policy that is registered at the beginning of playback is cleared when the `MediaPlayer` instance is deallocated. Your application must register a policy selector instance every time a new playback session is created.

For example:

```
// create a Custom Advertising Factory
private AdvertisingFactory createCustomAdvertisingFactory() {
    return new AdvertisingFactory() {
        ...
        @Override
        public AdPolicySelector createAdPolicySelector(MediaPlayerItem mediaPlayerItem)
        {
            return new CustomAdPolicySelector(mediaPlayerItem);
        }
        ...
    };
}

// register the custom advertising factory with media player
advertisingFactory = createCustomAdvertisingFactory();
mediaPlayer.registerAdClientFactory(advertisingFactory);
```

2. Implement any customizations.

## Skip ad breaks for a certain period of time

Default PSDK behavior is to force an ad break to play when the user seeks over an ad break. You could customize the behavior to skip an ad break if the time elapsed from a previous break completion is within a certain number of minutes.

The following example of a customized ad policy selector skips any ads within the next five minutes (wall clock time) after a user has watched an ad break.

1. Save the current system time whenever the user finishes watching an ad break.

```
@Override
public void onAdBreakComplete(AdBreak adBreak) {
    ...
    if (isShouldPlayUpcomingAdBreakRuleEnabled())
        CustomAdPolicySelector.setLastAdBreakPlayedTime(System.currentTimeMillis());
    ...
}
```

## 2. Extend the default ad policy selector to override the default behavior.

```

package com.adobe.mediacore.sample.advertising;

import com.adobe.mediacore.DefaultAdPolicySelector;
import com.adobe.mediacore.MediaPlayerItem;
import com.adobe.mediacore.timeline.advertising.AdBreakPlacement;
import com.adobe.mediacore.timeline.advertising.AdBreakPolicy;
import com.adobe.mediacore.timeline.advertising.AdPolicyInfo;
import com.adobe.mediacore.timeline.advertising.PlacementInformation;
import java.util.ArrayList;
import java.util.List;

public class CustomAdPolicySelector extends DefaultAdPolicySelector {

    private final long MIN_BREAK_INTERVAL = 300000; // 5 minutes interval for next ad break
    to be played
    private MediaPlayerItem _mediaPlayerItem;
    private static long _lastAdBreakPlayedTime;

    public CustomAdPolicySelector(MediaPlayerItem mediaPlayerItem) {
        super(mediaPlayerItem);
        _mediaPlayerItem = mediaPlayerItem;
        _lastAdBreakPlayedTime = 0;
    }

    @Override
    public AdBreakPolicy selectPolicyForAdBreak(AdPolicyInfo adPolicyInfo) {
        if (shouldPlayAdBreaks()) {
            AdBreakPlacement adBreakPlacement = adPolicyInfo.getAdBreakPlacements().get(0);

            // This condition will remove the pre-roll ad from live stream after watching
            if
(adBreakPlacement.getPlacement().getType().equals(PlacementInformation.Type.PRE_ROLL) &&
_mediaPlayerItem.isLive()) {
                return AdBreakPolicy.REMOVE_AFTER_PLAY;
            }
            if
(adBreakPlacement.getPlacement().getType().equals(PlacementInformation.Type.PRE_ROLL)) {
                return AdBreakPolicy.PLAY;
            }
            // This condition will remove every ad break that has been watched once. Comment
            this section if you want to play watched ad breaks again.
            if (adBreakPlacement.getAdBreak().isWatched()) {
                return AdBreakPolicy.SKIP;
            }
            return AdBreakPolicy.REMOVE_AFTER_PLAY;
        }
        return AdBreakPolicy.SKIP;
    }

    @Override
    public List<AdBreakPlacement> selectAdBreaksToPlay(AdPolicyInfo adPolicyInfo) {
        if (shouldPlayAdBreaks()) {
            List<AdBreakPlacement> adBreakPlacements = adPolicyInfo.getAdBreakPlacements();

            if (adBreakPlacements != null) {
                int size = adBreakPlacements.size();
                // Seek Forward Condition
                if (size > 0 && adPolicyInfo.getCurrentTime() <= adPolicyInfo.getSeekToTime())
            {
                List<AdBreakPlacement> adBreaks = new ArrayList<AdBreakPlacement>();
                AdBreakPlacement adBreakPlacement = adBreakPlacements.get(0);
                // Resume logic - This will be helpful in resuming the content from last
                saved playback session, and just play the pre-roll ad
                if(adPolicyInfo.getCurrentTime() == 0){

if(adBreakPlacement.getPlacement().getType().equals(PlacementInformation.Type.PRE_ROLL) &&
!adBreakPlacement.getAdBreak().isWatched()) {
                    //comment this line if you just need to seek to the user's last

```

```

known position without playing pre-roll ad. ZD#820
        adBreaks.add(adBreakPlacement);
        return adBreaks;
    } else{
        return null;
    }
} else {
    adBreakPlacement = adBreakPlacements.get(size -1);
    if (!adBreakPlacement.getAdBreak().isWatched()) {
        adBreaks.add(adBreakPlacement);
        return adBreaks;
    }
}
// Seek backward condition
} else if (size > 0 && adPolicyInfo.getCurrentTime() >
adPolicyInfo.getSeekToTime()) {
    List<AdBreakPlacement> adBreaks = new ArrayList<AdBreakPlacement>();
    AdBreakPlacement adBreakPlacement = adBreakPlacements.get(0);
    if(!adBreakPlacement.getAdBreak().isWatched()) {
        adBreaks.add(adBreakPlacement);
        return adBreaks;
    } else {
        return adBreaks;
    }
}
}
}
return null;
}
}

public static void setLastAdBreakPlayedTime(long lastAdBreakPlayedTime) {
    _lastAdBreakPlayedTime = lastAdBreakPlayedTime;
}

// used to decide if upcoming AdBreak should be played or not, Customer use case #2 - Five
minute AdBreak rule
private boolean shouldPlayAdBreaks(){
    long currentTime = System.currentTimeMillis();
    if (_lastAdBreakPlayedTime <= 0) {
        return true;
    }
    if (_lastAdBreakPlayedTime > 0 && (currentTime - _lastAdBreakPlayedTime) >
MIN_BREAK_INTERVAL) {
        return true;
    }
    // return false for not playing Ad if this Ad occurs with 5 minutes of last Ad
    playback
    return false;
}
}
}

```

### Save the video position and resume later

You can save the current playback position in a video and resume the user at the same position in a future session.

Dynamically inserted ads differ between user sessions, so saving the position *with* spliced ads will refer to a different position in a future session. The PSDK provides methods to retrieve the playback position, ignoring spliced ads.

Your application can then seek to a specific position while ignoring spliced ads.

1. When the user quits a video, retrieve and save the position in the video (excluding duration of ads).

Ad breaks can vary in each session due to ad patterns, frequency capping, and so on. Therefore, the current time of the video in one session could be different in a future session. When saving a position in the video, retrieve

the local time (which excludes the duration of all ads up to that position), which you can save either on the device or in a database on the server.

For example, if the user is at the 20th minute of the video and this includes five minutes of ads, current time will be 1200 seconds, while local time at this position will be 900.



**Note:** Local time and current time are the same for live/linear streams. In this case, `convertToLocalTime` has no effect. For VOD, local time remains unchanged while ads play.

```
// Save the user session when player activity stops
@Override
public void onStop(){
    super.onStop();
    ...
    prefs =
PreferenceManager.getDefaultSharedPreferences(getActivity().getApplicationContext());
    SharedPreferences.Editor editor = prefs.edit();
    editor.putLong(LAST_LOCAL_TIME, _mediaPlayer.getLocalTime()); //
get the local time where stream stopped playing and save it in System preferences
    editor.putString(LAST_MEDIA_RESOURCE, _contentInfo.toMediaResource().getUrl());
    editor.commit();
    ...
}
```

## 2. Restore the user session when player activity resumes.

```
@Override
public void onResume() {
    super.onResume();
    ...
    prefs =
PreferenceManager.getDefaultSharedPreferences(getActivity().getApplicationContext());
    if(prefs.getString(LAST_MEDIA_RESOURCE,
"nil").equals(_contentInfo.toMediaResource().getUrl())){
        _lastKnownLocalTime = prefs.getLong(LAST_LOCAL_TIME, 0); // get the last local
time saved in system preferences
        if(_lastKnownLocalTime > 0){
            _shouldResumePlayback = true;
        }
    }
    ...
}
```

## 3. Resume the user at the same position.

To resume the user to the same position saved from a previous session, use `seekToLocalTime`. Call this method *only* with local time values. Incorrectly calling it with the current time results in incorrect behavior. (To seek to the current time, use `seek`.) When your application receives the `onStateChanged` status change event, seek to the saved local time.

## 4. Provide the ad breaks to present to the user through the ad policy selector interface.

Implement a custom ad policy selector (by extending the default ad policy selector). Provide the ad breaks that must be presented to the user by implementing `selectAdBreaksToPlay`. This method includes a pre-roll ad break and the mid-roll ad breaks before the local time position. Your application can decide to play a pre-roll ad break and resume to the specified local time, play a mid-roll ad break and resume to the specified local time, or play no ad breaks.

## Metadata for ad insertion

Ad providers, such as the Adobe Ad Decisioning server, require configuration values to enable connection to the provider and for the ad resolver to work.

The PSDK includes the Ad Decisioning library. For your content to include advertising from the Ad Decisioning server, your application must provide the PSDK with the required `AuditudeSettings` information for connecting to the Ad Server:

- The `mediaID`, which identifies the specific content to play
- A default `mediaID`

Although optional, using this is highly recommended. It enables ads to be served even when your request to the ad server is invalid, content is incorrectly configured, Ad Decisioning is experiencing delays in propagating the data, or one of the Ad Decisioning back-end processes is malfunctioning or unavailable.

- The `zoneID`, assigned by Adobe to identify your company or Web site
- Ad server domain, specifies the domain of your assigned ad server
- Other targeting parameters

You can include other targeting parameters depending on your needs and on the ad provider.

### Set up metadata for ad insertion

Use the helper class `AuditudeSettings`, which extends the `MetadataNode` class, to set up Ad Decisioning metadata.

Advertising metadata is contained within the `MediaResource.Metadata` metadata property. When starting the playback of a new video, your application is responsible for setting the correct advertising metadata.

1. Build the `AuditudeSettings` instance.

```
AuditudeSettings auditudeSettings = new AuditudeSettings();
```

2. Set the Ad Decisioning `mediaID`, `zoneID`, domain, and, optionally, targeting parameters.

```
auditudeSettings.setZoneId("yourZoneId");
auditudeSettings.setMediaId("yourVideoId");
auditudeSettings.setDefaultMediaId("defVideoId");
auditudeSettings.setDomain("yourAuditudeDomain");
Metadata targetingParameters = new MetadataNode();
targetingParameters.setValue("desired_param", "desired_value");
auditudeSettings.setTargetingParameters(targetingParameters);
MetadataNode result = new MetadataNode();
result.setNode(DefaultMetadataKeys.AUDITUDE_METADATA_KEY.getValue(), auditudeSettings);
```

3. Create a `MediaResource` instance using the media stream URL and the previously created advertising metadata.

```
MediaResource mediaResource =
MediaResource.createFromUrl("http://example.com/media/test_media.m3u8", MetadataNode);
```

4. Load the `MediaResource` object through the `MediaPlayer.replaceCurrentItem(android)` method.

The `MediaPlayer` starts loading and processing the media stream manifest.

5. When the `MediaPlayer` transitions to the `INITIALIZED` status, get the media stream characteristics in the form of a `MediaPlayerItem` instance through the `MediaPlayer.getCurretItem` method.
6. Query the `MediaPlayerItem` instance to see whether the stream is live, whether it has alternate audio tracks, or whether it is protected.
7. Call `MediaPlayer.prepareToPlay` to start the advertising workflow.

After the ads have been resolved and placed on the timeline, the `MediaPlayer` transitions to the `PREPARED` state.

8. Call `MediaPlayer.play` to start the playback.

The PSDK now includes ads when your media plays.

### Enable ads in full-event replay

Full-event replay (FER) is a VOD asset that acts, in terms of ad insertion, as a live/DVR asset, so your application must take steps to ensure that ads are placed correctly.

For live content, the PSDK uses the metadata/cues presented in the manifest to determine where to place ads. However, sometimes live/linear content looks like VOD content, for example, when live content completes, an EXT-X-ENDLIST tag is appended to the live manifest. For HLS, the presence of the EXT-X-ENDLIST tag means that the stream is a VOD stream, and there is no way for the PSDK to automatically differentiate it from a normal VOD stream so that it can insert ads correctly.

Therefore, your application must notify the PSDK of this situation by specifying the `AdSignalingMode`.

For a FER stream, you do not want the Ad Decisioning server to provide the list of ad breaks that need to be inserted on the timeline prior to beginning playback, as would be usual for VOD. Instead, by specifying a different signaling mode, the PSDK reads all the cue points from the FER manifest and goes to the ad server for each cue point to request an ad break (similar to live/DVR).

Besides each request associated with a cue point, the PSDK makes an additional ad request for pre-roll ads.

1. Obtain from an external source (vCMS, etc.) the signaling mode that should be used.
2. Create advertising-related metadata as usual.
3. Specify the `AdSignalingMode` using `AdvertisingMetadata.setSignalingMode` if the default behavior must be overridden.

Valid values are `DEFAULT`, `SERVER_MAP`, and `MANIFEST_CUES`.

You must set the ad signaling mode before calling `prepareToPlay`. After the PSDK has started resolving and placing ads on the timeline, any change to the ad signaling mode is ignored. The recommended way is to set it when creating the advertising metadata for the resource (when creating the `AuditudeSettings` object).

4. Continue to playback as usual.

```
AuditudeSettings auditudeSettings = new AuditudeSettings();
auditudeSettings.setSignalingMode(AdSignalingMode.MANIFEST_CUES);
auditudeSettings.setDomain("your-auditude-domain");
auditudeSettings.setZoneId("your-auditude-zone-id");
auditudeSettings.setMediaId("your-media-id");
// set additional targeting parameters or custom parameters
MetadataNode mediaMetadata = new MetadataNode();
mediaMetadata.setNode(DefaultMetadataKeys.AUDITUDE_METADATA_KEY.getValue(), auditudeSettings);
MediaResource mediaResource = MediaResource.createFromUrl("your-stream-url", mediaMetadata);
final MediaPlayer mediaPlayer = DefaultMediaPlayer.create(getApplicationContext());
mediaPlayer.addListener(MediaPlayer.Event.PLAYBACK, new MediaPlayer.PlaybackEventListener()
{
    @Override
    public void onPrepared() {
        // the player is prepared and all ads have been placed
        mediaPlayer.play();
    }
    @Override
    public void onUpdated() {
    }
    @Override
    public void onTimedMetadata(TimedMetadata timedMetadata) {
    }
    @Override
    public void onTimelineUpdated() {
    }
}
```

```

@Override
public void onPlayStart() {
    // the playback has started we can update UI control
    // for example: remove starting/buffering slate if needed
}
@Override
public void onPlayComplete() {
    // playback has reached the end of stream ( ads included )
    // if we want to rewind we can call
    mediaPlayer.seek(mediaPlayer.getSeekableRange().getBegin());
}
@Override
public void onStateChanged(MediaPlayer.PlayerState state, MediaPlayerNotification notification)
{
    if (state == MediaPlayer.PlayerState.INITIALIZED) {
        mediaPlayer.prepareToPlay();
    }
}
@Override
public void onSizeAvailable(long height, long width) {
    // video size was update
    // you can use height and width to resize the media player view
}
});
mediaPlayer.replaceCurrentItem(mediaResource);

```

### Ad signaling mode

The ad signaling mode specifies from where the video stream should get advertising information.

Valid values are DEFAULT, SERVER\_MAP, and MANIFEST\_CUES.

The following table describes the effect of AdSignalingMode values for various HLS stream types.

	Default	Manifest cues	Ad server map
Video on Demand	<ul style="list-style-type: none"> <li>• Uses server map for placement detection</li> <li>• Ads are inserted</li> </ul>	<ul style="list-style-type: none"> <li>• Uses in-stream cues for placement detection</li> <li>• Pre-roll ads are inserted in the main stream</li> <li>• Mid-rolls ads replace main stream</li> </ul>	<ul style="list-style-type: none"> <li>• Uses server map for placement detection</li> <li>• Ads are inserted</li> </ul>
Live/linear	<ul style="list-style-type: none"> <li>• Uses manifest cues for placement detection</li> <li>• Ads replace main stream</li> </ul>	<ul style="list-style-type: none"> <li>• Uses in-stream cues for placement detection</li> <li>• Ads replace main stream</li> </ul>	Not supported

### Clickable ads

The PSDK provides you with information so that you can act upon click-through ads. As you create your player UI, you are responsible for deciding how to respond when a user clicks on a clickable ad.

