Adobe® Primetime

TVSDK 3.0 for Android Programmer's Guide

Contents

TVSDK 3.0.0 for Android Programmer's Guide....4

Product overview, audience, and this guide....4

Considerations and best practices....4

Primetime TVSDK features....5

Requirements....6

System and software requirements....6

Content and manifest requirements....6

#EXT-X-VERSION requirements....7

Content playback options....7

Set up the MediaPlayer....7

Work with MediaPlayer objects....8

Listen for Primetime Player events....15

Set up error handling....18

Configure the player user interface....18

Initialize the media player for a specific video....24

Implement fast forward and rewind....27

HTTP 302 redirect optimization....30

Work with cookies....30

Work with closed captions....31

Alternate audio....38

Enabling Background Audio....41

ID3 tags....42

Buffering....43

Parallel downloads....45

Adaptive bit rates (ABR) for video quality....46

Quality of service statistics....48

Playback and failover....49

Advertising....52

Advertising requirements....52

Inserting ads....53

Default and customized playback behavior with ads....55
Customize playback with ads....56
Just-in-Time Ad Resolving....64
Ad insertion metadata....66
Companion banner ads....69
Clickable ads....71
Repackage incompatible ads using Adobe Creative Repackaging Service (CRS)....74
Ad fallback for VAST and VMAP ads....74
Custom tags....75
VPAID 2.0 ad support....81
Ad measurements from Moat....82
Add custom ad markers....83
Customize opportunity generators and content resolvers....86
Delete and replace ads in VOD streams....91
Updating ad creative selection rules....101
Content protection....106
Widevine DRM....106
Primetime DRM interface overview....106
DRM authentication before playback....107
DRM authentication during playback....110
Video analytics....111
Initialize and configure video analytics....112
Implement custom metadata support....114
Implement chapter support....115
Set up video analytics reporting on the server side....116
Access video analytics reports....116
Events and notifications....117
Notifications and events for player status, activity, errors, and logging....117
Notification codes....118
Primetime player events summary....145
Billing metrics....150
Configure billing metrics....150
Transmit billing metrics....151

TVSDK 3.0.0 for Android Programmer's Guide

Product overview, audience, and this guide

This guide provides information about how to develop video player applications by using TVSDK for Android, which is implemented in Java.

- For a list of the features that are supported by TVSDK, see *Primetime TVSDK features*.
- For specific hardware and software requirements for using TVSDK, see *Requirements*.
- For a list of available APIs, see *TVSDK Android APIs*.

Product overview

TVSDK includes API descriptions and code samples to help you to integrate advanced video functionality, content protection, and advertising features into your player. You use Java to create a video player user interface. TVSDK helps you connect that user interface to its media player. This allows you to play videos and advertising based on media manifests. You can also use TVSDK to retrieve information about the video, handle security, and control and monitor playback.

Audience

This guide assumes that you understand how to develop applications and video players using Java. You implement your video player UI in Java, and incorporate the TVSDK features you require.

About this guide

This guide provides information that allows you to incorporate TVSDK features in a video player using Java on Android devices.

Namespace notation in this guide

Tip: *The TVSDK API namespace prefix* `com.adobe.mediacore` *is often omitted for brevity.*

Many API elements are referred to without their parent class designator if the context is clear.

Considerations and best practices

To use TVSDK most effectively, you should consider certain details of its operation and follow certain best practices.

Considerations

Remember the following information when using TVSDK:

- The TVSDK API is implemented in Java.
- Adobe Primetime does not currently work on Android emulators.

  You must use real devices for testing.
- Playback is supported only for HTTP Live Streaming (HLS) content.
- Main video content can be multiplexed (video and audio streams in the same rendition) or nonmultiplexed (video and audio streams in separate renditions).

- Currently, you need to run most TVSDK API operations on the UI thread, which is the main Android thread.

  Operations that run correctly on the main thread may throw an error and exit when run on a background thread.
- 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 the AVE.
  - Depending on encoder precision, the actual encoded media duration might differ from the durations that are recorded in the stream resource manifest.

    There is no reliable way to resynchronize between the ideal virtual timeline and the actual playout timeline. Progress tracking of the stream playback for ad management and Video Analytics must use the actual playout time, so reporting and user interface behavior might not precisely track the media and advertisement content.
  - The incoming user agent name for all media requests from the TVSDK on this platform is assigned the following string pattern:

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

    All ad-related calls use the Android default user agent or the custom user agent if you set it while setting up ad-insertion metadata.

Best practices

Here are recommended practices for TVSDK:

- Use HLS version 3.0 or above for program content.
- Run most TVSDK operations on the main (UI) thread, not on background threads.
- For TVSDK 3.0 for Android, lazy ad resolving is on by default.

  For content with no pre-roll or mid-roll, you can use `AdvertisingMetadata.setPreroll(false)` to accelerate content loading.

Primetime TVSDK features

TVSDK for Android 3.0 includes a variety of features that you can implement in your players.

TVSDK capabilities:

- **VOD and live/linear playback**
  - Management of the playback window, including methods that play, stop, pause, seek, and retrieve the playhead position
  - Support for full-event replay
  - Closed captioning (608, 708, WebVTT) and alternate forms of audio for increased accessibility
  - Controls for text styling in captions
  - DVR capability, fast forward, and fast rewind (the latter two are known as *trick-play mode*)
  - Adaptive bit rate (ABR) logic and initial set up of ABR controls
  - Live manifest failover support
  - Adjustable playback buffers
  - Fragment duration, size, and time-to-download tracking support
- **Advertising**

- VPAID 2.0
- Client-side ad stitching
  - Seamless ad insertion, including support for VAST/VMAP
  - Support for custom cue tags for ads
  - Support for marking, replacing, and deleting C3 ads
  - Customizable content/ad insertion workflow including blackout signaling

- **Content protection**
  - Access to digital rights management (DRM)-related services
  - Playback of HLS streams unencrypted or with Protected HTTP Live Streaming (PHLS)
  - Resolution-based output control, based on DRM policy
- **Video and ad tracking**
  - QoS event tracking
  - Notifications that help TVSDK and your application to communicate asynchronously about the status of videos, advertisements, and other elements. Notifications also log activity.
- **Logging**
  - Debug logging
  - Tracking support for fragment duration, size, and time-to-download

Requirements

TVSDK has specific requirements for media content, manifest content, DRM, and software versions.

System and software requirements

To use TVSDK, ensure that your hardware, operating system, and application versions all meet the minimum requirements listed below.

Operating system	Android 4.0 or later (minimum API level 14)
CPU	1 GHz Single Core or equivalent
RAM	256 MB
GPU	Hardware GPU required for playback
Architecture	x86 via Houdini, ARM64, ARMv7, and ARMv8

Content and manifest requirements

Check the restrictions and requirements for streams and playlists (manifests), including DRM encryption keys.

Adobe Access DRM	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	Must have the same bit-rate renditions as the renditions of the main content.

#EXT-X-VERSION requirements

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

Here is some information about the `#EXT-X-VERSION` tag, which specifies the HLS protocol version:

- The version must match the features and attributes in the HLS playlist; otherwise, playback errors might occur. For more information, see *HTTP Live Streaming specification*.
- Adobe recommends using at least Version 2 of HLS for playback in TVSDK-based clients.

Clients and servers must implement the versions in the following way:

Use at least this version	To use these features
`EXT-X-VERSION:2`	The IV attribute of the `EXT-X-KEY` tag.
`EXT-X-VERSION:3`	• Floating-point `EXTINF` duration values The duration tags (`#EXTINF:`<duration>,<title>) in version 2 were rounded to integer values. Version 3 and above require durations to be specified exactly, in floating point.
`EXT-X-VERSION:4`	• The `EXT-X-BYTERANGE` tag • The `EXT-X-I-FRAME-STREAM-INF` tag • The `EXT-X-I-FRAMES-ONLY` tag • The `EXT-X-MEDIA` tag • The `AUDIO` and `VIDEO` attributes of the `EXT-X-STREAM-INF` tag • TVSDK alternate audio

Content playback options

TVSDK provides tools for creating an advanced video player application (*your Primetime player*), that you can integrate with other Primetime components. It also provides a number of features designed to maximize the quality of video playback.

Set up the MediaPlayer

Instantiate a `MediaPlayer` and place a view of it into a frame layout.

1. Instantiate `MediaPlayer`, passing an `android.content.Context` object to the constructor:
   ```
   MediaPlayer mediaPlayer = new MediaPlayer(context);
   ```
2. Provide a frame layout (`android.widget.FrameLayout`) to hold a `ViewGroup` of `mediaPlayer`:
   ```
   FrameLayout playerFrame = (FrameLayout) _viewGroup.findViewById(R.id.playerFrame);
   ```
   Below is the code snippet to create `_viewGroup`.
   ```
   @Override
    public ViewGroup onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState) {
     _viewGroup = (ViewGroup) inflater.inflate(
   ```

```
    R.layout.fragment_player, container, false);
 return _viewGroup;
}
```

3. Place a view of `mediaPlayer` inside the frame layout:

```
playerFrame.addView(mediaPlayer.getView());
```

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

Work with MediaPlayer objects

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

About the MediaPlayerItem class

After you successfully load the `MediaResource` object, TVSDK creates an instance of the `MediaPlayerItem` class to provide access to that resource.

The `MediaResource` represents a request that is issued by the application layer to the `MediaPlayer` instance to load content.

The `MediaPlayer` resolves the media resource, loads the associated manifest file, and parses the manifest. This is the asynchronous part of the resource loading process. The `MediaPlayerItem` instance is produced after the resource has been resolved, and this instance is a resolved version of a `MediaResource`. TVSDK provides access to the newly created `MediaPlayerItem` instance through `MediaPlayer.CurrentItem`.

Tip: *You must wait for the resource to be successfully loaded before accessing the media player item.*

Lifecycle and statuses of the MediaPlayer object

The status of the media player determines which actions are legal.

For working with media player statuses:

• You can retrieve the current status of the `MediaPlayer` object with `MediaPlayer.getStatus()`.

• The list of statuses is defined in the *MediaPlayerStatus* enum.

Status-transition diagram for the lifecycle of a `MediaPlayer` instance:

The following table provides details about the lifecycle and statuses of the media player:

Status	Occurs when
IDLE	The media player's initial status. The player is created and is waiting for you to specify a media player item.
INITIALIZING	Your application calls `MediaPlayer.replaceCurrentItem()`. The media player item is loading.
INITIALIZED	TVSDK successfully set the media-player item.
PREPARING	Your application calls `MediaPlayer.prepareToPlay()`. The media player is loading the media player item and any associated resources.
PREPARED	TVSDK has prepared the media stream and has attempted to perform ad resolving and ad insertion (if enabled). The content is prepared and ads have been inserted in the timeline, or the the ad procedure failed. Buffering or playback can begin.
PLAYING/PAUSED	As the application plays and pauses the media, the media player moves between these statuses.
SUSPENDED	If the application navigates away from the playback, shuts down the device, or switches applications while the player is playing or paused, the media player is suspended and resources are released. Calling `MediaPlayer.restore()` returns the player to the status the player was in before it was SUSPENDED. The exception is if the player is SEEKING when suspended is called, the player is PAUSED and then SUSPENDED. ***Important:*** *Remember the following information:* • *The `MediaPlayer` automatically calls `suspend` only when the surface object that is used by the `MediaPlayerView` is destroyed.* • *The `MediaPlayer` automatically calls `restore()` only when a new surface object that is used by the `MediaPlayerView` is created.* If you always want playback to be paused when the MediaPlayer is restored, have your application call `MediaPlayer.pause()` in the Android Activity's `onPause()` method.
COMPLETE	The player has reached the end of the stream, and playback has stopped.

Status	Occurs when
RELEASED	Your application released the media player, which also releases any associated resources. You can no longer use this instance.
ERROR	An error occurred during the process. An error also might affect what the application can do next. For more information, see .

Tip: *You can use the status to provide feedback on the process, or example, a spinner while waiting for the next status change, or take the next steps in playing the media, such as waiting for the appropriate status before calling the next method.*

For example:

```
mediaPlayer.addEventListener(MediaPlayerEvent STATUS_CHANGED, new StatusChangeEventListener()
 {
    @Override
    public void onStatusChanged(MediaPlayerStatusChangeEvent event) {
        switch(event.getStatus()) {
            case INITIALIZED:
                mediaPlayer.prepareToPlay();
                break;
            case PREPARING:
                showBufferingSpinner();
                break;
            case PREPARED:
                hideBufferingSpinner();
                mediaPlayer.play();
                break;
            ...
        }
        ...
    }
});
```

MediaPlayerItem 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`.

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.

Method	Description
`List<ClosedCaptionsTrack> getClosedCationsTracks();`	Provides a list of available closed-caption tracks.
`ClosedCaptionsTrack get SelectedClosedCaptionsTrack();`	Retrieves the current closed caption track selected with `SelectClosedCaptionsTrack`.
`selectClosedCaptionsTrack ( ClosedCaptionsTrack closedCaptionsTrack)`	Sets a closed-caption track to be the current closed-caption track.
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.* TVSDK 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.
`AudioTrack getSelectedAudioTrack();`	Retrieves the audio track selected with `selectAudioTrack`.
`selectAudioTrack ( AudioTrack audioTrack )`	Selects an audio track to be the current audio track.
Timed metadata	
`boolean hasTimedMetadata();`	True if the stream has associated timed metadata.
`List<TimedMetadata> 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 bit rate profiles. For each profile, you can retrieve its bit rate and the height and width of the profile.
`Profile getSelectedProfile()`	Retrieves the currently selected profile.
Trick play	
`boolean isTrickPlaySupported();`	True if the player supports fast forward, rewind, and resume.
`List< Float> getAvailablePlaybackRates()`	Provides the list of available playback rates in the context of the trick-play feature.
`Float getSelectedPlaybackRate()`	Retrieves the currently selected playback rate.
`MediaPlayerItemConfig getConfig()`	Returns the `MediaPlayerItemConfig` instance associated with this item.

Method	Description
Media resource	
`MediaResource getResource();`	Returns the media resource associated with this item.
`int getResourceId()`	Returns the media identifier associated with this item. This ID is set when the item is loaded using `MediaPlayerItemLoader.load`.

Reuse or remove a MediaPlayer instance

You can reset, reuse, or release a `MediaPlayer` instance that you no longer need.

Reset or reuse a MediaPlayer instance

When you reset a `MediaPlayer` instance, it is returned to its uninitialized IDLE status as defined in `MediaPlayerStatus`.

This operation is useful in the following cases:

- You want to reuse a `MediaPlayer` instance but need to load a new `MediaResource` (video content) and replace the previous instance.

  Resetting allows you to reuse the `MediaPlayer` instance without the overhead of releasing resources, recreating the `MediaPlayer`, and reallocating resources.
- When the `MediaPlayer` is in ERROR status and needs to be cleared.

Important: *This is the only way to recover from the ERROR status.*

1. Call `reset` to return the `MediaPlayer` instance to its uninitialized status:
   ```
   void reset() throws MediaPlayerException;
   ```
2. Use `MediaPlayer.replaceCurrentResource()` to load another `MediaResource`.

   Tip: *To clear an error, load the same `MediaResource`.*
3. When you receive the `STATUS_CHANGED` event callback with `PREPARED` status, start the playback.

Release a MediaPlayer instance and resources

You should release a `MediaPlayer` instance and resources when you no longer need the `MediaResource`.

When you release a `MediaPlayer` object, the underlying hardware resources that are associated with this `MediaPlayer` object are deallocated.

Here are some reasons to release a `MediaPlayer`:

- Holding unnecessary resources can affect performance.
- Leaving an unnecessary `MediaPlayer` object instantiated can lead to continuous battery consumption for mobile devices.
- If multiple instances of the same video-codec are not supported on a device, playback failure might occur for other applications.

Release the `MediaPlayer`.

```
void release() throws MediaPlayerException;
```

After the `MediaPlayer` instance is released, you can no longer use it. If any method of the `MediaPlayer` interface is called after it is released, a `MediaPlayerException` is thrown.

Inspect the playback timeline

You can obtain a description of the timeline associated with the currently selected item being played by TVSDK. 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.

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

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

   A `TimelineMarker` object contains two pieces of information:

   • Position of the marker on the timeline (in milliseconds)
   • Duration of the marker on the timeline (in milliseconds)

3. Listen for the `MediaPlayerEvent.TIMELINE_UPDATED` event on the `MediaPlayer` instance, and implement the `TimelineUpdatedEventListener.onTimelineUpdated()` callback.

   The `Timeline` object can inform your application about changes that might 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();
}
```

Suspend and Restore MediaPlayer

Suspending and restoring the TVSDK `MediaPlayer` when a device screen is turned off and on must be handled by your application.

You can handle suspend and restore operations on `MediaPlayer` inside Android's broadcast receiver for screen on/off.

TVSDK can not determine when a Fragment (or Activity) is in the background or foreground. In addition, the Android `SurfaceView` does not get destroyed when the device screen is turned off (but the Activity is paused). However, `SurfaceView` *does* get destroyed when the device puts your application in the background. TVSDK cannot detect any of these changes, so they must be handled by your application.

The following sample code how your application can handle suspending and restoring the `MediaPlayer` when the device screen is turned on and off at the application level:

```
// Track the state of a fragment to determine if it is PAUSED or RESUMED
private boolean isActivityPaused = false;

/**
* Register the broadcast receiver to track screen on/screen off functions triggered from
device.
*/
private BroadcastReceiver mReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {

        // Call suspend when screen is turned off and mediaPlayer is not null and
        // mediaplayer status is not suspended/Error/Released state.
        if (intent.getAction().equals(Intent.ACTION_SCREEN_OFF)) {
            try {
                if (mediaPlayer != null &&
                  lastKnownStatus != MediaPlayerStatus.ERROR &&
                  lastKnownStatus != MediaPlayerStatus.RELEASED &&
                  lastKnownStatus != MediaPlayerStatus.SUSPENDED) {
                    PrimetimeReference.logger.i(LOG_TAG+"#screenOff:",
                    "Suspending mediaplayer as screen is turned off and mediaPlayer
                    status is " + lastKnownStatus.toString());
                    mediaPlayer.suspend();
                }
                else {
                    PrimetimeReference.logger.i(LOG_TAG+"#screenOff:",
                      "Not suspending mediaplayer since mediaplayer status is "
                      + lastKnownStatus.toString());
                }
            } catch(MediaPlayerException e) {
                PrimetimeReference.logger.e(LOG_TAG+"#screenOff:",
                  "MediaPlayer Exception for suspend() call");
            }
        }
        else if (intent.getAction().equals(Intent.ACTION_SCREEN_ON)) {
            // Call restore when the screen is turned on and mediaplayer is not in the
            // suspended status.  This is for the screen on condition when the device
            // does not have a lock and turning on the screen immediately brings the
            // fragment to the foreground.
            try {
                if(lastKnownStatus == MediaPlayerStatus.SUSPENDED && !isActivityPaused) {
                    PrimetimeReference.logger.i(LOG_TAG+"#screenOn:",
                      "Restoring mediaplayer since screen is turned on and mediaPlayer status
 is "
                      + lastKnownStatus.toString());
```

```
                    mediaPlayer.restore();
                }
                else {
                    PrimetimeReference.logger.i(LOG_TAG+"#screenOn:",
                      "Not restoring mediaplayer since mediaPlayer status is "
                      + lastKnownStatus.toString());
                }
            } catch(MediaPlayerException e) {
                PrimetimeReference.logger.e(LOG_TAG+"#screenOn:",
                  "MediaPlayer Exception for restore() call");
            }
        }
    }
};

/*
* Activity or Fragment's onPause() overridden method
*/
@Override
public void onPause() {
    PrimetimeReference.logger.i(LOG_TAG + "#onPause", "Player activity paused.");

    // Set the fragment paused status to true when app goes in background.
    isActivityPaused = true;
    super.onPause();
}

/*
* Activity or Fragment's onResume() overridden method
*/
@Override
public void onResume() {
    super.onResume();

    /**
    * When the device has a lock/pin the on resume will be called only after the device
      is unlocked.
    * Screen on does not call the onResume() method so we need to handle restore here
      explicitly.
    */
    if(lastKnownStatus == MediaPlayerStatus.SUSPENDED && isActivityPaused) {
        try {
            PrimetimeReference.logger.i(LOG_TAG + "#onResume",
              "Player restored as activity operations are resumed");
            mediaPlayer.restore();
        }
        catch(MediaPlayerException e) {
            PrimetimeReference.logger.i(LOG_TAG + "#onResume",
              "Exception occured while restoring mediaPlayer");
        }
    }
    // Set the fragment paused status to false when app comes in foreground.
    isActivityPaused = false;
}
```

Listen for Primetime Player events

Events from TVSDK indicate the status of the player, errors that occur, the completion of actions that you have requested, such as a video starting to play, or actions that occur implicitly, such as an ad completing.

Because your application needs to respond to many of these events, you need to implement event-handling routines and register these routines with TVSDK. The routines call the relevant TVSDK methods to respond appropriately.

More information about events:

• The real-time nature of video playback requires asynchronous (non-blocking) activity for many TVSDK operations.

- TVSDK supports an event-driven video player.

  It provides events that correspond to all important steps in the playing process. You register those events with your platform's event mechanism and create event handlers that will be called when those events occur. *Event Handlers* are also known as callback routines or event listeners. TVSDK provides a complete range of methods that can be used by the event handlers.

- Your application generally initiates non-blocking operations, such as requesting that a video start playing.

  TVSDK communicates asynchronously with your application by dispatching events, such as when the video starts playing and an event when the video finishes. Other events can indicate status changes in your player and error conditions. Your event handlers take appropriate actions.

Implement event listeners and callbacks

Event handlers enable you to respond to TVSDK events.

When an event occurs, TVSDK's event mechanism calls your registered event handler, passing it the event information.

TVSDK defines listeners as public internal interfaces inside the `MediaPlayer` interface.

Your application must implement event listeners for any TVSDK events that affect your application.

1. Determine which events your application must listen for.
   - Required events: Listen for all playback events.

     Important: *Listen for the status change event, which occurs when the player's status changes in ways that you need to know. The information it provides includes errors that might affect what your player can do next.*

   - For other events, depending on your application, see *Primetime player events summary*.
2. Implement and add an event listener for each event.

   For most events TVSDK passes arguments to the event listeners. Such values provide information about the event that can help you decide what to do next.

   The `MediaPlayerEvent` enumeration lists all the events that `MediaPlayer` dispatches. For more information, see *Primetime player events summary*.

   For example, if `mPlayer` is an instance of `MediaPlayer`, here is how you might add and structure an event listener:

```
mPlayer.addEventListener(MediaPlayerEvent.STATUS_CHANGED, new StatusChangeEventListener() {

    @Override
    public void onStatusChanged(MediaPlayerStatusChangeEvent event) {
        event.getMetadata();
        if (event.getMetadata() != null) {/* Do something */}
        if (event.getStatus() == MediaPlayerStatus.IDLE) {/* Do something */}
        else if (event.getStatus() == MediaPlayerStatus.INITIALIZED) {/* Do something */}
        else if (event.getStatus() == MediaPlayerStatus.PREPARED) {/* Do something */}
    }
});
```

Order of playback events

TVSDK dispatches events/notifications in generally expected sequences. Your player can implement actions based on events in the expected sequence.

The following examples show the order of some events that occur during playback.

When successfully loading a media resource through `MediaPlayer.replaceCurrentResource`, the order of events is:

1. `MediaPlayerEvent.STATUS_CHANGED` with status `MediaPlayerStatus.INITIALIZING`
2. `MediaPlayerEvent.STATUS_CHANGED` with status `MediaPlayerStatus.INITIALIZED`

Tip: *Load your media resource on the main thread. If you load a media resource on a background thread, this operation or subsequent operations might throw an error, such as* `MediaPlayerException`*, and exit.*

When preparing for playback through `MediaPlayer.prepareToPlay`, the order of events is:

1. `MediaPlayerEvent.STATUS_CHANGED` with status `MediaPlayerStatus.PREPARING`
2. `MediaPlayerEvent.TIMELINE_UPDATED` if ads were inserted.
3. `MediaPlayerEvent.STATUS_CHANGED` with status `MediaPlayerStatus.PREPARED`

For live/linear streams, during playback as the playback window advances and additional opportunities are resolved, the order of events is:

1. `MediaPlayerEvent.ITEM_UPDATED`
2. `MediaPlayerEvent.TIMELINE_UPDATED` if ads were inserted

Order of advertising events

When your playback includes advertising, TVSDK dispatches events/notifications in generally expected sequences. Your player can implement actions based on events within the expected sequence.

When playing ads, the order of events is:

- `MediaPlayerEvent.AD_RESOLUTION_COMPLETE`

The following events are dispatched for every ad within the ad break:

- `MediaPlayerEvent.AD_BREAK_START`
- `MediaPlayerEvent.AD_START`
- `MediaPlayerEvent.AD_PROGRESS (multiple times)`
- `MediaPlayerEvent.AD_CLICK (for each click)`
- `MediaPlayerEvent.AD_COMPLETE`
- `MediaPlayerEvent.AD_BREAK_COMPLETE`

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

```
mediaPlayer.addEventListener(MediaPlayerEvent.AD_RESOLUTION_COMPLETE, new
AdResolutionCompleteEventListener() {
        @Override
        public void onAdResolutionComplete() { ... }
    });

mediaPlayer.addEventListener(MediaPlayerEvent.AD_BREAK_START, new AdBreakStartedEventListener()
 {
         @Override
        public void onAdBreakStarted(AdBreakPlaybackEvent adBreakPlaybackEvent) { ... }
    });

mediaPlayer.addEventListener(MediaPlayerEvent.AD_START, new AdStartedEventListener() {
         @Override
        public void onAdStarted(AdPlaybackEvent adPlaybackEvent) { ... }
    });

mediaPlayer.addEventListener(MediaPlayerEvent.AD_PROGRESS, new AdProgressEventListener() {
         @Override
```

