Adobe® Primetime

PSDK 1.2 for Android Programmer's Guide

Updated 14 April 2014

Contents

PSDK for Android Programmer's Guide.....5

Overview.....5

Overview of the Player SDK.....5

Considerations.....5

Requirements.....6

Best practices.....6

PSDK changes for 1.2.....6

New features for 1.2.....8

Create a video player.....9

Set up the MediaPlayer.....9

Listen for Primetime Player events.....9

Set up error handling.....13

Configure the player user interface.....13

Initialize the MediaPlayer for a specific video.....19

Monitor quality of service statistics.....23

HTTP 302 redirect optimization.....25

Work with MediaPlayer objects.....25

About the MediaPlayerItem class.....25

Lifecycle and states of the MediaPlayer object.....25

MediaPlayer methods for accessing MediaResource information.....28

Reuse or remove a MediaPlayer instance.....29

Inspect the playback timeline.....30

Configure buffering.....31

Include advertising.....32

Requirements for advertising.....32

About ad insertion.....32

Metadata for ad insertion.....34

Clickable ads.....37

Repackage third-party ads.....40

Subscribe to custom tags.....40

Use a custom ad provider and custom opportunity detector....44
Add custom ad markers....48
TimeRange class....48
MediaPlayer and MediaResource classes....48
TimeRangeCollection class....49
Place TimeRange ad markers on the timeline....49
Control the seek-over-ads behavior....50
Set up closed captioning....51
Control closed-caption visibility....51
Control closed-caption styling....53
Set up alternate audio....57
Alternate audio tracks in the playlist....57
Access alternate audio tracks....58
Integrate with Video Analytics....58
Content security using DRM....58
DRM interface overview....58
DRM authentication before playback....59
DRM authentication during playback....62
Use the notification system....63
Notification content....63
Set up your notification system....64
Add real-time logging and debugging....64
Retrieve ID3 metadata....65
Playback and failover....66
Media playback and failover....66
Advertising insertion and failover for VOD....67
Primetime player classes summary....69
Mediacore classes....69
Info classes....70
Logging classes....70
Metadata classes....70
Notification classes....71
QoS and QoS metrics classes....71
Timeline classes 72

Timeline advertising classes....73
Utility classes....73
Notification codes....74
ERROR notification codes....74
WARNING notification codes....77
INFO notification codes....78
Glossary....80
A....80
C....80
D....80
F....80
H....80
I....81
L....81
M....81
N....81
P....82
Q....82
R....82
S....82
T....82
V....82
Copyright....82

PSDK 1.2 for Android Programmer's Guide

Overview

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
- Integration with Video Analytics
- Closed captioning (608, 708, WebVTT) and alternate forms of audio for increased accessibility
- DVR capability, including trick play
- Bit-rate controls and adaptive bit-rate logic
- Custom cue tags for ads
- Failover
- Debug logging
- Adjustable playback buffers
- 302 redirect optimization

Considerations

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

- Playback is supported only for HTTP Live Streaming (HLS) content.
- 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.

Requirements

The Android player requires specific properties for content and DRM.

- 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.

Best practices

These are recommended practices for the Adobe PSDK for Android .

- Use HLS Version 3.0 or above for program content.

PSDK changes for 1.2

Several API interfaces have changed for the 1.2 PSDK for Android .

Element	Description
MediaPlayer	The only supported implementation is the DefaultMediaPlayer class, which is final and is owned by PSDK. Custom implementations are not allowed.

Element	Description
prepareToPlay new parameter	`void prepareToPlay(long position)` New optional position parameter. Both in VoD and DVR, the playback might not start at the beginning of the playback window. As the start position affects the placement of the pre-roll ads and the ad resolving process, the position parameter allows you to specify the initial position of the playback. prepareToPlay() is still supported.
seek behavior	`void seek(long position)` No longer ignores invalid seek positions. Now adjusts the provided position to a location inside the seekable range. Otherwise, seeking too close to the live point or to the beginning of a rolling window could result in playback stalls or even unrecoverable playback errors.
replaceCurrentItem replaced with replaceCurrentResource	`void replaceCurrentResource(MediaResource resource)` Use replaceCurrentResource instead of replaceCurrentItem, which still works correctly in this release but is deprecated.
TimeRange behavior	Although for VoD streams any location in the stream is "seekable", this is not true for Live/Linear. The returned range is now inside the range returned by getPlaybackRange(). `TimeRange getSeekableRange()`
AuditudeMetadata renamed to AuditudeSettings	• All Auditude related settings are now in a base class called AuditudeSettings. • AuditudeMetadata extends the AuditudeSettings class (providing no additional logic) for compatibility. • AuditudeMetadata class is deprecated.
AuditudeAdProvider renamed to AuditudeResolver	• All Auditude ad resolving logic is now in a base class called AuditudeResolver • AuditudeAdProvider extends the AuditudeResolver class (providing no additional logic) for compatibility • AuditudeAdProvider class is deprecated
AuditudeAdTracker renamed to AuditudeTracker	• All Auditude ad tracking logic is now in a base class called AuditudeResolver • AuditudeAdTracker extends the AuditudeResolver class (providing no additional logic) for compatibility • AuditudeAdTracker class is deprecated
com.adobe.mediacore classes made private:	No longer public.

Element	Description
• AdHandler • ContentCache • ContentLoader • DefaultAdBreakPolicySelector • PlayerOperation • PlayOperation • SeekOperation	These were exposed only in development drops provided between 1.1 and 1.2.
MediaPlayerState new state INITIALIZING	A new state, INITIALIZING, comes between IDLE and INITIALIZED. The sequence is now IDLE - INITIALIZING - INITIALIZED - PREPARING - PREPARED - PLAYING
Version property apiVersion	New integer property apiVersion on the static Version class. `public static int getApiVersion()`

New features for 1.2

- **On-the-fly transcoding or creative repackaging**

  Some third-party ads (creatives) might be available only as a progressive download MP4 video, in which case they cannot be stitched into the HLS content stream. Primetime Ad Insertion and the Android PSDK provide an option called third-party creative repackaging, which addresses this situation.

  On the fly ad transcoding is supported through Adobe Ad Serving. The Adobe Ad Server account must be configured for creative repackaging on the Ad Server and then repackaging is seamlessly handled through the PSDK.

- **Support for 708 and WebVTT captions**

  These closed-caption types are automatically supported with no changes required in your application.

- **Full-event replay (FER)**

  Full-event replay (FER) is a VOD asset that acts, in terms of ad insertion, as a live/DVR asset. Your application can now ensure that ads are placed correctly by selecting the ad-signaling mode.

- **Support for DVR replay with ad insertion**

  This includes the addition of a new player state, INITIALIZED, and some changed behavior related to that state.

- **Use of MediaCodec**

  MediaCodec decoder replaces the Open MAX AL decoder for Android version 4.x and above. This improves performance and reliability and allows implementation of additional playback features.

- **302 redirect optimization**

When a request is redirected due to load balancing, the PSDK continues to use the redirected domain for future requests.