For the Android PSDK, only linear ads can be clickable.

### Respond to clicks on ads

When a user clicks on an ad or a related button, your application is responsible for responding. The PSDK provides you with information about the destination URL for the click.

1. Set up an event listener for the PSDK to provide you with click-through information.
  - a) Register an `AdPlaybackEventListener.onAdClick`.  
When an end user clicks a banner, the PSDK dispatches this notification, including information about the destination for the click.
2. Monitor user interactions on clickable ads.
3. When the user touches or clicks the ad or button, notify the PSDK.
  - a) To do this, call `notifyClick` on the `MediaPlayerView`.
4. Listen for the `onAdClick(AdBreak adBreak, Ad ad, AdClick adClick)` event from the PSDK.
5. Retrieve the click-through URL and related information.
  - a) Use the getter methods for the `AdClick` instance.
6. Pause the video.
7. Use the click-through information to display the ad click-through URL and any related information.  
For example, you could display it in one of the following ways:
  - Open the click-through URL in a browser within your application.  
On desktop platforms, the video ad playback area is typically used for invoking click-through URLs upon user clicks.
  - Redirect the user to their external mobile web browser.  
On mobile devices, the video ad playback area is used for other functions, such as hiding and showing controls, pausing playback, expanding to full screen, and so on. Therefore, on mobile devices, a separate view, such as a sponsor button, is usually presented to the user as a means to launch the click-through URL.
8. Close the browser window in which the click-through information is displayed and resume playing the video.

For example:

```
private final MediaPlayer.AdPlaybackEventListener _adPlaybackEventListener =
    new MediaPlayer.AdPlaybackEventListener() {
        ...
    }
@Override
public void onAdStart(AdBreak adBreak, Ad ad) {
    //TODO: Implement external tracking logic
    //TODO: Notify the user that an ad has started by an UI update
    // Notify the user that the ad is clickable
    if (ad.isClickable()) {
        //TODO: Modify the UI to give the user the possibility to
            choose to click upon the primary asset
    }
}
@Override
public void onAdClick(AdBreak adBreak, Ad ad, AdClick adClick) {
    // Open the native browser to display the click through url
    Uri uri = Uri.parse(adClick.getUrl());
    Intent intent = new Intent(ACTION_VIEW, uri);
    try {
        startActivity(intent);
    } catch (Exception e) {
        // Log exception
    }
}
```

### Separate the clickable ad process

It is good practice to separate your player's UI logic from the process that manages ad clicks.

One way to do this is to implement multiple Fragments for an Activity.

1. Implement one fragment to contain the MediaPlayer (to be responsible for video playback).

This Fragment should call notifyClick.

```
public class PlayerFragment extends SherlockFragment {
    ....
    public void notifyAdClick () {
        _mediaPlayer.getView().notifyClick();
    }
    ....
}
```

2. Implement a different fragment to display a UI element that indicates that an ad is clickable, monitor that UI element, and communicate user clicks to the fragment that contains the MediaPlayer.

This Fragment should declare an interface for fragment communication. The Fragment captures the interface implementation during its onAttach lifecycle method and can then call the Interface methods to communicate with the Activity.

```
public class PlayerClickableAdFragment extends SherlockFragment {
    private ViewGroup viewGroup;
    private Button button;
    OnAdUserInteraction callback;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
    savedInstanceState) {
        // the custom fragment is defined by a custom button
        viewGroup = (ViewGroup) inflater.inflate(R.layout.fragment_player_clickable_ad, container,
        false);
        button = (Button) viewGroup.findViewById(R.id.clickButton);
        // register a click listener to detect user interaction
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // send the event back to the activity
                callback.onAdClick();
            }
        });
        viewGroup.setVisibility(View.INVISIBLE);
        return viewGroup;
    }
    public void hide() {
        viewGroup.setVisibility(View.INVISIBLE);
    }
    public void show() {
        viewGroup.setVisibility(View.VISIBLE);
    }
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // attaches the interface implementation
        // if the container activity does not implement the methods from the interface an exception
        // will be thrown
        try {
            callback = (OnAdUserInteraction) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString()
            + " must implement OnAdUserInteraction");
        }
    }
    // user defined interface that allows fragment communication
    // must be implemented by the container activity
    public interface OnAdUserInteraction {
        public void onAdClick();
    }
}
```

```
}
}
```

## Pause and resume playback

When your application responds to clicks on ads, it should pause playback of the main video content.

Override the `onPause` and `onResume` from the Android Activity.

```
@Override
public void onResume() {
    super.onResume();
    requestAudioFocus();
    if (_lastKnownStatus == MediaPlayer.PlayerState.PAUSED) {
        _mediaPlayer.play();
    }
}
...
@Override
public void onPause() {
    super.onPause();
    if (_mediaPlayer != null) {
        if (_mediaPlayer.getStatus() == MediaPlayer.PlayerState.PLAYING ||
            _mediaPlayer.getStatus() == MediaPlayer.PlayerState.PAUSED) {
            _savedPlayerState = _mediaPlayer.getStatus();
            _lastKnownTime = _mediaPlayer.getCurrentTime();
        }
        if (_mediaPlayer.getStatus() == MediaPlayer.PlayerState.PLAYING) {
            _mediaPlayer.pause();
            _lastKnownStatus = MediaPlayer.PlayerState.PAUSED;
        }
    }
}
abandonAudioFocus();
}
```

## Repackage third-party ads

Some third-party ads cannot be stitched into the HLS content stream due to incompatible video formats. To address this situation, Primetime Ad Insertion and the Android PSDK provide third-party creative repackaging for progressive-download MP4 videos.

In some situations, you might need to incorporate ads (creatives) that are transcoded and served by a third party into your ad insertion workflow. For example, these could be ads that are served by an agency ad server, by your inventory partner, or by an ad network.

If a requested ad is available only as an MP4, the PSDK skips that ad and issues a request to the Primetime Ad Insertion back end to repackage the ad as HLS so that a compatible version will be available the next time that the ad is encountered. The back end generates multiple-bit-rate HLS renditions of the ad and stores these on the Primetime CDN. Then, the next time the PSDK receives an ad response that points to that ad, the PSDK uses the HLS version from the Primetime CDN.

To enable this optional feature, contact your Adobe representative.

## Configure and use custom HLS tags

Media streams can carry additional metadata in the form of tags contained within the playlist/manifest file, such as to indicate the placement of advertising. You can specify custom tag names and be notified when certain tags appear in the manifest file.

The PSDK provides out-of-the-box support for specific #EXT advertising tags. Your application could use custom tags for enhancing the advertising workflow or for supporting blackout scenarios. To support such advanced workflows, the PSDK allows you to specify additional tags in the manifest beyond the standard tags, and to receive notification when these tags appear in the manifest file by subscribing to the tags. You can subscribe to custom tags both for VoD and live/linear streams.

### Overview of custom tags

Here is an example of a customized VOD asset:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:7

#EXT-X-ASSET:AID=10

#EXTINF:9.9766,
seg1.ts

#EXTINF:9.9766,
seg2.ts

#EXTINF:9.9766,
seg3.ts

#EXT-X-AD:DURATION=10
#EXTINF:9.9766,
seg4.ts

#EXTINF:9.9766,
seg5.ts

#EXT-X-ENDLIST
```

Your application could set up the following:

- To be notified when #EXT-X-ASSET tags, or any other set of custom tag names to which you have subscribed, are present in the file.
- To have ad insertion occur when an #EXT-X-AD tag, or any other custom tag name, is found in the stream.

### Config class methods for tags

You can configure custom tag names in the PSDK globally with the `PSDKConfig` class.

The PSDK applies the global configuration automatically to any media stream that does not specify a stream-specific configuration.

`PSDKConfig` exposes these methods for managing the custom tags:

<b>Subscribe to specific custom tags</b>	
<code>public static String[] getSubscribedTags()</code>	Retrieves the current list of subscribed tags.
<code>public static void setSubscribedTags(String[] tags);</code>	<p>Sets the list of subscribed tags that will be exposed to the application.</p> <p>Your application is also automatically subscribed to all tags transmitted through <code>setAdTags</code>.</p>
<b>Customize the ad tags used by the default opportunity detector</b>	

<code>public static String[] getAdTags();</code>	Retrieves the current list of ad tags.
<code>public static void setAdTags(String[] tags);</code>	Sets the list of ad tags that will be used by default opportunity generator.

Note the following:

- The setter methods do not allow the tags parameter to contain null values. If encountered, the PSDK throws an `IllegalArgumentException`.
- The custom tag name must contain the # prefix. For example, #EXT-X-ASSET is a correct custom tag name; EXT-X-ASSET is incorrect.
- You cannot change the configuration after the media stream has been loaded.

### Timed metadata class

When the PSDK detects a subscribed tag within the playlist/manifest, it automatically tries to process it and expose it in the form of a `TimedMetadata` object. The class provides the following elements.

Property	Type	Description
id	long	Unique identifier of the timed metadata. Usually is extracted from the cue/tag ID attribute if present. Otherwise, a unique random value is provided. Use <code>getId</code> .
metadata	Metadata	The processed/extracted information from the playlist/manifest custom tag. Use <code>getMetadata</code> .
name	String	The name of the timed metadata. If the type is TAG, then the value represents the cue/tag name. If the type is ID3, then this is null. Use <code>getName</code> .
time	long	The time position, in milliseconds, relative to the start of the main content where this timed metadata is present in the stream. Use <code>getTime</code> .
type	Type	The type of the timed metadata. Use <code>getType</code> . <ul style="list-style-type: none"> <li>• TAG - indicates that the timed metadata was created from a tag contained within the playlist/manifest</li> <li>• ID3 - indicates that the timed metadata was created from an ID3 tag contained within the media stream</li> </ul>

Note the following:

- The PSDK automatically extracts the attributes list into key-value pairs and stores them in the metadata property.
- The PSDK automatically strips leading and trailing quotes (") from the values.
- If the extraction fails (due to a custom tag format), then the metadata property will be empty and it is your application's responsibility to extract the actual information. No error is thrown in this case.

Element	Description
<code>public enum Type { TAG, ID3 }</code>	Possible types for the timed metadata.
<code>public TimedMetadata(Type type, long time, long id, String name, Metadata metadata);</code>	Default constructor (time is the local stream time).
<code>public long getTime();</code>	The time position, relative to the start of the main content, where this metadata was inserted into the stream.

Element	Description
<code>public Metadata getMetadata();</code>	The metadata inserted into the stream.
<code>public Type getType();</code>	Returns the type of the timed metadata.
<code>public long getId();</code>	Returns the ID extracted from the cue/tag attributes if present. Otherwise, a unique random value is provided.
<code>public String getName();</code>	Returns the name of the cue (usually the HLS tag name).

### Subscribe to custom ad tags

The PSDK prepares `TimedMetadata` objects for subscribed tags every time they are encountered in the content manifest. You must subscribe to the tags before the playback starts.

To subscribe with the PSDK to be notified about custom tags within HLS manifests:

Set the custom ad tag names globally by passing an array that contains the custom tags to `setSubscribedTags` in `PSDKConfig`.

Include the `#` prefix when working with HLS streams.

For example:

```
String[] array = new String[3];
array[0] = "#EXT-X-ASSET";
array[1] = "#EXT-X-BLACKOUT";
array[2] = "#EXT-OATCLS-SCTE35";
PSDKConfig.setSubscribedTags(array);
```

### Add listeners for timed-metadata notifications

To receive notifications about tags in the manifest, implement the appropriate event listener(s).

You can monitor timed metadata by listening for the following events, which notify your application of related activity:

- `onTimedMetadata`: Every time that a unique subscribed tag is identified during parsing of the content, the PSDK prepares a new `TimedMetadata` object and dispatches this event. The object contains information about the tag name (what you subscribed to), the local time in the playback where this tag will appear, and other data.

Listen for events:

```
private final MediaPlayer.PlaybackEventListener playbackEventListener =
    new MediaPlayer.PlaybackEventListener() {
        @Override
        public void onTimedMetadata(TimedMetadata timedMetadata) {
            PMPDemoApp.logger.e(LOG_TAG + "#onTimedMetadata",
                "New timed metadata. Name " +
                timedMetadata.getName() + ", time " + timedMetadata.getTime() + ".");
        }
        ...
    };
...
mediaPlayer.addListener(MediaPlayer.Event.PLAYBACK, playbackEventListener);
```

If you are also interested in ID3 metadata, they use the same `onTimedMetadata` listener to indicate the presence of an ID3 tag. This should not cause any confusion, however, because you can use the `TimedMetadata`'s `type` property to differentiate between TAG and ID3.

## Store timed-metadata objects as they are dispatched

Your application is responsible for using the appropriate `TimedMetadata` objects at the appropriate times.

During content parsing, which happens prior to playback, the PSDK identifies subscribed tags and notifies your application about them. The time associated with each `TimedMetadata` is the local time on the playback timeline.

Your application must:

1. Keep track of the current playback time.
  2. Match the current playback time against the already-dispatched `TimedMetadata` objects.
  3. Use the `TimedMetadata` where the start time equals the current local playback time.
- The following example shows how to save `TimedMetadata` objects in an `ArrayList`.

```
private List<TimedMetadata> _timedMetadataList = new ArrayList<TimedMetadata>();
...
public void onTimedMetadata(TimedMetadata timedMetadata) {
    ...
    if (timedMetadata.getName().equalsIgnoreCase("#EXT-X-CUE")) {
        _timedMetadataList.add(timedMetadata);
    }
    ...
}
```

## Use timed metadata at the appropriate time

Use `TimedMetadata` when the current playback time matches the start time.

Use the saved `ArrayList` from the previous topic to actually use these saved `TimedMetadata` objects during playback.

1. Run a timer and query current playback time repeatedly.
2. Find all the `TimedMetadata` objects whose start time values match the current playback time. You can use these objects to perform various actions.

When checking whether the current playback time matches any `TimedMetadata` objects, put this as a condition as well: `shouldTriggerSubscribedTagEvent`. The timeline might change due to various ad behaviors. For example, one or more ad breaks might be moved from their original positions on the timeline.

`shouldTriggerSubscribedTagEvent` makes sure that the `TimeMetadata` object's start time matches the current playback time correctly.

For example:

```
_playbackClockEventListener = new Clock.ClockEventListener() {
    @Override
    public void onTick(String name) {
        getActivity().runOnUiThread(new Runnable() {
            @Override
            public void run() {

                /* handle timedmetadata object list */
                if (_mediaPlayer != null && _timedMetadataList != null &&
                    _timedMetadataList.size() > 0) {
                    if (_lastKnownStatus == MediaPlayer.PlayerState.PLAYING) {
                        long localTime = _mediaPlayer.getLocalTime();
                        Iterator<TimedMetadata> iterator = _timedMetadataList.iterator();
                        while (iterator.hasNext()) {
                            TimedMetadata timedMetadata = iterator.next();
                            long diff = localTime - timedMetadata.getTime();
                            if (diff >= 0 &&
                                diff <= PLAYBACK_CLOCK_INTERVAL &&
                                _mediaPlayer.shouldTriggerSubscribedTagEvent()) {

                                // use timed metadata object
                                PMPDemoApp.logger.i(LOG_TAG + "#METADATA_CLOCK", "Local time:
```



```

    public long getBegin() {...}
    public long getEnd() {...}
    public long getDuration() {...}
}

```

## MediaPlayer and MediaResource classes

A `MediaResource` represents the content that is about to be loaded by the `MediaPlayer` instance.

The PSDK library provides a simple means to load and prepare content for playback via the `replaceCurrentItem` method in the `MediaPlayer` interface. This method receives as the sole input argument an instance of the `MediaResource` class. The `MediaResource` class brings together the following information:

- A URL: This represents the location of the content that is about to be loaded.
- A type: This is the type of content that is about to be loaded. This is a simple enumeration in the `MediaResource` class that defines the types of content that can be loaded by the `MediaPlayer`. Possible values are `HLS` and `HDS`. Each value is associated with the string representing the file extensions commonly used, “m3u8” for `HLS` and “f4m” for `HDS`.
- Some metadata: An instance of the `Metadata` class (basically a dictionary-like structure). This structure may contain additional information about the content that is about to be loaded (such as information about the alternate/ad-content that should be placed inside the main content).

The metadata is the venue through which information related to alternate content is passed on to the PSDK. The `Metadata` interface defines the API for a generic key-value store, where both the key and the value are plain strings.

## TimeRangeCollection class

The `TimeRangeCollection` utility class abstracts the notion of an ordered collection of `TimeRange` specifications and provides services to translate itself into a `Metadata` instance.