```
        public void onAdProgress(AdPlaybackEvent adPlaybackEvent) { ... }
    });

mediaPlayer.addEventListener(MediaPlayerEvent.AD_COMPLETE, new AdCompletedEventListener() {
        @Override
        public void onAdCompleted(AdPlaybackEvent adPlaybackEvent) { ... }
    });

mediaPlayer.addEventListener(MediaPlayerEvent.AD_BREAK_COMPLETE, new
AdBreakCompletedEventListener() {
        @Override
        public void onAdBreakCompleted(AdBreakPlaybackEvent adBreakPlaybackEvent) { ... }
    });

Below event is for tracking ad clicks.

mediaPlayer.addEventListener(MediaPlayerEvent.AD_CLICK, new AdClickedEventListener() {
        @Override
        public void onAdClicked(AdClickEvent adClickEvent) { ... }
    });
```

Order of DRM events

TVSDK dispatches digital rights management (DRM) events in response to DRM-related operations such as when new DRM metadata becomes available. Your player can implement actions in response to these events.

To be notified about all DRM-related events, listen for `MediaPlayerEvent.DRM_METADATA`. TVSDK dispatches additional DRM events through the `DRMManager` class.

Order of loader events

TVSDK dispatches `MediaPlayerEvent.LOAD_INFORMATION_AVAILABLE` when loader events occur.

Set up error handling

You can set up one lace to handle errors.

1. Implement an event callback function for `MediaPlayerEvent.STATUS_CHANGED`.
   TVSDK passes event information, such as a `MediaPlayerStatusChangeEvent` object.
2. In the callback, when the returned status is `MediaPlayerStatus.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 status it remains in that status until you reset it using the `MediaPlayer.reset` method.

For example:

```
mediaPlayer.addEventListener(MediaPlayerEvent.STATUS_CHANGED,
  new StatusChangeEventListener() {
    @Override
    public void onStatusChanged(MediaPlayerStatusChangeEvent event) {
        if (event.getStatus() == MediaPlayerStatus.ERROR) {
        // handle TVSDK error here
        }
    }
});
```

Configure the player user interface

With TVSDK, you can control the basic playback experience for live and video on demand (VOD). TVSDK provides methods and properties on the player instance that you can use to configure the player user interface.

Wait for a valid status

Before you can use most of the TVSDK player methods, the player must be in a valid status.

Waiting for the player to be in the correct status ensures that the media resource has successfully loaded. If the player is not in at least the required status, many player methods throw `MediaPlayerException`.

The required status is usually PREPARED. When this occurs, the callback routine for `StatusChangeEventListener.onStatusChanged()` executes.

To confirm that the status is `PREPARED`, check `MediaPlayer.MediaPlayerStatus`.

Play and pause a video

You can add pause and play buttons to pause or play your video.

1. To create a pause or play button:
   a) Wait for the player to be in at least the prepared state.
   b) To start playback, call the `play` method:
   ```
   void play() throws MediaPlayerException;
   ```
   c) To pause playback, call the `pause()` method:
   ```
   void pause() throws MediaPlayerException;
   ```
2. Use the status changed event callback to check for errors or to take other appropriate actions.

   TVSDK calls this callback for `pause()` or `play()` and passes information about the status change, including the new status, such as paused or playing.

Identify whether the content is live or VOD

You might need to know whether the media content is live or video on demand (VOD).

1. Ensure that the player is in at least the `PREPARED` state.
2. Determine whether the`MediaPlayerItem` content is live (`true`) or VOD (`false`).
   ```
   boolean isLive();
   ```

Provide volume control

You can set up a user interface control to adjust the volume for the video.

1. In the callback routine for the volume control interface element, ensure that the player is in a valid status for this command.

   Tip: *Any status, except for RELEASED is valid.*

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

   For example:
   ```
   void setVolume(int volume) throws MediaPlayerException;
   ```

   The value for the volume represents the requested volume expressed as a proportion of the maximum volume, where `0` is silent and `1` is the maximum volume.

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

You can use TVSDK to retrieve information about the player's position in the media and display it on the seek bar.

1. Wait for the player to be in at least the PREPARED state.
2. Retrieve the current playhead time by using the `MediaPlayer.getCurrentTime` method.

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

   ```
   long getCurrentTime() throws MediaPlayerException;
   ```

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 MediaPlayerException;
      ```

   b) Use the `MediaPlayer.getPlaybackRange` method to get the virtual timeline time range.

      - For VOD, the range always begins with zero and the end value equals the sum of the main content duration and the durations of additional content in the stream (ads).
      - For a linear/live asset, the range represents the playback window range. This range changes during playback.

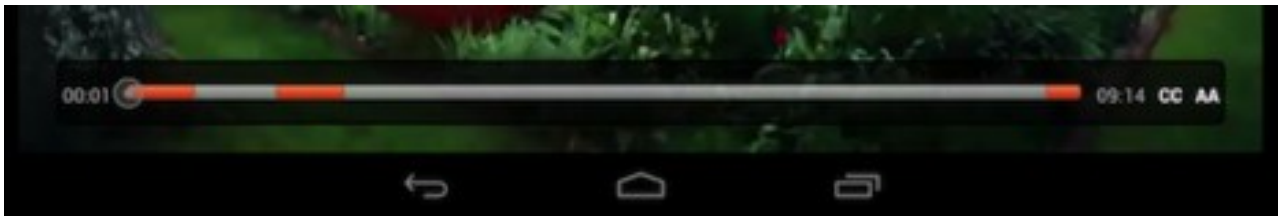
      TVSDK calls the `ITEM_Updated` callback to indicate that the media item was refreshed and that its attributes, including the playback range, were updated.

4. Use the methods that are available on `MediaPlayer` and on the `SeekBar` class in the Android SDK to set up the seek-bar parameters.

   For example, here is a possible layout that contains the seek bar 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 seek bar, as shown in the figure:



The following example uses the `Clock.java` helper class, which is available in `ReferencePlayer`, as the timer. 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 seek bar here.
    }
};
playbackClock.addClockEventListener(playbackClockEventListener);
```

On every clock tick, this example retrieves the media player's current position and updates the seek bar. 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() == MediaPlayerStatus.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 position

TVSDK supports seeking to a specific position (time) where the stream is a sliding-window playlist, in video on demand (VOD) and live streams.

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

1. Set up callbacks for seeking.

   Seeking is asynchronous, so TVSDK dispatches the following seek-related events:

   - `MediaPlayerEvent.SEEK_BEGIN`, where the seek starts.
   - `MediaPlayerEvent.SEEK_END`, where the seek is successful.
   - `MediaPlayerEvent.OPERATION_FAILED`, where the seek has failed.

2. Wait for the player to be in a valid status for seeking.
   The valid statuses are PREPARED, COMPLETE, PAUSED, and PLAYING.
3. Use the native `SeekBar` to set `OnSeekBarChangeListener`, which determines when the user is scrubbing.

4. Pass the requested seek position (milliseconds) to the `MediaPlayer.seek` method.

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

   You can seek only in the asset's seekable duration. For video on demand, that is from 0 through the asset's duration.

   Tip: *This step moves the play head to a new position in the stream, but the final computed position might differ from the specified seek position.*

5. Listen for `MediaPlayerEvent.OPERATION_FAILED` and take appropriate actions.

   This event passes the appropriate warning. Your application determines how to proceed, and the options include trying the seek again or continuing playback from the previous position.

6. Wait for TVSDK to call the `MediaPlayerEvent.SEEK_END` callback.
7. Retrieve the final adjusted play position using the callback's position parameter.

   This is important because the actual start position after the seek can be different from the requested position. Rules, including playback behavior is affected if a seek or other repositioning ends in the middle of an ad break or skips ad breaks, might apply.

8. Use the position information when displaying a seek scrub bar.

Seeking Example

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

```
//Use the native SeekBar to set an OnSeekBarChangeListener to
// see when the user is scrubbing.
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);
    }
};
```

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.

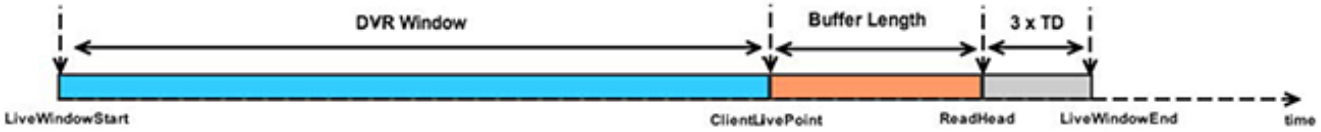
- For VOD, the length of the seekable window is the duration of the entire asset.
- For live streaming, the length of the DVR (seekable) window is defined as the time range that starts at the live playback window and ends at the client live point.

  Remember the following information:

  - The client live point is calculated by subtracting the buffered length from the live window end.

    The target duration is a value bigger than or equal to the maximum duration of a fragment in the manifest.

  - The default value is 10000 ms.
  - The control bar for live playback supports DVR by first positioning the thumb at the client live point when starting playback and by displaying a region that marks the area where seek is not allowed.



  To implement a control bar with DVR support, follow the steps in *Display a seek scrub bar with the current playback position* with the following differences:

  - You can 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 implement a control bar that is mapped for the playback range but that also displays the seekable range.

  For a control bar:

  1. Add an overlay to 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 `MediaPlayer.getSeekableRange`.

  For example:

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

  You can also choose to seek to the client live point using the `MediaPlayer.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`). You can override the default behavior.

Pass a position to `MediaPlayer.prepareToPlay`.

TVSDK considers the given position to be the starting point for the asset, and no seek operation is required. If the position is not inside the seekable range, TVSDK uses the default position. For more information, see *Load a media resource in the media player*.

For example:

```
long desiredPostion = // TODO : choose a value;
@Override
public void onStatusChanged(MediaPlayerStatusChangedEvent statusChangedEvent) {
    switch (statusChangedEvent.getStatus()) {
        case INITIALIZED:
            _mediaPlayer.prepareToPlay(desiredPosition);
            break;
        case PREPARING:
            showBufferingSpinner();
            break;
    }
}
```

Initialize the media player for a specific video

For each new video content, initialize a `MediaResource` instance with information about the video content and load the media resource.

Create a media resource

The `MediaResource` class represents the content to be loaded by the `MediaPlayer` instance.

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

   The `MediaResource` constructor requires the following parameters:

Constructor Parameter	Description
`url`	A string representing the URL of the manifest/playlist of the media.
`type`	One of the following members of the `MediaResource.Type` enum, corresponding to the indicated file type: • `HLS` - M3U8 • `ISOBMFF` - ISO base media file format (MP4) • `DASH` - MPEG-DASH media presentation description (MPD)
`metadata`	An instance of the `Metadata` class (a dictionary-like structure), which might contain additional information about the content that is about to be loaded, such as alternate or ad content to place inside the main content. If using advertising, set up `AuditudeSettings` before using this constructor (see *Ad insertion metadata*).

Important: *TVSDK supports playback for only specific types of content. If you attempt to load any other type of content, TVSDK dispatches an error event.*

For MP4 video-on-demand (VOD) content, TVSDK does not support trick play, adaptive bit rate (ABR) streaming, ad insertion, closed captions, or DRM.

The following code creates a `MediaResource` instance:

```
// To do: Create metadata here
MediaResource res = new MediaResource(
  "http://www.example.com/video/some-video.m3u8",
  MediaResource.Type.HLS,
  metadata);
```

At any time after this step, you can use `MediaResource` accessors (getters) to examine the resource's type, URL, and metadata.

2. Load the media resource using one of the following options:
   - The MediaPlayer instance.
   - `MediaPlayerItemLoader`

   For more information, see *Load a media resource using MediaPlayerItemLoader*.

Important: *Do not load the media resource on a background thread. Most TVSDK operations need to run on the main thread, and running them on a background thread can cause the operation to throw an error and exit.*

Load a media resource in the media player

Load a resource by directly instantiating a `MediaResource` and loading the video content to be played. This is one way of loading a media resource.

1. Set the media player to play the new resource.

   Replace the currently playable item by calling `MediaPlayer.replaceCurrentResource()` and passing an existing `MediaResource` instance.

   This starts the resource loading process.

2. Register the `MediaPlayerEvent.STATUS_CHANGED` event with the `MediaPlayer` instance. In the callback, check for at least the following status values:
   - `MediaPlayerStatus.PREPARED`
   - `MediaPlayerStatus.INITIALIZED`
   - `MediaPlayerStatus.ERROR`

   Through these events, the `MediaPlayer` object notifies your application when it has successfully loaded the media resource.

3. When the status of the media player changes to `INITIALIZED`, you can call `MediaPlayer.prepareToPlay()`.

   This status indicates that the media has been successfully loaded. The new `MediaPlayerItem` is ready for playback. Calling `prepareToPlay()` starts the advertising resolution and placement process, if any.

If a failure occurs, the media player switches to the `ERROR` status.

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(MediaPlayerEvent.STATUS_CHANGED,
  new StatusChangeEventListener() {
    @Override
    public void onStatusChanged(MediaPlayerStatus status) {
        if(event.getStatus() == MediaPlayerStatus.PREPARED) {
            // The resource is successfully loaded and available. The
            // MediaPlayer is ready to start the playback and can
            // provide a reference to the current playable item
            MediaPlayerItem playerItem = mediaPlayer.getCurrentItem();
            if (playerItem != null) {
                // We can look at the properties of the loaded stream
            }
        }
        else if (event.getStatus() == MediaPlayerStatus.ERROR) {
```

```
            //Something bad happened - the resource cannot be loaded.
            // The Metadata object in the event provides details.
        }
        else if (status == MediaPlayerStatus.INITIALIZED) {
            mediaPlayer.prepareToPlay();
        }
    }
}
```

Load a media resource using MediaPlayerItemLoader

Using `MediaPlayerItemLoader` helps you obtain information about a media stream without instantiating a `MediaPlayer` instance. This is especially useful in pre-buffering streams so that playback can begin without delay.

The `MediaPlayerItemLoader` class helps you exchange a media resource for the current `MediaPlayerItem` without attaching a view to a `MediaPlayer` instance, which would allocate video decoding hardware resources. Additional steps are necessary for DRM-protected content, but this manual does not describe them.

Important: *TVSDK does not support a single `QoSProvider` to work with both `itemLoader` and `MediaPlayer`. If your application uses Instant On, the application needs to maintain two `QoS` instances and manage both instances for the information. See Instant On for more information.*

1. Create an instance of `MediaPlayerItemLoader`.

```
private MediaPlayerItemLoader createLoader() {
    MediaPlayerItemLoader itemLoader =
      new MediaPlayerItemLoader(this, new MediaPlayerItemLoader.LoaderListener() {
        public void onError(PSDKErrorCode mediaErrorCode, String description) {
            //Do something
        }

        public void onLoadComplete(MediaPlayerItem playerItem) {
            loader.prepareBuffer();
        }

        public void onBufferingBegin() {
            //Do something
        }

        public void onBufferPrepared() {
            mPlayer.reset();
        }
    });

    itemLoader.setKeepRebufferingForLive(true);
    return itemLoader;
}
```

Tip: *Create a separate instance of `MediaPlayerItemLoader` for each resource. Do not use one `MediaPlayerItemLoader` instance to load multiple resources.*

2. Implement the `ItemLoaderListener` class to receive notifications from the `MediaPlayerItemLoader` instance.

```
private MediaPlayerItemLoader createLoader() {
    MediaPlayerItemLoader itemLoader =
      new MediaPlayerItemLoader(this, new MediaPlayerItemLoader.LoaderListener() {
        public void onError(PSDKErrorCode mediaErrorCode, String description) {
            //Do something
        }
        public void onLoadComplete(MediaPlayerItem playerItem) {
            loader.prepareBuffer();
        }
        public void onBufferingBegin() {
            //Do something
```

```
        }
        public void onBufferPrepared() {
            mPlayer.reset();
        }
    } );

    itemLoader.setKeepRebufferingForLive(true);
    return itemLoader;
}
```

In the `onLoadComplete()` callback, do one of the following:

- Ensure that anything that might affect buffering, for example, selecting WebVTT or audio tracks, is complete and call `prepareBuffer()` to take advantage of instant on.
- Attach the item to the `MediaPlayer` instance by using `replaceCurrentItem()`.

If you call `prepareBuffer()`, you receive the BUFFER_PREPARED event in your `onBufferPrepared` handler when the preparation is finished.

3. Call `load` on the `MediaPlayerItemLoader` instance and pass the resource to be loaded, and optionally the content ID, and a `MediaPlayerItemConfig` instance.

```
loader = createLoader();
MediaResource res = new MediaResource(mVideoUrl, MediaResource.Type.HLS, metadata);
loader.load(res, 233, getConfig());
```

4. To buffer from a point other than the beginning of the stream, call `prepareBuffer()` with the position (in milliseconds) at which to start buffering.
5. Use the `replaceCurrentItem()` and `play()` methods of `MediaPlayer` to start playing from that point.
6. Wait for idle status and call `replaceCurrentItem`.
7. Play the item.
   - If the item is loaded but not buffered:
     1. Wait for initialized status.
     2. Call `prepareToPlay()`.
     3. Wait for the PREPARED status.
     4. Call `play()`.
   - If the item is buffered:
     1. Wait for the buffer prepared event.
     2. Call `play()`.

Implement fast forward and rewind

When users fast forward or fast rewind through the media, they are in the *trick play* mode. To enter trick play mode, set the `MediaPlayer` playback rate to a value other than 1.

To switch the speed, you must set one value.

1. Move from normal play mode (1x) to trick play mode by setting the rate on the `MediaPlayer` to an allowed value.

   Remember the following information:

   - The `MediaPlayerItem` class defines the allowed playback rates.
   - TVSDK selects the closest allowed rate if the specified rate is not allowed.

The following example sets the player's internal playback rate to the requested rate:

```
import com.adobe.mediacore.MediaPlayer;
import com.adobe.mediacore.MediaPlayerItem;
import com.adobe.mediacore.MediaPlayerException;
import java.util.List;
import java.lang.Float;

private boolean setPlaybackRate(MediaPlayer player, float rate)
  throws MediaPlayerException {
    // Get list of playback rates that the media player supports
    MediaPlayerItem item = player.getCurrentItem();
    if (item == null) return false;
    List<Float> availableRates = player.getCurrentItem().getAvailablePlaybackRates();

    // Return false if requested rate is not supported
    if (availableRates.indexOf(rate) == -1) return false;

    // Otherwise set the playback rate to the requested rate
    // (this can throw MediaPlayerException)
    player.setRate(rate);
    return true;
}
```

2. You can optionally listen for rate-change events, which notifies you when you requested a rate change and when the rate change actually occurs.

   TVSDK dispatches the following events that are related to trick play:

   • `MediaPlayerEvent.RATE_SELECTED`, when the `rate` value changes to a different value.

   • `MediaPlayerEvent.RATE_PLAYING`, when playback resumes at the selected rate.

   TVSDK dispatches these events when the player returns from trick play mode to normal play mode.

Rate-change API elements

TVSDK includes methods, properties, and events to determine valid rates, current rates, whether trick play is supported, and other functionality that are related to fast forward and rewind.

Use the following API elements to change play rates:

• `PlaybackRateEvent.getRate`

• `MediaPlayerEvent.RATE_SELECTED`

• `MediaPlayerEvent.RATE_PLAYING`

• `MediaPlayerItem.isTrickPlaySupported`

• `MediaPlayerItem.getAvailablePlaybackRates`, which specifies valid rates.

Rate value	Effect on playback
2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0	Switches to fast-forward mode with the specified multiplier faster than normal (for example, 4 is 4 times faster than normal)
-2.0, -4.0, -8.0, -16.0, –32.0, -64.0 , –128.0	Switches to fast-rewind mode
1.0	Switches to normal play mode (calling `play` is the same as setting the rate property to 1.0)
0.0	Pauses (calling `pause` is the same as setting the rate property to 0.0)

Limitations and behavior for trick play

Limitations for trick play mode:

- The master playlist must contain Iframe-only segments.

  Only the key frames from the Iframe track are displayed on the screen.
- The audio track and closed captions are disabled.
- Play and pause are enabled.
- You can exit trick-play mode into any allowed playback rate (play or pause).
- When ads are incorporated in the stream:
  - You can go to trick play only while playing the main content. An error is dispatched if you try to switch to trick play during an ad break.
  - After starting trick play mode, ad breaks are ignored and no ad events are fired.
  - The timeline exposed by TVSDK to the player is not modified even if ad breaks are skipped.
  - The current time value jumps forward (on fast forward) or backward (on fast rewind) with the duration of the skipped ad break.

    This jump behavior for the current time allows the stream duration to remain unmodified during trick play. Your player can track the time relative only to the main content. No time jumps are performed on the values returned for the local time when skipping an ad.
  - The `MediaPlayerEvent.AD_BREAK_SKIPPED` event is dispatched immediately before an ad break is about to be skipped.

    Your player can use this event to implement custom logic related to the skipped ad breaks.
  - Exiting trick play invokes the same ad playback policy as when exiting seek.

    As with seeking, the behavior depends on whether your application's playback policy is different from the default. The default is that the last skipped ad break is played at the point where you come out of trick play.

Smoother trick play operations

If your system has access to hardware-assisted decoding, you can achieve smoother trick play than with the pure software TVSDK implementation by using iFrame format.

Using iFrame format results in trick play operations that are not smooth. Smoother trick play operation uses a normal (not iFrame) profile, hardware decoding support, and an increased frame rate. Different hardware-assisted decoding devices have different capabilities. Double speed requires 60 frames per second (FPS), and quadruple speed requires 120 FPS.

Important: *Adobe recommends that you limit playback to double speed for newer Android devices and not use the feature for older Android devices.*

To achieve smoother trick play, set `ABRControlParameters.maxPlayoutRate` to the desired multiple of normal speed (for example, 2.0 for double speed). If a subsequent call to `MediaPlayer.setRate()` has an argument that is less than or equal to the value you set for `maxPlayoutRate`, TVSDK uses a normal profile to achieve smoother trick play. Otherwise it uses an iFrame profile for the trickplay operation.

HTTP 302 redirect optimization

302 redirect optimization minimizes the number of 302 redirect responses, which allows your application to load balance more effectively.

If a main manifest request is redirected, and 302 optimization is enabled in your player, subsequent requests made for assets from that manifest will use the final domain location, which avoids additional 302 responses. This feature is enabled by default, and you can change this setting.

Disable or enable 302 redirect optimization

Use the `useRedirectedUrl` property to turn 302 redirect on (`true`) or off (`false`).

For example:

```
// Set useRedirectedUrl property to false
NetworkConfiguration networkConfiguration = new NetworkConfiguration();
networkConfiguration.setUseRedirectedUrl(false);

//Set NetworkConfiguration on MediaPlayerItemConfig
MediaPlayerItemConfig config = new MediaPlayerItemConfig ();
config.setNetworkConfiguration(networkConfiguration);

//Use this config when loading the MediaPlayerItem or calling replaceCurrentResource
```

Work with cookies

You can use TVSDK to send arbitrary data in cookie headers for session management, gate access, and so on.

Here is a sample request to the key server with some authentication:

1. Your customer logs in to your website in a browser and their login shows that this customer is allowed to view content.
2. Based on what is expected by the license server, your application generates an authentication token.

   This value is passed to TVSDK.
3. TVSDK sets this value in the cookie header.
4. When TVSDK makes a request to the key server to get a key to decrypt the content, the request contains the authentication value in the cookie header.

   The key server knows that the request is valid.

To work with cookies:

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

For example:

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

Tip: *When 302 redirect is enabled, the ad request may be redirected to a domain that is different from the domain to which the cookie belongs.*

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

Get string value for cookie when cookies are updated

The event MediaPlayerEvent.COOKIES_UPDATED is called when C++ cookies are updated. This cookiesUpdatedEvent has a method getCookieString() that returns a string value for the cookie.

A sample code snippet is below:

```
private final CookiesUpdatedEventListener cookiesUpdatedEventListener = new
CookiesUpdatedEventListener()
{
@Override
public void onCookiesUpdated(CookiesUpdatedEvent cookiesUpdatedEvent)
 {
 String cookieStr = cookiesUpdatedEvent.getCookieString();
 logger.i(LOG_TAG + "::MediaPlayer.CookiesUpdatedEventListener#onCookiesUpdated()", "cookieStr
 " + cookieStr);
 }
};
```

Work with closed captions

Closed captioning displays the audio portion of a video as text on the screen when the sound is inaudible or the viewer is hard of hearing.

Closed captions are typically in the same language as the audio and also display background sounds as text, but subtitles are typically in a different language and do not include background sounds.

TVSDK supports rendering these formats:

- 608 and 708 closed captioning, when delivered as part of the video transport stream over HLS as data packets in MPEG-2 video streams.
- WebVTT caption files, which are referenced from the M3U8 manifest files as defined in the HLS specifications.

These files are automatically available as closed-caption tracks in the Primetime player.

You can do the following:

- Select an available caption track to be the current track and listen for events that indicate additional available tracks.
- Switch closed captioning on (visible) or off (not visible) by using the `MediaPlayer` interface.
- Select styling options that dictate how closed captions are rendered by the underlying video engine.

Use the `MediaPlayerItem` interface to select formats such as the font or font color.

Select a current caption track from among available tracks

You can select a track from a list of currently available closed-caption tracks. This becomes the current track, which is displayed when visibility is on. Some tracks might not be available initially, so listen for the event that indicates that more have become available.

1. Wait for the media player to be in at least the `PREPARED` status.
2. Listen for these events:
    - `MediaPlayerEvent.STATUS_CHANGED` with status `MediaPlayerStatus.INITIALIZED`: The initial list of closed-caption tracks is available.

3. Get a list of all currently available closed-caption tracks.

   For example:

```
List<ClosedCaptionsTrack> ccTracks =
  mediaPlayer.getCurrentItem().getClosedCaptionsTracks();
```

4. Select an available track to be the current track.

   For example:

```
// Select the initial CC track.
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;
    }
}
```

5. Implement a listener for the event that indicates that more tracks are available. When TVSDK dispatches the event, retrieve the current list of available tracks.

   Retrieve the list each time that the event occurs to ensure that you always have the most current list.

Control closed-caption visibility

You can control the visibility of closed captions. When visibility has been enabled, the currently selected track is displayed. If you change which track is current, the visibility setting remains the same.

Tip: *If closed caption text is displayed when the player enters seek mode, the text no longer displays after the seek completes. Instead, after a few seconds, TVSDK displays the next closed caption text in the video after the ending seek position.*

The visibility values for closed captions are defined in `MediaPlayer.Visibility`*.*

```
enum Visibility {
    VISIBLE,
    INVISIBLE
}
```

1. Wait for the `MediaPlayer` to be in at least the PREPARED status.

   For more information, see *Wait for a valid status*.

2. To get the current visibility setting for closed captions, use the getter method in `MediaPlayer`, which returns a visibility value.