- **New events**
- **ID3 support**

  You can now extract information from the audio and video stream nested in ID3 tags.

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:

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 most of the `AdPlaybackEventListener` callbacks. In most cases, implement callbacks for playback events, which include notifications that the player's state has changed. Changes in state can indicate errors and also affect which actions your player can take.

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
onPlayComplete	The end of a media source has been reached.
onPlayStart	Playback of a media source has started.
onPrepared	The media player has successfully prepared the media.
onSizeAvailable (long height, long width)	The size of the media is available.
onStateChanged (*MediaPlayer.PlayerState* state, *MediaPlayerNotification* notification)	The state of the media player has changed. Your application should handle errors in this callback.
onTimedMetadata (*TimedMetadata* timedMetadata)	A new timed metadata is discovered in the manifest.
onTimelineUpdated	The media player has an updated timeline. The timeline might contain new ads or some ads might have been removed.
onUpdated	The media player has successfully updated the media in either of these situations: • For each manifest refresh for a live asset. When this refresh happens, old ad breaks might be removed from the timeline and new ad opportunities (cue points) might be 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 (*MediaPlayer.getTimeline*). (VOD does not change during playback.) • When a VOD or live asset has closed captioning and activity is first discovered for a closed captioning track.

The following example shows the normal progression of these events:

```
mediaPlayer.addEventListener(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
onAdBreakComplete (AdBreak adBreak)	An ad break has played completely.
onAdBreakStart (AdBreak adBreak)	An ad break has started.
onAdClick (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 `notifyClick` on the `MediaPlayerView` .
onAdComplete (AdBreak adBreak, Ad ad)	An ad has played completely.
onAdProgress (AdBreak adBreak, Ad ad, int percentage)	Ad playback has progressed.
onAdStart (AdBreak adBreak, Ad ad)	An ad has started.

The following example shows the normal progression of these 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-complete event.

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

Event	Meaning
onBufferComplete	Buffering is complete.
onBufferStart	Buffering has started.
onLoadInfo(loadInfo)	A fragment has successfully downloaded.
onOperationFailed(MediaPlayerNotification.*Warning* warning)	A recoverable error has occurred.
onSeekComplete(long adjustedTime)	Seeking is complete.
onSeekStart	Seeking is starting.

The following example shows the normal progression of these events:

```
mediaPlayer.addEventListener(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
onDRMMetadata (DRMMetadataInfo drmMetadataInfo) DRMMetadataInfoEvent.*DRM_METADATA_INFO_AVAILABLE*	New DRM metadata is available.

The following example shows the normal progression of these events:

```
mediaPlayer.addEventListener(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
onLoadComplete(*mediaPlayerItem* playerItem)	Media resource loading completed successfully.
onError	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` `MediaPlayer.ReplaceCurrentItem`).

For example:

```
mediaPlayer.addEventListener(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 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.

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

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 the PREPARED state , attempting to call the following methods throws an IllegalStateException.

2. Retrieve the current playhead time using the `DefaultMediaPlayer.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).

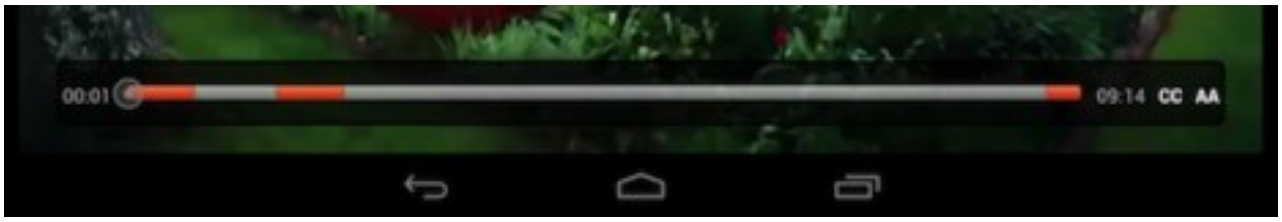
For a linear/live asset, the range represents the playback window range. This range changes during playback. The PSDK calls your `onUpdated` callback to indicate that the media item was refreshed and that its attributes (including the playback range) were updated.

4. 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>
```

5. 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:

   - If you seek into the middle of an ad break, the seek position is adjusted to the beginning of the ad break.
   - If you seek forward past one or more ad breaks, the playhead is positioned at the beginning of the last ad break previous to the specified seek position.

- 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 plus three times the target duration and ending at the client live point.

  The client live point is calculated by subtracting the buffered length and three times the target duration 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);
```

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` (Does Xbox have a 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.

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 or less) 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.
   The constructor is the only place where you can set these values.

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

5. Assign the ABR parameters 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
URL	Required. The location of the manifest/playlist for the content to be loaded.
Metadata	Required when using the `MediaResource` with the `createFromUrl` method. Otherwise, pass null. An instance of the `Metadata` 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.*