```

public final class TimeRangeCollection {
    // default constructor method
    public TimeRangeCollection(Type type) {...}
    // the list of timerange specifications provided at construction time
    public TimeRangeCollection(Type type, List<TimeRange> timeranges) {...}
    // timerange specs can also be added later
    public void addTimeRange(TimeRange timeRange) {...}
    // translate the set of timerange specs into a Metadata instance
    public Metadata toMetadata(Metadata options) {...}
}

```

The `type` parameter, which is the first positional parameter in the signature of the constructor methods, is an instance of the `TimeRangeCollection#Type` enumeration (part of the `TimeRangeCollection` class). The only value that is currently defined by this enumeration is `CUSTOM_AD_MARKERS`, which states that the `TimeRangeCollection` is a collection of time-range specifications associated with custom ad markers.

## Place TimeRange ad markers on the timeline

This example shows the recommended way to include `TimeRange` specifications on the playback timeline.

1. Translate the out-of-band ad-positioning information into a list of `TimeRange` specifications (that is, instances of the `TimeRange` class).
2. Use the set of `TimeRange` specifications to populate an instance of the `TimeRangeCollection` class.
3. Pass the `Metadata` instance, which can be obtained from the `TimeRangeCollection` instance, to the `replaceCurrentItem` method (part of the `MediaPlayer` interface).

4. Wait for the PSDK to transition to the PREPARED state by waiting for the `PlaybackEventListener#onPrepared` callback to be triggered.
  5. Start video playback by calling the `play` method (part of the `MediaPlayer` interface).
- Handling timeline conflicts: There might be cases when some `TimeRange` specifications overlap on the playback timeline. For example, the value of the start position corresponding to a `TimeRange` specification might be lower than the value of the end position that was already placed. In this case, the PSDK silently adjusts the start position of the offending `TimeRange` specification to avoid timeline conflicts. Through this adjustment, the new `TimeRange` becomes shorter than originally specified. If the adjustment is so extreme that it would lead to a `TimeRange` with a duration of zero ms, the PSDK silently drops the offending `TimeRange` specification.
  - When `TimeRange` specifications for custom ad breaks are provided, the PSDK attempts to translate these into ads that are grouped inside ad breaks. The PSDK looks for adjacent `TimeRange` specifications and clusters them into separate ad breaks. If there are time ranges that are not adjacent to any other time range, they are translated into ad breaks that contain a single ad.
  - It is assumed that the media player item that is being loaded points to a VOD asset. The PSDK checks this whenever your application tries to load a media resource whose metadata contains `TimeRange` specifications that can be used only in the context of the custom ad-markers feature. If the underlying asset is not of type VOD, the PSDK library throws an exception.
  - When dealing with custom ad markers, the PSDK deactivates other ad-resolving mechanisms (via Adobe Ad Decisioning (Auditude) or other ad provisioning system). You can use either one of the various ad-resolver modules provided by the PSDK or the custom ad-markers mechanism. When using the custom ad-markers API, the ad content is considered already resolved and placed on the timeline.

The following code snippet provides a simple example where a set of three `TimeRange` specifications are placed on the timeline as custom ad-markers.

```
// Assume that the 3 timerange specs are obtained through external means: CMS, etc.
// Use these 3 timerange specs to populate the TimeRangeCollection instance
TimeRangeCollection timeRanges = new TimeRangeCollection();
timeRanges.addTimeRange(new TimeRange(0,10000));
timeRanges.addTimeRange(new TimeRange(15000,20000));
timeRanges.addTimeRange(new TimeRange(25000,30000));

// create and configure a MediaResource instance
MediaResource mediaResource =
MediaResource.createFromUrl("www.example.com/video/test_video.m3u8",
    timeRanges.toMetadata(null));

// prepare the content for playback by creating
// NOTE: mediaPlayer is an instance of a properly configured MediaPlayer
mediaPlayer.replaceCurrentItem(mediaResource);
// wait for the PSDK to reach the PREPARED state
...
MediaPlayer.PlaybackEventListener playbackEventListener = new
    MediaPlayer.PlaybackEventListener() {
    @Override
    public void onPrepared() {
        // the PSDK is in the PREPARED state. We are allowed to start the playback
        mediaPlayer.play();
    }
}
```

## Control the seek-over-ads behavior

You can override the default behavior for how the PSDK handles seeks over ads when using custom ad markers.

The PSDK implements a default behavior when a seek operation is performed over an ad-break section: the playhead position is forcibly adjusted to the beginning of the last ad-break section that was skipped over.

You can enable this type of functionality for ad-content sections that result from the placement of custom ad markers. It is disabled by default.

1. Configure a `Metadata` instance with the appropriate key/value pair (the `DefaultMetadataKeys.METADATA_KEY_ADJUST_SEEK_ENABLED` enumeration entry) that enables the "adjustable-seek-position" feature.

The key value must be set to "true" as a literal string, not as a Boolean `true` value.

```
Metadata metadata = new MetadataNode();
metadata.setValue(DefaultMetadataKeys.METADATA_KEY_ADJUST_SEEK_ENABLED.getValue(), "true");
```

2. Create and configure a `MediaResource` instance, passing the additional configuration options to `TimeRangeCollection.toMetada`. This method receives additional configuration options via another generic metadata structure.

```
MediaResource mediaResource =
MediaResource.createFromUrl("www.example.com/video/test_video.m3u8",
    timeRanges.toMedatada(metadata));
```

## Customize opportunity detectors and content resolvers

An opportunity detector is a component of the PSDK that detects custom tags in a stream and identifies placement opportunities. These placement opportunities are then sent to the content resolver, which customizes the content/ad insertion workflow based on the placement opportunity properties and metadata.

The PSDK includes a default opportunity detector:

- (`SpliceOutPlacementOpportunityDetector`), which understands default ad cues

The PSDK also includes various default content resolvers that provide content to be inserted based on the metadata key in the player item:

- `AuditudeResolver` for `AUDITUDE_METADATA_KEY`
- `MetadataResolver` for `JSON_METADATA_KEY`
- `CustomAdMarkersContentResolver` for `CUSTOM_AD_MARKERS_METADATA_KEY`

You can override the default opportunity detectors and content resolvers to customize the advertising workflow in ways such as the following:

- Add support for custom tag detection
- Recognize custom tags for ad insertion
- Create a customized ad provider
- Black out content

### About opportunity detectors and content resolvers

The PSDK provides default opportunity detectors and content resolvers that place ads into the timeline based on nonstandard tags within the manifest. Your application might need to alter the timeline based on opportunities to do so that are identified within the manifest, such as indicators for a blackout period.

An *opportunity* represents a point of interest on the timeline and usually indicates an ad placement opportunity but an opportunity can also indicate a custom operation that might affect the timeline (such as a blackout period). An

*opportunity detector* is responsible for identifying specific opportunities (tags) in the timeline and notifying the PSDK that it has done so. Opportunities are identified in a timeline within `TimedMetadata` by the inclusion of a nonstandard (non-HLS) tag.

When your application is notified of such an opportunity (tag), it could alter the timeline in any manner, such as by inserting a series of ads, by switching to an alternate stream (blackouts), or by otherwise editing the timeline content. By default, the PSDK calls the appropriate *content resolver* to implement any timeline changes or actions required. Your application can use the default PSDK advertisement content resolver or register its own content resolver for different purposes.

The PSDK comes with a default content resolver that is responsible for resolving the default advertisement markers—`#EXT-X-CUE`—inside the PSDK. Whenever one of these tags is encountered, control is transferred to the default ad resolver, which resolves advertisements for the PSDK to insert into the timeline.

You can also use `PSDKConfig.setAdTags` to add more ad marker tags/cues for the PSDK to recognize and use `PSDKConfig.subscribedTags` to have the PSDK notify your application about additional tags, which might carry information needed for the advertising workflow.

One possible use of a custom resolver is for blackout periods. To handle blackouts, your application could implement and register a blackout opportunity detector that is responsible for handling blackout tags. Whenever the PSDK encounters this specific tag, it polls all the registered content resolvers to find the first one that handles the specified tag. In this case, it would be the blackout content resolver, which could, for example, replace the current item with alternate content on the player for the duration specified by the tag.

## Implement a custom opportunity detector

You can implement your own opportunity detectors by implementing the interface `PlacementOpportunityDetector`.

1. Create a custom `AdvertisingFactory` instance and override `createOpportunityDetector`. For example:

```
new AdvertisingFactory() {
    ...
    @Override
    public PlacementOpportunityDetector createOpportunityDetector(MediaPlayerItem item) {
        return new CustomPlacementOpportunityDetector();
    }
    ...
}
```

2. Register the ad client factory to the `MediaPlayer`. For example:

```
// register the custom advertising factory with media player
advertisingFactory = createCustomAdvertisingFactory();
mediaPlayer.registerAdClientFactory(advertisingFactory);
```

3. Create a custom opportunity detector class that extends the `PlacementOpportunityDetector` class.

- a) In the custom opportunity detector, override this function:

```
public List<PlacementOpportunity> process(List<TimedMetadata> timedMetadataList, Metadata metadata)
```

The `timedMetadataList` contains the list of available `TimedMetadata`, which is sorted. `Metadata` contains the targeting parameters and the custom parameters to be sent to the ad provider.

- b) For each `TimedMetadata`, create a `List<PlacementOpportunity>`. The list can be empty, but not null. `PlacementOpportunity` should have the following attributes:

```
PlacementOpportunity(
    String id, // can be id from timedMetadata
    PlacementInformation placementInformation // PlacementInformation object containing
    Type, time, duration
```

```

        Metadata metadata // ad metadata containing targeting params
        sent to the ad provider
    )

```

- c) After placement opportunities are created for all the detected timed metadata objects, simply return the `PlacementOpportunity` list.

This is a sample custom placement opportunity detector:

```

public class CustomPlacementOpportunityDetector implements PlacementOpportunityDetector {
    ...
    @Override
    public List<PlacementOpportunity> process(List<TimedMetadata> timedMetadataList, Metadata
metadata) {
        ...
        List<PlacementOpportunity> opportunities = new ArrayList<PlacementOpportunity>();
        for (TimedMetadata timedMetadata : timedMetadataList) {
            if (isOpportunity(timedMetadata)) { // check if given timedMetadata should be
considered as an opportunity
                // create an object of PlacementOpportunity and add it to the opportunities
                list
                    PlacementOpportunity opportunity =
createPlacementOpportunity(timedMetadata, airingId, metadata);
                    Opportunities.add(opportunity);
            }
        }
        return opportunities;
    }
    ...
}

```

## Implement a custom content resolver

You can implement your own content resolvers based on the default resolvers.

When the PSDK detects a new opportunity, the PSDK iterates through the registered content resolvers looking for one that is capable of resolving that opportunity. The first one that returns true is selected for resolving the opportunity. If no content resolver is capable, then that opportunity is skipped. Because the content resolving process is usually asynchronous, the content resolver is responsible for notifying the PSDK when the process has completed.

1. Create a custom `AdvertisingFactory` instance and override `createContentResolver`.

For example:

```

new AdvertisingFactory() {
    ....
    @Override
    public ContentResolver createContentResolver(MediaPlayerItem item) {
        Metadata metadata = _mediaPlayer.getCurrentItem().getResource().getMetadata();
        if (metadata != null) {
            if (metadata.containsKey(DefaultMetadataKeys.AUDITUDE_METADATA_KEY.getValue()))
            {
                return new AuditudeResolver(getActivity().getApplicationContext());
            } else if (metadata.containsKey(DefaultMetadataKeys.JSON_METADATA_KEY.getValue()))
            {
                return new MetadataResolver();
            } else if
(metadata.containsKey(DefaultMetadataKeys.CUSTOM_AD_MARKERS_METADATA_KEY.getValue())) {
                return new CustomAdMarkersContentResolver();
            } else if (metadata.containsKey(CustomAdResolver.CUSTOM_METADATA_KEY)) {
                return new CustomAdResolver();
            }
        }
        return null;
    }
    ....
}

```

## 2. Register the ad client factory to the MediaPlayer.

For example:

```
// register the custom advertising factory with media player
advertisingFactory = createCustomAdvertisingFactory();
mediaPlayer.registerAdClientFactory(advertisingFactory);
```

## 3. Pass an AdvertisingMetadata object to the PSDK as follows:

- a) Create an AdvertisingMetadata object and MetadataNode object.
- b) Save the AdvertisingMetadata object to MetadataNode.

```
MetadataNode result = new MetadataNode();
result.setNode(DefaultMetadataKeys.ADVERTISING_METADATA.getValue(), advertisingMetadata);
```

## 4. Create a custom ad resolver class that extends the ContentResolver class.

- a) In the custom ad resolver, override this protected function:

```
void doResolveAds(Metadata metadata, PlacementOpportunity placementOpportunity)
```

Metadata contains your AdvertisingMetadata. Use it for the following TimelineOperation vector generation.

- b) For each placement opportunity, create a Vector<TimelineOperation>.

The vector can be empty, but not null.

This sample TimelineOperation provides a structure for AdBreakPlacement:

```
AdBreakPlacement(
    AdBreak.createAdBreak(ads, // Vector<Ad>
                        time, // Ad Break start time. Note: local time on the timeline
                        duration, // Ad Break duration
                        tag() // An arbitrary string value that can be attached to the
    AdBreak object.
    ),
    placementInformation // Retrieved from PlacementOpportunity
)
```

- c) After ads are resolved, call one of the following functions:

- If the ad resolve succeeds: notifyResolveComplete(Vector<TimelineOperation> proposals)
- If the ad resolve fails: notifyResolveError(Error error)

For example, if it fails:

```
Metadata metadata = new MetadataNode();
metadata.setValue("NATIVE_ERROR_CODE", exception.getCause().toString());
error.setMetadata(metadata);
```

This sample custom ad resolver makes an HTTP request to the ad server and receives a JSON response.

```
public class CustomAdResolver extends ContentResolver {
    ...
    @Override
    protected void doResolveAds(Metadata metadata, PlacementOpportunity placementOpportunity)
    {
        ...
        if (resolveSuccess == true) {
            notifyResolveComplete(Vector<TimelineOperation> proposals);
        }
        else {
            notifyResolveError(Error error);
        }
    }
    ...
}
```

Sample JSON ad server response for a live stream:

```
{
  "response": {
    "breaks": [
```

```

    {
      "start": 0,
      "ads": [
        {
          "id": 1001,
          "primary_asset": {
            "url": "http://venkat-test.s3.amazonaws.com/ads/geico/playlist.m3u8",
            "duration": 30000,
            "id": "asset1",
            "resource_type": ""
          },
          "companion_assets": {
          }
        }
      ]
    },
    {
      "start": -1,
      "ads": [
        {
          "id": 1003,
          "primary_asset": {
            "url": "http://venkat-test.s3.amazonaws.com/ads/priceline/playlist.m3u8",
            "duration": 30000,
            "id": "asset3",
            "resource_type": ""
          },
          "companion_assets": {
          }
        }
      ]
    }
  ]
}

```

#### Sample JSON ad server response for VOD:

```

{
  "response": {
    "breaks": [
      {
        "start": 0,
        "ads": [
          {
            "id": 1001,
            "primary_asset": {
              "url": "http://venkat-test.s3.amazonaws.com/ads/geico/playlist.m3u8",
              "duration": 30000,
              "id": "asset1",
              "resource_type": ""
            },
            "companion_assets": {
            }
          },
          {
            "id": 1002,
            "primary_asset": {
              "url": "http://venkat-test.s3.amazonaws.com/ads/crescent/playlist.m3u8",
              "duration": 15000,
              "id": "asset2",
              "resource_type": ""
            },
            "companion_assets": {
            }
          }
        ]
      }
    ]
  }
}

```

```

    },
    {
      "start": 50000,
      "ads": [
        {
          "id": 1003,
          "primary_asset": {
            "url":
"http://venkat-test.s3.amazonaws.com/ads/priceline/playlist.m3u8",
            "duration": 30000,
            "id": "asset3",
            "resource_type": ""
          },
          "companion_assets": {
          }
        }
      ]
    },
    {
      "start": 100000000,
      "ads": [
        {
          "id": 1004,
          "primary_asset": {
            "url": "http://venkat-test.s3.amazonaws.com/ads/camry/playlist.m3u8",
            "duration": 15000,
            "id": "asset4",
            "resource_type": ""
          },
          "companion_assets": {
          }
        }
      ]
    }
  ]
}

```

## Set up closed captioning

Closed captioning allows people with hearing disabilities to have access to video programming by displaying the audio portion of the video as text on the screen.

Closed captioning differs from subtitles in that subtitles are typically not in the same language as the audio and subtitles do not typically include information about background sounds such as the sound of a door slamming or music.

Closed captions are part of the data packets of the MPEG-2 video streams inside the video transmission stream. The Android PSDK supports rendering 608 and 708 closed captioning, when delivered as part of the video transport stream over HLS. Additionally, it supports WebVTT as a sidecar caption option. You can:

- Switch closed captioning on or off (visible or not visible)
- Style the closed captions by selecting the font, font color, and other attributes.



**Note:** In the PSDK for Android, closed captions are always enabled. All default closed-caption tracks are considered to be present. Default tracks (such as CC1-CC4, CS1-CS6) are enumerated in `ClosedCaptionsTrack.DefaultCCTypes`. When playback begins, the PSDK looks for activity on any of these channels. If activity is found, the PSDK sets the `isActive` method for that track and dispatches the `MediaPlayer.PlaybackEventListener.onUpdated` event.



**Note:** Based on the HLS specifications, WebVTT files are referenced from the m3u8 manifest files, and are automatically available as CC tracks in the Primetime Player.

Beyond the simple closed-caption-related API exposed by the `MediaPlayerItem` interface, the `MediaPlayer` interface allows you to control various parameters that dictate how closed-captions data is rendered by the underlying video engine.

## Control closed-caption visibility

The PSDK provides methods to control the visibility of closed-caption data.

1. To get the current visibility status of a closed-caption track, use the getter method to return an instance of `MediaPlayer.Visibility`.