```
MediaPlayer.Visibility getCCVisibility() throws MediaPlayerException;
```

3. To change the visibility for closed captions, use the setter method, passing a visibility value from `MediaPlayer.Visibility`.

   For example:

```
mediaPlayer.setCCVisibility(MediaPlayer.Visibility visibility);
```

Allow users to change the caption track

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

1. Create a 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, that is, channels for which TVSDK has discovered data, 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 lists all the default closed caption 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

You can provide styling information for closed caption tracks using the `TextFormat` class, which sets the style for closed captions that are displayed by your player.

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 TVSDK methods.

1. Wait for the media player to be in at least the `PREPARED` status.
2. Create a `TextFormatBuilder` instance.

   You can provide all the closed-caption styling parameters now or set them later.

   TVSDK encapsulates closed-caption styling information in the `TextFormat` interface. The `TextFormatBuilder` class creates objects that implement this interface.

   ```
   public TextFormatBuilder(
      TextFormat.Font font,
      TextFormat.Size size,
      TextFormat.FontEdge fontEdge,
      java.lang.String fontColor,
      java.lang.String backgroundColor,
      java.lang.String fillColor,
      java.lang.String edgeColor,
      int fontOpacity,
      int backgroundOpacity,
      int fillOpacity
      java.lang.String bottomInset,
      java.lang.String safeArea)
   ```

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

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

   ```
   public TextFormat toTextFormat()
   ```

4. Optionally get the current closed-caption style settings by doing one of the following:

   - Get all the style settings with `MediaPlayer.getCCStyle`.

     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 MediaPlayerException if media player was already released.
     */
     public TextFormat getCCStyle() throws MediaPlayerException;
     ```

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

     ```
     public java.lang.String getFontColor();
     public java.lang.String getBackgroundColor();
     public java.lang.String getFillColor(); // retrieve the font fill color
     public java.lang.String getEdgeColor(); // retrieve the font edge color
     public TextFormat.Size getSize(); // retrieve the font size
     public TextFormat.FontEdge getFontEdge(); // retrieve the font edge type
     public TextFormat.Font getFont(); // retrieve the font type
     public int getFontOpacity();
     public int getBackgroundOpacity();
     public java.lang.String getBottomInset(java.lang.String bi);
     public java.lang.String getSafeArea(java.lang.String sa);
     ```

5. To change the style settings, do one of the following:

   - Use the setter method `MediaPlayer.setCCStyle`, passing an instance of the `TextFormat` interface:

```
/**
* 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 MediaPlayerException
*/
public void setCCStyle(TextFormat textFormat) throws MediaPlayerException;
```

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

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

```
// set font type
public void setFont(Font font)
public void setBackgroundColor(String backgroundColor)
public void setFillColor(String fillColor)
// set the font-edge color
public void setEdgeColor(String 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(String fontColor)
public void setBottomInset(String bi)
public void setSafeArea(String sa)
public void setTreatSpaceAsAlphaNum(bool)
```

     Important:

     Color Settings: *In Android TVSDK 2.X, an enhancement was made to color styling of closed captions. The enhancement allows for setting closed caption colors using a hex string representing RGB color values. The RGB hex color representation is the familiar 6 byte string you use in applications such as Photoshop:*

     • FFFFFF = Black
     • 000000 = White
     • FF0000 = Red
     • 00FF00 = Green
     • 0000FF = Blue

     and so on.

     In your application, whenever you pass color styling information to `TextFormatBuilder`, you still use the `Color` enumeration as before, but now you must add `getValue()` to the color to get the value as a string. For example:

```
tfb = tfb.setBackgroundColor(TextFormat.Color.RED.getValue());
```

Setting the closed-caption style is an asynchronous operation, so it might take up to a few seconds for the changes to appear on the screen.

Closed caption styling options

You can specify several caption styling options, and these options override the style options in the original captions.

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

Tip: *In options that define default values (for example, `DEFAULT`), that value refers to what the setting was when the caption was originally specified.*

Format	Description
Font	The font type. Can be set only to a value that is defined by the `TextFormat.Font` enumeration and represents, for example, monospaced with or without serifs. ***Tip:*** *The actual fonts that are available on a device might vary, and substitutions are used when necessary. Monospace with serifs is typically used as a substitute, although this substitution can be system specific.*
Size	The caption's size. Can be set only to a value defined by the `TextFormat.Size` enumeration: • `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. Can be set only to a value that is defined by the `TextFormat.FontEdge` enumeration.
Font color	The font color. Can be set only to a value defined by the `TextFormat.Color` enumeration.
Edge color	The color of the edge effect. Can be set to any of the values that are available for the font color.
Background color	The background character cell color.

Format	Description
	Can be set only to values that are available for the font color.
Fill color	The color of the background of the window in which the text is located. Can be set to any of the values that are available for the font color.
Font opacity	The opacity of the text. Expressed as a percentage from 0 (fully transparent) to 100 (fully opaque). `DEFAULT_OPACITY` for the font is 100.
Background opacity	The opacity of the background character cell. Expressed as a percentage from 0 (fully transparent) to 100 (fully opaque). `DEFAULT_OPACITY` for the background is 100.
Fill opacity	The opacity of the background of the caption window. Expressed as a percentage from 0 (fully transparent) to 100 (fully opaque). `DEFAULT_OPACITY` for fill is 0.
Bottom Inset	Vertical distance from the bottom of the caption window for captions to avoid. Expressed as a percentage of the caption window height (for example, "20%") or a number of pixels (for example, "20").
Safe area	A region around the edge of the screen between 0% to 25% where captions will not appear. By default, the safe area for WebVTT is 0%. This setting allows your application to override that default. If two values are provided, for example, the string "10%,20%", the first value is the horizontal safe area and the second value is the vertical safe area. If one value is provided, for example, the string "15%", both the vertical and horizontal axes use the specified safe area.

Caption formatting examples

Here are some examples that show you how to specify closed caption formatting.

Example 1: Specify format values 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.getValue(),
            TextFormat.Color.DEFAULT.getValue(),
            TextFormat.Color.DEFAULT.getValue(),
```

```
            TextFormat.Color.DEFAULT.getValue(),
            TextFormat.DEFAULT_OPACITY,
            TextFormat.DEFAULT_OPACITY,
            TextFormat.DEFAULT_OPACITY).toTextFormat();

        mediaPlayer.setCCStyle(tf);
        ...
    }
}
```

Example 2: Specify format values in 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,
    String fontColor, String backgroundColor,
    String fillColor, String 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);
...
```

Alternate audio

Alternate audio allows you to switch among available audio tracks for a video track. Users can select their preferred language track when the video is played.

When TVSDK creates the `MediaPlayerItem` instance for the current video, it creates an `AudioTrack` item for each available audio track. The item contains a `name` property, which is a string that typically contains a user-recognizable description of the language of that track. The item also contains information about whether to use that track by default. When it is time to play the video, you can ask for a list of available audio tracks, optionally allow the user select a track, and set the video to play with the selected track.

Tip: *Although rare, if an additional audio track becomes available after TVSDK creates the* `MediaPlayerItem`*, TVSDK fires a* `MediaPlayerItem.AUDIO_TRACK_UPDATED` *event.*

Added APIs

The following APIs have been added to support alternate audio:

`hasAlternateAudio`

If the specified media has an alternate audio track, other than default track, this boolean function returns `true`. If there is no alternate audio track, the function returns `false`.

```
boolean hasAlternateAudio();
```

`getAudioTracks`

This function returns list of all the current available audio tracks in a specified media.

```
List<AudioTrack> getAudioTracks();
```

`getSelectedAudioTrack`

This function that returns the currently selected alternate audio track and properties such as language. The auto-selection of track can also be extracted.

```
AudioTrack getSelectedAudioTrack();
```

`selectAudioTrack`

This function selects an alternate audio track to play.

```
void selectAudioTrack(AudioTrack audioTrack);
```

For example:

```
private void onPrepared() {
    // Select the AA track in PREPARED State
    boolean hasAlternateAudio = _mediaPlayer.getCurrentItem().hasAlternateAudio();
    if(hasAlternateAudio) {
        AudioTrack selectedAudioTrack =
          _mediaPlayer.getCurrentItem().getSelectedAudioTrack();

        if (selectedAudioTrack == null) {
            // Selecting default audio track
            // If index is 1 it will select alternate audio track
            selectedAudioTrack = _mediaPlayer.getCurrentItem().getAudioTracks().get(0);
        }
    }
    _mediaPlayer.getCurrentItem().selectAudioTrack(selectedAudioTrack);
}
```

AC-3 5.1 format

The Audio Codec 3 (AC-3, also known as Dolby Digital®) 5.1 format, allows content providers to compress the size of multichannel audio files without impairing the sound quality. AC-3 is a 5.1 format, which means that it provides five full-bandwidth channels for a richer user experience.

For more information, see *Dolby Digital 5.1*.

TVSDK supports the following AC-3 5.1 features:

- AC-3 surround audio
- Mixed/unmixed streams for surround audio type
- Ability to query the device to see if the surround audio codec is available on the device.

The results determine which preferred audio codec type is selected. The manifest with audio codec type that the device is not going to use is discarded. For example, if the AC-3 format has been selected, profiles with the Advanced Audio Coding (AAC) format are not considered.

- Passthrough mode

  In passthrough mode, instead of decoding the media from the AC-3 5.1 format to a multichannel pulse-code modulation (PCM) format, the TVSDK gets modified or unmodified (depending on the device) Dolby media from the Decoder. This media is sent to the audio device (speaker or receiver) so that the audio device can decode and play back the Dolby surround stream.

Important: *TVSDK supports the AC-3 5.1 features only on the device Amazon Fire TV 1st generation.*

The following AC-3 5.1 features are not supported:

- Multichannel AAC audio
- ABR across different codecs (AAC - AC3)

Selecting supported media

Here is the typical workflow that occurs when TVSDK finds a manifest with AC-3 and AAC media:

1. TVSDK queries which codec the device can support.
2. The codec with the higher quality is selected.

   Here is the order in which quality is selected:

   1. AC-3
   2. AAC

3. TVSDK ignores profiles with other audio codec types.

Tip: *The application cannot obtain information about ignored profiles.*

Determining the output mode

While processing AC-3 media, if an Android device is connected to the speaker system, the decision to play content in surround mode or stereo mode depends on how the device is configured.

	Surround sound	**Stereo speaker**
Device configuration Dolby on (or auto)	Device configuration Dolby on (or auto)	Stereo mode
Device configuration Dolby off	Stereo mode	Stereo mode

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. 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.

Alternate audio tracks allow users to switch between multiple language tracks for HTTP video streams (live/linear and VOD), and you do not have to modify, duplicate, or repackage the video for each audio track. You can provide multiple language tracks for a video asset before or after the asset's initial packaging.

Important: *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 `default` label. 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

Alternate audio uses `MediaPlayer` to play a video that is specified in an M3U8 HLS playlist and that can contain several alternate audio streams.

1. Wait for the `MediaPlayer` to be in at least the `MediaPlayerStatus.PREPARED` status.
2. Listen for the `MediaPlayerEvent.STATUS_CHANGED` event with status `MediaPlayerStatus.PREPARED`.

   This step means that the initial list of audio tracks is available.
3. Get the available audio tracks from the `MediaPlayerItem` instance.

   ```
   mediaPlayerItem.getAudioTracks()
   ```
4. (Optional) Present the available tracks to the user.
5. Set the selected audio track on the `MediaPlayerItem` instance.

   ```
   mediaPlayerItem.selectAudioTrack(audioTrack)
   ```

Enabling Background Audio

To enable audio playback when app is in background, app should call `enableAudioPlaybackInBackground` API of MediaPlayer with true as argument when player is in PREPARED state.

```
_mediaPlayer.enableAudioPlaybackInBackground(true);
```

App should pause playback when it loses its hold on audio focus during events like responding to the phone, etc. The following code snippet demonstrates how to implement the `OnAudioFocusChangeListener`:

```
/**
 * Register the AudioFocus Change listener to track Audio focus from device.
 */
 AudioManager.OnAudioFocusChangeListener onAudioFocusChangeListener = new
AudioManager.OnAudioFocusChangeListener(){
     @Override
     public void onAudioFocusChange(int focusChange){
          switch(focusChange){
               case AudioManager.AUDIOFOCUS_GAIN:
                    break;
               case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK:
                    /* lower output volume*/
                    break;
               case AudioManager.AUDIOFOCUS_LOSS:
               case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT:
                    if(_lastKnownStatus ==MediaPlayerStatus.PLAYING)
                         _mediaPlayer.pause();
                    break;
          }
     }
};

AudioManager audioManager = (AudioManager)
getActivity().getApplicationContext().getSystemService(Context.AUDIO_SERVICE);
audioManager.requestAudioFocus(onAudioFocusChangeListener, AudioManager.STREAM_MUSIC,
AudioManager.AUDIOFOCUS_GAIN);
```

ID3 tags

ID3 tags provide information about an audio or video file, such as the title of the file or the name of the artist. TVSDK detects ID3 tags at the transport stream (TS) segment level in HLS streams and dispatches an event. The application can extract data from the tag.

Important: *TVSDK recognizes ID3 metadata (version 2.3.0 or 2.4.0) in audio (AAC) and video (H.264) streams in any of its possible encodings (ASCII, UTF8, UTF16-BE, or UTF16-LE). It ignores ID3 tags that are not in one of the recognized versions or formats. Unspecified encoding is treated as UTF8.*

When TVSDK detects ID3 metadata, it issues a notification with the following data:

- TYPE = ID3
- NAME = ID3

1. Implement an event listener for `MediaPlayer.TimedMetadataEventListener#onTimedMetadata(TimeMetadata timeMetadata)` and register it with the `MediaPlayer` object.

   TVSDK calls this listener when it detects `ID3` metadata.

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

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){
            byte[] value = metadata.getByteArray(key);
        }
    }
}
```

Buffering

To provide a smoother viewing experience, TVSDK sometimes buffers the video stream. You can configure how the player buffers.

TVSDK defines a playback buffer length of at least 30 seconds and an initial buffer time before the media starts playing, of at least 2 seconds. After the application calls `play`, but before playback begins, TVSDK buffers the media up to the initial time to give a smooth start when it actually starts playing.

You can alter the buffer times by defining new buffering policies, and you can alter when the initial buffering occurs by using instant on.

Buffering time policies

Depending on your environment (including the device, the operating system, or the network conditions), you can set different buffering policies for your player, such as changing the minimum duration for initial buffering and for ongoing playback buffering.

After you call `play`, the media player begins buffering the video. When the media player has buffered the amount of video specified by the initial buffer time, playback begins. This process improves the start-up time because the player does not wait for the entire playback buffer to fill before starting playback. Instead, after the few initial seconds are buffered, playback begins.

While the video is being rendered, TVSDK continues to buffer new fragments until it is has buffered the amount that is specified by the playback buffer time. If the current buffer length drops below the playback buffer time, the player will download additional fragments. Once the current buffer length is above the playback buffer time by a few seconds, TVSDK will stop downloading fragments.

Tip: *If the initial buffer value is high, it might give your user a long initial buffering time before starting. That might provide smooth playback for a longer time; however, if network conditions are poor, initial playback could be delayed.*

If you enable instant on by calling `prepareBuffer`, the initial buffering begins at that moment, instead of waiting for `play`.

Set buffering times

The `MediaPlayer` provides methods to set and get the initial buffering time and playback buffering time.

Tip: *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 ongoing playback buffer time.*

1. Set up the `BufferControlParameters` object, which encapsulates the initial buffer time and playback buffer time control parameters.

   This class provides the following factory methods:

- To set the initial buffer time equal to the play buffer time:

```
public static BufferControlParameters createSimple(long bufferTime)
```

- To set the initial and play buffer times:

```
public static BufferControlParameters createDual(
  long initialBuffer,
  long bufferTime)
```

If the parameters are not valid, these methods throw `MediaPlayerException` with error code `PSDKErrorCode.INVALID_ARGUMENT`, such as when the following conditions are met:

- The initial buffer time is less than zero.
- The initial buffer time is greater than the buffer time.

2. To set the buffer parameters values, use this `MediaPlayer` method:

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

3. To get the current buffer parameter values, use this `MediaPlayer` method:

```
BufferControlParameters getBufferControlParameters()
```

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

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

Instant On

Enabling instant on means that one or more channels are preloaded. When users select a channel or switch channels, the content plays immediately. The buffering is complete by the time the user starts watching.

Without Instant On, TVSDK initializes the media to be played but does not start buffering the stream until the application calls `play`. The user sees no content until buffering is complete. With Instant On, you can launch multiple media player (or media-player item loader) instances, and TVSDK starts buffering the streams immediately. When a user changes the channel and the stream has buffered properly, calling `play` on the new channel starts playback immediately.

Although there are no limits to the number of `MediaPlayer` and `MediaPlayerItemLoader` instances that TVSDK can run, running more instances consumes more resources. Application performance can be affected by the number of instances that are running. For more information about `MediaPlayerItemLoader`, see *Load a media resource in the media player*.

Important: *TVSDK does not support a single `QoSProvider` to work with both `itemLoader` and `MediaPlayer`. If the customer uses Instant On, the application needs to maintain two QoS instances and manage both instances for the information.*

For more information about `MediaPlayerItemLoader`, see *Load a media resource using MediaPlayerItemLoader*.

Add a QoS Provider instance to mediaPlayerItemLoader

Create and attach a QoS Provider to a `mediaPlayerItemLoader` instance

```
// Create an instance of QoSProvider
private QOSProvider _qosProvider = new QOSProvider(this._context);

// Attach the QoSProvider instance to the mediaPlayerItemLoaderInstance
// (before calling load API on mediaPlayerItemLoader instance)
_qosProvider.attachMediaPlayerItemLoader(this._loader);
```

Once the playback starts, use the `_qosProvider` to get `timeToLoad` and `timeToPrepare` QoSdata. The remaining QoS metrics can be retrieved by using the `QoSProvider` attached to the `mediaPlayer`.

For more information about `MediaPlayerItemLoader`, see *Load a media resource using MediaPlayerItemLoader*.

Configure buffering for Instant On

TVSDK provides methods and statuses to allow you use Instant On with a media resource.

Note: *Adobe recommends using `MediaPlayerItemLoader` for InstantOn. To use `MediaPlayerItemLoader`, rather than `MediaPlayer`, see Load a media resource using MediaPlayerItemLoader.*

1. Confirm that the resource has loaded, and the player is prepared to play the resource.
2. Before calling `play`, call `prepareBuffer` for each `MediaPlayer` instance.

   `prepareBuffer` enables Instant On, and TVSDK starts buffering immediately and dispatches the `BUFFERING_COMPLETED` event when the buffer is full.

   Tip: *By default, `prepareBuffer` and `prepareToPlay` set up the media stream to start playing from the beginning. To start at another position, pass the position (in milliseconds) to `prepareToPlay`.*

```
@Override
public void onStatusChanged(MediaPlayerStatus status) {
    switch (status) {
        case INITIALIZED:
            // This example starts 5 seconds into the stream.
            mediaPlayer.prepareToPlay(5000);
            break;
        case PREPARING:
            break;
        case PREPARED:
            mediaPlayer.prepareBuffer();
            break;
        ...
    }
}
```

3. When you receive the `BUFFERING_COMPLETE` event, start playing the item or display visual feedback to indicate that the content is completely buffered.

   If you call `play`, playback should begin immediately.

Parallel downloads

Downloading video and audio in parallel, rather than in a series, reduces startup delays.

Parallel downloads allows video-on-demand (VOD) files to be played, optimizes the available bandwidth usage from a server, lowers the probability of getting into buffer under-run situations, and minimizes the delay between download and playback.

Without parallel downloads, TVSDK issues a request for the video segment, and after the video segment is loaded, it requests one or two audio segments. With parallel downloads, the audio and video segments are downloaded simultaneously, not sequentially. Also, because there are two connections and two HTTP requests per segment in parallel, the data reaches the screen faster.

Restriction: *This feature applies only to content where the audio and video are encoded into different files (unmuxed content) and does not apply to MP4 content, which is always muxed. HLS content is often unmuxed, especially with alternate audio.*

The HTTP connection might experience delays at the following stages:

- When establishing the TCP/IP connection to server

  Although the client and server have agreed to communicate, no HTTP communication has occurred yet. This type of delay depends on the infrastructure between the client and the server. This process requires finding a path through the internet between the client and the server and making sure all devices, such as routers and firewalls, on the route agree to the data transfer.

- When sending an HTTP request for a segment or a manifest over the TCP/IP connection.

  The server receives the request, processes it, and starts sending the data back to the client. The degree of delay depends on the load and the complexity of the software on the server and somewhat on the upload connection speed when the client sends the request.

Adaptive bit rates (ABR) for video quality

HLS and DASH streams provide different bit rate encodings (profiles) for the same short burst of video. TVSDK can select the quality level for each burst based on the current buffering level and the available bandwidth.

TVSDK constantly monitors the bit rate to ensure that the content is played at the optimal bit rate for the current network connection. You can set the adaptive bit rate (ABR) switching policy and the initial, minimum, and maximum bit rates for a multiple-bit-rate (MBR) stream. TVSDK automatically switches to the bit rate that provides the best playback experience in the specified configuration.

Initial bit rate	The desired playback bit rate (in bits per second) for the first segment. When playback starts, the closest profile, which is equal to or greater than the initial bit rate, is used for the first segment. If a minimum bit rate is defined, and the initial bit rate is lower than the minimum rate, TVSDK selects the profile with the lowest bit rate above the minimum bit rate. If the initial rate is above the maximum rate, TVSDK selects the highest rate below the maximum rate. If the initial bit rate is zero or undefined, the initial bit rate is determined by the ABR policy. `getABRInitialBitRate` returns an integer value that represents the byte-per-second profile.
Minimum bit rate	The lowest allowed bit rate to which the ABR can switch. ABR switching ignores profiles with a bit rate that is lower than this bit rate. `getABRMinBitRate` returns an integer value that represents the bits-per-second profile.
Maximum bit rate	The highest allowed bit rate to which the ABR can switch. ABR switching ignores profiles with a bit rate higher than this bit rate. `getABRMaxBitRate` returns an integer value that represents the bits-per-second profile.
ABR switching policy	Playback switches gradually to the highest-bit-rate profile when possible. You can set the policy for ABR switching, which determines how quickly TVSDK switches between profiles. The default is `ABR_MODERATE`.

	When TVSDK decides to switch to a higher bit rate, the player selects the ideal bit rate profile to switch to based on the current ABR policy: • `ABR_CONSERVATIVE`: Switches to the profile with the next higher bit rate when the bandwidth is 50% higher than the current bit rate. • `ABR_MODERATE`: Switches to the next higher bit rate profile when the bandwidth is 20% higher than the current bit rate. • `ABR_AGGRESSIVE`: Switches immediately to the highest bit-rate profile when the bandwidth is higher than the current bit rate. If the initial bit rate is zero, or is not specified but a policy is specified, playback starts with the lowest bit rate profile for a conservative policy, the profile closest to the median bit rate of available profiles for a moderate policy, and the highest bit rate profile for an aggressive policy. The policy works in the constraints of the minimum and maximum bit rates, if these rates are specified. `getABRPolicy` returns the current setting from the `ABRControlParameters` enum: `ABR_CONSERVATIVE`, `ABR_MODERATE`, or `ABR_AGGRESSIVE`. For more information, see *ABRControlParameters API doc*.

Keep the following information in mind:

- The TVSDK failover mechanism might override your settings, because TVSDK favors a continuous playback experience over strictly adhering to your control parameters.
- When the bit rate changes, TVSDK dispatches `onProfileChanged` events in `PlaybackEventListener`.
- You can change your ABR settings at any time, and the player switches to use the profile that most closely matches the most recent settings.

For example, if a stream has the following profiles:

- 1: 300000
- 2: 700000
- 3: 1500000
- 4: 2400000
- 5: 4000000

If you specify a range of 300000 to 2000000, TVSDK considers only profiles 1, 2 and 3. This allows applications to adjust to various network conditions, such as switching from wi-fi to 3G or to various devices such as a phone, a tablet, or a desktop computer.

To set ABR control parameters, set the parameters on the `ABRControlParameter` class.

Configure adaptive bit rates using ABRControlParameters

You can set ABR control values only with `ABRControlParameters`, but you can construct a new one at any time.

The following conditions apply to `ABRControlParameters`:

- At construction time, you must provide values for all parameters.
- After the construction, you cannot change individual values.

• If the parameters that you specify are outside of the allowed range, an `ArgumentError` is thrown.

1. Determine your initial, minimum, and maximum bit rates.
2. Determine the ABR policy:
   • `ABR_CONSERVATIVE`
   • `ABR_MODERATE`
   • `ABR_AGGRESSIVE`
3. Set the ABR parameter values in the `ABRControlParameters` constructor and assign the values to the Media Player.

```
public ABRControlParameters(int initialBitRate,
  int minBitRate,
  int maxBitRate,
  ABRControlParameters.ABRPolicy abrPolicy,
  int minTrickPlayBitRate,
  int maxTrickPlayBitRate,
  int maxTrickPlayBandwidthUsage,
  int maxPlayoutRate);
```

Quality of service statistics

Quality of service (QoS) provides a detailed view into how the video engine is performing. TVSDK provides detailed statistics about playback, buffering, and devices.