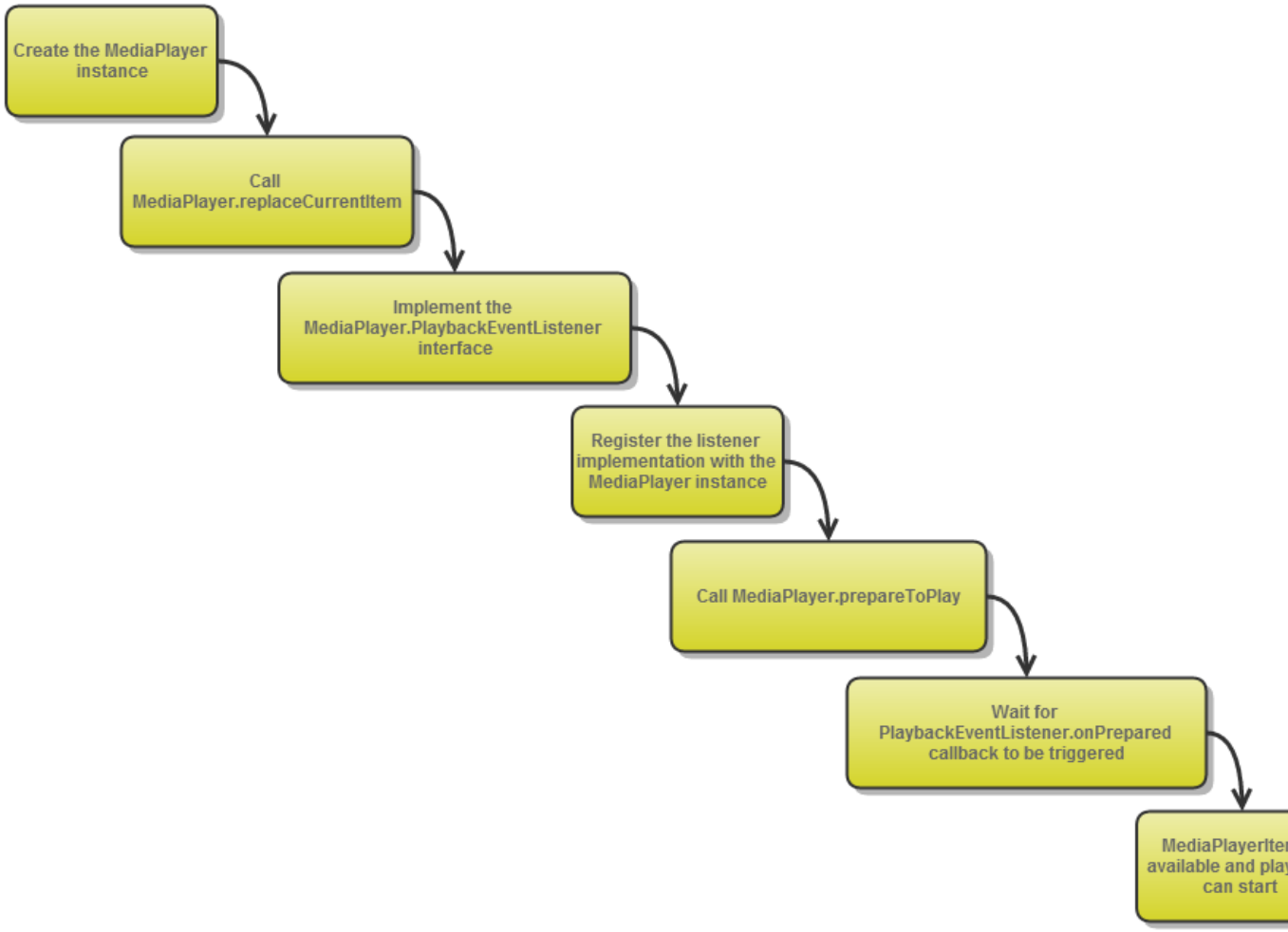
Method	Returns
getType	The type of the content to be loaded. Usually you can detect the type of the resource from the URL, but sometimes when custom naming schemes are involved, this is not possible. This is a simple enumeration, `MediaResource.Type`, which defines the types of content that can be loaded by the MediaPlayer. 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.
getUrl	The URL of the location of the manifest/playlist for the content to be loaded.
getMetadata	An instance of the `Metadata` 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.addEventListener(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.INITIALIAZED) {
    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

You can get information about each downloaded fragment.

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

   The PSDK provides the fragment URL, fragment size in bytes, and the duration of the download in ms.

> ***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 the frame rate, the profile bit rate, the total time spent in buffering, the number of buffering attempts, 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.

You can also get information about the device, such as 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 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. The PMPDemoApp demonstrates this. 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();

       setQosItem("Frame rate", (int) playbackInformation.getFrameRate());
       setQosItem("Dropped frames", (int)    playbackInformation.getDroppedFrameCount());
       setQosItem("Bitrate", (int) playbackInformation.getBitrate());
       setQosItem("Buffering time", (int) playbackInformation.getBufferingTime());
       setQosItem("Buffer length", (int) playbackInformation.getBufferLength());
       setQosItem("Buffer time", (int)    playbackInformation.   getBufferTime());
       setQosItem("Empty buffer count", (int) playbackInformation.getEmptyBufferCount());
       setQosItem("Time to load", (int) playbackInformation.getTimeToLoad());
       setQosItem("Time to start", (int) playbackInformation.getTimeToStart());
   }
});
}
};
```

4. Optionally read device-specific information.

```
// Show device information
DeviceInformation deviceInfo = new QOSProvider(getApplicationContext()).getDeviceInformation();