```
Visibility getCCVisibility() throws IllegalStateException;
```

The `MediaPlayer.Visibility` enumeration defines the possible states for the closed-caption track visibility.

```
enum Visibility {
    VISIBLE,
    INVISIBLE}

```

2. To control the visibility of a closed-caption track on the screen, use the setter method, passing an instance of `MediaPlayer.Visibility`.

For example:

```
mediaPlayer.setCCVisibility(Visibility.VISIBLE);
```

3. To set which closed-caption track is current, use the `MediaPlayerItem.selectClosedCaptionsTrack` method.
4. Retrieve the media player item from the media player, once it is prepared, using the `MediaPlayer.getCurrentItem` method.

For example:

```
// Select the initial CC track.
List<ClosedCaptionsTrack> ccTracks =
mediaPlayer.getCurrentItem().getClosedCaptionsTracks();
for (int i = 0; i < ccTracks.size(); i++) {
    ClosedCaptionsTrack track = ccTracks.get(i);
    if (track.getName().equals(INITIAL_CC_TRACK)) {
        mediaPlayer.getCurrentItem().selectClosedCaptionsTrack(track);
        selectedClosedCaptionsIndex = i;
    }
}

```

## Allow the user to change the track

This is an example of how you could create a button that allows a user to select a closed-caption track.

1. Create a simple button to change the closed-caption track.

```
<Button
    android:id="@+id/selectCC"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"
    android:layout_marginRight="10dp"
    android:onClick="selectClosedCaptioningClick"
    android:text="CC" />

```

- Convert the list of available closed caption tracks to a string array. The closed-caption tracks that have activity (data was discovered for that channel) are marked accordingly:

```
/**
 * Converts the closed captions tracks to a string array.
 *
 * @return array of CC tracks
 */
private String[] getCCsAsArray() {
    List<String> closedCaptionsTracksAsStrings = new ArrayList<String>();
    MediaPlayerItem currentItem = mediaPlayer.getCurrentItem();
    if (currentItem != null) {
        List<ClosedCaptionsTrack> closedCaptionsTracks =
            currentItem.getClosedCaptionsTracks();
        Iterator<ClosedCaptionsTrack> iterator = closedCaptionsTracks.iterator();
        while (iterator.hasNext()) {
            ClosedCaptionsTrack closedCaptionsTrack = iterator.next();
            String isActive = closedCaptionsTrack.isActive() ? " (" +
                getString(R.string.active) + ")" : "";
            closedCaptionsTracksAsStrings.add(closedCaptionsTrack.getName() + isActive);
        }
    }
    return closedCaptionsTracksAsStrings.toArray(new
        String[closedCaptionsTracksAsStrings.size()]);
}
```

- When the user clicks the button, display a dialog that shows all the default CC tracks.

```
public void selectClosedCaptioningClick(View view) {
    Log.i(LOG_TAG + "#selectClosedCaptions", "Displaying closed captions chooser dialog.");
    final String items[] = getCCsAsArray();
    final AlertDialog.Builder ab = new AlertDialog.Builder(this);
    ab.setTitle(R.string.PlayerControlCCDialogTitle);
    ab.setSingleChoiceItems(items, selectedClosedCaptionsIndex, new
        DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                // Select the new closed captioning track.
                MediaPlayerItem currentItem = mediaPlayer.getCurrentItem();
                ClosedCaptionsTrack desiredClosedCaptionsTrack =
                    currentItem.getClosedCaptionsTracks().get(whichButton);
                boolean result = currentItem.selectClosedCaptionsTrack(desiredClosedCaptionsTrack);

                if (result) {
                    selectedClosedCaptionsIndex = whichButton;
                }
                // Dismiss dialog.
                dialog.cancel();
            }
        })
        .setNegativeButton(R.string.PlayerControlCCDialogCancel, new
            DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int whichButton) {
                    // Just cancel the dialog.
                }
            })
        );
    ab.show();
}
```

## Control closed-caption styling

The styling information associated with the closed-caption track is abstracted by the `TextFormat` interface. This class encapsulates closed-caption styling information such as the font type, size, color, and background opacity.

## Set closed-caption styles

You can style the closed-caption text with PSDK methods.

1. Create a `TextFormatBuilder` instance.

You can optionally provide all the closed-caption styling parameters at construction time, or set them later.

The PSDK encapsulates closed-caption styling information inside the `TextFormat` interface. The `TextFormatBuilder` is a utility class that allows you to obtain a reference to an object that implements this interface.

```
public TextFormatBuilder(
    Font font,
    Size size,
    FontEdge fontEdge,
    Color fontColor,
    Color backgroundColor,
    Color fillColor,
    Color edgeColor,
    int fontOpacity,
    int backgroundOpacity,
    int fillOpacity)
```

2. To obtain a reference to an object that implements the `TextFormat` interface, you can call the `TextFormatBuilder.toTextFormat` public method.

This returns a `TextFormat` object that can be applied to the media player.

```
public TextFormat toTextFormat()
```

3. To get the current closed-caption style settings, you can do either of the following:

- Get all the settings at once with the `MediaPlayer.getCCStyle` getter method. The return value is an instance of the `TextFormat` interface.

```
/**
 * @return the current closed captioning style.
 * If no style was previously set, it returns a TextFormat object
 * with default values for each attribute.
 * @throws IllegalStateException if media player was already released.
 */
public TextFormat getCCStyle() throws IllegalStateException;
```

- Get the settings one at a time through the `TextFormat` interface getter methods.

```
public Color getFontColor();
public Color getBackgroundColor();
public Color getFillColor(); // retrieve the font fill color
public Color getEdgeColor(); // retrieve the font edge color
public Size getSize(); // retrieve the font size
public FontEdge getFontEdge(); // retrieve the font edge type
public Font getFont(); // retrieve the font type
public int getFontOpacity();
public int getBackgroundOpacity();
```

4. To change the styling settings, you can do either of the following:

- Use the setter method `MediaPlayer.setCCStyle`, passing an instance of the `TextFormat` interface as the single input parameter.

```
/**
```

```

* Sets the closed captioning style. Used to control the closed captioning font,
* size, color, edge and opacity.
*
* This method is safe to use even if the current media stream doesn't have closed
* captions.
*
* @param textFormat
* @throws IllegalStateException
*/
public void setCCStyle(TextFormat textFormat) throws IllegalStateException;

```

- Use the `TextFormatBuilder` class, which defines individual setter methods.

The `TextFormat` interface defines an immutable object: there are only getter methods and no setters. You can set the closed-caption styling parameters only when dealing with the `TextFormatBuilder` class.

```

// set font type
public void setFont(Font font)
public void setBackgroundColor(Color backgroundColor)
public void setFillColor(Color fillColor)
// set the font-edge color
public void setEdgeColor(Color edgeColor)
// set the font size
public void setSize(Size size)
// set the font edge type
public void setFontEdge(FontEdge fontEdge)
public void setFontOpacity(int fontOpacity)
public void setBackgroundOpacity(int backgroundOpacity)
// set the font-fill opacity level
public void setFillOpacity(int fillOpacity)
public void setFontColor(Color fontColor)

```



**Note:** Setting the closed-caption style is an asynchronous operation and it may take up to a few seconds to see the actual changes on the screen.

## Closed caption styling options

The PSDK allows you to specify several caption styling options. These settings override any style options in the original captions.

Format	Description
Font	<p>The font type. Can be set to values such as <code>monospaced_with_serifs</code>, <code>cursive</code>, and others. The value <code>default</code> refers to whatever font the caption originally specified. Valid values are defined by the <code>TextFormat.Font</code> enumeration.</p> <p>Note that the specific fonts that are actually available on any particular device may vary. Substitutions are used when necessary. The <code>MONOSPACED_WITH_SERIFS</code> is typically the font that is used as a substitute; however, this, too, can be system specific.</p>
Size	<p>The caption's size. Valid values are defined by the <code>TextFormat.Size</code> enumeration.</p> <p>Meanings are:</p> <ul style="list-style-type: none"> <li>• <code>MEDIUM</code> - The standard size</li> <li>• <code>LARGE</code> - Approximately 30% larger than medium</li> <li>• <code>SMALL</code> - Approximately 30% smaller than medium</li> </ul>

Format	Description
	<ul style="list-style-type: none"> <li>• <code>DEFAULT</code> - The default size for the caption; the same as medium</li> </ul>
Font edge	The effect used for the font edge, such as raised or none. Valid values are defined by the <code>TextFormat.FontEdge</code> enumeration.
Font color	The font's color. Can be set to any one of a wide variety of values, such as <code>BLACK</code> , <code>RED</code> , or <code>DARK_CYAN</code> . The value <code>DEFAULT</code> refers to whatever color the caption originally specified. Valid values are defined by the <code>TextFormat.Color</code> enumeration.
Edge color	The color of the edge effect. Can be set to any of the values defined for the font color.
Background color	The background character cell color. Can be set to any of the values defined for the font color.
Fill color	The color of the background of the window that the text is in. Can be set to any of the values defined for the font color.
Font opacity	The opacity of the text. Expressed as a percentage from 0 to 100 with 100 being fully opaque and 0 being fully transparent.
Background opacity	The opacity of the background character cell. Expressed as a percentage from 0 to 100 with 100 being fully opaque and 0 being fully transparent.
Fill opacity	The opacity of the background of the window that the text is in. Expressed as a percentage from 0 to 100 with 100 being fully opaque and 0 being fully transparent.

### Caption formatting samples

Also refer to the `PMPDemoApp`, which includes these operations in the scrub-bar controls.

#### Example 1: Specify format explicitly

```
private final MediaPlayer.PlaybackEventListener _playbackEventListener =
new MediaPlayer.PlaybackEventListener() {
    @Override
    public void onPrepared() {
        // Set CC style.
        TextFormat tf = new TextFormatBuilder(TextFormat.Font.DEFAULT,
TextFormat.Size.DEFAULT,
TextFormat.FontEdge.DEFAULT,
TextFormat.Color.DEFAULT,
TextFormat.Color.DEFAULT,
TextFormat.Color.DEFAULT,
TextFormat.Color.DEFAULT,
TextFormat.DEFAULT_OPACITY,
TextFormat.DEFAULT_OPACITY,
TextFormat.DEFAULT_OPACITY).toTextFormat();
mediaPlayer.setCCStyle(tf);
    }
}
```

#### Example 2: Specify format with parameters

```
/**
 * Constructor using parameters to initialize a TextFormat.
 *
 * @param font
 * The desired font.
```

```

* @param size
* The desired text size.
* @param fontEdge
* The desired font edge.
* @param fontColor
* The desired font color.
* @param backgroundColor
* The desired background color.
* @param fillColor
* The desired fill color.
* @param edgeColor
* The desired color to draw the text edges.
* @param fontOpacity
* The desired font opacity.
* @param backgroundOpacity
* The desired background opacity.
* @param fillOpacity
* The desired fill opacity.
*/
public TextFormatBuilder(Font font, Size size, FontEdge fontEdge,
Color fontColor, Color backgroundColor,
Color fillColor, Color edgeColor,
int fontOpacity, int backgroundOpacity,
int fillOpacity);
/**
* Creates a TextFormat with the parameters supplied to this builder.
*/
public TextFormat toTextFormat();
/**
* Sets the text font.
* @param font The desired font
* @return This builder object to allow chaining calls
*/
public TextFormatBuilder setFont(Font font);
....

```

## Set up alternate audio

Alternate (late-binding) audio is the support of multiple language tracks for HTTP video streams (live/linear and VOD) without having to duplicate and repackage the video for each audio track. Late binding of an audio track allows you to easily provide multiple language tracks for a video asset at any time before or after the asset's initial packaging.

### Alternate audio tracks in the playlist

The playlist for a video can specify an unlimited number of alternative audio tracks for the main video content. With the PSDK, you can use these alternative tracks. For example, you might want to add different languages to your video content or allow the user to switch between different tracks on their device while the content is playing. This is known as late-binding audio.

Late-binding audio is the support of multiple language tracks for HTTP video streams (live/linear and VOD) without having to modify, duplicate, or repackage the video for each audio track. Late binding of an audio track allows you to easily provide multiple language tracks for a video asset at any time before or after the asset's initial packaging.

In order for the alternate audio to be mixed with the video track of the main media, the timestamps of the alternate track must match the timestamps of the audio in the main track.

The main audio track is included in the audio tracks collection with the label "default". Metadata for the alternate audio streams is included in the playlist in the #EXT-X-MEDIA tags with TYPE=AUDIO.

For example, an M3U8 manifest that specifies multiple alternate audio streams might look like this:

```
#EXTM3U
```

```
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="bipbop_audio",LANGUAGE="eng",NAME="BipBop Audio 1",
  AUTOSELECT=YES,DEFAULT=YES
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="bipbop_audio",LANGUAGE="eng",NAME="BipBop Audio 2",
  AUTOSELECT=NO,DEFAULT=NO,URI="alternate_audio_aac/prog_index.m3u8"
#EXT-X-MEDIA:TYPE=SUBTITLES,GROUP-ID="subs",NAME="English",AUTOSELECT=YES,FORCED=NO,
  LANGUAGE="eng",URI="subtitles/eng/prog_index.m3u8"
#EXT-X-MEDIA:TYPE=SUBTITLES,GROUP-ID="subs",NAME="English (Forced)",DEFAULT=YES,
  AUTOSELECT=YES,FORCED=YES,LANGUAGE="eng",URI="subtitles/eng_forced/prog_index.m3u8"
#EXT-X-MEDIA:TYPE=SUBTITLES,GROUP-ID="subs",NAME="Français",AUTOSELECT=YES,FORCED=NO,
  LANGUAGE="fra",URI="subtitles/fra/prog_index.m3u8"
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=263851,CODECS="mp4a.40.2, avc1.4d400d",
  RESOLUTION=416x234,AUDIO="bipbop_audio",SUBTITLES="subs"
gear1/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=577610,CODECS="mp4a.40.2, avc1.4d401e",
  RESOLUTION=640x360,AUDIO="bipbop_audio",SUBTITLES="subs"
gear2/prog_index.m3u8
...
```

## Access alternate audio tracks

The PSDK for Android supports late-binding audio. The PSDK uses MediaPlayer to play a video specified in an M3U8 HLS playlist, which can contain several alternate audio streams.

Access the available audio tracks using methods and properties in the MediaPlayerItem class.

You can present the available tracks to the user through the player user interface. Refer to the PMPDemoApp, which includes these operations in the scrub-bar controls.

## Content security using DRM

You can use the features of the Primetime digital rights management (DRM) system to provide secure access to your video content.

Primetime DRM provides a scalable, efficient workflow for digital rights management to help you deliver and protect your premium video content. You protect and manage the rights to your video content by creating licenses for each digital media file. The PSDK gives you the ability to implement and deploy content protection.

Refer to the DRM sample player code included in the PSDK package.

Refer to the Adobe Access Help Resource Center for complete DRM documentation.

### DRM interface overview

The key elements of the digital rights management (DRM) system include MediaPlayer methods, the DRM class, and the DRMHelper class.

- A reference to the object that implements the DRM subsystem:

```
MediaPlayer.getDRMManager();
```

- The DRMHelper helper class, which is useful when implementing DRM workflows. You can see DRMHelper in the PMPDemoApp.
- A DRMHelper metadata loader method, which loads DRM metadata when it is located in a separate URL from the media.