TVSDK also provides information about the following downloaded resources:

• Playlist/manifest files
• File fragments
• Tracking information for files

Track at the fragment level using load information

You can read quality of service (QoS) information about downloaded resources, such as fragments and tracks, from the `LoadInformation` class.

1. Implement and register the `MediaPlayerEvent.LOAD_INFORMATION_AVAILABLE` event listener.
2. Call `event.getLoadInformation()` to read the relevant data from the `event` parameter that is passed to the callback.

   For more about `LoadInformation`, see *2.5 for Android (Java)* API docs.

Read QOS playback, buffering, and device statistics

You can read playback, buffering, and device statistics from the `QOSProvider` class.

The `QOSProvider` class provides various statistics, including information about buffering, bit rates, frame rates, time data, and so on. It also provides information about the device, such as the manufacturer, model, operating system, SDK version, manufacturer's device ID, and screen size/density.

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

   The `QOSProvider` constructor takes a player context so that it can retrieve device-specific information.

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

3. (Optional) Read the playback statistics.

   One solution to read playback statistics is to have a timer, that periodically fetches the new QoS values from the `QOSProvider`.

   For example:

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

                setQosItem("Perceived Bandwidth", (int)
playbackInformation.getPerceivedBandwidth());
                playbackInformation.getPerceivedBandwidth());
            }
        });
    };
};
```

4. (Optional) Read the device-specific information.

```
// Show device information
DeviceInformation deviceInfo = new QOSProvider(parent.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.aboutDeviceResolutin);
String orientation = parent.getResources().getConfiguration().orientation ==
  Configuration.ORIENTATION_LANDSCAPE ? "landscape" : "portrait";
tv.setText(parent.getString(R.string.aboutDeviceResolution) + " " +
  deviceInfo.getWidthPixels() + "x" + deviceInfo.getHeightPixels() +
  " (" + orientation + ")");
```

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 that is played locally.

Important: *Primetime cannot protect from failures such as an ISP outage or a cable disconnection.*

Primetime streaming provides failover protection to protect playback from certain remote server failures or operational failures, which makes a better viewing experience. Despite transmission problems, TVSDK implements failover protection to minimize playback interruptions and achieve a seamless playback. The video player automatically switches to a backup media set when entire renditions or fragments are unavailable.

Media playback and failover

For live and video-on-demand (VOD) media, TVSDK starts playback by downloading the playlist that is associated with the middle-resolution bit rate and downloads the media segments that are defined by that playlist. It quickly selects the high-resolution bit rate playlist and its associated media and continues the downloading process.

Missing playlist failover

When an entire playlist is missing, for example, when the M3U8 file that is specified in a top-level manifest file does not download, TVSDK attempts to recover. If it cannot be recovered, your application determines the next step.

If the playlist that is associated with the middle-resolution bit rate is missing, TVSDK searches for a variant playlist at the same resolution. If it finds the same resolution, TVSDK starts downloading the variant playlist and the segments from the matching position. If the player does not find the same resolution playlist, it will try to cycle through other bitrate playlists and their variants. An immediately lower bitrate is the first choice, then its variant, and so on. If all of the lower bitrate playlists and their variants are exhausted in the attempt to find a valid playlist, TVSDK will go to the top bitrate and count down from there. If a valid playlist cannot be found, the process fails, and the player moves to the ERROR state.

Your application can determine how to handle this situation. For example, you might want to close the player activity and direct the user to the catalog activity. The event of interest is the `STATUS_CHANGED` event, and the corresponding callback is the `onStatusChanged` method. Here is code that monitors whether the player changes its internal status to `ERROR`:

```
...
case ERROR:
getActivity().finish(); // this is where we close the current activity (the Player activity)
break;
...
```

Missing segment failover

When a segment is missing, for example when a particular segment fails to download, TVSDK attempts to recover through a variety of failover attempts. If it cannot recover, it issues an error.

If a segment is missing on the server because, for example, the manifest file is not present, the segment cannot be downloaded, and so on, TVSDK attempts to fail over by attempting the following options:

1. Attempt a failover to the same segment, at the same bit rate, in a variant file.
2. Switch to an alternate bit rate (ABR switch) in the same file.
3. Cycle through every available bit rate in every available variant.
4. Skip the segment and issue a warning.

When TVSDK cannot obtain an alternative segment, it triggers a `CONTENT_ERROR` error notification. This notification contains an inner notification with the `DOWNLOAD_ERROR` code. If the stream with the problem is an alternate audio track, TVSDK generates the `AUDIO_TRACK_ERROR` error notification.

If the video engine is continuously unable to obtain segments, it limits continuous segment skips to 5, after which playback is stopped and TVSDK issues a `NATIVE_ERROR` with the code 5.

Restriction:

Here are some restrictions you should be aware of:

- *The adaptive bit rate (ABR) control parameters are not taken into consideration when a failover occurs.*

  This is because the failover mechanism is designed to use any of the currently available playlists, regardless of their bit rate profile, as backup streams.

- *During a failover operation, there can be a profile switch.*

  If an error occurs during the download of one of the playlist segments, ABR control parameters such as min/max allowed bit rate are ignored.

Advertising insertion and failover for VOD

The video-on-demand (VOD) ad-insertion process consists of the ad resolving, ad insertion, and ad playback phases. For ad tracking, TVSDK must inform a remote tracking server about the playback progress of each ad. When unexpected situations arise, TVSDK takes appropriate action.

Ad-resolving phase

TVSDK contacts an ad delivery service, such as Adobe Primetime ad decisioning, and attempts to obtain the primary playlist file that corresponds to the video stream for the ad. During the ad-resolving phase, TVSDK makes an HTTP call to the remote ad-delivery server and parses the server's response.

TVSDK supports the following types of ad providers:

- Metadata ad provider

  The ad data is encoded in plain-text JSON files.

- Primetime ad decisioning ad provider

  TVSDK sends a request, including a set of targeting parameters and an asset identification number, to the Primetime ad decisioningback-end server. Primetime ad decisioningresponds with a synchronized multimedia integration language (SMIL) document that contains the required ad information.

- Custom ad markers provider

  Handles the situation where ads are burned into the stream, from the server side. TVSDK 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 TVSDK uses to perform the ad tracking.

One of the following failover situations can occur during this phase:

- The data cannot be retrieved because, for example, of the lack of connectivity or a server-side error, such as a resource cannot be found, and so on.
- The data was retrieved, but the format is invalid.

  This might occur because, for example, the parsing of the inbound data failed.

TVSDK issues a warning notification about the error and continues processing.

Ad-insertion phase

TVSDK inserts the alternate content (ads) into the timeline that corresponds to the main content.

When the ad-resolving phase is complete, TVSDK has an ordered list of ad resources that are grouped into ad breaks. Each ad break is positioned on the main content timeline by using a start-time value that is expressed in

milliseconds (ms). Each ad in an ad break has a duration property that is also expressed in ms. The ads in an ad break are chained, and as a result, the duration of an ad break is equal to the sum of the durations of the individual composing ads.

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

TVSDK issues a warning notification about the error and continues processing.

Ad-playback phase

TVSDK downloads the ad segments and renders them on the device's screen.

Now, TVSDK has resolved ads, positioned the ads on the timeline, and attempts to render the content on the screen.

The following main classes of errors might occur during the following phases:

• When connecting to the host server.
• When downloading the manifest file.
• When downloading the media segments.

TVSDK forwards the triggered events to your application, including notification events that are triggered when:

• A failover happens.
• The profile is changed because of the failover algorithm.
• All failover options have been considered, and no additional action can be taken automatically.

  Your application needs to take the appropriate action.

Regardless of whether errors occur, TVSDK calls `onAdBreakComplete` for each `onAdBreakStart` and `onAdComplete` for every `onAdStart`. However, if segments could not be downloaded, there might be gaps in the timeline. When the gaps are large enough, the values in the playhead position and the reported ad progress might exhibit discontinuities.

Advertising

You can insert ads in your VOD and live/linear content by using the Adobe Primetime ad decisioning interface.

Primetime ad decisioningworks with TVSDK to identify ad opportunities, resolve ads, and insert resolved ads in your video streams.

Advertising requirements

To incorporate ads in your video content, ensure that the advertising and main video content meets the following requirements:

• The advertising content's HLS version cannot be higher than the main content's HLS version.
• Ads do not have to be multiplexed (with no restrictions), regardless of whether the main content is multiplexed.

Inserting ads

Ad insertion resolves ads for video-on-demand (VOD), for live streaming, and for linear streaming with ad tracking and ad playback. TVSDK makes the required requests to the ad server, receives information about ads for the specified content, and places the ads in the content in phases.

An *ad break* contains one or more ads that play in sequence. TVSDK inserts ads in the main content as members of one or more ad breaks.

Note: *If the ad has errors, TVSDK ignores the ad.*

VOD ad resolving and insertion

For video-on-demand (VOD) content, TVSDK inserts ad breaks by splicing the ads in the main content so that the timeline duration increases.

Before playback, TVSDK resolves known ads, inserts ad breaks in the main content as described by a timeline that is returned from Adobe Primetime ad decisioning, and recomputes the virtual timeline, if necessary.

TVSDK inserts ads in the following ways:

- **Pre-roll**, which is placed before the content.
- **Mid-roll**, which is in the middle of the content.
- **Post-roll**, which is placed after the content.

Tip: *After playback starts, no additional changes can occur in the content.*

Ads cannot be:

- Inserted
- Deleted

  For example, you cannot delete built-in ads from the content to offer an ad-free experience.

- Replaced

  For example, you cannot replace built-in ads with targeted ads.

Live/linear ad resolving and insertion

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

Before and during playback, TVSDK resolves known ads, replaces parts of the main content with ad breaks of the same duration, and recomputes the virtual timeline, if necessary. The positions of the ad breaks are specified by cue points that are defined by the manifest.

TVSDK inserts ads in the following ways:

- **Pre-roll**, which is placed before the content.
- **Mid-roll**, which is placed in the middle of the content.

TVSDK accepts the ad break even if the duration is longer or shorter than the cue point replacement duration. By default, TVSDK supports the `#EXT-X-CUE` cue as a valid ad marker when resolving and placing ads. This marker requires the metadata field `DURATION` value to be expressed in seconds and the cue's unique ID. For example:

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

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

After playback starts, the video engine periodically refreshes the manifest file. TVSDK resolves any new ads and inserts the ads when a cue point is encountered in the live or linear stream that was defined in the manifest. After ads are resolved and inserted, TVSDK computes the virtual timeline again and dispatches a `TimelineItemsUpdatedEventListener.onTimelineUpdated` event.

Implement an early ad break return

For live stream ad insertion, you might need to exit from an ad break before all the ads in the break are played to completion.

For example, the duration of the ad break in certain sports events might not be known before the break starts. TVSDK provides a default duration, but if the game resumes before the break finishes, the ad break must be exited. Another example is an emergency signal during an ad break in a live stream.

1. Subscribe to `#EXT-X-CUE-OUT`,`#EXT-X-CUE-IN`, and `#EXT-X-CUE`, which are the splice out/splice in markers.
   For more information about to how to splice out/in ad markers, see *Opportunity generators and content resolvers*
2. Use a custom `ContentFactory`.
3. In `retrieveGenerators`, use the `SpliceInPlacementOpportunityGenerator`.

   For example:

   ```
   public List<OpportunityGenerator> retrieveGenerators(MediaPlayerItem item) {
       List<OpportunityGenerator> generators = new ArrayList<OpportunityGenerator>();
       generators.add(SpliceInPlacementOpportunityDetector(item));
       return generators;
   }
   ```

   For more information about using a custom `ContentFactory`, see step 1 in *Implement a custom opportunity generator*.

4. On the same custom `ContentFactory`, implement `retrieveResolvers` and include `AuditudeResolver` and `SpliceInCustomResolver`.

   For example:

   ```
   List<ContentResolver> contentResolvers = new ArrayList<ContentResolver>();
   contentResolvers.add(new AuditudeResolver(getActivity().getApplicationContext()));
   contentResolvers.add(new SpliceInCustomResolver());
   return contentResolvers;
   ```

Client ad tracking

TVSDK automatically tracks ads for VOD and live/linear streaming.

Notifications are used to inform your application about an ad's progress, including information about when an ad begins and when it ends.

Client error handling for broken VMAP

When TVSDK encounters a broken VMAP in an ad server response, it dispatches an 1109 (`NETWORK_AD_URL_FAILED`) error.

Depending upon the nature of the ad server response, and upon your ad loading settings, your player could receive different numbers of 1109 errors when TVSDK encounters a broken VMAP in an ad server response.

Let's consider a scenario in which the ad server response points to VMAP XML. Let's also say that the ad server response has four available ad slots, each of which points to the same VMAP. Finally, let's say that this VMAP is broken.

In this scenario, if lazy ad resolving is enabled (*t_enable-lazy-ad-resolving.xml*), TVSDK will dispatch two 1109 errors (not one as might be expected): one error is dispatched on each parsing pass over the timeline. This is because when lazy ad resolving is enabled, TVSDK parses the ads in 2 passes: the first pass happens just before the content playback starts for pre-roll ads, and the second pass happens after playback starts, for mid-roll and post-roll ads.

Note: *In this scenario, if you disable lazy ad resolving, TVSDK will fire only one 1109 error (only one parsing pass).*

Secure Ad loading over HTTPS

Adobe Primetime provides an option to request first call to the Primetime ad server and CRS related calls over HTTPS.

The feature is not enabled by default. Use the following to enable secure ad loading.

```
AuditudeSettings auditudeSettings = new AuditudeSettings();
auditudeSettings. getForceHttpsConfiguration().setAdServerCalls(true);
```

Default and customized playback behavior with ads

The behavior of media playback is affected by seeking, pausing, fast forward or rewind, and advertising.

To override the default behavior, use `AdBreakPolicySelector` .

Important: *TVSDK does not provide a way to disable seeking during ads. Adobe recommends that you configure your application to disable seeking during ads.*

Here is the playback behavior for live/linear content:

- Resuming playback after a pause results in the playback of the content that was buffered at the time of the pause.

  If the resuming position is still in the playback range, the playback should be continuous. Otherwise, TVSDK jumps to the new live point. You can also perform a seek operation and select a different playback point.

- TVSDK resolves ads between cues after the position at which the application enters live playback.

  Playback begins after the first cue is resolved. The default value for entering live playback is the client live point, but you can choose a different position. All cues before the initial position are resolved after the application performs a seek in the DVR window.

The following table describes how TVSDK handles ads and ad breaks during playback:

Video activity	**Default TVSDK behavior policy**	**Customization available through `AdBreakPolicySelector`**
During normal play, an ad break is encountered.	• For live/linear, plays the ad break, even if the ad break has already been watched. • For VOD, plays the ad break and marks the ad break as watched.	Specify a different policy for the ad break by using `selectPolicyForAdBreak`.

Video activity	Default TVSDK behavior policy	Customization available through `AdBreakPolicySelector`
Your application seeks forward over ad break(s) into main content.	Plays the last unwatched ad break that was skipped over and resumes playback at the desired seek position when the break(s) playback is complete.	Select which skipped break to play by using `selectAdBreaksToPlay`.
Your application seeks backward over ad break(s) into main content.	Skips to the desired seek position without playing ad breaks.	Select which skipped break to play by using `selectAdBreaksToPlay`.
Your application seeks forward into an ad break.	Plays from the beginning of the ad in which the seek ended.	Specify a different ad policy for the ad break and for the specific ad where the seek ended by using `selectPolicyForSeekIntoAd`.
Your application seeks backward into an ad break.	Plays from the beginning of the ad in which the seek ended.	Specify a different ad policy for the ad break and for the specific ad in which the seek ended by using `selectPolicyForSeekIntoAd`.
Your application seeks forward or backward over watched ad break(s) into main content.	If the last ad break skipped has already been watched, skips to the user-selected seek position.	Select which of the skipped breaks to play using `selectAdBreaksToPlay` and determine which breaks have already been watched by using `AdBreak.isWatched` . ***Important:*** *By default, TVSDK marks an ad break as watched immediately after entering the first ad in the ad break.*
Your application seeks forward or backward over one or more ad breaks and drops into a watched ad break.	Skips the ad break and seeks to the position immediately following the ad break.	Specify a different ad policy for the ad break (with the watched status set to true) and for the specific ad where the seek ended by using `selectPolicyForSeekIntoAd`.
Your application enters trick-play (DVR mode). Play rate can be negative (rewind) or greater than 1 (fast forward).	Skips all ads during fast forward or rewind, plays the last break skipped after trick play ends, and skips to the user-selected trick play position when that break finishes playback.	Select which of the skipped breaks to play after trick play ends using `selectAdBreaksToPlay`.
Your application seeks forward over ads that were inserted using custom ad markers.	Skips to the user-selected seek position.	For more information, see *Display a seek scrub bar with the current playback position*

Customize playback with ads

When playback reaches an ad break, passes an ad break, or ends in an ad break, TVSDK defines some default behavior for the positioning of the current playhead.

Tip: *You can override the default behavior by using the `AdBreakPolicySelector` class.*

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

You can customize ad playback behavior in the following ways:

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

  On the failover playback session, to allow the user to skip ads and resume to the previous failed position, you can disable pre-roll and/or mid-roll ads. TVSDK provides methods to enable skipping pre-roll and mid-roll ads.

API elements for ad playback

TVSDK provides classes and methods that you can use to customize the playback behavior of content that contains advertising.

The following API elements are useful for customizing playback:

API element	Content that supports advertising
`AdvertisingMetadata`	Control whether an ad break should be marked as having been watched by a viewer, and if yes, when to mark it. Set and get the watched policy using `setAdBreakAsWatched` and `getAdBreakAsWatched`.
`AdBreakPolicy`	Enumerates possible playback policies for ad breaks.
`AdPolicy`	Enumerates possible playback policies for ads.
`AdPolicySelector`	Interface that allows customization of TVSDK ad behavior.
`DefaultAdPolicySelector`	Class that implements the default TVSDK behavior. Your application can override this class to customize the default behaviors without implementing the complete interface.
`MediaPlayer`	• `getLocalTime` This is the local time of the playback, excluding the placed ad breaks. • `seekToLocal`. Here, the seek occurs relative to a local time in the stream. • `getTimeline.convertToLocalTime`. The virtual position on the timeline is converted to the local position. ***Important:*** *`getLocalTime` in `MediaPlayer` returns the current time relative to the original content, without dynamically spliced ads. `getLocalTime` in `AdBreak` returns the start time of the break relative to the original content.*

API element	Content that supports advertising
`AdBreak`	`isWatched` property. Indicates whether the viewer has watched the ad.

Use the default playback behavior

You can choose to use default ad behaviors.

To use default behaviors, complete one of the following tasks:

- If you implement your own `AdvertisingFactory` class, return null for `createAdPolicySelector`.
- If you do not have a custom implementation for the `AdvertisingFactory` class, TVSDK uses a default ad policy selector.

Set up customized playback

You can customize or override ad behaviors.

Before you customize or override ad behaviors, register the ad policy instance with TVSDK.

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

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

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

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

To customize ad behaviors:

1. Implement the `AdPolicySelector` interface and all of its methods.
2. Assign the policy instance to be used by TVSDK through the advertising factory.

   Attention: *Custom ad policies that are registered at the beginning of playback are cleared when the `MediaPlayer` instance is deallocated. Your application must register a policy selector instance each time a new playback session is created.*

   For example:

```
class CustomContentFactory extends ContentFactory {
    ...
    @Override
    public AdPolicySelector retrieveAdPolicySelector(MediaPlayerItem mediaPlayerItem) {
        return new CustomAdPolicySelector(mediaPlayerItem);
    }
    ...
}

// register the custom content factory with media player
MediaPlayerItemConfig config =  new MediaPlayerItemConfig();
config.setAdvertisingFactory(new CustomContentFactory());

// this config will should be later passed while loading the resource
mediaPlayer.replaceCurrentResource(resource, config);
```

3. Implement your customizations.

Skip ad breaks for a period of time

By default, TVSDK forces an ad break to play when the user seeks over an ad break. You can customize the behavior to skip an ad break if the time elapsed from a previous break completion is within a certain number of minutes.

Important: *If you have to complete an internal seek to forgive an ad, there might be a slight pause during the playback.*

To override the default TVSDK ad break behavior, you can extend the default ad policy selector. There are four ad break policies available:

- PLAY
- SKIP

Note: *The SKIP ad break policy might not work as expected for live streams when an ad is present at the live point. For example, for a pre-roll, SKIP will cause a seek to the end of the ad break, which could be greater than the live point. In this case, TVSDK may seek to the middle of an ad.*

- REMOVE_AFTER
- REMOVE

Note: *The `REMOVE` ad break policy is slated for deprecation. Adobe recommends that you use the `SKIP` ad break policy in place of `REMOVE`.*

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

1. When the user finishes watching an ad break, save the current system time.

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

2. Extend `AdPolicySelector`.

```
package com.adobe.mediacore.sample.advertising;

import com.adobe.mediacore.MediaPlayerItem;
import com.adobe.mediacore.MediaPlayerItemConfig;
import com.adobe.mediacore.timeline.advertising.policy.*;
import com.adobe.mediacore.timeline.advertising.AdBreakTimelineItem;
import com.adobe.mediacore.metadata.AdvertisingMetadata;

import java.util.ArrayList;
import java.util.List;

public class CustomAdPolicySelector implements AdPolicySelector {

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

    public CustomAdPolicySelector(MediaPlayerItem mediaPlayerItem) {
        _mediaPlayerItem = mediaPlayerItem;
```

```
        _lastAdBreakPlayedTime = 0;

        if (mediaPlayerItem != null) {
            watchedPolicy = extractWatchedPolicy(mediaPlayerItem.getConfig());
        }
    }

    @Override
    public AdBreakPolicy selectPolicyForAdBreak(AdPolicyInfo adPolicyInfo) {
        if (shouldPlayAdBreaks() && adPolicyInfo.getAdBreakTimelineItems() != null) {

            AdBreakTimelineItem item = adPolicyInfo.getAdBreakTimelineItems().get(0);

            // This condition will remove the pre-roll ad from live stream after watching
            if (item.getTime() == 0 && _mediaPlayerItem.isLive()) {
                return AdBreakPolicy.REMOVE_AFTER_PLAY;
            }
            if (item.getTime() == 0) {
                return AdBreakPolicy.PLAY;
            }

            // This condition will remove every ad break that has been watched once.
            // Comment this section if you want to play watched ad breaks again.
            if (item.isWatched()) {
                return AdBreakPolicy.SKIP;
            }

            return AdBreakPolicy.REMOVE_AFTER_PLAY;
        }

        return AdBreakPolicy.SKIP;
    }

    @Override
    public List<AdBreakTimelineItem> selectAdBreaksToPlay(AdPolicyInfo adPolicyInfo) {

        if (shouldPlayAdBreaks()) {

          List<AdBreakTimelineItem> timelineItems = adPolicyInfo.getAdBreakTimelineItems();

            AdBreakTimelineItem item;
            List<AdBreakTimelineItem> selectedItems = new ArrayList<AdBreakTimelineItem>();


            if (timelineItems != null && timelineItems.size() > 0) {

                // Seek Forward Condition
                if (adPolicyInfo.getCurrentTime() <= adPolicyInfo.getSeekToTime()) {
                    item = timelineItems.get(0);

                    // Resume logic - This will be helpful in resuming the content
                    // from last saved playback session, and just play the pre-roll ad
                    if(adPolicyInfo.getCurrentTime() == 0) {
                        if(item.getTime() == 0 && !item.isWatched()) {
                            // comment this line if you just need to seek to the user's
                            // last known position without playing pre-roll ad. ZD#820
                            selectedItems.add(item);
                            return selectedItems;
                        }
                        else{
                            return null;
                        }
                    } else {
                        item = timelineItems.get(timelineItems.size()-1);
                        if (!item.isWatched()) {
                            selectedItems.add(item);
                            return selectedItems;
                        }
                    }
```

```
                // Seek backward condition
            } else if (adPolicyInfo.getCurrentTime() > adPolicyInfo.getSeekToTime()) {
                item = timelineItems.get(0);

                if(!item.isWatched()) {
                    selectedItems.add(item);
                    return selectedItems;
                } else {
                    return null;
                }
            }
        }
    }
    return null;
}

@Override
public AdPolicy selectPolicyForSeekIntoAd(AdPolicyInfo adPolicyInfo) {
    // Simple Ad Policy selector
    // if the first ad in the break was watched,
    // skip to the next add after the seek position
    // otherwise, play the ads in the break from the beginning

    List<AdBreakTimelineItem> timelineItems = adPolicyInfo.getAdBreakTimelineItems();
    if (timelineItems != null && timelineItems.size() > 0) {
        if (timelineItems.get(0).isWatched()) {
            return AdPolicy.SKIP_TO_NEXT_AD_IN_AD_BREAK;
        }
    }

    // Resume play from the next ad in the break
    return AdPolicy.PLAY_FROM_AD_BREAK_BEGIN;

}

@Override
public AdBreakWatchedPolicy selectWatchedPolicyForAdBreak(AdPolicyInfo adPolicyInfo) {
    return watchedPolicy;
}

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

private boolean shouldPlayAdBreaks() {
    long currentTime = System.currentTimeMillis();

    if (_lastAdBreakPlayedTime <= 0) {
        return true;
    }

    if (_lastAdBreakPlayedTime > 0 &&
      (currentTime - _lastAdBreakPlayedTime) > MIN_BREAK_INTERVAL) {
        return true;
    }

    // return false for not playing Ad if this
    // Ad occurs with 5 minutes of last Ad playback
    return false;
}

private AdBreakWatchedPolicy extractWatchedPolicy(MediaPlayerItemConfig config) {
    if (config != null) {
        AdvertisingMetadata metadata = config.getAdvertisingMetadata();
        if (metadata != null) {
            return metadata.getAdBreakWatchedPolicy();
        }
    }
```

```
        return AdBreakWatchedPolicy.WATCHED_ON_BEGIN;
    }
}
```

Save the video position and resume later

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

Dynamically inserted ads differ between user sessions, so saving the position **with** spliced ads refers to a different position in a future session. TVSDK provides methods to retrieve the playback position while ignoring spliced ads.

1. When the user quits a video, your application retrieves and saves the position in the video.

   Tip: *Ad durations are not included.*

   Ad breaks can vary in each session due to ad patterns, frequency capping, and so on. The current time of the video in one session might be different in a future session. When saving a position in the video, the application retrieves the local time, which you can save on the device or in a database on the server.

   For example, if the user is at the 20th minute of the video, and this position includes five minutes of ads, `getCurrentTime` will return 1200 seconds, while `getLocalTime` at this position will return 900 seconds.

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

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

2. Restore the user session when player activity resumes.