tv = (TextView) view.findViewById(R.id.aboutDeviceModel);
tv.setText(parent.getString(R.string.aboutDeviceModel) +
" " + deviceInfo.getManufacturer() + " - " + deviceInfo.getModel());
tv = (TextView) view.findViewById(R.id.aboutDeviceSoftware);
tv.setText(parent.getString(R.string.aboutDeviceSoftware) +
" " + deviceInfo.getOS() + ", SDK: " + deviceInfo.getSDK());
tv = (TextView) view.findViewById(R.id.aboutDeviceResolution); String orientation =
parent.getResources().getConfiguration().orientation ==
Configuration.ORIENTATION_LANDSCAPE ? "landscape" : "portrait";
tv.setText(parent.getString(R.string.aboutDeviceResolution) +
" " + deviceInfo.getWidthPixels() + "x" + deviceInfo.getHeightPixels() +
" (" + orientation + ")");
```

HTTP 302 redirect optimization

If a URL request is redirected, subsequent requests made to similar locations use the final URL, thereby avoiding 302 responses. This is enabled automatically.

Work with MediaPlayer objects

The `MediaPlayer` object represents your media player and a `MediaPlayerItem` represents audio or video on your player.

About the MediaPlayerItem class

Successfully loading a media resource creates an instance of the `MediaPlayerItem` class.

The `MediaResource` represents a request issued by the application layer to the `MediaPlayer` instance to load some content. The `MediaPlayer` starts by resolving the media resource and continues by loading the associated manifest file and parsing it (this is the asynchronous part of the resource loading process). At the end of the resource resolving process, the `MediaPlayerItem` instance is produced, so the `MediaPlayerItem` instance is basically the resolved version of a `MediaResource`.

The PSDK provides access to the newly created `MediaPlayerItem` instance through `MediaPlayer.getCurrentItem`. Wait for the resource to be successfully loaded before accessing the media player item.

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.

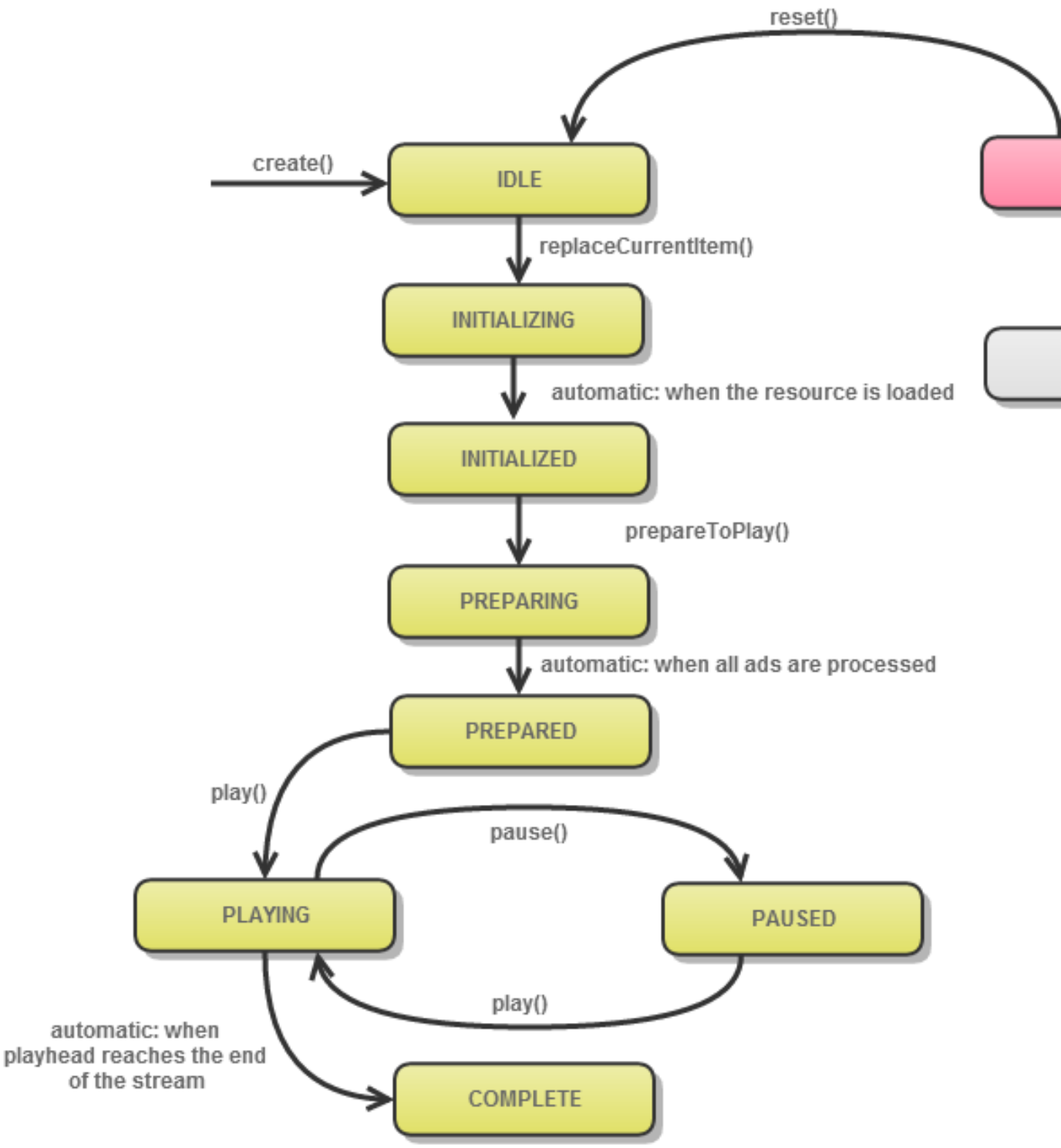
The ERROR state also changes what can happen next.

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 between PLAYING and PAUSED.

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	List<String> getAdTags()	Provides the list of ad tags that are used for the ad placement process.
Live stream	boolean isLive();	True if the stream is live; false if it is VOD.
DRM protected	boolean isProtected();	True if the stream is DRM protected.
	List<DRMMetadataInfo> getDRMMetadataInfos();	Lists all the DRM metadata objects discovered in the manifest.
Closed captions	boolean hasClosedCaptions();	True if closed-caption tracks are available.
	List<ClosedCaptionsTrack> getClosedCationsTracks();	Provides a list of available closed-caption tracks.
	ClosedCaptionsTrack get SelectedClosedCaptionsTrack();	Retrieves the closed caption track selected with SelectClosedCaptionsTrack.
Alternate audio tracks	boolean hasAlternateAudio();	True if the stream has alternate audio tracks. ***Note:*** *The main (default) audio track is also part of the alternate audio track list.* 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 `MediaPlayerItem.hasAlternateAudio` returns false is when the stream has no audio at all. If the content has only one audio track, this method returns true, and `MediaPlayerItem.getAudioTracks` returns a list with a single element (the default audio track).
	List<AudioTrack> getAudioTracks();	Provides a list of available alternate audio tracks.

Purpose	Method	Description
	AudioTrack getSelectedAudioTrack();	Retrieves the currently selected audio track selected with `SelectAudioTrack`.
Timed metadata	boolean hasTimedMetadata();	True if the stream has associated timed metadata.
	List<TimeMetadata> getTimedMetadata();	Provides a list of the timed metadata objects associated with the stream.
Multiple profiles (bit rates)	boolean isDynamic();	True if the stream is a multiple bit-rate (MBR) stream.
	List<Profile> getProfiles();	Provides a list of the associated profiles.
Trick play	boolean isTrickPlaySupported();	***Important:*** *The trick play feature is not currently supported. Because of this,* `isTrickPlaySupported` *always returns false.*
	List<Integer> getAvailableSpeeds();	***Important:*** *The trick play feature is not currently supported. Because of this,* `getAvailableSpeeds` *always returns an empty list.*
Media resource	MediaResource getResource();	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 consists of a single 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.

Requirements for advertising

Advertising and main content must meet certain requirements for ad insertion to work correctly.

- The advertising content HLS version 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.

About ad insertion

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

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 advertising workflow 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.

Playback behavior

The behavior of media playback is affected by seeking, pausing, and the inclusion of advertising.

For both live/linear streaming and VOD:

- When a user seeks forward past ad breaks, the seek position automatically adjusts to the beginning of the latest ad break before the ending seek position.
- When seeking backward, if the seek position is inside an ad break, the seek position adjusts to the beginning of that ad break.

Note: *The PSDK does not provide a way to disable seeking; your application must disable it if you want seeking disabled.*

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. The 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.

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=identiferForThisCue ...
```

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 includes the Ad Decisioning ad tracking library, which automatically tracks ads for live/linear ad tracking.

The PSDK tracks ad playback, reporting back to the Ad Decisioning server when an ad begins and when an ad reaches 25%, 50%, 75%, and 100% complete.

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.

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` 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.addEventListener(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	• Uses server map for placement detection • Ads are inserted	• Uses in-stream cues for placement detection • Pre-roll ads are inserted in the main stream • Mid-rolls ads replace main stream	• Uses server map for placement detection • Ads are inserted
Live/linear	• Uses manifest cues for placement detection • Ads replace main stream	• Uses in-stream cues for placement detection • Ads replace main stream	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 a `MediaPlayer.AdPlaybackEventListener`.
   This enables your application to receive PSDK events that provide information about the destination URL for a click on an ad.
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. 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 (creatives) might be available only as a progressive download MP4 video, in which case they cannot be stitched into the HLS content stream. Primetime Ad Insertion and the Android PSDK provide an option called third-party creative repackaging, which addresses this situation.

In some situations, you might need to incorporate ads 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.

Subscribe to custom tags

The PSDK allows you to specify custom tag names used in the ad placement process and to receive notification when certain tags appear in the manifest file.

Overview of custom tags

Here is an example VOD asset:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:7
#EXTINF:9.9766,
seg1.ts
#EXTINF:9.9766,
seg2.ts
#EXT-X-ASSET:AID=10
#EXTINF:9.9766,
seg1.ts
#EXT-X-AD:DURATION=10
#EXTINF:9.9766,
seg2.ts
#EXTINF:9.9766,
#EXT-X-CUE-OUT:
seg1.ts
#EXTINF:9.9766,
seg2.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.

PSDKConfig methods

The PSDKConfig class exposes two static setter methods and two static getter methods.