```
public static void loadDRMMetadata(final DRMManager drmManager,
  final String drmMetadataUrl,
  final DRMLoadMetadataListener loadMetadataListener);
```

- A DRMHelper method to check the metadata to determine whether authentication is needed.

```
/**
 * Return whether authentication is needed for the provided
 * DRMMetadata.
 *
 * @param drmMetadata
 * The desired DRMMetadata on which to check whether auth is needed.
 * @return whether authentication is required for the provided metadata
 */
public static boolean isAuthNeeded(DRMMetadata drmMetadata);
```

- DRMHelper method to perform authentication.

```
/**
 * Helper method to perform DRM authentication.
 *
 * @param drmManager
 * the DRMManager, used to perform the authentication.
 * @param drmMetadata
 * the DRMMetadata, containing the DRM specific information.
 * @param authenticationListener
 * the listener, on which the user can be notified about the
 * authentication process status.
 * @param authUser
 * the DRM username provider by the user.
 * @param authPass
 * the DRM password provided by the user.
 */
public static void performDrmAuthentication(final DRMManager drmManager,
final DRMMetadata drmMetadata,
final String authUser,
final String authPass,
final DRMAAuthenticationListener authenticationListener);
```

- Events that notify your application about various DRM activities and status.

Relevant SDK elements:

- [com.adobe.ave.drm.DRMManager](#)
- [com.adobe.ave.drm.DRMMetadata](#)
- [com.adobe.ave.drm.DRMPolicy](#)
- [com.adobe.ave.drm.DRMAuthenticationMethod](#)
- [com.adobe.ave.drm.DRMAAuthenticationCompleteCallback](#)
- [com.adobe.ave.drm.DRMOperationErrorCallback](#)

## DRM authentication before playback

When the DRM metadata for a video is separate from the media stream, perform authentication before beginning playback.

A video asset can have an associated DRM metadata file. For example:

- "url": "http://www.domain.com/asset.m3u8"
- "drmMetadata": "http://www.domain.com/asset.metadata"

When this is the case, use `DRMHelper` methods to download the contents of the DRM metadata file, parse it, and check whether DRM authentication is needed.

1. Use `loadDRMMetadata` to load the metadata URL content and parse the downloaded bytes to a `DRMMetadata`.

Like any other network operation, this method is asynchronous, creating its own thread.

```
public static void loadDRMMetadata(
    final DRMMManager drmManager,
    final String drmMetadataUrl,
    final DRMLoadMetadataListener loadMetadataListener);
```

For example:

```
DRMHelper.loadDRMMetadata(drmManager, metadataURL, new DRMLoadMetadataListener());
```

2. Because the operation is asynchronous, it is a good idea to make the user aware of that. Otherwise, he'll wonder why his playback is not beginning. For example, show a spinner wheel while the DRM metadata is being downloaded and parsed.
3. Implement the callbacks in the `DRMLoadMetadataListener`. The `loadDRMMetadata` calls these event handlers (dispatches these events).

```
public interface DRMLoadMetadataListener {
    public void onLoadMetadataUrlStart();
    /**
     * @param authNeeded
     *       whether DRM authentication is needed.
     * @param drmMetadata
     *       the parsed DRMMetadata obtained. */
    public void onLoadMetadataUrlComplete(boolean authNeeded, DRMMetadata drmMetadata);
    public void onLoadMetadataUrlError();
}
```

- `onLoadMetadataUrlStart` detects when the metadata URL loading has begun.
- `onLoadMetadataUrlComplete` detects when the metadata URL has finished loading.
- `onLoadMetadataUrlError` indicates that the metadata failed to load.

4. When loading completes, inspect the `DRMMetadata` object to see whether DRM authentication is needed.

```
public static boolean isAuthNeeded(DRMMetadata drmMetadata);
```

For example:

```
@Override
public void onLoadMetadataUrlComplete(boolean authNeeded, DRMMetadata drmMetadata) {
    Log.i(LOG_TAG + "#onLoadMetadataUrlComplete",
        "Loaded metadata URL contents. Auth needed:" + authNeeded + ".");
    if (!authNeeded) {
        // Auth is not required. Start player activity.
        showLoadingSpinner(false);
        startPlayerActivity(ASSET_URL);
        return;
    }
}
```

5. If authentication is not needed, begin playback.
6. If authentication is needed, perform the authentication by acquiring the license.

```
/**
 * Helper method to perform DRM authentication.
 *
 * @param drmManager
 *       the DRMMManager, used to perform the authentication.
 * @param drmMetadata
 *       the DRMMetadata, containing the DRM specific information.
 * @param authenticationListener
 *       the listener, on which the user can be notified about the
 *       authentication process status.
 */
public static void performDrmAuthentication(
    final DRMMManager drmManager,
    final DRMMetadata drmMetadata,
    final String authUser,
```

```
final String authPass,
final DRMAuthenticationListener authenticationListener);
```

This example, for simplicity, explicitly codes the user's name and password.

```
DRMHelper.performDrmAuthentication(drmManager, drmMetadata, DRM_USERNAME, DRM_PASSWORD, new
DRMAuthenticationListener() {
@Override
public void onAuthenticationStart() {
    Log.i(LOG_TAG + "#onAuthenticationStart", "DRM authentication started.");
    // Spinner is already showing.
}
@Override
public void onAuthenticationError(long majorCode, long minorCode, Exception e) {
    Log.e(LOG_TAG + "#onAuthenticationError",
        "DRM authentication failed. " + majorCode + " 0x" + Long.toHexString(minorCode));

    showToast(getString(R.string.drmAuthenticationError));
    showLoadingSpinner(false);
}
@Override
public void onAuthenticationComplete(byte[] authenticationToken) {
    Log.i(LOG_TAG + "#onAuthenticationComplete", "Auth successful. Launching content.");
    showLoadingSpinner(false);
    startPlayerActivity(ASSET_URL);
}
});
```

7. This also implies network communication, hence this is also an asynchronous operation. Use an event listener to check the authentication status.

```
public interface DRMAuthenticationListener {
/**
 * Called to indicate that DRM authentication has started.
 */
public void onAuthenticationStart();
/**
 * Called to indicate that DRM authentication has been successful.
 *
 * @param authenticationToken
 * the obtained token, which can be stored locally.
 */
public void onAuthenticationComplete(byte[] authenticationToken);
/**
 * Called to indicate that an error occurred while performing the DRM
 authentication.
 *
 * @param majorCode
 * the major code.
 * @param minorCode
 * the minor code.
 * @param e
 * the exception thrown.
 */
public void onAuthenticationError(long majorCode, long minorCode, Exception e);
}
```

8. If authentication is successful, start playback.
9. If authentication is not successful, notify the user and do not start playback.

Your application must handle any authentication errors. Failing to successfully authenticate before playing places the PSDK into an error state. That is, the PSDK changes its state to ERROR, an error is generated containing the error code from the DRM library, and the playback stops. Your application must resolve the issue, reset the player, and reload the resource.

## DRM authentication during playback

When the DRM metadata for a video is included in the media stream, perform authentication during playback.

Consider the license rotation feature, where an asset is encrypted with multiple DRM licenses. Each time that new DRM metadata is discovered, use DRMHelper methods to check whether the DRM metadata requires DRM authentication.

### Note:

*This tutorial does not handle domain bound licenses. Ideally, before starting playback, check whether you are dealing with a domain bound license. If yes, perform domain authentication (if needed) and join the domain.*

1. When new DRM Metadata is discovered in an asset, an event is dispatched at the application layer.

```
mediaPlayer.addEventListener(MediaPlayer.Event.DRM,
    new MediaPlayer.DRMEventListener() {
        public void onDRMMetadata(final DRMMetadataInfo drmMetadataInfo) {
        }
    });
```

2. Use the DRMMetadata to check whether authentication is needed. If not, do nothing; playback continues uninterrupted.
3. Otherwise, perform DRM authentication. Since this operation is asynchronous and is handled in a different thread, it has have no impact on the user interface nor on video playback.
4. If authentication fails, the user cannot continue viewing the video and playback ceases. Otherwise, playback will continue uninterrupted.

```
private final MediaPlayer.DRMEventListener drmEventListener = new
MediaPlayer.DRMEventListener() {
    @Override
    public void onDRMMetadata(final DRMMetadataInfo drmMetadataInfo) {
        Log.i(LOG_TAG + "::MediaPlayer.DRMEventListener#onDRMMetadata",
            "DRM metadata available: " + drmMetadataInfo + ".");
        if (drmMetadataInfo == null ||
!DRMHelper.isAuthNeeded(drmMetadataInfo.getDRMMetadata())) {
            Log.i(LOG_TAG + "#onDRMMetadata", "DRM auth is not needed.");
            return;
        }
        // Perform DRM auth.
        // Possible logic might take into consideration a threshold between
        // the current player time and the
        // DRM metadata start time. For the time being, we resolve it as soon
        // as we receive the DRM metadata.
        DRMManager drmManager = mediaPlayer.getDRMManager();
        if (drmManager == null) {
            Log.e(LOG_TAG + "#onDRMMetadata", "DRMManager is null.");
            return;
        }
        DRMHelper.performDrmAuthentication(drmManager,
            drmMetadataInfo.getDRMMetadata(),
            CatalogActivity.DRM_USERNAME,
            CatalogActivity.DRM_PASSWORD,
            new DRMAuthenticationListener()
        {
            @Override
            public void onAuthenticationStart() {
                metadata at ["Log.i(LOG_TAG + "#onAuthenticationStart",
                    "DRM authentication started for DRM metadata at ["
                    + drmMetadataInfo.getPrefetchTimestamp() + "].");
            }
        }
        @Override
        public void onAuthenticationError(long majorCode, long minorCode, Exception e) {
```

```

        Log.e(LOG_TAG + "#onAuthenticationError",
            "DRM authentication failed. " + majorCode + " 0x" + Long.toHexString(minorCode));
        handler.post(new Runnable() {
            @Override
            public void run() {
                showToast(getString(R.string.drmAuthenticationError));
            }
        });
    }

    @Override
    public void onAuthenticationComplete(byte[] authenticationToken) {
        Log.i(LOG_TAG + "#onAuthenticationComplete",
            "Auth successful for DRM metadata at ["
                + drmMetadataInfo.getPrefetchTimestamp() + "].");
    }
}
};

```

## Use the notification system

The notification portion of the PSDK library allows you to create a logging and debugging system that can be useful for diagnostic and validation purposes.

`MediaPlayerNotification` notifications provide information about *player status* as information, warnings, or errors. Your application can retrieve data that describes what caused the error or warning. Errors that stop the playback of the video also cause a change of the `state` of the player.



### Note:

*The PSDK also uses notification to refer to `MediaPlayer` event notifications, which the PSDK dispatches to provide information about player activity. You implement event listeners to capture and respond to those. Many event notifications also cause `MediaPlayerNotification` status notifications.*

## Notification content

`MediaPlayerNotification` notifications provide information related to the player's status.

The PSDK provides a chronologically sorted list of `MediaPlayerNotification` notifications. Each notification contains the following:

- Time stamp
- Diagnostic metadata, including a numeric code, a name for the notification, metadata keys, and related inner notifications

The diagnostic metadata consists of the following elements:

Element	Description
type	Describes the notification event type. Depending on the platform, this property is an enumerated type with possible values of INFO, WARN, and ERROR. This is the first, and highest-level, classification criterion for the notification events.
code	The numerical representation assigned to the notification event. <ul style="list-style-type: none"> <li>• Error notification events, from 100000 to 199999</li> </ul>

Element	Description
	<ul style="list-style-type: none"> <li>• Warning notification events, from 200000 to 299999</li> <li>• Information notification events, from 300000 to 399999</li> </ul>
name	A string containing a human-readable description of the notification event, such as <code>PLAYBACK_START</code> .
metadata	Metadata containing relevant information about the notification stored as key/value pairs. For example, a key named <code>URL</code> would provide a URL related to the notification, such as an invalid URL that caused an error.
innerNotification	A reference to another <code>MediaPlayerNotification</code> object that directly impacted this notification. An example might be a notification about an ad-insertion failure that directly corresponds to a time-line insertion conflict.

You can store this information locally for later analysis or send it to a remote server for logging and graphical representation.

## Set up your notification system

You can listen for notifications and you can add your own notifications to the notification history.

The core of the Primetime Player notification system is the `Notification` class, which represents a standalone notification.

The `NotificationHistory` class provides a mechanism for accumulating notifications. It stores a log of notification (`NotificationHistoryItem`) objects that represents a collection of `Notifications`.

Your application can perform two basic operations with notifications:

- Listen for notifications
  - Add notifications to the notification history
1. Listen for state changes.
  2. Implement the `MediaPlayer.PlaybackEventListener.onStateChanged` callback.
  3. The PSDK passes two parameters to the callback:
    - The new state (`MediaPlayer.PlayerState`)
    - A `MediaPlayerNotification` object, which contains the following:

## Add real-time logging and debugging

You can use notifications to do real-time logging inside your video application.

The notification system allows you to gather logging and debugging information for diagnostics and validation without overly stressing the system.

The logging back end is not part of a production setup and is not expected to handle high-load traffic. So, if your implementation does not need to be absolutely complete, take efficiency of data transmission into consideration so that you do not overload your system.

1. Create a timer-based execution thread for your video application that periodically queries the data gathered by the PSDK's notification system.

This is how the `PMPDemoApp` implements logging.

The logging back end is not part of a production setup and is not expected to handle high-load traffic. So, if your implementation does not need to be absolutely complete, you should take efficiency of data transmission into consideration so as not to overload your system.

2. If the timer's interval is too large and the size of the event list is too small, the notification event list will overflow. To avoid overflowing the notification event list, do either or both of the following:
  - Decrease the time interval that drives the thread that polls for new events.
  - Increase the size of the notification list.
3. Serialize the new notification event entries in JSON format and send them to a remote server for postprocessing. The remote server could then graphically display the provided data in real-time.
4. To detect the loss of notification events, look for gaps in the sequence of event index values.

Each notification event has an index value that is automatically incremented by the `session.NotificationHistory` class.

## Retrieve ID3 metadata

With notifications, you can detect ID3 metadata from HLS streams embedded in ID3 tags at the Transport Stream segment level. You can then extract the information from the audio and video stream nested in the ID3 tag.

Conventionally, ID3 tags are used in conjunction with audio files. ID3 contains information related to the file, such as the name of the artist, track title, and album title. The PSDK recognizes ID3 metadata as follows:

- ID3 v2.4.0. Versions that are not compliant are ignored.
- ID3 tags in H.264 video and AAC audio streams, not in audio-only streams.
- ID3 data encoded as UTF8 or as UTF16-BE (no BOM). Unspecified encoding is treated as UTF8.

When the PSDK determines that received data is ID3 metadata, it issues a notification with the following data:

- InfoCode = 303007
- TYPE = ID3
- NAME = not present
- ID = 0

1. **DHLS:** Implement an event listener for `MediaPlayer.PlaybackEventListener#onTimedMetadata(TimedMetadata timedMetadata)` and register it with the `MediaPlayer` object. This listener is called when ID3 metadata is detected.



### Note:

*Custom ad cues use the same `onTimedMetadata` listener to indicate the detection of a new tag. This should not cause any confusion, however, because custom ad cues are detected at the manifest level and ID3 tags are embedded in the stream.*

2. Retrieve the metadata.

```
@Override
public void onTimedMetadata(TimedMetadata timedMetadata) {
    TimedMetadata.Type type = timedMetadata.getType();
    if (type.equals(TimedMetadata.Type.ID3)){
```

```

    long time = timeMetadata.getTime();
    Metadata metadata = timedMetadata.getMetadata();
    Set<String> keys = metadata.keySet();
    for (String key : keys){
        String value = metadata.getValue(key);
    }
}
}

```

## Playback and failover

Streaming over the Internet requires a constant and stable connection to play a stream from a remote server. However, the variability of a viewer's Internet connection or streaming playback means that remote playback might not have the quality of media played locally.

Primestime cannot protect from all failures—for example, an ISP failure or a cable disconnection. However, Primestime HLS (HTTP Live Streaming) provides failover protection to protect playback from certain remote server failures and operation failures, making a better experience for viewers. The Adobe Video Engine (AVE) also implements failover protection to provide the least number of playback interruptions and to achieve a seamless playback experience over some remote server turbulence and operational errors in publication. This feature allows the video player to automatically failover to a backup media set when entire renditions or fragments are unavailable.

## Media playback and failover

For both live and VOD media, the AVE starts playback by downloading the playlist associated with the middle-resolution bit rate, then downloading segments of the middle-resolution bit-rate media defined by the playlist. It then quickly selects the high-resolution bit-rate playlist and its associated media and continues downloading from there.

### Missing playlist failover

What happens when an entire playlist is missing, such as when the m3u8 file referenced by a top-level manifest file fails to download.

If the playlist associated with the middle-resolution bit rate is missing, the AVE searches for a variant playlist at the same resolution.

- If found, it starts downloading the variant playlist and the segments specified by that playlist. It quickly switches to downloading the high-resolution playlist and the segments specified by that playlist.
- If not found, it starts downloading the high-resolution playlist and the segments specified by that playlist.

If the high-resolution playlist is missing, the AVE switches to a variant playlist for the same high resolution and starts downloading segments from there.

If the missing playlist still cannot be found, the AVE cycles through all variants and bit rates to attempt to find a valid playlist. If none is found, the process fails and the PSDK moves into the ERROR state. Your application can choose how to handle this. The PSDK demo application handles this situation by closing the player activity and directing the user to the catalog activity. (See the `PlayerFragment.java` file; the `MediaPlayer.PlaybackEventListener` class implements callbacks that are attached to various events. The event of interest here is the `STATE_CHANGED` event, whose corresponding callback is the `onStateChanged` method. The code monitors whether the player changes its internal state to `ERROR`.)