```
@Override
public void onResume() {
    super.onResume();
    ...
    prefs =
      PreferenceManager.getDefaultSharedPreferences(getActivity().getApplicationContext());

    if (prefs.getString(LAST_MEDIA_RESOURCE, "nil").
      equals(_contentInfo.toMediaResource().getUrl())) {
        _lastKnownLocalTime =
          prefs.getLong(LAST_LOCAL_TIME, 0); // get the last local time
                                             // saved in system preferences
        if(_lastKnownLocalTime > 0) {
            _shouldResumePlayback = true;
        }
    }
    ...
}
```

3. To resume the video at the same position:
   • To resume playing the video from the position that was saved from a previous session, use `seekToLocalTime`.

   Tip: *This method is called only with local time values. If the method is called with current time results, incorrect behavior occurs.*

   • To seek to the current time, use `seek`.
4. When your application receives the `onStatusChanged` status change event, seek to the saved local time.

```
private final MediaPlayer.PlaybackEventListener _playbackEventListener =
  new MediaPlayer.PlaybackEventListener() {
    @Override
    public void onPrepared() {
        ...
        if(_shouldResumePlayback){
            if(_lastKnownLocalTime >= 0) {
                 _mediaPlayer.seekToLocalTime(_lastKnownLocalTime);
            }
        }
        ...
    }
    ...
}
```

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

Partial Ad break insertion

You can enable a TV-like experience of being able to join in the middle of an ad, in live streams. The Partial Ad break feature allows you to mimic a TV-like experience where, if the client starts a live stream inside a midroll, it will start within that midroll. It is similar to switching to a TV channel and the commercials run seamlessly.

For example, If a user joins in the middle of a 90-second ad break (three 30-second ads), 10 seconds into the second ad (that is, at 40 seconds into the ad break), the following happens:

• The second ad is played for the remaining duration (20 sec) followed by the third ad.
• Ad trackers for the partially played ad (the second ad) are not fired. Only the tracker for the third ad is fired.

This behavior is not enabled by default. To enable this feature work in your app, do the following:

1. Disable the live prerolls, using AdvertisingMetadata class's method setEnableLivePreroll.

```
advertisingMetadata.setLivePreroll(false)
advertisingMetadata.setPreroll(false)
```

2. Switch ON the preference for Partial Ad-break Insertion. Use the new method setPartialAdBreakPref in MediaPlayer interface to switch this feature ON. Use getPartialAdBreakPref method to find the current state of this preference.

```
MediaPlayer mediaPlayer = new MediaPlayer(getActivity().getApplicationContext());
 mediaPlayer.setPartialAdBreakPref(true);
```

Just-in-Time Ad Resolving

Ad resolving and ad loading may cause an unacceptable delay for a user waiting for playback to start. The Lazy Ad Loading and Lazy Ad Resolving features can reduce this startup delay. Lazy Ad Resolving has changed significantly in the 3.0 version. In Lazy Ad loading prior to 3.0, ad resolution was broken into two steps, resolving just pre-rolls ads prior to the PREPARED status, and mid-rolls and post-rolls after the PREPARED status. This has changed and ad breaks are now resolved at a specified interval prior to the position of the ad break.

• Basic ad resolving and loading process:

1. TVSDK downloads a manifest (playlist) and *resolves* all of the ads.
2. TVSDK *loads* all of the ads and places them on the timeline.
3. TVSDK moves the player into the PREPARED status, and content playback begins.

The player uses the URLs in the manifest to obtain the ad content (creatives), ensures that the ad content is in a format that TVSDK can play, and TVSDK places the ads on the timeline. This basic process of resolving and loading ads can cause an unacceptably long delay for a user waiting to play their content, especially if the manifest contains several ad URLs.

• *Lazy Ad Loading*:

1. TVSDK downloads a playlist and *resolves* all of the ads.
2. TVSDK *loads* pre-roll ads, moves the player into the PREPARED status, and content playback begins.
3. TVSDK *loads* the remaining ads and puts them on the timeline as playback occurs.

This feature improves upon the basic process by putting the player into the PREPARED status before all ads are loaded.

• *Lazy Ad Resolving*:

1. TVSDK downloads the playlist.
2. TVSDK resolves and loads any pre-roll ads, moves the player into the PREPARED status, and content playback begins.
3. TVSDK resolves and each of the remaining ad breaks individually based on the following calculation:

   ```
   AdvertisingMetadata::getDelayAdLoadingTolerance() + PlayBufferTime::playBufferTime + the value defined in EXT-X-TARGETDURATION
   ```

   By default, for content with a 6 second Target duration, this will be 5.0 + 30.0 + 6.0 seconds (41 seconds)
4. If an ad break occurs within 10 seconds of the start position, it will get resolved along with pre-roll ads prior to the PREPARED status.

Important: Factors to consider with Lazy Ad Resolving:

- *Lazy Ad Resolving is only supported for VOD streams only with modes SERVER_MAP and MANIFEST_CUES.*
- *Lazy Ad Resolving is not enabled by default. If disabled, all ads are resolved on VOD streams before playback starts.*
- *Lazy Ad Resolving is incompatible with the Instant On feature. For more information about Instant On, see Instant On.*
- *With Lazy Ad Resolving, while seeking forward over an ad break, the closest ad break to the seek position will be resolved during the seek.*

- *With Lazy Ad Resolving, if multiple ad breaks exist at the same time (VMAP), they will be resolved at the same time.*
- *It is not recommended to reduce the value of setDelayAdLoadingTolerance() below the default value (5 seconds). Doing so could cause the player to "buffer" unnecessarily.*
- *Lazy Ad Resolving does not affect pre-roll ads.*
- *Lazy Ad Resolving is currently supported with the Auditude-Plugin. It is recommended to not set setDelayAdLoadingto true if you are using a custom resolver.*

Enable lazy ad resolving

You can enable or disable the Lazy Ad Resolving feature using the existing Lazy Ad Loading mechanism (Lazy Ad Resolving is disabled by default).

You can enable or disable Lazy Ad Resolving by calling `AdvertisingMetadata.setDelayLoading` with true or false.

- Use the Boolean *hasDelayAdLoading* and *setDelayAdLoading* methods in AdvertisingMetadata to control the timing of ad resolution and the placement of ads on the timeline:
  - If *hasDelayAdLoading* returns false, TVSDK waits until all ads are resolved and placed before transitioning to the PREPARED state.
  - If *hasDelayAdLoading* returns true, TVSDK resolves only the initial ads and transitions to the PREPARED state.
    - The remaining ads are resolved and placed during playback.
  - When *hasPreroll* or *hasLivePreroll* return false, TVSDK assumes that there is no preroll ad and starts the playback of the content immediately. These are default set to true.

APIs relevant to lazy ad resolution:

```
Class:    com.adobe.mediacore.metadata.AdvertisingMetadata
Methods:
    public final boolean hasDelayAdLoading() // Check if Lazy Ad Resolving enabled
    public final void setDelayAdLoading() // Enable or disable Lazy Ad Resolving
    public final void setDelayAdLoadingTolerance() // Sets the Lazy Ad Resolving Tolerance
level.  This value is added to the BufferControlParameters::playBufferTime and the content's
EXT-X-TARGETDURATION to determine when ad breaks should resolve
    public final double getDelayAdLoadingTolerance() // Gets the Lazy Ad Resolving Tolerance
level
    public final boolean hasPreroll() // Check for existence of pre-roll ads
    public final void setPreroll() // Set pre-roll true or false
    public final boolean hasLivePreroll() // Check for live pre-roll ads
    public final void setLivePreroll() // Set live pre-roll true or false


Class:     com.adobe.mediacore.timeline.TimelineMarker
Methods:
    public double getDuration() // Returns the current duration of the TimelineMarker.  This
will be zero if the ad break has not yet resolved.
    public Placement.Type getPlacementType() // Returns whether
```

To accurately reflect ads as cues on a scrub bar, listen for the *TimelineEvent* event and redraw the scrub bar every time you receive this event.

When Lazy Ad Resolving is enabled for VOD streams, all ad breaks are placed on the timeline, however, many of the ad breaks will not be resolved yet. The application can determine whether or not to draw these markers by checking the `TimelineMarker::getDuration()`. If the value is greater than zero, then the ads within the ad break have been resolved.

TVSDK dispatches this event when an ad break has been resolved, and also when the player transitions to the PREPARED status.

```
mediaPlayer.addEventListener(MediaPlayerEvent.TIMELINE_UPDATED, timelineUpdatedEventListener);
/**
* ...
*/
public void onTimelineUpdated(TimelineEvent event) {
for (PlaybackManagerEventListener listener : eventListeners) {
  listener.onUpdate(getLocalSeekRange(), event.getTimeline());
}
```

Upgrading from 2.5.x Lazy Ad Resolving to 3.0.0 Lazy Ad Resolving (API/Workflow changes):

The com.adobe.mediacore.timeline.TimelineMarker interface now contains a new method:

Placement.Type getPlacementType()

This method will return a placement type of Placement.Type.PRE_ROLL, Placement.Type.MID_ROLL or Placement.Type.POST_ROLL. If an ad break is unresolved, the `getDuration()`method on the TimelineMarker interface will return 0.

Note: *This interface not always cast into the type com.mediacore.timeline.advertising.AdBreakTimelineItem if it has not yet been resolved. It will be able to be casted if the `getDuration()` method of the TimelineMarker is greater than 0.*

Event changes:

`kEventAdResolutionComplete` is now depreciated and is now triggered immediately after the player enters the PREPARED status. Applications that previously only listened to this event to draw the scrub bar should change this to listen for `kEventTimelineUpdated` only. After individual ad breaks are resolved, a new `kEventTimelineUpdated` event will be dispatched.

Ad insertion metadata

To allow the ad resolver to work, ad providers, such as Adobe Primetime ad decisioning, require configuration values to enable your connection to the provider.

TVSDK includes the Primetime ad decisioninglibrary. For your content to include advertising from the Primetime ad decisioningserver, your application must provide the following required `AuditudeSettings` information:

- `mediaID`, which is a unique identifier for the video to be played.

  The publisher assigns the mediaID when submitting video content and ad information to the Adobe Primetime ad decisioning server. This ID is used by Primetime ad decisioningto retrieve related advertising information for the video from the server.

- (Optional)`defaultMediaId`, which specifies the ads that are served when the following conditions are met:
  - Your request to the ad server is invalid, or the content is incorrectly configured.
  - Primetime ad decisioningis experiencing delays in propagating the data.
  - One of the Primetime ad decisioning back-end processes is malfunctioning or is unavailable.

  Tip: *Adobe recommends using `defaultMediaId`.*

- Your `zoneID`, which is assigned by Adobe, identifies your company or website.

• The domain of your assigned ad server.
• Other targeting parameters.

You can include these parameters depending on your needs and the needs of the ad provider.

Set up ad insertion metadata

Use the helper class `AuditudeSettings`, which extends the `MetadataNode` class, to set up Adobe Primetime ad decisioning metadata.

Tip: *Adobe Primetime ad decisioning was previously known as Auditude.*

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

1. Build the `AuditudeSettings` instance.
```
AuditudeSettings auditudeSettings = new AuditudeSettings();
```
2. Set the Adobe Primetime ad decisioning `mediaID`, `zoneID`, `<ph conkeyref="phrases/primetime-sdk-name"/>`, and the optional targeting parameters.
```
auditudeSettings.setZoneId("yourZoneId");
auditudeSettings.setMediaId("yourVideoId");
auditudeSettings.setDefaultMediaId("defVideoId");
auditudeSettings.setDomain("yourAuditudeDomain");

// Optionally set user agent
auditudeSettings.setUserAgent("yourUserAgent");

Metadata targetingParameters = new Metadata();
targetingParameters.setValue("desired_param", "desired_value");
auditudeSettings.setTargetingParameters(targetingParameters);
```

Tip: *The media ID is consumed by TVSDK as a string, that is converted to an md5 value, and is used for the `u` value in the Primetime ad decisioning URL request. For example:*

*`http://ad.auditude.com/adserver?`**`u`**`=c76d04ee31c91c4ce5c8cee41006c97d`*
`&z=114100&l=20150206141527&of=1.4&tm=15&g=1000002`

3. Create a `MediaResource` instance by using the media stream URL and the previously created advertising metadata.
```
MediaResource mediaResource = new MediaResource(
"http://example.com/media/test_media.m3u8", MediaResource.Type.HLS, Metadata);
```
4. Load the `MediaResource` object through the `MediaPlayer.replaceCurrentResource` 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.CurrentItem` method.
6. (Optional) Query the `MediaPlayerItem` instance to see whether the stream is live, regardless of whether it has alternate audio tracks or the stream is protected.

   This information can help you prepare the UI for the playback. For example, if you know there are two audio tracks, you can include a UI control that toggles between these tracks.
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.

TVSDK now includes ads when your media plays.

Enable ads in full-event replay

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

For live content, TVSDK uses the metadata/cues in the manifest to determine where to place ads. However, sometimes live/linear content might resemble VOD content. For example, when live content completes, an `EXT-X-ENDLIST` tag is appended to the live manifest. For HLS, the `EXT-X-ENDLIST` tag means that the stream is a VOD stream. To correctly insert ads, TVSDK cannot automatically differentiate this stream from a typical VOD stream.

Your application must tell TVSDK whether the content is live or VOD by specifying the `AdSignalingMode`.

For a FER stream, the Adobe Primetime ad decisioning server should not provide the list of ad breaks that need to be inserted on the timeline before starting the playback. This is the typical process for VOD content. Instead, by specifying a different signaling mode, TVSDK reads all the cue points from the FER manifest and goes to the ad server for each cue point to request an ad break. This process is similar to live/DVR content.

Tip: *In addition to each request that is associated with a cue point, TVSDK makes an additional ad request for pre-roll ads.*

1. From an external source, such as vCMS, obtain the signaling mode that should be used.
2. Create the advertising-related metadata.
3. If the default behavior must be overwritten, specify the `AdSignalingMode` by using `AdvertisingMetadata.setSignalingMode`.

   The valid values are `DEFAULT`, `SERVER_MAP`, and `MANIFEST_CUES`.

   Important: *You must set the ad signaling mode before calling `prepareToPlay`. After TVSDK starts to resolve and place ads on the timeline, changes to the ad signaling mode are ignored. Set the mode when you create the `AuditudeSettings` object.*

4. Continue to playback.

```
MediaPlayer mediaPlayer =
  new MediaPlayer(getActivity.().getApplicationContext());

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

MediaPlayerItemConfig itemConfig =
  new MediaPlayerItemConfig(getActivity().getApplicationContext());
MediaResource mediaResource =
  new MediaResource("http://example.com/media/test_media.m3u8",
                    MediaResource.Type.HLS, Metadata);

mediaPlayer.addEventListener(MediaPlayerEvent.STATUS_CHANGED,
  new StatusChangeEventListener() {
    @Override
    public void onStatusChanged(MediaPlayerStatusChangeEvent event) {
        if (status == MediaPlayerStatus.INITIALIZED) {
            mediaPlayer.prepareToPlay();
        }
```

```
        else if( event.getStatus() == MediaPlayerStatus.PREPARED ) {
            // TVSDK is in the PREPARED state, so start the playback
            mediaPlayer.play();
        }
        else if( event.getStatus() == MediaPlayerStatus.COMPLETE ) {
            // playback has reached the end of stream ( ads included )
            // if we want to rewind we can call
            mediaPlayer.seek(mediaPlayer.getSeekableRange().getBegin());
        }
    }
});

mediaPlayer.replaceCurrentResource(mediaResource, itemConfig);
```

Ad signaling mode

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

The valid values are `DEFAULT`, `SERVER_MAP`, and `MANIFEST_CUES`.

The following table describes the effect of `AdSignalingMode` values for the various types of HLS streams:

	Default	Manifest cues	Ad server map
Video on Demand (VOD)	• 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

Companion banner ads

TVSDK supports companion banner ads, which are ads that accompany a linear ad and often remain on the page after the linear ad ends. Your application is responsible for displaying the companion banners that are provided with a linear ad.

Best practices for companion banner ads

When displaying companion ads, follow these recommendations:

• Attempt to present as many of a video ad's companion banner ads as will fit in your player's layout.
• Present a companion banner only if you have a location that matches the ad's specified height and width.

Important: *Do not resize the ad.*

• Begin presenting the companion banner(s) as soon as possible after the ad begins.
• Do not overlay the main ad/video container with companion banners.
• You can display companion banners after the ad ends.

The standard practice is to display each companion banner until you have a replacement for the ad.

Companion banner data

The content of an `AdAsset` describes a companion banner.

Each `AdAsset` provides information about displaying the asset.

Available information	Description
width	Width of the companion banner in pixels.
height	Height of the companion banner in pixels.
resource type	The resource type for this companion banner: • html: The data is in HTML code. • iframe: The data is an iframe URL (src).
static URL	Sometimes, the companion banner also has a `staticURL` that is a direct URL to the image or to a `.swf` (flash banner). If you do not want to use html or iframe, you can use a direct URL to an image or swf to display the banner in the Flash stage instead. In this case, you can use the `staticURL` to display the banner. ***Important:*** *You must check whether the static URL is a valid string, because this property might not always be available.*

Display banner ads

To display banner ads, you need to create banner instances and allow TVSDK to listen for ad-related events.

TVSDK provides a list of companion banner ads that are associated with a linear ad through the `AdPlaybackEventListener.onAdBreakStart` event.

Manifests can specify companion banner ads by:

- An HTML snippet
- The URL of an iFrame page
- The URL of a static image or an Adobe Flash SWF file

For each companion ad, TVSDK indicates which types are available for your application.

Add a listener for the `AdPlaybackEventListener.onAdBreakStart` event that does the following:

- Clears existing ads in the banner instance.
- Gets the list of companion ads from `Ad.getCompanionAssets`.
- If the list of companion ads is not empty, iterate over the list for banner instances.

  Each banner instance (an `AdAsset`) contains information, such as width, height, resource type (html, iframe, or static), and data that is required to display the companion banner.
- If a video ad has no companion ads booked with it, the list of companion assets contains no data for that video ad.
- To show a standalone display ad, add the logic to your script to run a normal DFP (DoubleClick for Publishers) display ad tag in the appropriate banner instance.
- Sends the banner information to a function on your page that displays the banners in an appropriate location.

This is usually a `div`, and your function uses the `div ID` to display the banner.

Clickable ads

TVSDK provides you with information so that you can act on click-through ads. As you create your player UI, you must decide how to respond when a user clicks on a clickable ad.

For TVSDK for Android, only linear ads are clickable.

Respond to clicks on ads

When a user clicks on an ad or a related button, your application must respond. TVSDK provides you with information about the destination URL for the click.

1. To set up an event listener for TVSDK, and provide the click-through information, register `AdClickedEventListener.onAdClicked`.

   When a user clicks on an ad or a related button, TVSDK dispatches this notification, including information about the destination for the click.
2. Monitor user interactions on clickable ads.
3. When the user touches or clicks the ad or button, to notify TVSDK, call `notifyClick` on the `MediaPlayerView`.
4. Listen for the `onAdClick(AdClickEvent event)` event from TVSDK.
5. To retrieve the click-through URL and related information, use the getter methods for the `AdClickEvent` instance.
6. Pause the video.

   For more information about pausing the video, see *Pause and resume playback*.
7. Use the click-through information to display the ad click-through URL and the related information.

   You can, for example, display the information in one of the following ways:

   • In your application, by opening the click-through URL in a browser.

   On desktop platforms, the video ad playback area is used to invoke click-through URLs at user clicks.

   • Redirect users 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. On these devices, a separate view such as a sponsor button, is used 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 AdStartedEventListener adStartedEventListener =
  new AdStartedEventListener() {
    @Override
    public void onAdStarted(AdPlaybackEvent adPlaybackEvent) {
        Ad ad = adPlaybackEvent.getAd();
        if (ad == null) {
            return;
        }

        _pubOverlay.startAd(adPlaybackEvent.getAdBreak(), ad);

        if (areClickableAdsEnabled() && ad.isClickable()) {
            _isClickableAdPlaying = true;
            _playerClickableAdFragment.show();
        }
    }
```

```
};

private AdCompletedEventListener adCompletedEventListener =
  new AdCompletedEventListener() {
    @Override
    public void onAdCompleted(AdPlaybackEvent adPlaybackEvent) {
        Ad ad = adPlaybackEvent.getAd();
        _pubOverlay.stopAd(adPlaybackEvent.getAdBreak(), ad);

        _isClickableAdPlaying = false;
        if (ad.isClickable()) {
            _playerClickableAdFragment.hide();
        }
    }
};

private AdClickedEventListener adClickedEventListener =
  new AdClickedEventListener() {
    @Override
    public void onAdClicked(AdClickEvent adClickEvent) {
        AdClick adClick = adClickEvent.getAdClick();
        Ad ad = adClickEvent.getAd();

        String url = adClick.getUrl();
        if (url == null || url.trim().equals("")) {
        } else {
            Uri uri = Uri.parse(url);
            Intent intent = new Intent(ACTION_VIEW, uri);
            try {
                startActivity(intent);
            } catch (Exception e) {
            }
        }
    }
};
```

Separate the clickable ad process

You should 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`.

   This fragment should call `notifyClick()` and will be responsible for video playback.

   ```
   public class PlayerFragment extends SherlockFragment {
       ...
       public void notifyAdClick () {
           _mediaPlayer.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 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 a user clicks an ad, your application should pause playback of the main video content.

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

```
@Override
public void onResume() {
    super.onResume();
    requestAudioFocus();
    if (_lastKnownStatus == MediaPlayerStatus.PAUSED) {
        _mediaPlayer.play();
    }
}
...

@Override
public void onPause() {
    super.onPause();
    if (_mediaPlayer != null) {
        if (_mediaPlayer.getStatus() == MediaPlayerStatus.PLAYING ||
```

```
          _mediaPlayer.getStatus() == MediaPlayerStatus.PAUSED) {
            _savedPlayerStatus = _mediaPlayer.getStatus();
            _lastKnownTime = _mediaPlayer.getCurrentTime();
        }
        if (_mediaPlayer.getStatus() == MediaPlayerStatus.PLAYING) {
            _mediaPlayer.pause();
            _lastKnownStatus = MediaPlayerStatus.PAUSED;
        }
    }
}
abandonAudioFocus();
```

Repackage incompatible ads using Adobe Creative Repackaging Service (CRS)

Some third-party ads (or creatives) cannot be stitched into the HTTP Live Streaming (HLS) content stream because their video format is incompatible with HLS. Primetime ad insertion and TVSDK can optionally attempt to repackage incompatible ads into compatible M3U8 videos.

Ads served from various third parties, such as an agency ad server, your inventory partner, or an ad network, are often delivered in incompatible formats, such as the progressive download MP4 format.

When TVSDK first encounters an incompatible ad, the player ignores the ad and issues a request to the creative repackaging service (CRS), which is part of the Primetime ad insertion back end, to repackage the ad into a compatible format. CRS attempts to generate multiple bit rate M3U8 renditions of the ad and stores these renditions on the Primetime Content Delivery Network (CDN). The next time TVSDK receives an ad response that points to that ad, the player uses the HLS-compatible M3U8 version from the CDN.

To activate this optional CRS feature, contact your Adobe representative.

Note: *For CRS Version 3.0 (and earlier) customers, beginning with CRS Version 3.1 the following changes have improved both security and performance:*

- *CRS 3.1 continues with `https:` if the content being repackaged uses `https:`. This reduces the potential for some players to present unsecure content.*
- *CRS 3.1 greatly minimizes network calls, improving video startup time.*

For more information about CRS, see *Creative Packaging Service (CRS)*.

Enable CRS in TVSDK applications

To enable CRS in your TVSDK applications, you must set the following information in your Auditude settings:

Enable CRS in `AuditudeSettings`.

```
...
auditudeSettings.setCreativeRepackagingEnabled(true);
auditudeSettings.setCreativeRepackagingFormat("application/x-mpegURL");
```

Ad fallback for VAST and VMAP ads

For Digital Video Ad Serving Template (VAST) ads (or creatives) that have the fallback rule enabled, TVSDK treats an ad with an invalid media type as an empty ad and attempts to use fallback ads in its place. You can configure some aspects of fallback behavior.

The VAST/Digital Video Multiple Ad Playlist (VMAP) specification states that for ads that have VAST fallback enabled, empty ads automatically trigger the use of fallback ads. When a VAST ad is empty, TVSDK looks for a valid HLS

media type replacement among the fallback ads. When a VAST ad in a wrapper has an invalid media type, TVSDK treats this ad as empty. You can configure whether TVSDK should do the same for ads inline in a VMAP. For more information about the VAST `fallbackOnNoAd` feature, see *Digital Video Ad Serving Template (VAST) 3.0*.

Note: Zero Length Ads *- When TVSDK encounters a VAST response that contains an ad of zero duration, or an ad break with no ads, it fires AD_BREAK_START / AD_BREAK_COMPLETE events for those zero-length ad breaks. This behavior applies only for VOD streams. TVSDK fires these events even when your app is using the SKIP ad policy.*

TVSDK does not fire AD_BREAK_START / AD_BREAK_COMPLETE events for Live streams, or when a user employs trickplay or seek to go past the zero-length ad.

Define fallback ad behavior for VMAP inline ads

You can enable fallback when a VMAP inline ad contains an invalid media type.

Set `setFallbackOnInvalidCreativeEnabled` to `true` to have VMAP fall back when the media type for a linear/inline ad is invalid for HLS.

The default value is `false`. If a linear ad fails because it is has an invalid media type, or because the ad cannot be repackaged, this flag allows Primetime ad decisioningto follow the same fallback behavior as if the ad was an empty VAST wrapper.