Those methods are:

- setAdTags sets the custom ad cues to be used in the ad placement process; getAdTags is its counterpart.
- setSubscribedTags sets the tag names for which to notify your application; getSubscribedTags is its counterpart.

```
/**
 * This class 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,
 * for which the application wants to be notified about.
 */
public class PSDKConfig {
 public static final String DEFAULT_AD_TAG = "#EXT-X-CUE";

 /**
 * Sets an array of ad tags. The player will use them when it will perform the ad placement.
 * These tags are used in addition to the default ad markers already supported. These
 * values will be taken into consideration only by the next media player(s) created.
 * @param tags The array of tags, containing the tag strings in the following format (eg
   #EXT-X-AD, #EXT-X-CUE-OUT)
 */
 public static void setAdTags(String[] tags);

 /**
 * Sets an array of tags, for which the application is interested to be notified about.
 * These values will be taken into consideration only by the next
 * media player(s) created.
 * @param tags The array of tags, containing the tag strings in the following format
 * (eg #EXT-X-AD, #EXT-X-CUE-OUT)
 */
 public static void setSubscribedTags(String[] tags);

 /**
```

```
 * @return the array of tags, that the player will use when it will perform
 * the ad placement.
 */
 public static String[] getAdTags();

 /**
 * @return the array of tags, for which the application is interested to be notified about.
 */
 public static String[] getSubscribedTags();
}
```

Note: *The setter methods do not allow the array parameter to contain null values. If encountered, the PSDK throws an IllegalArgumentException.*

Notifications for ads

The event MediaPlayer.PlaybackEventListener.onTimedMetadata notifies your application at the first occurrence in the manifest of each tag that matches a subscribed tag name.

Your application is automatically subscribed to all tags transmitted through setSubscribedTags and through setAdTags.

```
/**
 * Interface definition of a set of callback to be invoked during playback.
 */
 interface PlaybackEventListener extends EventListener {
 /**
  * Called when a new timed metadata is discovered in the manifest.
  * @param timedMetadata the object encapsulating the metadata information,
  * such as the relative time, type, tag name and id etc.
  */
 void onTimedMetadata(TimedMetadata timedMetadata);
 ...
}
```

TimedMetadata class

The TimedMetadata class encapsulates the tag/cue information that is found in the manifest.

```
public final class TimedMetadata {
 public enum Type {
  TAG, ID3
 }
 /**
  * Default constructor.
  *
  * @param time
  * local stream time where this
  * @param metadata
  */
 public TimedMetadata(Type type, long time, long id, String name, Metadata metadata);

 /**
  * The time position, relative to the start of the main content, where this
  * metadata was inserted into the stream.
  *
  * @return insertion time in milliseconds.
  */
 public long getTime();

 /**
  * The metadata inserted into the stream.
  *
  * @return parsed metadata
  */
 public Metadata getMetadata();
```

```
 /**
  * @return the type of the timed metadata.
  */
 public Type getType();

 /**
  * Returns the ID extracted from the cue attributes if present. Otherwise,
  * default unique value will be provided.
  *
  * @return
  */
 public long getId();

 /**
  * @return the name of the cue (usually the HLS tag name)
  */
 public String getName();
}
```

Configure and use custom ad tags

Before you can use custom ad tags, you need to subscribe to the tag names.

1. Set the custom ad tag names:

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

2. 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.addEventListener(MediaPlayer.Event.PLAYBACK, playbackEventListener);
   ```

3. Retrieve the list of timed metadata.

   The list of metadata is populated when the asset is first loaded. For live content, every time the manifest is refreshed, this list is updated, so that old entries are removed and new ones are added.

   ```
   private final MediaPlayer.PlaybackEventListener _playbackEventListener = new
   MediaPlayer.PlaybackEventListener() {
   @Override
   public void onPrepared() {
    // Called when media is prepared.
    List<TimedMetadata> timedMetadata = _mediaPlayer.getCurrentItem().getTimedMetadata();
    ...
   }
   @Override
   public void onUpdated() {
    // Called when manifest is refreshed, for a live asset.
    List<TimedMetadata> timedMetadata = _mediaPlayer.getCurrentItem().getTimedMetadata();
    ...
   }
   ...
   }
   ```

If you are also listening for ID3 metadata, they use the same onTimedMetadata listener to indicate the presence of an ID3 tag. This should not cause any confusion, however, because the custom ad cues are detected at the manifest level and the ID3 tags are embedded in the stream.

Use a custom ad provider and custom opportunity detector

The PSDK allows you to set a custom ad provider that will take care of issuing ad-resolving requests. It also allows you to define a custom opportunity detector for a live stream.

Each time a cue point is discovered in the manifest, you can choose:

- Whether this cue point is to be used for ad placement
- Which custom parameters found in the cue are to be sent to the ad provider

For example:

```
#EXT-X-CUE-OUT:CAID="0x120130605080613704361 38",DURATION="60"
#EXTINF:9.9766,
fileSequence38483.ts
#EXTINF:9.9766,
fileSequence38484.ts
#EXTINF:9.9766,
fileSequence38485.ts
#EXTINF:9.9766,
fileSequence38486.ts
#EXTINF:9.9766,
fileSequence38487.ts
#EXTINF:9.9766,
fileSequence38488.ts
```

For the cue point marked by the #EXT-X-CUE-OUT, your application could decide whether to transform this cue point into a placement opportunity that will be resolved by the ad provider. You could send the airing aid (CAID) and the duration value to the ad provider, as well.

With the PSDK, you can set an ad provider and a custom opportunity detector. You can also set an existing ad provider, such as the Adobe Ad Serving (Auditude) ad provider.

Note: *Currently SCTE-35 cues must be decoded by your application to be used for ad placement.*

Set an AdClientFactory on the MediaPlayer

To set an AdClientFactory on the MediaPlayer:

```
/**
 * Sets a custom AdClientFactory to be used in the ad placement process.
 * This factory can create a custom ad provider, to be used in the ad resolving.
 * It can also set a custom ad opportunity detector, that iterates through the timed metadata

 * and creates a list of PlacementOpportunity. This opportunities will be resolved by the
ad provider.
 *
 * @param adClientFactory
 * the custom ad client factory.
 */
void registerAdClientFactory(AdClientFactory adClientFactory);
```

This is the AdClientFactory:

```
public interface AdClientFactory {
 /**
  * Create an ad provider, which will issue the ad resolving requests.
```

```
  * @param item
  * the current media player item
  * @return
  * the ad provider created
  */
 public AdProvider createAdProvider(MediaPlayerItem item);

 /**
  * Create an opportunity detector.
  * Each time a cue point is discovered in the manifest, the detector
  * will decide whether it is going to be used in the ad placement process.
  * @param item
  * the current media player item
  * @return
  * the opportunity detector created
  */
 public PlacementOpportunityDetector createOpportunityDetector(MediaPlayerItem item);
}
```

This is the PlacementOpportunityDetector:

```
public interface PlacementOpportunityDetector {
 /**
  * Processes the existing timed metadata and returns a list of PlacementOpportunity
  * to be resolved by the ad provider.
  * If the TimedMetadata is refused the return value is null.