```

case ERROR:
PMPDemoApp.logger.e(LOG_TAG + "::MediaPlayer.PlayerStateEventListener#onStateChanged()",
                    "Error: " + notification + ".");
getActivity().finish(); // this is where we close the current activity (the Player activity)
break;

```

## Missing segment failover

What happens when a segment is missing, such as when a particular segment fails to download. If a high-resolution segment is missing on the server (that is, the file is not present or has been removed), the AVE looks for a variant playlist associated with the high-resolution bit rate.

- If found, it uses that variant playlist from that point on, downloading segments starting with the missing segment.
- If not found, or if found but the same segment is missing from the variant playlist, the AVE switches back to the mid-resolution bit rate starting at the missing segment. It attempts to return to the high-resolution bit rate and, if successful, uses the high-resolution bit rate from that point on.

If a missing segment still cannot be found, the AVE cycles through all variants and bit rates to attempt to find the missing segment. If not found, it continues with the next segment in the current resolution or appropriate resolution depending on the failover algorithm and continues playing in that resolution.

If no alternatives can be found for a segment download, the PSDK triggers a `CONTENT_ERROR` error notification. This notification will contain an inner notification with the code `DOWNLOAD_ERROR`. If the stream with the problem is an alternate audio track, the PSDK generates the `AUDIO_TRACK_ERROR` error notification.

The PSDK will not continue to try a faulty server forever, if there are other servers that perform consistently better.

There is a documented limitation in the AVE failover strategy that says that the ABR control parameters are no longer taken into consideration once a failover scenario occurs. This is due to the fact that the failover mechanism is designed to use as backup streams any of the currently available playlists (regardless of their bit-rate profile). In other words, it is possible during a failover operation to actually have a profile switch (a change in the stream's bit-rate). If there is an error with either the playlist download or with the download of one of the segments, the ABR control parameters (such as min/max allowed bit-rate) are ignored.

## Advertising insertion and failover for VOD

The VOD ad-insertion process consists of three main phases: ad-resolving, ad-insertion, and ad-playback.

In addition, for ad tracking, the PSDK needs to inform a remote tracking server about the playback progress of each ad.

Things might not work as expected during each phase of the ad-insertion/playback process or during the ad-tracking process. This section describes the possible errors and the corresponding PSDK behavior.

### Ad-resolving phase

The PSDK contacts an ad delivery service, such as Primetime Ad Serving (Auditude), and attempts to obtain the primary playlist file corresponding to the video stream for the ad. During the ad-resolving phase, the PSDK makes an HTTP call to the remote ad-delivery server and parses the server's response.

The PSDK supports the following types of ad providers:

- Metadata ad provider: The ad data is encoded in plain-text JSON files.
- Auditude ad provider: A request is sent to the Auditude back-end server including a set of targeting parameters and an asset identification number. Auditude responds with a SMIL document that contains the ad information that the PSDK requires.
- Custom ad markers provider: Handles the situation where ads are burned into the stream, from the server side. The PSDK does not perform the actual ad insertion, but it needs to keep track of the ads that were inserted on the server side. This provider sets the ad markers that are used by the PSDK to perform the ad tracking.

Two failover scenarios can occur during this phase:

- The data cannot be retrieved for reasons such as lack of connectivity or a server-side error (resource cannot be found and so on).
- The data was retrieved, but the format is invalid (that is, the parsing of the inbound data fails). The PSDK issues a warning notification about the error and continues processing.

The PSDK issues a warning notification about the error and continues processing.

### **Ad-insertion phase**

The PSDK inserts the alternate content (ads) into the timeline corresponding to the main content.

When the ad-resolving phase is complete, the PSDK is in possession of an ordered list of ad resources grouped into ad breaks. Each ad break is positioned on the main content timeline using a start-time value expressed in milliseconds (ms). Each ad inside an ad break has a duration property also expressed in ms. The ads inside an ad break are chained together one after another. As a result, the duration of an ad break is equal to the summation of the durations of the individual composing ads.

Failover can occur in this phase with conflicts that might arise on the timeline during ad insertion. For specific combinations of ad break start-time/duration values, it could be possible for ad segments to overlap. This happens when the last portion of an ad break intersects the beginning of the first ad in the next ad break. In these situations, the PSDK discards the later ad break. The PSDK continues the ad-insertion process with the next item on the list until all ad breaks are either inserted or discarded.

The PSDK issues a warning notification about the error and continues processing.

### **Ad-playback phase**

The AVE downloads the ad segments and renders them on the device screen.

At this point, the PSDK has done most of its job in terms of ad resolving and positioning on the timeline. It now relies on the AVE to render the content on the screen.

Three main classes of errors can occur in this phase:

- Errors when connecting to the host server
- Errors during the download of the manifest file
- Errors during the download of the media segments

For all three classes of errors, the PSDK forwards any events triggered by the AVE engine to your application, specifically:

- Notification events that are triggered when a failover happens
- Notification event when the profile is changed due to the failover algorithm
- Notification event triggered when all failover options have been considered and no further action can be taken automatically (that is, your application needs to intervene and do something)

Whether errors occur or not, the PSDK calls `onAdBreakComplete` for each `onAdBreakStart` and `onAdComplete` for every `onAdStart`. However, in cases where segments cannot be downloaded, there could be gaps in the actual timeline. When the gaps are large enough, the values in the playhead position could exhibit discontinuities, as could the reported ad progress.

## Get information about profile switch

When the media player switches its current profile to a new profile, you can retrieve information about the switch, such as when it switched or why a different bit rate was used.

1. Listen for the `ProfileChangedEvent` event.

The PSDK media player dispatches this event when its adaptive bit-rate switching algorithm switches to another profile due to network or machine conditions. (When the bit rate or the period changes.)

2. When the event occurs, check its properties for information about the switch:

- `profile`: Identifier for the new profile being used.
- `time`: The stream time at which the switch occurred.
- `description`: Textual description of the reason for a bit-rate change, as a string of semicolon-separated key/value pairs. Includes a maximum of one `Reason` and one `Bitrate`. If the information is not available or the bit rate did not change, this string is empty.

Key name	Possible values
Reason	<ul style="list-style-type: none"> <li>• Network Adaptation</li> <li>• Seek</li> <li>• Profile Not Supported</li> <li>• Failover</li> </ul>
Bitrate	<ul style="list-style-type: none"> <li>• <code>up</code>: The bit rate increased</li> <li>• <code>down</code>: The bit rate decreased</li> </ul>

Examples of returned `description` strings:

```
"Bitrate::=up;Reason::=Network Adaptation;"
```

```
"Bitrate::=down;Reason::=Failover;"
```

## Primetime player classes summary

You can use the Primetime Player API to customize the behavior of the player.

### Mediacore classes

These classes describe your media player and its resources.

Package: [com.adobe.mediacore](http://com.adobe.mediacore)

Name	Description
<a href="#">ABRControlParameters</a>	Class that encapsulates all adaptive bit rate control parameters.
<a href="#">AdClientFactory</a>	Interface for creating opportunity detectors.
<a href="#">AdvertisingFactory</a>	Factory class that allows customization of the Ad Decisioning process.
<a href="#">BufferControlParameters</a>	Class that encapsulates all buffer control parameters.
<a href="#">DefaultAdPolicySelector</a>	Default implementation for ad playback behaviors.

Name	Description
<a href="#"><i>DefaultMediaPlayer</i></a>	Default class implementation of MediaPlayer interface.
<a href="#"><i>MediaPlayer</i></a>	Public interface for the DefaultMediaPlayer class. Includes enumerations for Event, PlayerState, and Visibility.
<a href="#"><i>MediaPlayer.AdPlaybackEventListener</i></a>	Interface definition of a set of callbacks to be invoked during ad playback.
<a href="#"><i>MediaPlayer.DRMEventListener</i></a>	Interface definition of a callback to be invoked when protected metadata becomes available.
<a href="#"><i>MediaPlayer.EventListener</i></a>	Marker interface used to unify event listener registration.
<a href="#"><i>MediaPlayer.PlaybackEventListener</i></a>	Interface definition of a set of callback to be invoked during playback.
<a href="#"><i>MediaPlayer.QOSEventListener</i></a>	Interface definition of a set of callback to be invoked during QoS.
<a href="#"><i>MediaPlayerItem</i></a>	Interface that represents audio-video media.
<a href="#"><i>MediaPlayerItemLoader</i></a>	Class that loads a media player resource and creates the corresponding MediaPlayerItem object.
<a href="#"><i>MediaPlayerItemLoader.LoaderListener</i></a>	Interface that defines the listener methods associated with the MediaPlayerItemLoader object.
<a href="#"><i>MediaPlayerView</i></a>	Class for the view that will be used by the MediaPlayer for video rendering.
<a href="#"><i>MediaResource</i></a>	Class that wraps all information about a media resource. Includes enumeration for Type of media resource.
<a href="#"><i>PlacementOpportunityDetector</i></a>	Interface used for processing in-manifest cues that will be used as placements for the ad decisioning process.
<a href="#"><i>PSDKConfig</i></a>	Class that encapsulates the custom tags used by the media player when performing the ad placement, in addition to the default cue tags. It also includes the tag names that the application wants to be notified about.
<a href="#"><i>TextFormat</i></a>	Interface that encapsulates different attributes describing a text style (for example, the closed captions style).
<a href="#"><i>TextFormatBuilder</i></a>	Class methods for setting the formatting of text.
<a href="#"><i>Version</i></a>	Class that provides the PSDK version and description.

## Info classes

These classes provide information about the media.

Package: [\*com.adobe.mediacore.info\*](#)

Name	Description
<a href="#"><i>AudioTrack</i></a>	Class that extends Track to define the audio track abstraction.
<a href="#"><i>ClosedCaptionsTrack</i></a>	Class that extends Track to define the closed caption abstraction.

Name	Description
<a href="#">Profile</a>	Class that contains media profile information, such as the height, width, and bit rate.
<a href="#">Track</a>	Class that defines the track abstraction to use in defining AudioTrack and ClosedCaptionsTrack.

## Logging classes

These classes enable you to customize logging.

Package: [com.adobe.mediacore.logging](#)

Name	Description
<a href="#">Log</a>	Class. Provides access to the log system.
<a href="#">LogEntry</a>	Class. Defines an entry log and holds information about a log message.
<a href="#">LogFactory</a>	Interface that enables custom logging.
<a href="#">Logger</a>	Interface. The methods required to implement a custom logger for the PSDK.

## Metadata classes

These classes provide metadata for advertising, namespaces, and tracking.

Package: [com.adobe.mediacore.metadata](#)

Name	Description
<a href="#">AdvertisingMetadata</a>	Class that provides properties that should be configured for resolving ads for a given media item. All the required properties must be set to configure the player for successfully resolving ads.
<a href="#">AuditudeMetadata</a>	Deprecated. Use <a href="#">AuditudeSettings</a> .
<a href="#">AuditudeSettings</a>	Class that extends Java AdvertisingMetadata specifically for Ad Decisioning. Provides properties to be configured for resolving Ad Decisioning ads for a given media item. You must set all the required properties, including zone ID, media ID, and ad server URL, to configure the player for successfully resolving ads.
<a href="#">Metadata</a>	Defines the generic interface for configuring all available metadata for your player and additional objects.
<a href="#">MetadataNode</a>	Generic data-structure-like class for storing arbitrary key-value pairs.
<a href="#">TimedMetadata</a>	Class for the raw representation of the timed metadata inserted into a media stream.

## Notification classes

These classes describe messages about errors, warnings, and some activities that the PSDK issues for logging and debugging purposes.

Package: [com.adobe.mediacore](#)

Package: [com.adobe.mediacore.session](#)

Class name	Description
MediaPlayerNotification. <a href="#">Error</a>	Class that describes the notification for an error that causes the player to stop playback. This is a <a href="#">MediaPlayerNotification</a> of notification type ERROR.
MediaPlayerNotification. <a href="#">Info</a>	Class that describes an informational notification. This is a <a href="#">MediaPlayerNotification</a> of notification type INFO.
MediaPlayerNotification. <a href="#">Warning</a>	Class that describes a warning notification. This is a <a href="#">MediaPlayerNotification</a> of notification type WARNING.
<a href="#">MediaPlayerNotification</a>	Class that provides informational messages, warnings, and errors. Encapsulates the object representation of a single notification event within <a href="#">NotificationHistory</a> .
MediaPlayerNotification. <a href="#">NotificationCode</a>	Returns the description associated with the provided notification code.
<a href="#">NotificationHistory</a>	Class that stores a log of notification objects. A circular list of <a href="#">NotificationHistory.Item</a> objects that provides access to a notification events history list. That is, it maintains a list of elements, each element containing a separate instance of the <a href="#">MediaPlayerNotification</a> class. (In <a href="#">session</a> package.)
NotificationHistory. <a href="#">Item</a>	Class that defines an entry in the circular list in <a href="#">NotificationHistory</a> and holds the notification and its timestamp.
MediaPlayerNotification. <a href="#">EntryType</a>	Class that contains the supported notifications types.

## QoS and QoS metrics classes

These classes provide information that help you to determine how well the player is performing.

Package: [com.adobe.mediacore.qos](#)

Package: [com.adobe.mediacore.qos.metrics](#)

Name	Description
metrics. <a href="#">BufferingMetrics</a>	Provides information about how much time the player spent while buffering and how often a buffering event occurred.
<a href="#">DeviceInformation</a>	Provides information about the platform and operating system on which the PSDK runs: <ul style="list-style-type: none"> <li>• Version of the platform OS</li> <li>• Version number of the PSDK library</li> <li>• Device's model name</li> <li>• Device manufacturer's name</li> <li>• Device UUID</li> <li>• Width/height of the device screen</li> </ul>
<a href="#">LoadInfo</a>	Contains various QoS information about loading various resources (files, manifest or playlist, fragments/segments, tracks, and so on).
<a href="#">PlaybackInformation</a>	Provides information on how the playback is performing. This includes the frame rate, the profile bit rate, the total time spent in buffering, the number of buffering attempts,

Name	Description
	the time it took to get the first byte from the first video fragment, the time it took to render the first frame, the currently buffered length, and the buffer time.
metrics. <a href="#">PlaybackLoadMetrics</a>	Provides information on how much time it took for the media to load, how much it took for the player to render the first frame or, in case of an error, to fail.
metrics. <a href="#">PlaybackMetrics</a>	Provides information on how the playback is behaving. This includes the frame rate, bit rate, buffer length, and so on.
metrics. <a href="#">PlaybackSessionMetrics</a>	Provides information on how many seconds the player spent while actually playing and how much time the video was actually on screen.
<a href="#">QOSProvider</a>	Provides essential QoS metrics for both playback and the device.

## Timeline classes

These classes provide information about the timeline of the particular media, including the placement of ads.

Package: [com.adobe.mediacore.timeline](#)

Name	Description
<a href="#">PlacementOpportunity</a>	An opportunity class represents an "interesting" point on the timeline.
<a href="#">Timeline</a>	Interface that provides an iterator for processing timeline markers. Represents the timeline of the content, including ad breaks.
<a href="#">TimelineItem</a>	Class. Generic immutable representation of a timeline item.
<a href="#">TimelineMarker</a>	Interface that represents a marker on the timeline. This marks a region of interest on the actual timeline. Currently, the regions of interest are the ads, which you might want to mark, for example, with a different color on the scrub bar UI. Each marker is defined by a position and a duration (each expressed in milliseconds).
<a href="#">TimelineOperation</a>	Base class for all operations that affect the timeline.

## Timeline advertising classes

These classes provide information about ads that occur within a timeline.

Package: [com.adobe.mediacore.timeline.advertising](#)

Package: [com.adobe.mediacore.timeline.advertising.auditude](#)

Name	Description
<a href="#">Ad</a>	Class that defines the Ad abstraction and holds all ad information. It is defined by a unique ID, a duration, and a MediaResource. The MediaResource contains the URL where the actual ad content resides.
<a href="#">AdAsset</a>	Class that represents an asset to be displayed.
<a href="#">AdBreak</a>	Class that gives a unified view on several ads that will be played at some point during playback.
<a href="#">AdBreakPlacement</a>	Ad break placement operation class.

Name	Description
<a href="#">AdBreakPolicy</a>	Enumeration that defines the ad playback policy related to the user bypassing ads while seeking.
<a href="#">AdClick</a>	Class that represents a click instance associated with an asset. This instance contains information about the click-through URL and the title that can be used to provide additional information to the user.
<a href="#">AdPolicyInfo</a>	Interface that defines properties for AdPolicySelector API calls. These properties provide the context for enforcing each ad behavior.
<a href="#">AdPolicySelector</a>	An ad policy selector interface for enforcing ad behaviors. Applications can conform to this interface by implementing all the required methods or by extending the existing default policy selector class to customize specific behaviors.
auditude.AuditadeAdProvider	Deprecated. Use AuditadeResolver.
auditude. <a href="#">AuditadeResolver</a>	Class that handles primetime ad resolving in the Ad Decisioning process.
auditude. <a href="#">AuditadeTracker</a>	Class that implements the ContentTracker interface and defines Primetime ad-tracking events.
<a href="#">ContentResolver</a>	Class that handles the ad-resolving part in the Ad Decisioning process.
<a href="#">ContentTracker</a>	Interface that defines the protocol that you must implement if you want to create an ad-tracking module that is designed to integrate with the PSDK library or a custom ad tracker.  This interface requires that you define the way ad-progress events are reported to the remote ad-tracking system.
<a href="#">PlacementInformation</a>	Class that abstracts a placement information request. Each resolved ad must have one placement information attached to it. The placement information describes where the ad is intended to be placed on the timeline. It contains information such as: <ul style="list-style-type: none"> <li>• Placement position (in ms)</li> <li>• Type of the placement (pre-roll, mid-roll, or post-roll)</li> <li>• Duration of the main content chunk that is about to be replaced</li> </ul>

## Utility classes

These classes provide ways to process various types of information.

Package: [com.adobe.mediacore.utils](#)

Class name	Description
<a href="#">DateUtils</a>	Methods for processing dates.
<a href="#">NumberUtils</a>	Helper methods related to numbers.
<a href="#">StringUtils</a>	Helper methods related to strings.
<a href="#">TimeRange</a>	Methods for creating and interpreting time ranges.

## Notification codes

The PSDK notification system includes various classes of notification events produced by the PSDK.

These notification events expose numerical codes that are grouped into ranges of integer values. The ranges provide two levels of classification.

Notification events are grouped in the following top-level classifications:

- Error notification events, from 100000 to 199999
- Warning notification events, from 200000 to 299999
- Information notification events, from 300000 to 399999

Inside each top-level range, subranges further classify notifications. Each subrange contains up to 1000 values, so each top-level range can contain up to 100 second-level subranges.

### ERROR notification codes

Code	Name	InnerNotification	Metadata Keys	Comments
<b>Playback</b>				
101000	PLAYBACK_ERROR	None	DESCRIPTION	
101004	CONTENT_ERROR	DOWNLOAD_ERROR		An Error has occurred while downloading a fragment or segment(both video and audio).
101008	SEEK_ERROR	None	NATIVE_ERROR_CODE DESIRED_SEEK_POSITION DESIRED_SEEK_PERIOD	An error has occurred while performing a seek operation.
101009	PAUSE_ERROR	None	DESCRIPTION	An error has occurred while performing a pause operation.
101102	PERIOD_INFO_ERROR	None	DESCRIPTION	An error has occurred while retrieving information about a content period.
101103	RETRIEVE_TIME_ERROR	None	DESCRIPTION	An error has occurred while attempting to retrieve the playback position.
101104	GET_QOS_DATA_ERROR	None	DESCRIPTION	An error has occurred while attempting to retrieve the QOS information.

Code	Name	InnerNotification	Metadata Keys	Comments
101200	DOWNLOAD_ERROR	None	URL	An error has occurred while attempting to download data.
<b>Invalid resource</b>				
102100	RESOURCE_LOAD_ERROR	None	DESCRIPTION RESOURCE	An error has occurred while loading a resource item.
102101	RESOURCE_INSERT_FAILED	None	CONTENT_ID	An error has occurred while placing a resource on the playback timeline.
<b>Ad processing</b>				
104000	AD_RESOLVER_FAIL	AD_METADATA_INVALID AD_RESOLVER_INITIALIZATION_FAIL AD_RESOLVER_RESOLVE_FAIL AD_RESOLVER_SERVER_UNREACHABLE	None	None
104001	AD_RESOLVER_METADATA_INVALID	None	DESCRIPTION	Ad resolving failed due to invalid ad-metadata format.
104003	AD_RESOLVER_RESOLVE_FAIL	None	NATIVE_ERROR_CODE	Ad plugin failed to resolve ads.
104005	AD_INSERTION_FAIL	None	PROPOSED_AD_BREAK	Ad resolving phase has failed.
<b>Native</b>				
106000	NATIVE_ERROR	None	NATIVE_ERROR_CODE NATIVE_ERROR_NAME DESCRIPTION DESCRIPTION	The low-level AVE library issued an error. DESCRIPTION is the error description. It is usually the URL of the media.
			<b>DRM details</b> DRM_ERROR_STRING NATIVE_SUBERROR_CODE	May also be returned for DRM errors.
106001	ENGINE_CREATION_ERROR	None	DESCRIPTION	An error has occurred while instantiating the AVE low-level library.
106002	ENGINE_RELEASE_ERROR	None	DESCRIPTION	An error has occurred while releasing the AVE low-level library.

Code	Name	InnerNotification	Metadata Keys	Comments
106003	<del>ENGINE_RESOURCES_ERROR</del>	None	DESCRIPTION	An error has occurred while releasing the GPU resources utilised by the AVE library.
106004	ENGINE_RESET_ERROR	None	DESCRIPTION	An error has occurred while resetting the AVE library.
106005	ENGINE_SET_VIEW_ERROR	None	DESCRIPTION	An error has occurred while attaching a view to the AVE library.
<b>Configuration</b>				
107000	SET_VOLUME_ERROR	None	DESCRIPTION VOLUME	An error has occurred while attempting to set the volume level.
107001	SET_BUFFER_TIME_ERROR	None	DESCRIPTION PLAY_BUFFER_TIME	An error has occurred while attempting to change the buffering parameters.
107002	SET_CC_VISIBILITY_ERROR	None	DESCRIPTION	An error has occurred while attempting to change the visibility of the CC tracks.
107003	SET_CC_STYLING_ERROR	None	DESCRIPTION	An error has occurred while attempting to change the styling options for the CC tracks.
107004	SET_ABR_PARAMETERS_ERROR	None	DESCRIPTION	An error has occurred while attempting to change the ABR control parameters.
107005	SET_BUFFER_PARAMETERS_ERROR	None	DESCRIPTION INITIAL_BUFFER_TIME PLAY_BUFFER_TIME	An error has occurred while attempting to change the buffering control parameters.
<b>Alternate audio</b>				
109000	AUDIO_TRACK_ERROR	DOWNLOAD_ERROR	AUDIO_TRACK_NAME AUDIO_TRACK_LANGUAGE	An error related to an audio track occurred.
<b>Generic</b>				

Code	Name	InnerNotification	Metadata Keys	Comments
199999	GENERIC_ERROR	None	None	Marks a generic error event. Not actually issued by the PSDK. It's just a marker for the end of the range of numerical codes corresponding to PSDK error events.

## WARNING notification codes

Most warnings contain relevant metadata (for example, the URL of the resource that failed to be downloaded). Some warnings contain metadata to specify whether the problem occurred in the main video content, in the alternate audio content, or in an ad.

Code	Name	InnerNotification	Metadata Keys	Comments
<b>Playback</b>				
200000	PLAYBACK_OPERATION_FAIL	AUDIO_TRACK_ERROR SEEK_ERROR	DESCRIPTION	A playback-related operation has failed, but playback may continue.
<b>Ad-resolving</b>				
201000	AD_RESOLVER_FAIL	AD_RESOLVER_RESOLVE_FAIL RESOURCE_PLACEMENT_FAILED AD_RESOLVER_METADATA_INVALID	None	The ad-resolver has failed to resolve/insert the ad content. Playback may continue.
<b>Trick mode</b>				
280000	TRICKLAY_RATE_CHANGE_FAIL	None	DESCRIPTION	Rate change failed.
<b>Native</b>				
209100	NATIVE_WARNING	None	<b>AVE</b> NATIVE_ERROR_CODE NATIVE_ERROR_NAME DESCRIPTION	The low-level AVE library issued an error. DESCRIPTION is the error description. It is usually the URL of the media.  See <a href="#">DRM errors exposed through AVE</a> .
			<b>DRM</b> NATIVE_SUBERROR_CODE DRM_ERROR_STRING	DRM minor error code and DRM server error string. See <a href="#">NATIVE_SUBERROR_CODE values</a> .
<b>Generic</b>				

Code	Name	InnerNotification	Metadata Keys	Comments
299999	GENERIC_WARNING	None	None	Marks a generic warning event. Not actually issued by the PSDK. It's just a marker for the end of the range of numerical codes corresponding to PSDK warning events.

## INFO notification codes

Code	Name	InnerNotification	Metadata Keys	Comments
<b>Playback</b>				
300000	PLAYBACK_START	None	None	Notifies that playback has started
300001	PLAYBACK_COMPLETE	None	None	Notifies that playback has completed
300002	SEEK_START	None	SEEK_TIME	Notifies that a seek operation was initiated
300003	SEEK_COMPLETE	None	SEEK_TIME	Notifies that a seek operation has completed.
300004	CONTENT_CHANGE	None	CONTENT_ID CURRENT_MEDIA_TIME	Notifies that the current playback time has crossed the border between main/alternate content.
300005	PLAYER_STATE_CHANGE	Any ERROR notification.	STATE	Notifies that the player state has changed. When state is ERROR, the inner notification is the error notification object that triggered the switch to the ERROR state.
300006	CONTENT_MARKER	None	CONTENT_ID CURRENT_MEDIA_TIME	Content marker received.
300100	LOAD_INFO_AVAILABLE	None	FRAGMENT_URL FRAGMENT_SIZE FRAGMENT_DOWNLOAD_DURATION PERIOD_INDEX	Provides info related to the way video segments are being downloaded.
300101	VIDEO_SIZE_CHANGED	None	HEIGHT WIDTH	Notifies that the size of the video playback window has changed.
<b>Adaptive bit rates (ABR)</b>				
302000	BITRATE_CHANGE	None	BITRATE CURRENT_MEDIA_TIME	Notifies that the bitrate of the video has changed
<b>Ad Processing</b>				

Code	Name	Notification	Metadata Keys	Comments
303000	TIMELINE_CHANGE	None	CONTENT_ID PERIOD_INDEX	Notifies that the timeline has changed (e.g. alternate content was added / removed).
303001	AD_BREAK_PLACEMENT_COMPLETE	None	PROPOSED_AD_BREAK ACCEPTED_AD_BREAK	Notifies that a proposed ad break was accepted by the PSDK and placed (in its entirety or just partially) on the playback timeline.
303002	AD_BREAK_START	None	AD_BREAK	Notifies that the playback of a particular ad-break has started.
303003	AD_BREAK_COMPLETE	None	AD_BREAK	Notifies that the playback of a particular ad-break has completed.
303004	AD_START	None	AD_BREAK AD	Notifies that the playback of a particular ad has started.