```
AuditudeSettings result = new AuditudeSettings();
result.setFallbackOnInvalidCreative(true);
```

Ad fallback behavior for VAST and VMAP

When Primetime ad decisioningencounters a VAST ad (creative) that is empty or that has a media type that is invalid for HLS, it evaluates the fallback ads to determine what to return.

In TVSDK, the only valid media type is `application/x-mpegURL` (M3U8).

When there are stand-alone fallback ads, the Primetime ad decisioningplug-in examines these ads in the following order and returns the first ad with a valid media type:

1. If repackaging is enabled, the first occurrence of an ad with an invalid media type is treated like other invalid media types.

   If repackaging fails, this process applies to additional occurrences of the ad.
2. If TVSDK finds no valid fallback ads, it returns the original ad with the invalid media type.
3. If a fallback ad with a valid MIME type is returned instead of the original ad, Primetime ad decisioningsends error code 403 to the VAST error URL, if available.
4. If a fallback ad is a wrapper and returns multiple ads, only ads with the correct media type are returned.

Important: *This behavior is always enabled for ads in VAST wrappers. For VAST ads inline in a VMAP, the behavior is disabled by default, but your application can enable it.*

Custom tags

Media streams can carry additional metadata in the form of tags in the playlist/manifest file, and this file indicates the placement of advertising. You can specify custom tag names and be notified when certain tags appear in the manifest file.

HLS content tags

Important: *This feature is not available for Safari on Apple computers, because TVSDK uses the video tag, rather than Flash or MSE, to play HLS content.*

TVSDK provides out-of-the-box support for specific `#EXT` advertising tags. Your application can use custom tags to enhance the advertising workflow or to support blackout scenarios. To support advanced workflows, TVSDK allows you to specify and subscribe to additional tags in the manifest. You can be notified when these tags appear in the manifest file.

Tip: *You can subscribe to custom tags both for VOD and live/linear streams.*

Limitation: *When HLS is played by using the Video tag in Safari, and not by using Flash Fallback, this feature will not be available in Safari.*

Using custom HLS tags

Here is an example of a customized VOD asset:

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

#EXT-X-ASSET:AID=10

#EXTINF:9.9766,
seg1.ts

#EXTINF:9.9766,
seg2.ts

#EXTINF:9.9766,
seg3.ts

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

#EXTINF:9.9766,
seg5.ts

#EXT-X-ENDLIST
```

Your application can set up the following scenarios:

- A notification when `#EXT-X-ASSET` tags, or any other set of custom tag names to which you have subscribed, exist in the file.
- Insert ads when an `#EXT-X-AD` tag, or any other custom tag name, is found in the stream.

You can subscribe to any of the following tags as custom tags:

- `EXT-PROGRAM-DATE-TIME`
- `EXT-X-START`
- `EXT-X-AD`
- `EXT-X-CUE`
- `EXT-X-ENDLIST`

You will be notified with a `TimedMetadata` event during the parsing of manifest files.

There are some advertising tags, such as `EXT-X-CUE`, to which you are already subscribed. These ad tags are also used by the default opportunity generator. You can specify which ad tags are used by the default opportunity generator by setting the `adTags` property.

Example of a customized VOD asset

Here is an example of a customized VOD asset:

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

#EXT-X-ASSET:AID=10

#EXTINF:9.9766,
seg1.ts

#EXTINF:9.9766,
seg2.ts

#EXTINF:9.9766,
seg3.ts

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

#EXTINF:9.9766,
seg5.ts

#EXT-X-ENDLIST
```

Your application could set up the following scenarios:

- A notification when `#EXT-X-ASSET` tags, or any other set of custom tag names to which you have subscribed, exist in the file.
- Insert ads when an `#EXT-X-AD` tag, or any other custom tag name, is found in the stream.

Config class methods for tags

You can globally configure custom tag names in TVSDK with the `MediaPlayerItemConfig` class.

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

`MediaPlayerItemConfig` exposes these methods to manage the custom tags:

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

`public final void setAdTags(String[] tags);`	Sets the list of ad tags that will be used by default opportunity generator.

Remember the following:

- The setter methods do not allow the tags parameter to contain null values.

  If encountered, TVSDK throws an `IllegalArgumentException`.

- The custom tag name must contain the `#` prefix.

  For example, `#EXT-X-ASSET` is a correct custom tag name, but `EXT-X-ASSET` is incorrect.

- You cannot change the configuration after the media stream has been loaded.

Timed metadata class

When TVSDK detects a subscribed tag in the playlist/manifest, the player automatically tries to process and expose the tag in the form of a `TimedMetadata` object.

The class provides the following elements:

Property	Type	Description
`id`	long	Unique identifier of the timed metadata. This value is usually extracted from the cue/tag ID attribute. Otherwise, a unique random value is provided. Use `getId`.
`metadata`	Metadata	The processed/extracted information from the playlist/manifest custom tag. Use `getMetadata`.
`name`	String	The name of the timed metadata. If the type is `TAG`, the value represents the cue/tag name. If the type is `ID3`, it is null. Use `getName`.
`time`	long	The time position, in milliseconds, relative to the start of the main content where this timed metadata is present in the stream. Use `getTime`.
`type`	Type	The type of the timed metadata. Use `getType`. • TAG - indicates that the timed metadata was created from a tag in the playlist/manifest. • ID3 - indicates that the timed metadata was created from an ID3 tag in the media stream.

Remember the following:

- TVSDK automatically extracts the attributes list into key-value pairs and stores the attributes in the metadata property.

Tip: *Complex data in custom tags in the manifest, such as strings with special characters, must be in quotes. For example:*

```
#EXT-CUSTOM-TAG:type=SpliceOut,ID=1,time=71819.7222,duration=30.0,url=
"www.example.com:8090?parameter1=xyz&parameter2=abc"
```

- If the extraction fails because of a custom tag format, the metadata property will be empty and your application must extract the actual information. In this case, no error is thrown.

Element	Description
`public enum Type {TAG, ID3}`	Possible types for the timed metadata.
`public TimedMetadata(Type type, long time, long id, String name, Metadata metadata);`	Default constructor (time is the local stream time).
`public long getTime();`	The time position, relative to the start of the main content, where this metadata was inserted in the stream.
`public Metadata getMetadata();`	The metadata inserted in the stream.
`public Type getType();`	Returns the type of the timed metadata.
`public long getId();`	Returns the ID extracted from the cue/tag attributes. Otherwise, a unique random value is provided.
`public String getName();`	Returns the name of the cue, which is usually the HLS tag name.

Subscribe to custom tags

TVSDK prepares `TimedMetadata` objects for subscribed tags each time these objects are encountered in the content manifest.

Before the playback starts, you must subscribe to the tags.

To be notified about custom tags in HLS manifests:

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

Important: *You must include the # prefix when working with HLS streams.*

For example:

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

Add listeners for timed metadata notifications

To receive notifications about tags in the manifest, you need to implement the appropriate event listeners.

You can monitor timed metadata by listening for `onTimedMetadata`, which notify your application of related activity. Each time a unique subscribed tag is identified during parsing of the content, TVSDK prepares a new `TimedMetadata` object and dispatches this event. The object contains the name of the tag to which you subscribed, the local time in the playback where this tag will appear, and other data.

Listen for events.

```
private final TimedMetadataEventListener timedMetadataEventListener = new
TimedMetadataEventListener() {
    @Override
    public void onTimedMetadata(TimedMetadataEvent timedMetadataEvent) {
        TimedMetadata timedMetadata = timedMetadataEvent.getTimedMetadata();

        TimedMetadata.Type type = timedMetadata.getType();
        if (type.equals(TimedMetadata.Type.ID3)){
            Metadata metadata = timedMetadata.getMetadata();
            Set<String> keys = metadata.keySet();
            for (String key : keys) {
                String value = metadata.getValue(key);
            }
        } else if (_mediaPlayer.getPlaybackRange() != null &&
_mediaPlayer.getPlaybackRange().getDuration() > 0) {
            displayRanges();
        }
    }
};
```

ID3 metadata uses the same `onTimedMetadata` listener to indicate the presence of an ID3 tag. This should not cause any confusion, however, because you can use the `TimedMetadata type` property to differentiate between TAG and ID3. For more information about ID3 tags, see *ID3 tags*.

Store timed metadata objects as they are dispatched

Your application must use the appropriate `TimedMetadata` objects at the appropriate times.

During content parsing, which happens before playback, TVSDK identifies subscribed tags and notifies your application about these tags.

Tip: *The time that is associated with each `TimedMetadata` is the local time on the playback timeline.*

To store timed metadata objects as they are dispatched:

1. Keep track of the current playback time.
2. Match the current playback time to the dispatched `TimedMetadata` objects.
3. Use the `TimedMetadata` where the start time equals the current local playback time.

   The following example shows how to save `TimedMetadata` objects in an `ArrayList`.

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

VPAID 2.0 ad support

Video player ad-serving interface definition (VPAID) 2.0 provides a common interface to play video ads. It provides a rich media experience for users and allows publishers to better target ads, track ad impressions, and monetize video content.

The following features are supported:

- Version 2.0 of the VPAID specification

  For more information, refer to *IAB VPAID 2.0*.

- Linear VPAID ads with video-on-demand (VOD) content
- JavaScript VPAID ads

  VPAID ads must be JavaScript-based, and the ad response must identify the media type of the VPAID ad as `application/javascript`.

The following features are not supported:

- Version 1.0 of the VPAID specification
- Skippable ads
- Nonlinear ads, such as overlay ads, dynamic companion ads, minimizable ads, collapsible ads, and expandable ads
- Preloading VPAID ads
- VPAID ads in live content
- Flash VPAID ads

API

The following API elements support VPAID 2.0 ads:

- The `getCustomAdView` method of `MediaPlayer` returns a `CustomAdView` object, representing the web view that renders the VPAID ad (see *API References*).
- `MediaPlayer.setCustomAdTimeout(int milliseconds)` sets the timeout on the VPAID loading process. The default timeout value is 10 seconds.

While the VPAID ad is playing:

- The VPAID ad is displayed in a view container above the player view, so code that relies on taps by users on the player view does not work.
- Calls to `pause` and `play` on the player instance pause and resume the VPAID ad.
- VPAID ads do not have a predefined duration, because the ad can be interactive.

  The ad duration and total ad break duration that are specified in the ad server response might not be accurate.

Implement VPAID 2.0 integration

To add VPAID 2.0 support, add a custom ad view and appropriate listeners.

To add VPAID 2.0 support:

1. Add the custom ad view to the player interface when the player is in the PREPARED state.

```
...
private FrameLayout _playerFrame;
```

```
    ...
    case PREPARED:
        ...
        addCustomView();
...
private void addCustomView() {
    ...
    WebView view = (WebView)_mediaPlayer.getCustomAdView();
    ...
    _playerFrame.addView(view);
```

2. Create listeners and process the events described in *Events*.

Important: *In a VPAID 2.0 workflow, for custom ad views it is very important to maintain your `CustomAdView` instance across `AdBreak` starts (event `AD_BREAK_START`) and `AdBreak` completes (event `AD_BREAK_COMPLETE`), from the time you create the custom ad view through to when you dispose of it. That is, do not create a custom ad view on every ad break start and dispose of it on every ad break complete.*

In addition, you should only create your custom ad view when your player is in the PREPARED state,

Only dispose of the custom ad view when reset is called. For example:

```
// on reset
if (_mediaPlayer != null) {
    _mediaPlayer.disposeCustomAdView();
    ...
}
```

Finally, before you dispose of your custom ad view, you must remove it from the `FrameLayout`. For example:

```
if (_playerFrame != null)
    _playerFrame.removeAllViews();
```

.

Ad measurements from Moat

TVSDK takes information from FreeWheel and other ad servers providing VAST responses. FreeWheel provides, within VAST responses, information from the Moat service. The Moat service counts ad impressions with an accuracy that better shows whether creatives capture or neglect an audience's interests.

Moat is a service measuring ad viewing across many uses, from browsers to within applications. Moat generates marketing analytics data in real-time across multiple platforms.

The VAST response XML has a property and an element your code can read, the outermost `Ad id` property and the outermost `Extension` element. Either way, your code can use TVSDK to save both the `Ad id` information and the `Extension` information, then organize the information in a tree structure. With this organization, your code can pick up the data from any level and pass it along to wherever it needs to go. The value of the outermost `Ad id` property enables your code to coordinate information from the associated campaign.

For example, FreeWheel can return data in an Extensions element. Below is a sample element.

```
<?xml version="1.0"?>
<Extensions>
  <Extension type="FreeWheel">
    <Parameter name="moat">
      <MeasurementInfo renditionID="6398737" type="MediaFile">

<MoatID><![CDATA[169843;56705;17860255;17860316;2509639;g8912342;103311138;g436558;530633]]></MoatID>

      </MeasurementInfo>
      <MeasurementInfo renditionID="6398739" type="MediaFile">
```

```
<MoatID><![CDATA[169843;56705;17860255;17860316;2509639;g8912342;103311138;g436558;530633]]></MoatID>

      </MeasurementInfo>
    </Parameter>
  </Extension>
</Extensions>
```

Freewheel can also set the `id` property in the `Ad` element, as shown in the sample below.

```
<Ad id="118566" sequence="1">
```

For API information, see the API documentation for the class `NetworkAdInfo`.

Add custom ad markers

By using 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, for example, from a live event, and the result of the recording is one HLS stream. The recording contains the main content and advertising-related content in one HLS video-on-demand (VOD) stream. The recording process does not keep track of the ad-related segments, so the information that is related to the positioning of the ads in the main content is lost.

You might be able to obtain the information that is related to the positioning of the ad-content periods from other out-of-band sources, such as external CMS systems. You can 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 TVSDK, such as when ads are resolved by Adobe Primetime ad decisioning. However, TVSDK provides the following abstractions that define the way ad-related content is represented on the timeline:

- The ad break

  An ad break is an ordered list of individual consecutive ads.

- An individual ad

Playback events are triggered separately for ad breaks and ads at the start and end point for each ad.

TVSDK dispatches ad tracking events to your application, so you can implement your own tracking logic. If you set custom ad markers, you receive the `onAdBreakStart`, `onAdStart`, `onAdProgress`, `onAdComplete`, and `onAdBreakComplete` events.

TimeRange class

Custom ad markers allow you to pass a set of `TimeRange` specifications that represent timeline segments to TVSDK.

Each `TimeRange` specification in the set represents a segment on the playback timeline that is maintained internally by TVSDK and that must be appropriately marked as an ad-related period.

The `TimeRange` class is a simple data structure that exposes the start position and the end position on the timeline. These two read-only properties abstract the idea of a time range in the playback timeline.

Tip: *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.

TVSDK provides the means to load and prepare content for playback by using the `replaceCurrentResource` method in `MediaPlayer`. This method takes two arguments, an instance of `MediaPlayerResource` and, optionally, an instance of `MediaPlayerItemConfig`, which you can use to pass application-defined custom parameters.

- For more details see *Reuse or remove a MediaPlayer instance*.
- For details of `MediaPlayerResource`, see *Create a media resource*

ReplaceTimeRange class

The `ReplaceTimeRange` utility class is an extension of the `TimeRange` class to be used with `CustomRangeMetadata`.

```
public class ReplaceTimeRange extends TimeRange {
    // Default constructor method
    public ReplaceTimeRange() {
        ...
    }

    // Details of begining, duration and replaceDuration
    // provided at construction time
    public ReplaceTimeRange(long begin, long duration, long replaceDuration) {
        ...
    }

    // Replace duration
    public long getReplaceDuration() {
        ...
    }
}
```

Placing custom ad markers on the timeline

This example shows the recommended way to include custom ad markers on the playback timeline.

1. Translate the out-of-band ad-positioning information into a list/array of `RepaceTimeRange` class.
2. Create an instance of `CustomRangeMetadata` class, and use its `setTimeRangeList` method with the list/array as its argument to set its time range list.
3. Use its `setType` method to set the type to `MARK_RANGE`.
4. Use the `MediaPlayerItemConfig.setCustomRangeMetadata` method with the `CustomRangeMetadata` instance as its argument to set the custom range metadata.
5. Use the `MediaPlayer.replaceCurrentResource` method with the `MediaPlayerItemConfig` instance as its argument to set make the new resource the current one.
6. Wait for a `STATE_CHANGED` event, which reports that the player is in the `PREPARED` state.
7. Start video playback by calling `MediaPlayer.play`.

Here is the result of completing the tasks in this example:

- If a `ReplaceTimeRange` overlaps another on the playback timeline, for example, the start position of a `ReplaceTimeRange` is earlier than an already placed end position, TVSDK silently adjusts the start of the offending `ReplaceTimeRange` to avoid the conflict.

  This makes the adjusted `ReplaceTimeRange` shorter than originally specified. If the adjustment leads to a duration of zero, TVSDK silently drops the offending `ReplaceTimeRange`.

- TVSDK looks for adjacent time ranges for custom ad breaks and clusters them into separate ad breaks.

  Time ranges not adjacent to any other time range are translated into ad breaks that contain a single ad.

- If the application tries to load a media resource whose configuration contains `CustomRangeMetadata` that can be used only in the context custom ad markers, TVSDK throws an exception if the underlying asset is not of type VOD.
- When dealing with custom ad markers, TVSDK deactivates other ad-resolving mechanisms (for example, Adobe Primetime ad decisioning).

  You can use any TVSDK ad-resolver module or the custom ad markers mechanism. When you use custom ad markers, the ad content is considered resolved and is placed on the timeline.

The following code snippet places three time ranges on the timeline as custom ad-markers.

```
// Assume that the 3 time ranges are obtained through external means
// Use them to populate the ReplaceTimeRange instance
List<ReplaceTimeRange> timeRanges = new ArrayList<ReplaceTimeRange>();
timeRanges.add(new ReplaceTimeRange(0,10000, 0));
timeRanges.add(new ReplaceTimeRange(15000,20000, 0));
timeRanges.add(new ReplaceTimeRange(25000,30000, 0));

CustomRangeMetadata customRangeMetadata = new CustomRangeMetadata();
customRangeMetadata.setTimeRangeList(timeRanges);
customRangeMetadata.setType(CustomRangeMetadata.CustomRangeType.MARK_RANGE);

//Create a MediaResource instance
MediaResource mediaResource = MediaResource.createFromUrl(
        "www.example.com/video/test_video.m3u8", timeRanges.toMedatada(null));

// Create a MediaPlayerItemConfig instance
MediaPlayerItemConfig config =
  new MediaPlayerItemConfig(getActivity().getApplicationContext());

// Set customRangeMetadata
config.setCustomRangeMetadata(customRangeMetadata);

// Prepare the content for playback by calling replaceCurrentResource
// NOTE: mediaPlayer is an instance of a properly configured MediaPlayer
mediaPlayer.replaceCurrentResource(mediaResource, config);

// wait for TVSDK to reach the PREPARED state
mediaPlayer.addEventListener(MediaPlayerEvent.STATE_CHANGED,
  new StatusChangeEventListener() {
    @Override
    public void onStatusChanged(MediaPlayerStatusChangeEvent event) {

    if( event.getStatus() == MediaPlayerStatus.PREPARED ) {
        // TVSDK is in the PREPARED state, so start the playback
        mediaPlayer.play();
    }
    ...
}
```

Control playback behavior for seeking over custom ad markers

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

By default, when a user seeks into or past ad sections that result from the placement of custom ad markers, TVSDK skips the ads. This might differ from the current playback behavior for standard ad breaks. You can set TVSDK to reposition the playhead to the beginning of the most recently skipped custom ad when the user seeks past one or more custom ads.

1. Call `CustomRangeMetadata.setAdjustSeekPosition` with `true`.
```
customRangeMetadata.setAdjustSeekPosition (true);
```
2. Use `customRangeMetadata` in `MediaPlayerItemConfig`.
```
// Set customRangeMetadata
config.setCustomRangeMetadata(customRangeMetadata);

// prepare the content for playback by calling replaceCurrentResource
mediaPlayer.replaceCurrentResource(mediaResource, config);
```

Customize opportunity generators and content resolvers

An opportunity generator identifies placement opportunities by custom tags in a stream, ad signaling mode custom markers, and so on. The opportunity generator sends these placement opportunities to the content resolver, which customizes the content/ad insertion workflow based on the placement opportunity's properties and metadata.

TVSDK includes the following default opportunity generators:

- `ManifestCuesOpportunityGenerator` generates opportunities from the default ad cues (`#EXT-X-CUE`).
- `AdSignalingModeOpportunityGenerator` generates an initial opportunity for the specified ad signaling mode. This ignores any cues or timed metadata information.
- `CustomMarkerOpportunityGenerator` generates opportunities to replace baked-in C3 ads.
- `AuditudeResolver`'s opportunity generator produces opportunities when lazy ad resolving is on.

TVSDK also includes default content resolvers:

- `CustomRangeResolver`
- `JSONResolver`
- `AuditudeResolver`, which can communicate with Primetime ad decisioning.

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

- Recognize custom tags for ad insertion

  For more information, see *Customize opportunity generators and content resolvers*.
- Create a customized ad provider.
- Black out content.

Opportunity generators and content resolvers

TVSDK provides default opportunity generators and content resolvers that place ads in the timeline, and these generators and resolvers are based on nonstandard tags in the manifest. Your application might need to alter the timeline based on opportunities that are identified in the manifest, such as indicators for a blackout period.

An *opportunity* represents a point of interest on the timeline that usually indicates an ad placement opportunity. This opportunity can also indicate a custom operation that might affect the timeline, such as a blackout period. An *opportunity generator* identifies specific opportunities (tags) in the timeline and notifies TVSDK that these opportunities have been tagged. Opportunities are identified in a timeline in by including a nonstandard (non-HLS) tag.

When your application is notified about an opportunity, your application might alter the timeline by inserting a series of ads, by switching to an alternate stream (blackouts), or by otherwise editing the timeline content. By default, TVSDK calls the appropriate *content resolver* to implement the required timeline changes or actions. Your application can use the default TVSDK advertisement content resolver or register its own content resolver.

You can also use `MediaPlayerItemConfig.setAdTags` to add more ad marker tags/cues so that TVSDK can recognize and use `MediaPlayerItemConfig.subscribedTags` and notify your application about additional tags that might have advertising workflow information.

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

Implement a custom opportunity generator

You can implement your own opportunity generators by implementing the `OpportunityGenerator` class.

1. Implement your custom `ContentFactory` by implementing the `ContentFactory` interface and overriding `retrieveGenerators`.

   For example:

   ```
   class MyContentFactory extends ContentFactory {
       @Override
       public List<OpportunityGenerator> retrieveGenerators(MediaPlayerItem item) {
           List<OpportunityGenerator> generators = new ArrayList<OpportunityGenerator>();
           generators.add(MyOpportunityGenerator(item));
           return generators;
       }
       ...
   }
   ```

2. Register the `ContentFactory` to the `MediaPlayer`.

   For example:

   ```
   // register the custom content factory with media player
   MediaPlayerItemConfig config =  new MediaPlayerItemConfig();
   config.setAdvertisingFactory(new MyContentFactory());

   // this config will should be later passed while loading the resource
   mediaPlayer.replaceCurrentResource(resource, config);

   // OR use MediaPlayerItemLoader to pre-load a resource
   id = 23;
   itemLoader.load(resource, id, config);
   ```

3. Create a custom opportunity generator class that implements the `OpportunityGenerator` class.

   ```
   public class CustomOpportunityGenerator implements OpportunityGenerator
   {...}
   ```

   a) In the custom opportunity generator, override `doConfigure`, `doUpdate` and `doCleanup`:

   ```
   @Override
    public void configure(MediaPlayerItem item, Context context,
    OpportunityGeneratorClient client, AdSignalingMode mode, long playhead, TimeRange
   playbackRange) {
   }

   protected void update(long playhead, TimeRange playbackRange){
   }
   ```

```
protected void cleanup(){
}
```

To obtain the timed metadata:

```
List<TimedMetadata> tList = getItem().getTimedMetadata();
```

b) For each `TimedMetadata` or group of `TimedMetadata`, create an opportunity with the following attributes:

```
Opportunity(
  String id,                      // Can be id from timedMetadata
  Placement placementInformation, // Placement object containing Type, time, duration
  Metadata metadataSettings,      // Ad metadata with targeting params sent to the ad
provider
  Metadata customParams           // Metadata for customizing resolving and/or tracking
process.
);
```

c) For each opportunity created, call `resolve` on the `OpportunityGeneratorClient:getClient().resolve(opportunity);`.

This is a sample custom placement opportunity detector:

```
public class MyOpportunityGenerator implements OpportunityGenerator {


    @Override
     public void configure(MediaPlayerItem item, Context context,
     OpportunityGeneratorClient client, AdSignalingMode mode, long playhead, TimeRange
playbackRange) {
     }

       MediaPlayerItem item = getItem();
       MediaPlayerItemConfig itemConfig = item.getConfig();

       if (itemConfig == null || itemConfig.getAdvertisingMetadata() == null) {
           // no ad metadata, no ads
           return;
       }

       AdvertisingMetadata metadata = item.getConfig().getAdvertisingMetadata();


       AdSignalingMode mode = itemConfig.getAdSignalingMode();

       if (mode == AdSignalingMode.CUSTOM_RANGES)
       {
           // don't override custom ad ranges
           return;
       }

       Placement.Type pType = (mode == AdSignalingMode.MANIFEST_CUES) ?
                 Placement.Type.PRE_ROLL : Placement.Type.SERVER_MAP;
       Placement.Mode pMode = Placement.Mode.DEFAULT;
       Placement placement = new Placement(pType, playhead,
                 Placement.UNKNOWN_DURATION, pMode);

       Opportunity opportunity = new Opportunity("initialOpportunity", placement,
                 metadata, null);

       OpportunityGeneratorClient client = getClient();
       client.resolve(opportunity);
   }

   @Override
   protected void update(long playhead, TimeRange playbackRange) {

...
timedMetadataList = getItem().getTimedMetadata();
       for (TimedMetadata timedMetadata : timedMetadataList) {
```

```
        if (isOpportunity(timedMetadata)) {   // check if given timedMetadata should
                                              // be considered as an opportunity
  // create a PlacementOpportunity object and add it to the opportunities list
              Opportunity opportunity = new Opportunity("id", placement, metadata, null);
              client.resolve(opportunity)
          }
        }
    }

    @Override
    protected void cleanup() {}
}
```

Implement a custom content resolver

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

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

1. Implement your own custom `ContentFactory`, by extending the `ContentFactory` interface and overriding `retrieveResolvers`.

   For example:

   ```
   class MyContentFactory extends ContentFactory {
       @Override
       public List<ContentResolver> retrieveResolvers(MediaPlayerItem item) {
           List<ContentResolver> resolvers = new ArrayList<ContentResolver>();
           MediaPlayerItemConfig itemConfig = item.getConfig();
           if(itemConfig) {
               CustomRangeMetadata customRanges = itemConfig.getCustomRangeMetadata();
               if (customRanges) {
                   List<ReplaceTimeRange> timeRanges = customRanges.getTimeRangeList();

                   if (timeRanges && timeRanges.size() > 0)
                   {
                   // CustomRangeResolver is only activated by the presence of CustomRanges in
    configuration
                   resolvers.add(new CustomRangeResolver());
                   }
               }
               AdvertisingMetadata metadata = itemConfig.getAdvertisingMetadata();
               if (metadata) {
                   if (metadata instanceOf AuditudeSettings)
                       resolvers.add(new AuditudeResolver(getContext());
                   }
               }
           // add your custom resolver if any
           resolvers.add(MyOpportunityGenerator(item));
           return resolvers;
       }
       ...
   }
   ```

2. Register the `ContentFactory` to the `MediaPlayer`.

   For example:

   ```
   //Register the custom content factory with the media player
   MediaPlayerItemConfig config = new MediaPlayerItemConfig();
   config.setAdvertisingFactory(new MyContentFactory());

   //Pass this config while loading the resource
   mediaPlayer.replaceCurrentResource(resource, config);
   ```

```
// OR use MediaPlayerItemLoader to pre-load a resource
id = 23;
itemLoader.load(resource, id, config);
```

3. Pass an `AdvertisingMetadata` object to TVSDK as follows:
   a) Create an `AdvertisingMetadata` object.
   b) Save the `AdvertisingMetadata` object to `MediaPlayerItemConfig`.