  * @param timedMetadataList the list of available timed metadata, which is sorted.
  * @param metadata the metadata which can contain the targeting params and the
  * custom parameters to be sent to the ad provider.
  */
 public List<PlacementOpportunity> process(List<TimedMetadata> timedMetadataList, Metadata
metadata);

 public PlacementOpportunity process(TimedMetadata timedMetadata, Metadata metadata);

 /**
  * When the current time has changed due to a seek or due to a live refresh,
  * the PlacementOpportunityDetector has to update its logic.
  *
  * @param currentTime new current time
  * @param playbackRange new playback range of the current item
  */
 public void updateByTime(long currentTime, TimeRange playbackRange);
}
```

Set the custom ad client factory

Add the following code:

```
MediaPlayer mediaPlayer = DefaultMediaPlayer.create(getActivity().getApplicationContext());
// Use custom ad client factory.
mediaPlayer.registerAdClientFactory(
 new AdClientFactory() {
  public AdProvider createAdProvider(MediaPlayerItem item) {
   // Use the Auditude ad provider.
   return new AuditudeAdProvider();
  }

  public PlacementOpportunityDetector createOpportunityDetector(MediaPlayerItem item) {
   // Use the custom placement opportunity.
   long initialTime = _mediaPlayer != null ? _mediaPlayer.getCurrentTime() : -1;
   return new CustomPlacementOpportunityDetector(item, initialTime);
  }
 }
);
```

Create a custom opportunity detector

The custom opportunity detector is a class that monitors the playback timeline and detects ad placement opportunities.

The packager scans the playlist files for special cues and detects if they contain any ad placement information. An example of a cue entry for HLS stream is:

```
#EXT-X-CUE:TYPE=SpliceOut,ID=7687, TIME=578123.41,DURATION=30.0,AVAIL-NUM=0,AVAILS-EXPECtED=0,
 PROGRAM-ID=0
```

1. Add the following code:

```
public class CustomPlacementOpportunityDetector implements PlacementOpportunityDetector {
 // ...

 @Override
 public List<PlacementOpportunity> process(List<TimedMetadata> timedMetadataList, Metadata
metadata) {
  PMPDemoApp.logger.i(LOG_TAG + "#process", "Processing ["
   + timedMetadataList.size()
   + "] timed metadata, in order to provide placement opportunities.");
  List<PlacementOpportunity> opportunities = new ArrayList<PlacementOpportunity>();

  for (TimedMetadata timedMetadata : timedMetadataList) {
   if (isPlacementOpportunity(timedMetadata)) {
    boolean shouldProcessTimedMetadata = timedMetadata.getTime() >
_timeOfLastDetectedOpportunity;
   if (shouldProcessTimedMetadata) {
    _timeOfLastDetectedOpportunity = timedMetadata.getTime();

   // The airingId (CAID) is in another tag. Iterate through the timedMetadata list and find

   // the CAID associated with this ad cue point.

   String airingId = getAiringIdForTime(timedMetadata.getTime(), timedMetadataList);
   PlacementOpportunity opportunity = createPlacementOpportunity(timedMetadata, airingId,
metadata);
```

2. Also add:

```
if (opportunity != null) {
  PMPDemoApp.logger.i(LOG_TAG + "#process", "Created new placement opportunity at time "
   + opportunity.getPlacementInformation().getTime()
   + ", with a duration of "
   + opportunity.getPlacementInformation().getDuration()
   + "ms.");
  opportunities.add(opportunity);
 } else {
  PMPDemoApp.logger.w(LOG_TAG + "#process", "Ad placement opportunity creation has failed.
Probably has invalid metadata."
   + " opportunity time = " + String.valueOf(timedMetadata.getTime())
   + ", metadata: " + timedMetadata.getMetadata()
   + ", processing time = " + String.valueOf(_timeOfLastDetectedOpportunity) +
   "].");
  }
 } else {
  PMPDemoApp.logger.w(LOG_TAG + "#process", "Ad placement opportunity
   skipped. It was either already processed or its position is in the past (previous to the
 play
   head).
   [" + " opportunity time= " + String.valueOf(timedMetadata.getTime()) +
   ", processing time=" + String.valueOf(_timeOfLastDetectedOpportunity) +
   "].");
   }
  }
 }
}
```

Iterate through the timedMetadataList and find the CAID

You need to iterate through the timedMetadataList to find and return the CAID (airing ID) associated with the provided local time.

1. Step through the timedMetadataList:

```
private String getAiringIdForTime(long time, List<TimedMetadata> timedMetadataList) {

 for (TimedMetadata timedMetadata : timedMetadataList) {
  if (timedMetadata.getTime() == time
   && timedMetadata.getMetadata() != null
   && timedMetadata.getMetadata().containsKey(OPPORTUNITY_ID_KEY)) {
  return timedMetadata.getMetadata().getValue(OPPORTUNITY_ID_KEY);
  }
 }
 return null;
}
```

2. Then, add the following code:

```
private PlacementOpportunity createPlacementOpportunity(TimedMetadata timedMetadata, String

 airingId, Metadata metadata) {
 long time = timedMetadata.getTime();
 long duration = 0;

 Metadata rawMetadata = timedMetadata.getMetadata();
 if (rawMetadata.containsKey(OPPORTUNITY_DURATION_KEY)) {
  duration = NumberUtils.parseNumber(rawMetadata.getValue(OPPORTUNITY_DURATION_KEY),0) *
1000;
 }

 if (duration <= 0) {
 return null;
 }

 // The custom params to be sent to the ad provider.
 MetadataNode customParams = new MetadataNode();
 customParams.setValue(PSDK_AVAIL_DURATION_KEY, String.valueOf(duration/1000));
 customParams.setValue(PSDK_ASSET_ID_KEY, String.valueOf(getAiringIdAsLong(airingId)));
 ((MetadataNode) metadata).setNode(DefaultMetadataKeys.CUSTOM_PARAMETERS.getValue(),
customParams);

 return new PlacementOpportunity(
  String.valueOf(timedMetadata.getId()),
  new PlacementInformation(Type.MID_ROLL, time, duration), metadata
 );
}
```

Check if a timed metadata represents an ad placement opportunity

You need to check if a timed metadata represents an ad placement opportunity.

- `@param timedMetadata` is the timed metadata to be checked
- `@return true` if the timed metadata is an ad placement
- `@return false` if the timed metadata is not an ad placement

Add the following code:

```
private boolean isPlacementOpportunity(TimedMetadata timedMetadata) {
 Metadata metadata = timedMetadata.getMetadata();
 return timedMetadata.getName().equals(OPPORTUNITY_TAG_NAME)
  && metadata != null
```