303005	AD_COMPLETE	None	AD_BREAK AD	Notifies that the playback of a particular ad has completed.
303006	AD_PROGRESS	None	AD_BREAK AD PROGRESS	Notifies that the playback of a particular ad has reached a certain percentage of that particular ad.
303007	TIMED_METADATA_ADD	None	TYPE ID NAME TIME	Notifies that a new timed metadata was discovered in the manifest.
303008	AD_CLICK	None	AD_BREAK AD AD_CLICK	Returns info about an ad that user clicked. (?)
<b>Late-binding audio (LBA)</b>				
304000	AUDIO_TRACK_CHANGE	None	TRACK_ID CURRENT_MEDIA_TIME	Notifies that the audio track has been changed.
<b>DRM</b>				
305000	DRM_METADATA_AVAILABLE	None	PREFETCH_TIMESTAMP	Notifies that new DRM data available.

Code	Name	Notification	Metadata Keys	Comments
<b>Generic</b>				
399999	GENERIC_INFO	None	None	Marks a generic information event. Not actually issued by the PSDK. It's just a marker for the end of the range of numerical codes corresponding to PSDK informational events.

## Details for the NATIVE\_ERROR notification

When the Primetime SDK handles a native error (notification type of NATIVE\_ERROR), it sets some or all of the following Metadata key values.

Metadata key name	Metadata value
NATIVE_ERROR_CODE	Native error code from the AVE. These codes represent the following: <ul style="list-style-type: none"> <li>• DRM errors (codes 3300 to 3367). These are the same as the equivalent Flash Player error codes.</li> <li>• Video playback errors (-1 to 89).</li> <li>• Cryptography errors (300 to 307).</li> </ul>
NATIVE_ERROR_NAME	A string that contains the error's name; for example, <code>AAXS_InvalidVoucher</code> or <code>DECODER_FAILED</code> .
NATIVE_SUBERROR_CODE	For DRM errors, suberror codes are also returned. These codes correspond to the <code>DRMErrorEvents</code> suberror code that is returned by Flash Player. When reporting errors to Adobe, include this numeric value for troubleshooting assistance.
DRM_ERROR_STRING	For DRM errors, this is your custom error string from your DRM server deployment, if you defined any. Also include this when reporting errors to Adobe.
DESCRIPTION	String description of the error.

The PSDK receives these error codes and strings from the AVE.

### DRM error values

The Video Encoder interface of the AVE returns these DRM notifications in the NATIVE\_ERROR metadata object. When reporting DRM errors to Adobe, also include the NATIVE\_SUBERROR\_CODE and DRM\_ERROR\_STRING for troubleshooting assistance.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Meaning
3300	<code>AAXS_InvalidVoucher</code>	Reacquire license from the server.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Meaning
3301	AAXS_AuthenticationFailed	Ask user to reenter user credentials and retry license acquisition.
3302	AAXS_RequireSSL	Not in use.
3303	AAXS_ContentExpired	
3304	AAXS_AuthorizationFailed	Current user not authorized to view content. Log in as different user.
3305	AAXS_ServerConnectionFailed	Check network connection.
3306	AAXS_ClientUpdateRequire	Use a newer version of the PSDK fro Android.
3307	AAXS_InternalFailure	
3308	AAXS_WrongLicenseKey	Reacquire license from the server.
3309	AAXS_CorruptedAdditionalHeader	Redownload the content.
3310	AAXS_AppIDMismatch	The Android application does not match the one in use.
3311	AAXS_AppVersionMismatch	Not in use.
3312	AAXS_LicenseIntegrity	Redownload the license from the server.
3313	AAXS_WriteMicrosafeFailed	A file system error occurred.
3314	AAXS_CorruptedDRMMetadata	Rdownload the DRMMetadata.
3315	AAXS_PermissionDenied	Not in use.
3316	AAXS_NOTUSED_MOVED	Was ADOBECPSHIM_MinorErr_MissingAdobeCPModule. Moved to 3344 due to conflict with Flash error code.
3317	AAXS_LoadAdobeCPFailed	You might have used a corrupted version of the PSDK; make sure that you have the latest version.
3321	AAXS_I15nFailed	Retry the operation. Hopefully Adobe run Indiv server is functional again.
<b>Global Store corruption errors</b>		
3322	AAXS_DeviceBindingFailed	In response, either undo device changes or deindiv, client will re-indiv
3323	AAXS_CorruptGlobalStateStore	Retry the operation at least once and if that gives the same error, then deindiv, client will auto re-indiv.
<b>Server Store corruption errors</b>		
3325	AAXS_CorruptServerStateStore	Try the operation again. The offending server store has been deleted internally.
3326	AAXS_StoreTamperingDetected	Call DRMManager.resetDRM.
3327	AAXS_ClockTamperingDetected	Fx the clock or reacquire license.
<b>Authentication/License/Domain server errors</b>		
3328	AAXS_ServerErrorTryAgain	Server is misbehaving. Try the request again.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Meaning
3329	AAXS_ApplicationSpecificError	Server returned an error in application specific namespace. Suberror code is application specific, not FMRMS known
3330	AAXS_NeedAuthentication	Authenticate the user and then acquire the license again.
<b>License Enforcement errors that aren't covered above</b>		
3331	AAXS_ContentNotYetValid	Content is not yet valid.
3332	AAXS_CachedLicenseExpired	Reacquire the license by going to the server.
3333	AAXS_PlaybackWindowExpired	Tell the user that they can no longer play this content until the policy expires.
3334	AAXS_InvalidDRMPlatform	This platform is not allowed to play back the content.
3335	AAXS_InvalidDRMVersion	Upgrade to the latest version of the PSDK for Android.
3336	AAXS_InvalidRuntimePlatform	This platform is not allowed to play back the content.
3337	AAXS_InvalidRuntimeVersion	Upgrade to the latest version of the PSDK for Android.
3338	AAXS_UnknownConnectionType	Cannot detect the connection type. "Connection" refers to external display connection, for example, when you try to output your tablet's display to a TV or external monitor.
3339	AAXS_NoAnalogPlaybackAllowed	Can't play back on analog device. Connect a digital external display device (monitor/tv).
3340	AAXS_NoAnalogProtectionAvail	Can't play back because connected analog external display device (monitor/tv) doesn't have the correct capabilities .
3341	AAXS_NoDigitalPlaybackAllowed	Can't play back on digital external display device (monitor/tv). This should never happen.
3342	AAXS_NoDigitalProtectionAvail	The connected digital external display device (monitor/tv) doesn't have the correct capabilities.
3346	AAXS_MigrationFailed	License migration failed.
3347	AAXS_InsufficientDeviceCapabilites	The playback device doesn't have sufficient device capabilities.
3348	AAXS_HardStopIntervalExpired	Hard stop interval expired.
3349	AAXS_ServerVersionTooHigh	Server is running at a version that is higher than the max supported by client.
3350	AAXS_ServerVersionTooLow	Server is running at a version that is lower than the min supported by client.
3351	AAXS_DomainTokenInvalid	Domain token was invalid. In response, reregister with the domain.
3352	AAXS_DomainTokenTooOld	Domain token is older than that required by the license. In response, reregister with the domain.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Meaning
3353	AAXS_DomainTokenTooNew	Domain token is newer than that required by the license.
3354	AAXS_DomainTokenExpired	Domain token has expired.
3355	AAXS_DomainJoinFailed	Domain join failed.
3356	AAXS_NoCorrespondingRoot	A root license for a V3 leaf license was not found.
3357	AAXS_NoValidEmbeddedLicense	No valid embedded license found.
3358	AAXS_NoACPProtectionAvail	Can't play back because connected analog device doesn't have ACP protection.
3359	AAXS_NoCGMSAProtectionAvail	Can't play back because connected analog device doesn't have CGMSA protection.
3360	AAXS_DomainRegistrationRequired	Domain registration required.
3361	AAXS_NotRegisteredToDomain	Machine is not registered to the domain for the specified metadata.
3362	AAXS_OperationTimeoutError	Asynchronous operation took longer than maxOperationTimeout. Only returned by iOS DRMNative Framework currently.
3363	AAXS_UnsupportedIOSPlaylistError	The M3U8 playlist passed in had unsupported content. Only returned by iOS DRMNative Framework.
3364	AAXS_NoDeviceId	The framework requested the device ID, but the returned value was empty.
3365	AAXS_IncognitoModeNotAllowed	This browser/platform combination does not allow DRM protected playback when in incognito mode.
3366	AAXS_BadParameter	A bad parameter was passed to one of the Access APIs. Only returned by iOS DRMNative Framework or AVE.
3367	MAX_ERRORS	Max value of this enumeration.

### Video playback error values

The Video Encoder interface of the AVE returns these video playback notifications in the NATIVE\_ERROR metadata object.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Description
-1	END_OF_PERIOD	End of period.
0	SUCCESS	Operation successful.
1	ASYNC_OPERATION_IN_PROGRESS	Asynchronous operation. The operation request has been made. Success/failure information will be available later.
2	EOF	Operation not possible due to end of file (EOF) condition.
3	DECODER_FAILED	The decoder failed at runtime.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Description
4	DEVICE_OPEN_ERROR	Failed to open hardware decoder.
5	FILE_NOT_FOUND	Resource cannot be located.
6	GENERIC_ERROR	Generic error.
7	IRRECOVERABLE_ERROR	An error condition that the Video Engine cannot recover from.
8	LOST_CONNECTION_RECOVERABLE	Network error, trying to recover.
9	NO_FIXED_SIZE	The size of the resource cannot be determined.
10	NOT_IMPLEMENTED	Feature not implemented.
11	OUT_OF_MEMORY	Out of memory.
12	PARSE_ERROR	Error while parsing the media file.
13	SIZE_UNKNOWN	The resource has a size, but it is unknown.
14	UNDER_FLOW	Underflow condition.
15	UNSUPPORTED_CONFIG	Configuration is not supported.
16	UNSUPPORTED_OPERATION	Operation is not supported.
17	WAITING_FOR_INIT	Not yet initialized.
18	INVALID_PARAMETER	Invalid parameter.
19	INVALID_OPERATION	Operation not permitted.
20	OP_ONLY_ALLOWED_IN_PAUSED_STATE	The operation is allowed only while paused.
21	OP_INVALID_WITH_AUDIO_ONLY_FILE	Operation cannot be used on audio only files.
22	PREVIOUS_STEP_SEEK_IN_PROGRESS	Previous seek operation is still in progress.
23	SOURCE_NOT_SPECIFIED	Resource not specified.
24	RANGE_ERROR	Specified value is out of range.
25	INVALID_SEEK_TIME	Invalid seek time.
26	FILE_STRUCTURE_INVALID	The file specified does not conform to the expected syntax.
27	COMPONENT_CREATION_FAILURE	An essential component could not be created.
28	DRM_INIT_ERROR	Failed to create DRM context.
29	CONTAINER_NOT_SUPPORTED	Container type is not supported.
30	SEEK_FAILED	Seek failed.
31	CODEC_NOT_SUPPORTED	Unsupported codec.
32	NETWORK_UNAVAILABLE	Network is not available.
33	NETWORK_ERROR	Error getting data from the Network.
34	OVERFLOW	Overflow.
35	VIDEO_PROFILE_NOT_SUPPORTED	Unsupported video profile.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Description
36	PERIOD_NOT_LOADED	An operation was attempted on a HOLD period or a period that has not yet been loaded.
37	INVALID_REPLACE_DURATION	The replace duration specified is invalid or extends past the end of the stream.
38	CALLED_FROM_WRONG_THREAD	API can't be called from the wrong thread. Mostly, for API elements that should be called from Main thread only.
39	FRAGMENT_READ_ERROR	Fragment read error. No failover present. Engine will try to read the next fragment.
40	ABORTED	The operation was aborted by an explicit Abort or Destroy call.
41	UNSUPPORTED_HLS_VERSION	Cannot play this version of HLS media.
42	CANNOT_FAIL_LOVER	Cannot fail over.
43	HTTP_TIME_OUT	HTTP download has timed out.
44	NETOWRK_DOWN	The user's network connection is down. Playback could stop any moment and will resume when the connection is available.
45	NO_USABLE_BITRATE_PROFILE	No usable bit-rate profile found in the stream.
46	BAD_MANIFEST_SIGNATURE	The manifest has a bad signature. It failed the manifest signing test.
47	CANNOT_LOAD_PLAYLIST	Cannot load a playlist.
48	REPLACEMENT_FAILED	Replacement specified in an Insert API could not succeed. This means that the insertion succeeded but replacement did not. Replacement could fail if the manifest to be replaced has been removed from the timeline.
49	SWITCH_TO_ASYMMETRIC_PROFILE	DRM is switching to an asymmetric profile. All the profiles are expected to be aligned in duration. If not, this warning will be thrown, and there may be jumps in the playback.
50	LIVE_WINDOW_MOVED_BACKWARD	Live window is expected to move forward only. If not, this warning will be thrown, and the window will not be read. Because of that, there may be jumps (or stop / long pause) in the playback.
51	CURRENT_PERIOD_EXPIRED	Live window moved beyond the current period.
52	CONTENT_LENGTH_MISMATCH	The content-length reported by the HTTP server did not match the actual media size.
53	PERIOD_HOLD	The media reader is unable to read further because it has reached the time set by setHoldAt API.
54	LIVE_HOLD	The media reader is unable to load segments because it has reached the end of the live window. Segment loading will resume when the server ads new media to the live window. This state is usually reached if:

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Description
		<ul style="list-style-type: none"> <li>The bufferTime is too high (equal to or higher than the live window duration).</li> <li>A combination of one or more of insert/erase API replaced more media than it added.</li> <li>The next period is a live period with a pending media replacement (due to InsertBy API call)</li> </ul>
55	BAD_MEDIA_INTERLEAVING	Indicates that the audio and video interleaving in the media is not done properly. This is a packaging error. The warning is dispatched when the difference exceeds two seconds.
56	DRM_NOT_AVAILABLE	
57	PLAYBACK_NOT_AUTHORIZED	HLS playback has not been enabled in the Flash Player. See <code>AuthorizedFeatures.enableHLSPlayback</code> .
58	BAD_MEDIA_SAMPLE_FOUND	The decoder received a bad sample that cannot be decoded. This is usually not a fatal error but indicates that there may be glitches in the audio/video. Too many instances of this error indicate a bad encoding or bad file.
59	RANGE_SPANS_READ_HEAD	After playback has started, the Insert/Replace range should not contain the read head.
60	POSTROLL_WITH_LIVE_NOT_ALLOWED	Post-roll insertions are not allowed on a live media. They are, however, allowed after the server marks the media as complete.
61	INTERNAL_ERROR	A very rare issue that should never have happened.
62	SPS_PPS_FOUND_OUTSIDE_AVCC	The stream does not follow the packaging recommendation of always putting H264 SPS/PPS in an AVCC. Seek / playback issues might be seen.
63	PARTIAL_REPLACEMENT	Replacement specified in an Insert API was only partially done. This happens when <code>replaceDuration</code> spans over the timeline duration.
64	RENDITION_M3U8_ERROR	Rendition playlist had an error loading. This is only for AVE, not for FlashPlayer.
65	NULL_OPERATION	Operation does not do anything.
66	SEGMENT_SKIPPED_ON_FAILURE	Segment cannot be played and is skipped on failure.
67	INCOMPATIBLE_RENDER_MODE	Incompatible render mode.
68	PROTOCOL_NOT_SUPPORTED	The Web protocol used in the URL is not supported.
69	PARSE_ERROR_INCOMPATIBLE_VERSION	Error while parsing media file.
70	MANIFEST_FILE_UNEXPECTEDLY_CHANGED	Manifest file was changed in an unexpected manner.
71	CANNOT_SPLIT_TIMELINE	Cannot perform a split operation on a timeline.
72	CANNOT_ERASE_TIMELINE	Cannot perform an erase operation on a timeline.

Metadata code value for NATIVE_ERROR_CODE key	Metadata name value for NATIVE_ERROR_NAME key	Description
73	DID_NOT_GET_NEXT_FRAGMENT	Did not get the next fragment.
74	NO_TIMELINE	No timeline present in an internal data structure.
75	LISTENER_NOT_FOUND	No listener found in an internal data structure.
76	AUDIO_START_ERROR	Unable to start audio.
77	NO_AUDIO_SINK	No audio sink present in an internal data structure.
78	FILE_OPEN_ERROR	Unable to open file.
79	FILE_WRITE_ERROR	Unable to write to a file.
80	FILE_READ_ERROR	Unable to read from a file.
81	ID3PARSE_ERROR	There was an error parsing ID3 data.
82	SECURITY_ERROR	Loading the content failed because of security restrictions.
83	TIMELINE_TOO_SHORT	The timeline duration is too short. If this is a live stream, frequent buffering may happen.
84	AUDIO_ONLY_STREAM_START	The stream has been switched to an audio-only stream.
85	AUDIO_ONLY_STREAM_END	The stream has been switched from audio-only to a stream with video.
87	KEY_NOT_FOUND	Key cannot be found.
88	INVALID_KEY	The key is invalid.
89	KEY_SERVER_NOT_FOUND	Key server does not return a key.

### Crypto error values

The crypto module of the AVE returns these notifications in the NATIVE\_ERROR metadata object.

Metadata code value for NATIVE_ERROR_CODE	Metadata name value for NATIVE_ERROR_NAME key	Meaning
300	CRYPTO_ALGORITHM_NOT_SUPPORTED	Algorithm being used is not supported.
301	CRYPTO_ERROR_CORRUPTED_DATA	Data is corrupted.
302	CRYPTO_ERROR_BUFFER_TOO_SMALL	Buffer too small.
303	CRYPTO_ERROR_BAD_CERTIFICATE	Bad certificate.
304	CRYPTO_ERROR_DIGEST_UPDATE	Digest update.
305	CRYPTO_ERROR_DIGEST_FINISH	Digest finish.
306	CRYPTO_ERROR_BAD_PARAMETER	Bad parameter.
307	CRYPTO_ERROR_UNKNOWN	Unknown crypto error.

## Glossary

### A

#### **ABR**

Adaptive bit rate.

Based on the bandwidth conditions and the quality of playback (frame rate), the video engine automatically switches the quality level (bit rate) to provide the best playback experience.

#### **ABR algorithm**

The algorithm that determines which bit rate the client bandwidth can handle.

#### **adaptive set**

A set of renditions, typically from the same CDN. Within a set, renditions represent different bit rates.

#### **ad break**

One or more ads grouped to play in sequence. Even a single ad is enclosed in an ad break.

#### **AVE**

Adobe Video Engine.

### C

#### **CDN**

Content delivery network.

#### **cue point**

An #EXT tag in an M3U8 manifest file that the PSDK uses for splicing ads into a live stream.

### D

#### **DRM**

Digital rights management. Ensuring that content can be accessed or viewed only by users who have the appropriate permissions.

#### **DVR**

Digital video recorder.

## F

### **full-event replay**

A VOD asset that acts, in terms of ad insertion, as a live/DVR asset, so that your application must ensure that ads are placed correctly.

### **fragment**

One segment of one representation of the media presentation.

## H

### **HDS**

HTTP Dynamic Streaming

This technology is from Adobe. The format works in OSMF 1.6 and later. HTTP Dynamic Streaming reproduces much of the functionality of RTMP delivery, providing the publisher a choice in delivery options. The primary benefit that HTTP offers is its ability to cache content, which is important for enterprise customers who deploy internal caching systems to optimize network usage.

### **HLS**

HTTP Live Streaming

This Apple technology is a required format for delivery to Apple devices. It works in HTML5 browsers, AIR for iOS player, and native iOS apps.

## I

### **iframe**

An HTML iframe. An inline frame places another HTML document in a frame. An inline frame can be the target frame for links (URLs) defined by other elements, and it can be selected by the user agent as the focus for printing, viewing its source, and so on. The content of the element is used as alternative text to be displayed if the browser does not support iframes.

### **I-frame**

In video compression, an I-frame (also called a key frame) is an Intra-coded picture, in effect a fully specified picture, like a conventional static image file. P-frames and B-frames hold only part of the image information, so they need less space to store than an I-frame, and thus improve video compression rates.

## K

### **key frame**

See I-frame.

## L

### **linear video**

Linear video is streamed video with advertisements already stitched into the stream. An example of linear video could be a programmed television show.

### **live video**

Video that is being streamed from a live event. Advertisements are typically overlaid on live video. An example of live video could be a live sporting event.

## M

### **main content**

Represents the movie or streaming video.

### **MBR**

Multiple bit rate. A video asset that has more than one bit-rate rendition available.

### **manifest**

An M3U8 document/file that defines the protocol for transferring multimedia data streams, including specifying the data format and the actions to be taken by the sender and the receiver. Also called a playlist.

### **multiplexed**

Audio and video are contained in the same rendition rather than in separate renditions.

### **muxed**

Multiplexed.

## N

### **national ads**

Ads that are placed in the main content, but not targeted.

## P

### **PHDS**

Protected HTTP Dynamic Streaming.

### **PHLS**

Protected HTTP Live Streaming.

### **playlist**

See manifest.

## Q

### QoS

Quality of service.

## R

### rendition

One representation of the media presentation.

### RTA

Real-time analytics.

## S

### SBR

Single bit rate.

### segment

See fragment.

## T

### targeted ads

Ads that are served to a user based on received information about that user.

## V

### VOD

Video on demand.

Video that you watch at your convenience. An example of VOD is a video that you can download on your device from a video publishing service.

## Copyright

© 2014 Adobe Systems Incorporated. All rights reserved.

Adobe Primetime PSDK 1.3 BETA for Android Programmer's Guide

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.