```
AdvertisingMetadata advertisingMetadata = new AdvertisingMetadata();

advertisingMetadata.setDelayAdLoading(true);
...

mediaPlayerItemConfig.setAdvertisingMetadata(advertisingMetadata);
```

4. Create a custom ad resolver class that extends the `ContentResolver` class.
   a) In the custom ad resolver, override `doConfigure`, `doCanResolve`, `doResolve`, `doCleanup`:

```
void doConfigure(MediaPlayerItem item);
boolean doCanResolve(Opportunity opportunity);
void doResolve(Opportunity opportunity);
void doCleanup();
```

   You get your `advertisingMetadata` from the item passed in `doConfigure`:

```
MediaPlayerItemConfig itemConfig = item.getConfig();

AdvertisingMetadata advertisingMetadata =
  mediaPlayerItemConfig.getAdvertisingMetadata();
```

   b) For each placement opportunity, create a `List<TimelineOperation>`.

   This sample `TimelineOperation` provides a structure for `AdBreakPlacement`:

```
AdBreakPlacement(
    new AdBreak( ads,    // Vector<Ad>
                 tracker // Content Tracker
    ),
    placementInformation // Retrieved from Opportunity
);
```

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

   • If the ad resolve succeeds, call `process(List<TimelineOperation> proposals)` and `notifyCompleted(Opportunity opportunity)` on the `ContentResolverClient`

```
_client.process(timelineOperations);
_client.notifyCompleted(opportunity);
```

   • If the ad resolve fails, call `notifyResolveError` on the `ContentResolverClient`

```
_client.notifyFailed(Opportunity opportunity, PSDKErrorCode error);
```

   For example:

```
_client.notifyFailed(opportunity, UNSUPPORTED_OPERATION);
```

This sample custom ad resolver resolves an opportunity and serves a simple ad:

```
public class CustomContentResolver extends ContentResolver {
    protected void doConfigure(MediaPlayerItem item){}

    protected boolean doCanResolve(Opportunity opportunity) {
        return true;
    }

    protected void doResolve(Opportunity opportunity) {
```

```
        _client.process(createAdBreakPlacementsFor(opportunity.getPlacement()));
        _client.notifyCompleted(opportunity);
    }

   private List<TimelineOperation> createAdBreakPlacementsFor(Placement placementInformation)
{
        List<Ad> ads = new ArrayList<Ad>();
        AdAsset adAsset = new AdAsset("101", 15000, new MediaResource(
          "http: . . ..m3u8", MediaResource.Type.HLS, null), null, null);

        Ad ad = Ad.linearFromAsset("101", adAsset, null, null, false);
        ads.add(ad);
        AdBreak adBreak = new AdBreak(ads, null, AdInsertionType.CLIENT_INSERTED);

        List<TimelineOperation> result = new ArrayList<TimelineOperation>();

        result.add(new AdBreakPlacement(placementInformation, adBreak));
        return result;
    }

    protected void doCleanup() {}
}
```

Delete and replace ads in VOD streams

TVSDK supports the programmatic deleting and replacing of ad content in VOD streams.

The delete and replace feature extends the custom ad markers feature. Custom ad markers mark sections of the main content as ad-related content periods. In addition to marking these time ranges, you can also delete and replace time ranges.

Custom time range operations

The `CustomRangeMetadata` class identifies different types of time ranges in a VOD stream: mark, delete, and replace. For each of these custom time range types, you can perform corresponding operations, including deleting and replacing ad content.

For ad deletion and replacement, TVSDK uses the following *custom time range operation* modes:

- **MARK**

  This mode was referred to as custom ad markers in previous versions of TVSDK. The mode marks the beginning and ending times for ads that are already placed into the VOD stream. When there are time range markers of type `MARK` in the stream, an initial placement of `Mode.MARK` is generated by `CustomMarkerOpportunityGenerator` and resolved by `CustomRangeResolver`. No ads are inserted.

- **DELETE**

  For `DELETE` time ranges, an initial `placementInformation` of type `Mode.DELETE` is created and resolved by `CustomRangeResolver`. `DeleteRangeTimelineOperation` defines the ranges to be removed from the timeline, and TVSDK uses `removeByLocalTime` from the Adobe Video Engine (AVE) API to complete this operation. If there are DELETE ranges and Adobe Primetime ad decisioning metadata, the ranges are deleted first, then the `AuditudeResolver` resolves ads using the typical Adobe Primetime ad decisioning workflow.

- **REPLACE**

  For `REPLACE` time ranges, two initial `placementInformations` are created, one `Mode.DELETE` and one `Mode.REPLACE`. `CustomRangeResolver` deletes the time ranges first and then the `AuditudeResolver` inserts ads of the specified `replaceDuration` into the timeline. If no `replaceDuration` is specified, the server determines what to insert.

To support these custom time range operations, TVSDK provides the following:

- Multiple content resolvers

  A stream can have multiple content resolvers based on the ad signaling mode and ad metadata. The behavior changes with different combinations of ad signaling modes and ad metadata.

- Multiple initial opportunities using `CustomMarkerOpportunityGenerator`.
- A new ad signaling mode, `CUSTOM_RANGES`.

  Ads are placed based on Time Range data from an external source, such as a JSON file.

Content resolvers for ad deletion / replacement

You can use multiple content resolvers to handle different timeline operations.

```
public List<ContentResolver> retrieveResolvers(MediaPlayerItem item) {
    List<ContentResolver> resolvers = new ArrayList<ContentResolver>();
    MediaPlayerItemConfig itemConfig = item.getConfig();
    if(itemConfig) {
        CustomRangeMetadata customRanges = itemConfig.getCustomRangeMetadata();
        if (customRanges) {
            List<ReplaceTimeRange> timeRanges = customRanges.getTimeRangeList();

            if (timeRanges && timeRanges.size() > 0) {
                //CustomRangeResolver is activated by the presence of CustomRanges
                resolvers.add(new CustomRangeResolver());
            }
        }
        AdvertisingMetadata metadata = itemConfig.getAdvertisingMetadata();
        if (metadata) {
            if (metadata instanceOf AuditudeSettings)
            resolvers.add(new AuditudeResolver(getContext());

        }
    }
    //Add your custom resolver if any
    resolvers.add(MyOpportunityGenerator(item));
    return resolvers;
}
```

Effect on ad insertion and deletion from ad signaling mode and ad metadata combinations

You can mark, delete, and replace time ranges in VOD streams by using different ad signaling mode and ad metadata combinations. Different combinations of signaling mode and metadata result in different behaviors.

Tip: *When there is a conflict between time ranges and ad signaling modes, TVSDK gives the time ranges priority.*

The following table provides the details about the signaling mode and metadata combination behaviors:

Ad Signaling Mode	Ad Metadata	Resolvers Created	`PlacementInformations` created	Resulting behavior
Server Map				
	Delete	Delete	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)`	Ranges deleted

Ad Signaling Mode	Ad Metadata	Resolvers Created	`PlacementInformations` created	Resulting behavior
	Delete, Auditude	Delete, Auditude	• `PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)`, • `PlacementInfo (Type.SERVER_MAP, Mode.INSERT)`	Ranges deleted, Ads inserted
	Auditude	Auditude	`PlacementInfo (Type.SERVER_MAP, Mode.INSERT)`	Ads inserted
	Replace, Auditude	Delete, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE), PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.REPLACE)`	Ranges replaced
	Mark	CustomAd	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked
	Mark, Auditude	CustomAd, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked, no ads inserted
Manifest Cues				
	Auditude	Auditude	`PlacementInfo (Type.PRE_ROLL, Mode.INSERT)`	Ads inserted
	Delete, Auditude	Delete, Auditude	• `PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)` • `PlacementInfo (Type.PRE_ROLL, Mode.INSERT)`	Ranges deleted, ads inserted
	Mark, Auditude	Mark, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked, no ads inserted
	Delete	Delete	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)`	Ranges deleted
	Mark	CustomAd	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked

Ad Signaling Mode	Ad Metadata	Resolvers Created	`PlacementInformations` created	Resulting behavior
	Replace, Auditude	Delete, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE), PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.REPLACE)`	Ranges replaced
Custom Time Range				
	Delete	Delete	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)`	Ranges deleted
	Delete, Auditude	Delete, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)`	Ranges deleted, no ads inserted
	Auditude	Auditude	None	No ads inserted
	Replace, Auditude	Delete, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE), PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.REPLACE)`	Ranges replaced with ads
	Mark	CustomAd	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked
	Mark, Auditude	Custom Ad, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked, no ads inserted
Not set (default)				
	Delete	Delete	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE)`	Ranges deleted
	Delete, Auditude	Delete, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE), PlacementInfo (Type.SERVER_MAP, Mode.INSERT)`	Ranges deleted, ads inserted
	Auditude	Auditude	`PlacementInfo (Type.SERVER_MAP, Mode.INSERT)`	Ads inserted

Ad Signaling Mode	Ad Metadata	Resolvers Created	`PlacementInformations` created	Resulting behavior
	Replace, Auditude	Delete, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.DELETE), PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.REPLACE)`	Ranges replaced with ads
	Mark	CustomAd	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked
	Mark, Auditude	CustomAd, Auditude	`PlacementInfo (Type.CUSTOM_TIME_RANGE, Mode.MARK)`	Ranges marked

Use cases to delete and replace ads

Here are the use cases to delete and replace ads:
Mark ranges

You can designate time intervals in VOD content as ad breaks.

The `TimeRanges` between the `begin` and `end` in `localTime` will be marked as an `AdBreak` in the timeline. Other ad settings are ignored.

Tip: *If you want to only mark certain ranges in the content as ads, with no dynamic ad insertion, create a `CustomRangeMetadata` instance, and specify the type as a `MARK` operation with the defined custom ranges.*

Tp mark the ranges:

```
{
    "properties": [],
    "stream": {
        "manifests": [
            {
                "url":
"http://d398890tia84ty.cloudfront.net/e2e-vod/cloudfront_vod_hls_tos_30fps.m3u8",
                "type": "hls"
            }
        ],

    "metadata": {
        "time-ranges": {
                "type": "mark",
            "adjust-seek-position" : true,
                "time-range-list": [
                  {
                     "begin": 0,
                     "end": 15000
                  },
                {
                     "begin": 69000,
                     "end": 99000
                  },
                {
                    "begin": 251000,
                    "end": 281000
```

```
                    },
                  {
                      "begin": 514000,
                      "end": 544000
                    }
                ]

            }
        }
    },
    "title": "VOD - MARK TimeRanges and no ads",
    "thumbnail": {
        "large": "http://example.com",
        "small": "http://example.com"
       },
    "type": "vod",
    "id": "vod_004"
}
```

Replace time ranges with an ad

You can insert ads into VOD content.

The `TimeRanges` between the `begin` and `end` in `localTime` are removed from the timeline. These ranges are replaced by an `AdBreak` of `begin` to `begin+replaceDuration`. If the `replacement-duration` does not exist as a parameter, the server makes the determination on the returned `Adbreak`.

Tip: *You should always provide a `replacement-duration` for custom ranges. If no ads are intended to replace this custom range, provide a `replacement-duration` of 0.*

To replace the ranges with Primetime ad decisioningads:

```
{
    "properties": [],
    "stream": {
        "manifests": [
            {
                "url":
"http://d398890tia84ty.cloudfront.net/e2e-vod/cloudfront_vod_hls_tos_30fps.m3u8",
                "type": "hls"
            }
        ],
        "metadata": {
            "time-ranges": {
                "type": "replace",
                "time-range-list": [
                    {
                        "begin": 0,
                        "end": 15000,
                        "replacement-duration": 15000
                    },
                    {
                             "begin": 69000,
                             "end": 99000,
                             "replacement-duration": 30000
                    },
                    {
                        "begin": 251000,
                        "end": 281000,
                        "replacement-duration": 30000
                    },
                    {
                        "begin": 514000,
                        "end": 544000,
                        "replacement-duration": 30000
                    }
```

```
                ]
            },
            "ad": {
                "targeting": [
                    {
                        "value": "MulAdsAvail12346",
                        "key": "osmfKeyMulAdsAvail12346"
                    }
                ],
            "domain": "sandbox2.auditude.com",
            "mediaid": "psdk_000105",
            "zoneid": "121781"
            }
        }
    },
    "title": "VOD - Replace TimeRange with Auditude Ads",
    "thumbnail": {
        "large": "http://example.com",
        "small": "http://example.com"
    },
    "type": "vod",
    "id": "vod_003"
}
```

Delete ranges

You can remove `TimeRanges` between `begin` and `end` in `localTime` from the timeline.

Tip: *To only remove certain ranges from the content, create a `CustomRangeMetadata` instance and specify the type as a `DELETE` operation with the defined custom ranges.*

The ad map must be used as defined by the ad server.

To delete ranges with an Adobe Primetime ad decisioning ad:

```
{
    "properties": [],
    "stream": {
        "manifests": [
            {
                "url":
"http://d398890tia84ty.cloudfront.net/e2e-vod/cloudfront_vod_hls_tos_30fps.m3u8",
                "type": "hls"
            }
        ],
        "metadata": {
            "time-ranges": {
                "type": "delete",
                "time-range-list": [
                    {
                        "begin": 0,
                        "end": 20000
                    },
                    {
                        "begin": 69000,
                        "end": 99000
                    },
                    {
                        "begin": 251000,
                        "end": 281000
                    },
                    {
                        "begin": 514000,
                        "end": 544000
                    }
                ]
            },
```

```
            "ad": {
                "targeting": [
                    {
                        "value": "MulAdsAvail12346",
                        "key": "osmfKeyMulAdsAvail12346"
                    }
                ],
                "domain": "sandbox2.auditude.com",
                "mediaid": "psdk_000105",
                "zoneid": "121781"
            }
        }
    },
    "title": "VOD - DELETE TimeRange with xm-replace_text Phrase Ads",
    "thumbnail": {
        "large": "http://example.com",
        "small": "http://example.com"
    },
    "type": "vod",
    "id": "vod_003"
},
```

Examples of deleting and replacing ads

Here are some examples of the process to delete and replace ads.

Here is an example of using the `DELETE_RANGE`:

```
// Assume that the 3 timerange specs are obtained through external means,
// like a CMS. Assume mediaPlayer is an instance of a properly configured MediaPlayer
// Use these 3 timerange specs to populate the RepaceTimeRange list
List<ReplaceTimeRange> timeRanges = new ArrayList<ReplaceTimeRange>();
timeRanges.add(new ReplaceTimeRange(0,10000, 0));
timeRanges.add(new ReplaceTimeRange(15000,20000, 0));
timeRanges.add(new ReplaceTimeRange(25000,30000, o));

CustomRangeMetadata customRangeMetadata = new CustomRangeMetadata();
customRangeMetadata.setTimeRangeList(timeRanges);
customRangeMetadata.setType(CustomRangeMetadata.CustomRangeType.DELETE_RANGE);

// create a MediaResource instance
MediaResource mediaResource = ... ;

// create a MediaPlayerItemConfig instance
MediaPlayerItemConfig config =
  new MediaPlayerItemConfig(getActivity().getApplicationContext());

// Set customRangeMetadata
config.setCustomRangeMetadata(customRangeMetadata);

// prepare the content for playback by calling replaceCurrentResource
mediaPlayer.replaceCurrentResource(mediaResource, config);
```

Here is an example of using the `REPLACE_RANGE`:

```
// Assume that the 3 timerange specs are obtained through external means, like
//  a CMS. Assume mediaPlayer is an instance of a properly configured MediaPlayer
// Use these 3 timerange specs to populate the RepaceTimeRange list
List<ReplaceTimeRange> timeRanges = new ArrayList<ReplaceTimeRange>();
timeRanges.add(new ReplaceTimeRange(0,10000, 10000));
timeRanges.add(new ReplaceTimeRange(15000,20000, 20000));
timeRanges.add(new ReplaceTimeRange(25000,30000, 30000));

CustomRangeMetadata customRangeMetadata = new CustomRangeMetadata();
customRangeMetadata.setTimeRangeList(timeRanges);
customRangeMetadata.setType(CustomRangeMetadata.CustomRangeType.REPLACE_RANGE);

// create a MediaResource instance
```

```
MediaResource mediaResource = ... ;

// create a MediaPlayerItemConfig instance
MediaPlayerItemConfig config = new MediaPlayerItemConfig(getActivity()
    .getApplicationContext());

// Set Auditude settings, which are used for ad replacement, for this
//  MediaPlayerItemConfig instance,

...

// Set customRangeMetadata
config.setCustomRangeMetadata(customRangeMetadata);

// prepare the content for playback by calling replaceCurrentResource
mediaPlayer.replaceCurrentResource(mediaResource, config);
```

Ad deletion and replacement error handling

TVSDK handles time range errors according to the specific problem by merging or reordering the improperly defined time ranges.

TVSDK manages `timeRanges` errors through default merging and reordering processes. First, the player sorts customer-defined time ranges by the *begin* time. Based on this sorting order, if there are subsets and intersections among the ranges, TVSDK merges adjacent ranges and joins these ranges.

TVSDK handles time-range errors with the following options:

- **Out of order**

  TVSDK reorders the time ranges.

- **Subset**

  TVSDK merges the time-range subsets.

- **Intersect**

  TVSDK merges the intersecting time ranges.

- **Replace ranges conflict**

  TVSDK selects the replace duration from the earliest `timeRange` that appears in the conflicting group.

TVSDK handles signaling-mode conflicts with ad metadata in the following ways:

- If the ad signaling mode conflicts with the time-range metadata, the time-range metadata always has priority.

  For example, if the ad signaling mode is set as server map or manifest cues, and there are also MARK time rangesin the ad metadata, the resulting behavior is that the ranges are marked, and no ads are inserted.

- For REPLACE ranges, if the signaling mode is set as the server map or manifest cues, the ranges are replaced as specified in the REPLACE ranges, and there is no ad insertion through server map or manifest cues.

  For more information, see the *Signaling Mode / Metadata Combination Behaviors* table in *Effect on ad insertion and deletion from ad signaling mode and ad metadata combinations*.

Remember the following:

- When the server does not return valid `AdBreaks`, TVSDK generates and processes a `NOPTimelineOperation` for the empty AdBreak, and no ad plays.

• Although C3 ad delete/replacement is intended to be supported only for VOD, if specified in the ad metadata, time ranges are also processed for live streams.

Time range error examples

TVSDK responds to erroneous time range specifications by merging or replacing the time ranges as appropriate.

DELETE time range

In the following example, four intersecting DELETE time ranges are defined. TVSDK merges the four time ranges into one, so that the actual delete range is from 0-50s.

```
"time-ranges": {
    "type": "delete",
    "time-range-list": [
        {
            "begin": 10000,
            "end": 35000
        },
        {
            "begin": 20000,
            "end": 50000
        },
        {
            "begin": 0,
            "end": 30000
        },
        {
            "begin": 30000,
            "end": 40000
        }
    ]
}
```

REPLACE time range

In the following example, four REPLACE time ranges are defined with conflicting time ranges. In this case, TVSDK replaces 0-50s with 25s of ads. It goes with the first replacement duration in the sort order, because there are conflicts in subsequent ranges.

```
"time-ranges": {
    "type": "replace",
    "time-range-list": [
        {
            "begin": 10000,
            "end": 35000,
            "replace-duration": 15000
        },
        {
            "begin": 20000,
            "end": 50000,
            "replace-duration": 20000
        },
        {
            "begin": 0,
            "end": 30000,
            "replace-duration": 25000
        },
        {
            "begin": 30000,
            "end": 40000,
            "replace-duration": 30000
        }
    ]
}
```

Updating ad creative selection rules

You can use the TVSDK configuration file (`AdobeTVSDKConfig.json`) to update the priorities for ad creative selection on VAST/VMAP responses. You can also use this configuration file to define the source URL transformation rules for ad creatives.

When your video player makes a request to an ad server, the VAST/VMAP response usually includes multiple ad creatives (`MediaFile` elements), each of which provides a URL to a different container-codec version. In some cases, ad creatives in the VAST/VMAP response each provide a different bitrate for the ad. If you want to specify your own priority and transformation rules for these ad creatives, you can do so in the `AdobeTVSDKConfig.json` configuration file.

Important:

- *Do not change the name of the TVSDK configuration file. The name must remain `AdobeTVSDKConfig.json`.*
- *This file must be placed in the `assets/` folder of your project.*
- *Changing audio tracks when ad is playing does not change the audio track. A player should not allow users to change the audio track when an ad is playing.*

You can specify two types of rules in `AdobeTVSDKConfig.json`: *Priority* rules and *Normalize* rules.

Ad Rules change

The Ad rules are specified using a JSON file. The format of the JSON file remains the same in both versions of the TVSDK. However, in TVSDK v3.0, the Ad rules JSON file must be hosted on a location accessible via a HTTP URL. The application can use an instance of AuditudeSettings:

```
//TVSDK v3.0 AuditudeSettings result = new AuditudeSettings();
result.setCRSRulesJsonURL(<http url of
AdobeTVSDKConfig.json>);
```

Priority rules

The priority rule defines the priority order of the ad creatives that will be selected for playback from a VAST/VMAP response.

Table 1: A Priority rule has the following attributes and possible values:

Key	Type	Values	Description
`priority`	`Array`		An array of lowercase mime-types that define the priority in which source creatives must be selected for playback.
`item`	`String`	`host`	Currently only `host` is supported. This attribute must be present when `matches` and `values` attributes are defined.
`matches`	`String`	`multiple`	Possible values: • `eq` - equals • `ne` - not equals

Key	Type	Values	Description
			• `co` - contains • `nc` - not contains • `sw` - starts with • `ew` - ends with
`type`	`String`	`priority`	The value must always be `priority`
`values`	`Array`		TVSDK will use the `matches` attribute on the `item` of the source creative and match against the values defined in this array
`stream`	`String`		Value can be `vod` or `live`

```
{
    "ads": {
        "rules": {
            "default": [
                {
                    "type": "priority",
                    "stream": "vod",
                    "priority": [
                        "application/x-mpegurl",
                        "application/vnd.apple.mpegurl",
                        "application/x-shockwave-flash",
                        "video/mp4",
                        "video/m4v",
                        "video/x-flv",
                        "video/webm"
                    ]
                },
                {
                    ...
                },
            ]
        }
    }
}
```

Normalize rules

The normalize rule defines a URL transformation to apply to a source creative URL obtained from a VAST/VMAP response.

Table 2: The normalize rule has the following attributes and possible values:

Key	Type	Values	Description
`type`	`String`	`normalize`	The value must always be `normalize`.
`item`	`String`	`host`	Currently only `host` is supported. This attribute must be present when

Key	Type	Values	Description
			`matches` and `values` attributes are defined.
`matches`			Possible values: • `eq` - equals • `ne` - not equals • `co` - contains • `nc` - not contains • `sw` - starts with • `ew` - ends with
`values`	`Array`		TVSDK will use the `matches` attribute on the `item` of the source creative and match against the values defined in this array.
`find`	`regex`		A regular expression to apply on the source creative URL to match.
`replace`	`regex`		A regular expression to apply on the source creative URL to replace based on the match.