```
  && metadata.containsKey(OPPORTUNITY_DURATION_KEY);
 }
}
```

Add custom ad markers

With custom ad markers, you can mark specific sections of the main content as ad-related content periods.

This feature is most useful when content is being recorded (possibly from a live event) and the result of the recording is a single HLS stream. The recording itself contains both main content and advertising-related content in a single HLS VOD stream. The recording process does not keep track of the ad-related segments and the information related to the positioning of the ads inside the main content is lost.

You might be able to obtain the information related to the positioning of the ad-content periods from other out-of-band sources (such as external CMS systems).The PSDK allows you to define custom markers, through which this out-of-band information can be passed to the timeline manager subsystem. The intention is to mark the content sections that match the specified ad-related content in such a way that all ad-specific playback events are triggered in the same manner as if these custom ad-periods were explicitly placed on the player's timeline.

Ad tracking is not handled internally by the PSDK, such as when ads are resolved by Ad Decisioning (Auditude). However, the PSDK defines two abstractions that define the way ad-related content is represented on the timeline: the ad break and the individual ad. An ad break is an ordered list of individual consecutive ads. The ad-related playback events are triggered separately for ad breaks and ads: there are callbacks for ad-break start/complete events and separate ad start/complete events.

All ad tracking events are dispatched to your application, so you can implement your own tracking logic. If you set the custom ad markers, you will receive the onAdBreakStart, onAdStart, onAdProgress, onAdComplete, and onAdBreakComplete events.

TimeRange class

The main purpose of custom ad markers is to enable you to pass on to the PSDK a set of TimeRange specifications that represent timeline segments.

Each TimeRange specification in the set represents a segment on the playback timeline (maintained internally by the PSDK) that must be marked appropriately as an ad-related period.

The TimeRange class is a simple data structure that exposes two read-only properties, the start position and the end position on the timeline, which abstracts the idea of a time range inside the playback timeline. Both values are expressed in milliseconds. Here is a summary of the TimeRange class:

```
public final class TimeRange {
    // the start/end values are provided at construction time
    public static TimeRange createRange(long begin, long duration) {...}

    // only getters are available
    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 three pieces of 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. Currently, 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.
- 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 toMedatada(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-rage 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.toMedatada(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 in 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 the actual literal string and not the true Boolean 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 the TimeRangeCollection#toMetada method. 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));
```

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" />
```

2. 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()]);
}
```

3. 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	The font type. Can be set to values such as `monospaced_with_serifs`, `cursive`, and others. The value `default` refers to whatever font the caption originally specified. Valid values are defined by the `TextFormat.Font` enumeration. Note that the specific fonts that are actually available on any particular device may vary. Substitutions are used when necessary. The `MONOSPACED_WITH_SERIFS` is typically the font that is used as a substitute; however, this, too, can be system specific.
Size	The caption's size. Valid values are defined by the `TextFormat.Size` enumeration. Meanings are: • `MEDIUM` - The standard size • `LARGE` - Approximately 30% larger than medium • `SMALL` - Approximately 30% smaller than medium • `DEFAULT` - The default size for the caption; the same as medium
Font edge	The effect used for the font edge, such as raised or none. Valid values are defined by the `TextFormat.FontEdge` enumeration.
Font color	The font's color. Can be set to any one of a wide variety of values, such as `BLACK`, `RED`, or `DARK_CYAN`. The value `DEFAULT` refers to whatever color the caption originally specified. Valid values are defined by the `TextFormat.Color` enumeration.
Edge color	The color of the edge effect. Can be set to any of the values defined for the font color.

Format	Description
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.

Integrate with Video Analytics

The PSDK allows you to track content and ads, as well as monitor important milestones and key events.

You can monitor the following milestones and key event data:

• Playback started
• Quartile checkpoints (25%, 50%, and 75% of the total duration)
• Playback stopped

You can use notifications to collect this information. You can then use the information in video analysis of your choice.

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.

The PSDK supports AAXS integration as custom DRM workflows. This means that your application must implement the DRM authentication workflows before playing the stream using the Flash DRMManager. To enable this, the MediaPlayer provides you with the DRM manager for authentication.

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 DRMAuthenticationListener authenticationListener);
```

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

Relevant SDK elements:

- *com.adobe.ave.drm.DRMManager*
- *com.adobe.ave.drm.DRMMetdata*
- *com.adobe.ave.drm.DRMPolicy*
- *com.adobe.ave.drm.DRMAuthenticationMethod*
- *com.adobe.ave.drm.DRMAuthenticationCompleteCallback*
- *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 DRMManager 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 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.
*/
public static void performDrmAuthentication(
     final DRMManager 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 uninterruptedly.

```
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));
           finish();
           }
       });
   }

   @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 notificatons.

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. • Error notification events, from 100000 to 199999 • Warning notification events, from 200000 to 299999 • Information notification events, from 300000 to 399999
name	A string containing a human-readable description of the notification event, such as PLAYBACK_START.
metadata	Metadata containing relevant information about the notification stored as key/value pairs. For example, a key named `URL` would provide a URL related to the notification, such as an invalid URL that caused an error.
innerNotification	A reference to another `MediaPlayerNotification` 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. Implement a MediaPlayer.PlaybackEventListener#onTimedMetadata(TimeMetadata timeMetadata) listener 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.

Primetime cannot protect from all failures-for example, an ISP failure or a cable disconnection. However, Primetime 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 PDSK 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.

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*

Name	Description
ABRControlParameters	Class that encapsulates all adaptive bit rate control parameters.
AdvertisingFactory	Factory class that allows customization of the AdDecisioning process.
BufferControlParameters	Class that encapsulates all buffer control parameters.
DefaultMediaPlayer	Default class implementation of MediaPlayer interface.
MediaPlayer	Public interface for the DefaultMediaPlayer class. Includes enumerations for Event, PlayerState, and Visibility.
MediaPlayer.AdPlaybackEventListener	Interface definition of a set of callbacks to be invoked during ad playback.
MediaPlayer.DRMEventListener	Interface definition of a callback to be invoked when protected metadata becomes available.
MediaPlayer.EventListener	Marker interface used to unify event listener registration.
MediaPlayer.PlaybackEventListener	Interface definition of a set of callback to be invoked during playback.
MediaPlayer.QOSEventListener	Interface definition of a set of callback to be invoked during QoS.
MediaPlayerItem	Interface that represents audio-video media.
MediaPlayerItemLoader	Class that loads a media player resource and creates the corresponding MediaPlayerItem object.
MediaPlayerItemLoader.LoaderListener	Interface that defines the listener methods associated with the MediaPlayerItemLoader object.
MediaPlayerView	Class for the view that will be used by the MediaPlayer for video rendering.
MediaResource	Class that wraps all information about a media resource. Includes enumeration for Type of media resource.
PlacementOpportunityDetector	Interface used for processing in manifestcues that will be used as placement for the ad decisioning process.

Name	Description
TextFormat	Interface that encapsulates different attributes describing a text style (for example, the closed captions style).
TextFormatBuilder	Class methods for setting the formatting of text.
Version	Class that provides the PSDK version and description.

Info classes

These classes provide information about the media.

Package: *com.adobe.mediacore.info*

Name	Description
AudioTrack	Class that extends Track to define the audio track abstraction.
ClosedCaptionsTrack	Class that extends Track to define the closed caption abstraction.
Profile	Class that contains media profile information, such as the height, width, and bit rate.
Track	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
Log	Class. Provides access to the log system.
LogEntry	Class. Defines an entry log and holds information about a log message.
LogFactory	Interface that enables custom logging.
Logger	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
AuditudeMetadata	Deprecated. Use AuditudeSettings.
AuditudeSettings	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,

Name	Description
	media ID, and ad server URL, to configure the player for successfully resolving ads.
Metadata	Defines the generic interface for configuring all available metadata for your player. Extends MetadataNode. Individual components extend this class to provide additional accessors for key-value metadata associated with PSDK objects.
MetadataNode	Generic data-structure-like class for storing arbitrary key-value pairs.
TimedMetadata	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. *Error*	Class that describes the notification for an error that causes the player to stop playback. This is a *MediaPlayerNotification* of notification type ERROR.
MediaPlayerNotification. *Info*	Class that describes an informational notification. This is a *MediaPlayerNotification* of notification type INFO.
MediaPlayerNotification. *Warning*	Class that describes a warning notification. This is a *MediaPlayerNotification* of notification type WARNING.
MediaPlayerNotification	Class that provides informational messages, warnings, and errors. Encapsulates the object representation of a single notification event within *NotificationHistory*.
MediaPlayerNotification. *NotificationCode*	Returns the description associated with the provided notification code.
NotificationHistory	Class that stores a log of notification objects. A circular list of *NotificationHistory.Item* 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 *MediaPlayerNotification* class. (In *session* package.)
NotificationHistory. *Item*	Class that defines an entry in the circular list in *NotificationHistory* and holds the notification and its timestamp.
MediaPlayerNotification. *EntryType*	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. *BufferingMetrics*	Provides information about how much time the player spent while buffering and how often a buffering event occurred.
DeviceInformation	Provides information about the platform and operating system on which the PSDK runs: • Version of the platform OS • Version number of the PSDK library • Device's model name • Device manufacturer's name • Device UUID • Width/height of the device screen
LoadInfo	Offers information about an individual fragment that is being downloaded. This includes the fragment URL, its size in bytes, and total download duration in ms.
PlaybackInformation	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, 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. *PlaybackLoadMetrics*	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. *PlaybackMetrics*	Provides information on how the playback is behaving. This includes the frame rate, bit rate, buffer length, and so on.
metrics. *PlaybackSessionMetrics*	Provides information on how many seconds the player spent while actually playing and how much time the video was actually on screen.
QOSProvider	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
PlacementOpportunity	An opportunity class represents an "interesting" point on the timeline.
Timeline	Interface that provides an iterator for processing timeline markers. Represents the timeline of the content, including ad breaks.
TimelineItem	Class. Generic immutable representation of a timeline item.
TimelineMarker	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).
TimelineOperation	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
Ad	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.
AdAsset	Class that represents an asset to be displayed.
AdBreak	Class that gives a unified view on several ads that will be played at some point during playback.
AdBreakPlacement	Ad break placement operation class.
AdBreakPolicy	Enumeration that defines values for whether an ad break is to remain on the timeline or be removed.
AdBreakPolicySelector	Interface that instructs the media player how to proceed after the ad break has been viewed by the user.
AdClick	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.
auditude.AuditudeAdProvider	Deprecated. Use AuditudeResolver.
auditude. *AuditudeResolver* *AuditudeAdResolver*	
auditude. *AuditudeTracker*	Class.
ContentResolver	Class that handles the ad-resolving part in the Ad Decisioning process.
ContentTracker	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.
PlacementInformation	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: • Placement position (in ms) • Type of the placement (pre-roll, mid-roll, or post-roll) • Duration of the main content chunk that is about to be replaced

Utility classes

These classes provide ways to process various types of information.

Package: *com.adobe.mediacore.utils*

Class name	Description
DateUtils	Methods for processing dates.
NumberUtils	Helper methods related to numbers.
StringUtils	Helper methods related to strings.
TimeRange	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
Playback				
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.

Code	Name	InnerNotification	Metadata Keys	Comments
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.
101200	DOWNLOAD_ERROR	None	URL	An error has occurred while attempting to download data.
Invalid resource				
102100	RESOURCE_LOAD_ERROR	None	DESCRIPTION RESOURCE	An error has occurred while loading a resource item.
102101	RESOURCE_PLACEMENT_FAILED	None	CONTENT_ID	An error has occurred while placing a resource on the playback timeline.
Ad processing				
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.
Native				
106000	NATIVE_ERROR	None	NATIVE_ERROR_CODE NATIVE_ERROR_NAME DESCRIPTION	The low-level AVE library issued an error.
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	ENGINE_RESOURCES_RELEASE_ERROR	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.
Configuration				
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.
Alternate audio				
109000	AUDIO_TRACK_ERROR	DOWNLOAD_ERROR	AUDIO_TRACK_NAME AUDIO_TRACK_LANGUAGE	An error related to an audio track occurred.
Generic				

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
Playback				
200000	PLAYBACK_OPERATION_FAIL	AUDIO_TRACK_ERROR SEEK_ERROR	DESCRIPTION	A playback-related operation has failed, but playback may continue.
Ad-resolving				
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.
Trick mode				
280000	TRICKPLAY_RATE_CHANGE_FAIL	None	DESCRIPTION	Rate change failed.
Native				
209100	NATIVE_WARNING	None	NATIVE_ERROR_CODE NATIVE_ERROR_NAME DESCRIPTION	The low-level AVE library has issued an warning event. Playback may continue.
Generic				
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	Inner Notification	Metadata Keys	Comments
Playback				
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.
Adaptive bit rates (ABR)				
302000	BITRATE_CHANGE	None	BITRATE CURRENT_MEDIA_TIME	Notifies that the bitrate of the video has changed
Ad Processing				
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

Code	Name	Inner Notification	Metadata Keys	Comments
				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. (?)
Late-binding audio (LBA)				
304000	AUDIO_TRACK_CHANGE	None	TRACK_ID CURRENT_MEDIA_TIME	Notifies that the audio track has been changed.
DRM				
305000	DRM_METADATA_AVAILABLE	None	PREFETCH_TIMESTAMP	Notifies that new DRM data available.
Generic				
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.

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.

AVE

Adobe Video Engine.

C

cue point

An #EXT tag in an M3U8 file that the PSDK uses for splicing ads into a live stream.

D

DRM

Digital rights management.

F

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.

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

Live video is 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

Multipe bit rate.

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.

Q

QoS

Quality of service.

R

rendition

One representation of the media presentation.

RTA

Real-time analytics.

S

SBR

Single bit rate.

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

Adobe Primetime Player SDK for Android Programmer's Guide, Version 1.2

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