```
{
    "ads": {
        "rules": {
            "default": [
                {
                ...
                }
                {
                    "type": "normalize",
                    "item": "host",
                    "matches": "ew",
                    "values": [
                        "redirector.gvt1.com"
                    ],
                    "find": "videoplayback/(.*?)/expire/.*?/(.*?)/signature/.*?/",
                    "replace": "videoplayback/$1/expire//$2/signature//"
                }
            ]
        }
    }
}
```

Sample creative selection rules

In the `AdobeTVSDKConfig.json` you can specify default rules as well as rules for specific zones.

Sample default rules

The following is an example of an `AdobeTVSDKConfig.json` file that defines only default rules:

```
{
    "ads": {
        "rules": {
```

```
            "default": [
                {
                    "type": "priority",
                    "stream": "vod",
                    "priority": [
                        "application/x-mpegurl",
                        "application/vnd.apple.mpegurl",
                        "application/x-shockwave-flash",
                        "video/mp4",
                        "video/m4v",
                        "video/x-flv",
                        "video/webm"
                    ]
                },
                {
                    "type": "priority",
                    "stream": "live",
                    "priority": [
                        "application/x-mpegurl",
                        "application/vnd.apple.mpegurl",
                        "video/mp4",
                        "video/m4v",
                        "video/x-flv",
                        "video/webm"
                    ]
                },
                {
                    "type": "normalize",
                    "item": "host",
                    "matches": "ew",
                    "values": [
                        "redirector.gvt1.com"
                    ],
                    "find": "videoplayback/(.*?)/expire/.*?/(.*?)/signature/.*?/",
                    "replace": "videoplayback/$1/expire//$2/signature//"
                }
            ]
        }
    }
}
```

Sample default rules with additional zone rules

The following is an example of an `AdobeTVSDKConfig.json` file that defines default rules, plus additional rules for a specific zone ID (in this case, zone **"1234"**):

```
{
    "ads": {
        "rules": {
            "default": [
                {
                    "type": "priority",
                    "stream": "vod",
                    "priority": [
                        "application/x-mpegurl",
                        "application/vnd.apple.mpegurl",
                        "application/x-shockwave-flash",
                        "video/mp4",
                        "video/m4v",
                        "video/x-flv",
                        "video/webm"
                    ]
                },
                {
                    "type": "priority",
                    "stream": "live",
                    "priority": [
                        "application/x-mpegurl",
                        "application/vnd.apple.mpegurl",
```

```
                        "video/mp4",
                        "video/m4v",
                        "video/x-flv",
                        "video/webm"
                    ]
                },
                {
                    "type": "normalize",
                    "item": "host",
                    "matches": "ew",
                    "values": [
                        "redirector.gvt1.com"
                    ],
                    "find": "videoplayback/(.*?)/expire/.*?/(.*?)/signature/.*?/",
                    "replace": "videoplayback/$1/expire//$2/signature//"
                }
            ],
            "1234": [
                {
                    "type": "priority",
                    "matches": "nc",
                    "item": "host",
                    "values": [
                        "my.domain.com",
                        "a.bcd.com"
                    ],
                    "priority": [
                        "application/x-shockwave-flash",
                        "video/mp4",
                        "video/x-flv",
                        "video/quicktime",
                        "video/webm",
                        "application/x-mpegurl",
                        "application/vnd.apple.mpegurl",
                        "application/javascript"
                    ]
                }
            ]
        }
    }
}
```

Applying creative selection rules

TVSDK applies creative selection rules in the following ways:

- TVSDK applies all `default` rules first, followed by the zone-specific rules.
- TVSDK ignores any rules that are not defined for the current zone ID.
- Once TVSDK applies the default rules, the zone-specific rules can further change the creative priorities based on the `host` (domain) matches on the creative selected by the `default` rules.
- In the included sample rules file with additional zone rules, once TVSDK applies the `default` rules, if the M3U8 creative domain does not contain `my.domain.com` or `a.bcd.com` and the ad zone is `1234`, the creatives are re-ordered and the Flash VPAID creative is played first if available. Otherwise an MP4 ad is played, and so on down to JavaScript.
- If an ad creative is selected that TVSDK cannot play natively (`.mp4`, `.flv`, etc.), TVSDK issues a repackaging request.

Note that the ad types that can be handled by TVSDK are still defined through the `validMimeTypes` setting in `AuditudeSettings`.

Content protection

You can use the features of the Primetime Digital Rights Management (DRM) system to provide secure access to your video content. Alternatively, you can use third-party DRM solutions as an alternative to Adobe's integrated solution.

Contact your Adobe representative for the most up-to-date information on the availability of third-party DRM solutions.

Widevine DRM

You can use the Android native Widevine DRM with DASH streams.

Call the following `com.adobe.mediacore.drm.DRMManager` API before starting play:

```
public static void setProtectionData(
    String drm,
    String licenseServerURL,
    Map<String, String> requestProperties)
```

Arguments:

- `drm` - `"com.widevine.alpha"` for Widevine.
- `licenseServerURL` - The URL of the Widevine license server that receives license requests.
- `requestProperties` - Contains extra headers to include in the outgoing license request.

For example, when using content packaged for Expressplay DRM, use the following code before playing:

```
DRMManager.setProtectionData(
  "com.widevine.alpha",
  "https://wv.service.expressplay.com/hms/wv/rights/?ExpressPlayToken=token",
  null);
```

Primetime DRM interface overview

The key client-side element of the Primetime DRM solution is the DRM Manager. The sample application that is included with the Android SDK also includes a `DRMHelper` class that can be used to make certain DRM operations easier to implement.

Primetime DRM provides a scalable, efficient workflow to implement content protection in TVSDK applications. You protect and manage the rights to your video content by creating a license for each digital media file.

For more information, see the DRM sample player code that is included in the TVSDK package.

Here are the most important API elements for working with DRM:

- A reference in the media player to the DRM manager object that implements the DRM subsystem:

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

Tip: *This API will return a valid `DRMManager` object only after the `MediaPlayerEvent.DRM_METADATA` is fired. If you call `getDRMManager()` before this event fires, it might return NULL.*

- The `DRMHelper` helper class, which is useful when implementing DRM workflows.
- 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 DRM metadata and determine whether authentication is required.

```
/**
* 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.

For more information about DRM, see the *DRM documentation*.

DRM authentication before playback

When the DRM metadata for a video is separate from the media stream, you should authenticate before you beginning the 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"`

In this example, you can 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`.

   Tip: *This method is asynchronous and creates 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. Notify the user that this operation is asynchronous, it is a good idea to make the user aware of that.

   If users do not know that the operation is asynchronous, they might wonder why playback has not yet started. You can, 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.

```
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();
}
```

   Here are additional details about the handlers:

   • `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. After the loading is complete, inspect the `DRMMetadata` object to determine whether DRM authentication is required.

```
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. Complete one of the following tasks:

   • If authentication is not required, begin playback.

   • If authentication is required, complete 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);
```

In this example, for simplicity, the user's name and password are explicitly coded:

```
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(int major,
                                      int minor,
                                      String errorString,
                                      String serverErrorURL) {
        Log.e(LOG_TAG +
              "#onAuthenticationError",
              "DRM authentication failed. " +
              major + " 0x" + Long.toHexString(minor));
        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);
    }
});
```

6. Use an event listener to check the authentication status.

   This process implies network communication, so this is also an asynchronous operation.

```
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 major
    *       the major code.
    * @param minorC
    *       the minor code.
    * @param errorString
    *       the exception thrown.
    * @param serverErrorURL
    *       the URL of the server
    *       on which the error occurred
    */
    public void onAuthenticationError(int major,
                                      int minor,
                                      String errorString,
```

```
                                        String serverErrorURL);
}
```

7. If authentication is successful, start the playback.
8. 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 TVSDK in an error state, 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, you can perform authentication during playback.

With license rotation, an asset is encrypted with multiple DRM licenses. Each time that new DRM metadata is discovered, the `DRMHelper` methods are used to check whether the DRM metadata requires DRM authentication.

Tip: *Before starting playback, determine whether you are dealing with a domain bound license and whether domain authentication is required. If yes, complete the domain authentication and join the domain.*

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

```
mediaPlayer.addEventListener(MediaPlayerEvent.DRM_METADATA,
                             drmMetadataInfoEventListener);

DRMMetadataInfoEventListener drmMetadataInfoEventListener =
  new DRMMetadataInfoEventListener() {
    @Override
    public void onDRMMetadataInfo(DRMMetadataInfoEvent drmMetadataInfoEvent) {
        ...
    }
};
```

2. Use the `DRMMetadata` to check whether authentication is needed.
   - If authentication is not required, you do not need to do anything, and playback continues uninterrupted.
   - If authentication is required, complete DRM authentication.

   Since this operation is asynchronous and is handled in a different thread, it has no impact on the user interface nor on video playback.

3. If authentication fails, the user cannot continue viewing the video, and playback stops.

For example:

```
DRMMetadataInfoEventListener drmMetadataInfoEventListener =
  new DRMMetadataInfoEventListener() {
    @Override
    public void onDRMMetadataInfo(DRMMetadataInfoEvent drmMetadataInfoEvent) {
        final DRMMetadataInfo drmMetadataInfo =
          drmMetadataInfoEvent.getDRMMetadataInfo();

        if (drmMetadataInfo == null ||
          !DRMHelper.isAuthNeeded(drmMetadataInfo.getDRMMetadata())) {
            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) {
            return;
        }

        SharedPreferences sharedPreferences =
          PreferenceManager.getDefaultSharedPreferences(getActivity());
      String authUser = sharedPreferences.getString(PrimetimeReference.SETTINGS_DRM_USERNAME,

          getResources().getString(R.string.drmUsername));
      String authPass = sharedPreferences.getString(PrimetimeReference.SETTINGS_DRM_PASSWORD,

          getResources().getString(R.string.drmPassword));

        DRMHelper.performDrmAuthentication(drmManager, drmMetadataInfo.getDRMMetadata(),
          authUser, authPass, new DRMAuthenticationListener() {

            @Override
            public void onAuthenticationStart() {
                ...
            }

            @Override
            public void onAuthenticationError(int major,
                                              int minor,
                                              String erroString,
                                              String serverErrorURL) {
                if (getActivity() == null) {
                    return;
                }
                _handler.post(new Runnable() {
                    @Override
                    public void run() {
                        showToast(getString(R.string.drmAuthenticationError));
                        getActivity().finish();
                    }
                });
            }

            @Override
            public void onAuthenticationComplete(byte[] authenticationToken) {
            }

        });
    }
};
```

Video analytics

You can track video use by integrating TVSDK with Adobe Analytics.

Video tracking in TVSDK uses the **Adobe Analytics Video Essentials** service, which provides video engagement metrics, such as video views, video completes, ad impressions, time spent on video, and so on. For more information about this service, contact your Adobe representative.

The following procedure summarizes the steps to activate video tracking in your player:

1. Initialize and/or configure the following video tracking components:

   Tip: *In Android, these components are part of TVSDK.*

   - JSON configuration file
   - Video analytics metadata object
   - Global metadata object

2. Set up video analytics reporting on the server side by using Adobe Analytics Admin Tools.

Initialize and configure video analytics

You can configure your player to track and analyze video use.

Before activating video tracking (video heartbeats), ensure that you have the following:

• TVSDK 3.0 for Android.
• Configuration / Initialization Information

Contact your Adobe representative for your specific video-tracking account information:

`ADBMobileConfig.json`	***Important:*** *This JSON config file name must remain `ADBMobileConfig.json`. The name and the path of this configuration file cannot be changed. The path to this file must be `<source root>/assets`.*
AppMeasurement tracking server endpoint	The URL of the Adobe Analytics (formerly SiteCatalyst) back-end collection endpoint.
Video analytics tracking server endpoint	The URL of the video analytics back-end collection endpoint. This is where all video heartbeat tracking calls are sent. ***Tip:*** *The URL of the visitor tracking server is the same as the URL of the analytics tracking server. For information about implementing the Visitor ID Service, see Implement ID Service.*
Account name	Also known as the Report Suite ID (RSID).
Marketing Cloud organization ID	A string value required for instantiating the Visitor component.

To configure video tracking in your player:

1. Confirm that load-time options in the `ADBMobileConfig.json` resource file are correct.

```
{
    "version" : "1.1",
    "analytics" : {
        "rsids" : "adobedevelopment",
        "server" : "10.131.129.149:3000",
        "charset" : "UTF-8",
        "ssl" : false,
        "offlineEnabled" : false,
        "lifecycleTimeout" : 5,
        "batchLimit" : 50,
        "privacyDefault" : "optedin",
        "poi" : []
    },
    "marketingCloud": {
        "org": "ADOBE PROVIDED VALUE"
    },
    "target" : {
        "clientCode" : "",
        "timeout" : 5
    },
    "audienceManager" : {
```

```
        "server" : ""
    }
}
```

This JSON-formatted configuration file is bundled as a resource with TVSDK. Your player reads these values only at load time, and the values remain constant while your application runs.

To configure load-time options:

1. Confirm that the `ADBMobileConfig.json` file contains the appropriate values (provided by Adobe).
2. Confirm that this file is located in the `assets/` folder.

   This folder must be located in the root of your application source tree.
3. Compile and build your application.
4. Deploy and run the bundled application.

   For more information about these AppMeasurement settings, see *Measuring Video in Adobe Analytics*.

2. Initialize and configure video heartbeat tracking metadata.

Important: *You can stop the video analytics module midstream and reinitialize it again as necessary. Before the module is reinitialized, ensure that the video analytics metadata is also updated to the correct content metadata. To recreate the metadata, repeat the first two steps below (sub-steps **a** and **b**).*

a) Create an instance of the Video Analytics metadata.

   This instance contains all of the configuration information that is needed to enable video heartbeat tracking. For example:

```
private VideoAnalyticsMetadata getVideoAnalyticsTrackingMetadata() {
    VideoAnalyticsMetadata vaMetadata = new VideoAnalyticsMetadata();

    vaMetadata.setTrackingServer("example.com");
    vaMetadata.setChannel("test-channel");
    vaMetadata.setVideoName("myvideo");
    vaMetadata.setVideoId("myvideoid");
    vaMetadata.setPlayerName("PSDK Player");
    vaMetadata.setUseSSL(false);
    vaMetadata.debugLogging = true; // Set to NO for production deployment.
    vaMetadata.setEnableChapterTracking(true);
    // use this API to override the default asset length -1 for live streams
    vaMetadata.setAssetDuration(SAMPLE_ASSET_DURATION);

    return vaMetadata;
}
```

b) Initialize the Video Analytics provider.

   After creating a media player instance, you must create a Video Analytics provider instance and provide the application context to it.

Tip: *Always create a new provider instance for each content playback session and remove the previous reference after you detach the media player instance.*

```
VideoAnalyticsProvider videoAnalyticsProvider = new VideoAnalyticsProvider(appContext);
```

c) Set the Video Analytics metadata on the `videoAnalyticsProvider` instance.

```
videoAnalyticsProvider.setVideoAnalyticsMetadata(vaMetadata);
```

d) Attach the media player instance to the `videoAnalyticsProvider` instance:

```
videoAnalyticsProvider.attachMediaPlayer(mediaPlayer);
```

e) Destroy the Video Analytics provider.

Before you begin a new content playback session, destroy the previous instance of the video provider. After you receive the content complete event (or notification), wait a few minutes before you destroy the video analytics provider instance. Destroying the instance immediately might interfere with the Video Analytics provider's ability to send a "video complete" ping.

```
if (videoAnalyticsProvider) {
    videoAnalyticsProvider.detachMediaPlayer();
    videoAnalyticsProvider = null;
}
```

f) Manually mark the Live/Linear stream as complete.

If you have various episodes on one live stream, you can manually mark an episode as complete by using the complete API. This ends the video tracking session for the current video episode, and you can start a new tracking session for the next episode.

Tip: *This API is optional and does not work for VOD video tracking.*

```
if (videoAnalyticsProvider) {
    videoAnalyticsProvider.trackVideoComplete();
}
```

Implement custom metadata support

You can provide custom metadata on content, ads, and chapter tracking calls by using callback functions.

Callback functions are invoked just before the tracking call is made, so your application can attach the metadata that is specific to an ad or chapter.

Invoke callback functions for content, ads, and chapters.

```
// Video Metadata Block
// In a separate public class Implement an instance
// of VideoAnalyticsMetadata.VideoMetadataBlock

public class VideoMetadataBlockImpl
  implements VideoAnalyticsMetadata.VideoMetadataBlock {

    private final String video_id;
    private final String player_version;

    public VideoMetadataBlockImpl(String id, String version) {
        this.video_id = id == null ? "" : id;
        this.player_version = version == null ? "" : version;
    }
    @Override
    public HashMap<String, String> call() {
        HashMap<String, String> result = new HashMap<String, String>();
        result.put("videoid", video_id);
        result.put("mysdkversion", player_version);
        return result;
    }
}
// Create an instance of the above created
// public class and assign it to vaMetadata
vaMetadata.setVideoMetadataBlock(
  new VideoMetadataBlockImpl("1234", "1.2.3.4"));

// Ad Metadata Block that is invoked on every ad start
// In a separate public class Implement an instance of
// VideoAnalyticsMetadata.AdMetadataBlock

public class AdMetadataBlockImpl
```

```
  implements VideoAnalyticsMetadata.AdMetadataBlock {

    private final String ad_id;
    private final String ad_sdkversion;


    public AdMetadataBlockImpl(String id, String version) {
        this.ad_id = id == null ? "" : id;
        this.ad_sdkversion = version == null ? "" : version;
    }

    @Override
    public HashMap<String, String> call() {
        HashMap<String, String> result = new HashMap<String, String>();\
        result.put("myadid", ad_id);
        result.put("myad-sdkversion", ad_sdkversion);
        return result;
    }
}
// Create an instance of above created
// public class and assign it to vaMetadata
vaMetadata.setAdMetadataBlock(
  new AdMetadataBlockImpl("ad-1234", "1.2.3.4"));

// Chapter Metadata Block that is invoked on every chapter start
// In a separate public class Implement an instance of
// VideoAnalyticsMetadata.ChapterMetadataBlock

public class ChapterMetadataBlockImpl
  implements VideoAnalyticsMetadata.ChapterMetadataBlock {

    private final String chapter_id;
    private final String chapter_sdkversion;

    public ChapterMetadataBlockImpl(String id, String version) {

        this.chapter_id = id == null ? "" : id;
        this.chapter_sdkversion = version == null ? "" : version;
    }

    @Override
    public HashMap<String, String> call() {

        HashMap<String, String> result = new HashMap<String, String>();
        result.put("mychapterid", chapter_id);
        result.put("mychapter-sdkversion", chapter_sdkversion);
        return result;

        }
}
// Create an instance of above created public class and
// assign it to vaMetadata
vaMetadata.setChapterMetadataBlock(
  new ChapterMetadataBlockImpl("chapter-1234", "1.2.3.4"));
```

Implement chapter support

You can define and track *custom* chapters for video tracking in TVSDK-based applications.

Custom chapters are managed by the application, and are based on CMS data or on another way that the application uses to define chapters.

: Default chapters are not supported in the 3.0 Android TVSDK.

Define and track custom chapters.

```
// First, enable chapter tracking by setting
// enableChapterTracking to true:

vaMetadata.enableChapterTracking(true);
// For custom chapter definitions, provide
// an array list of chapters through the metadata.
// For example: 3 chapters of 60 second duration each

List<VideoAnalyticsChapterData> chapters =
  new ArrayList<VideoAnalyticsChapterData>();

Int chapterDuration = 60;
for (var i = 0; i < 3; i++) {
    VideoAnalyticsChapterData chapterData =
      new VideoAnalyticsChapterData(i * chapterDuration, (i + 1) * chapterDuration);
    chapterData.setName("chapter_" + (i+1));
    chapters.add(chapterData);
}

vaMetadata.setChapters(chapters);
```

Set up video analytics reporting on the server side

Your Adobe representative will handle most aspects of the server-side setup for Adobe Analytics reporting. For more information, see *Analytics Help and Reference - Report Suite Manager*.

1. To enable the conversion level for the Report Suite ID (RSID):
   a) Access **Admin Tools**.
   b) Select **Report Suites**.
   c) Select the RSID that you want to set up.
   d) Click **Edit Settings** > **General** > **General Account Settings**.
   e) In the **Conversion Level** combo box, select **Enabled, no Shopping Cart**.
   f) Click **Save**.
2. To enable video tracking:
   a) Access **Admin Tools**.
   b) Select **Report Suites**
   c) Select the RSID to set up.
   d) Click **Video Management** > **Edit Settings** > **Video Reporting**.
   e) Select the settings and click **Save**.

Access video analytics reports

Video analytics reports are routed to the Adobe Analytics reporting platforms.

For more information on Adobe Analytics set up, see the *Adobe Analytics* documentation.

1. Select the video-tracking enabled RSID.
2. Click **Video** > **Video Engagement** > **Video Overview**.
3. Select a video clip.

Events and notifications

Events and notifications help you manage the asynchronous aspects of the video application.

Notifications and events for player status, activity, errors, and logging

`MediaPlayerStatus` objects provide information about changes in player status. `Notification` objects provide information about warnings and errors. Errors that stop the playback of the video also cause a change in the status of the player. You implement event listeners to capture and respond to events (`MediaPlayerEvent` objects).

Your application can retrieve notification and status information. Using this information, you could also create a logging system for diagnostics and validation.

Notification content

`MediaPlayerNotification` provides information that is related to the player's status.

TVSDK provides a chronological list of `MediaPlayerNotification` notifications, and each notification contains the following information:

- A time stamp
- Diagnostic metadata that consists of the following elements:
  - `type`: INFO, WARN, or ERROR.
  - `code`: A numerical representation of the notification.
  - `name`: A human-readable description of the notification, such as SEEK_ERROR
  - `metadata`: Key/value pairs that contain relevant information about the notification. For example, a key named `URL` provides a value that is a URL related to the notification.
  - `innerNotification`: A reference to another `MediaPlayerNotification` object that directly impacts this notification.

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.

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

To receive notifications, listen for notifications as follows:

1. Implement the `NotificationEventListener.onNotification()` callback.
2. TVSDK passes a `NotificationEvent` object to the callback.
   Notifications types are enumerated in the `Notification.Type` enum:
   - `ERROR`
   - `INFO`
   - `WARNING`

Add real-time logging and debugging

You can use notifications to implement real-time logging in your video application.

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

Important: *The logging back end is not part of a production setup and is not expected to handle high-load traffic. If your implementation does not need to be absolutely complete, consider the efficiency of data transmission to avoid overloading your system.*

Here is an example of how to retrieve notifications:

1. Create a timer-based execution thread for your video application that periodically queries the data gathered by the TVSDK notification 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 this overflow, do one 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 latest notification event entries in JSON format and send the entries to a remote server for postprocessing.
   The remote server can 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.

Notification codes

The TVSDK notification system produces various error, warning, and informational notices that provide diagnostic metadata.

Notification objects provide information that is related to the player's status. TVSDK provides a chronologically sorted list of notification objects. Each notification contains the following metadata:

Element	Description
`type`	The notification type. Depending on the platform, this property is an enumerated type with possible values of INFO, WARN, and ERROR. This is the top-level grouping for notifications.
`code`	The following numerical representations are assigned to the notifications: • Error notification events, from 100000 to 199999 • Warning notification events, from 200000 to 299999 • Information notification events, from 300000 to 399999 Each top-level range, such as errors, is divided into subranges, such as 101000 through 101999 representing playback errors.
`name`	A string that contains a human-readable description of the notification event, such as `SEEK_ERROR`.
`metadata`	Key/value pairs that contain additional relevant information about the notification. For example, a key named `URL` would provide a value that is a URL related to the notification, such as an invalid URL that caused an error.

Element	Description
`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. Not all notifications provide an inner notification.

Details for the NATIVE_ERROR notification

When TVSDK handles a native error, it returns some or all of the following metadata key values as strings.

Metadata key name	Metadata value
`NATIVE_ERROR_CODE`	Native error code from the AVE. These codes represent the following: • DRM errors (codes 3300 to 3367). These are the same as the equivalent Flash Player error codes • Video playback errors (-1 to 89) • Cryptography errors (300 to 307)
`NATIVE_ERROR`	Short description of the notification (for example, `AAXS_InvalidVoucher` or `DECODER_FAILED`).
`DESCRIPTION`	Long description of the notification (for example, Ad resolving operation has failed).
`PSDK_ERROR_CODE`	`com.adobe.mediacore.PSDKErrorCode` numeric value as a string (for example, "13").
`PSDK_ERROR`	`com.adobe.mediacore.PSDKErrorCode` as a string (for example, `kECNetworkError`).
`WARNING`	Description of the warning.
`ERROR`	Description of the error.
DRM	
`NATIVE_SUBERROR_CODE`	Minor error from the DRM module.
`DRM_ERROR_STRING`	Description of the error.
`DRM_ERROR_SERVER_URL`	URL of the DRM server to which TVSDK tried to talk.
Ad manifest load failure	
`AD_URL`	URL of the content that failed to load.
`AD_TYPE`	Type of ad (a constant from the `MediaResource.Type` enum).
`AD_DURATION`	Ad duration in milliseconds.

Metadata key name	Metadata value
AD_ID	ID assigned to the ad.
File errors	
DOWNLOAD_ERROR	Description of the error during media file download.
URL	URL of file being downloaded.
MANIFEST_ERROR	Description of error during manifest file download.
CONTENT_ERROR	Description of error during fragment (for example, ts) download.
Audio track errors	
AUDIO_TRACK_NAME	Name of the audio track that failed to load, as specified in the manifest.
AUDIO_TRACK_LANGUAGE	Language of the audio track, as specified in the manifest.
Seek errors	
DESIRED_SEEK_PERIOD	ID of the period (integer).
DESIRED_SEEK_POSITION	Position (in milliseconds) sought (double).
Miscellaneous	
AUDITUDE_ERROR_CODE	Auditude error code (number).

NATIVE_ERROR: DRM values

The Video Encoder interface of the Adobe video engine returns these DRM notifications in the NATIVE_ERROR metadata object.

When reporting DRM errors to Adobe, ensure that you include the NATIVE_SUBERROR_CODE and DRM_ERROR_STRING for troubleshooting assistance.

Tip: *This list provides TVSDK-specific information about the errors. For complete descriptions, see ActionScript Run-Time Errors ActionScript Reference for the Adobe Flash Platform.*

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
3300	AAXS_InvalidVoucher	• What the distributor's software should do: • If you are using Google Chrome, and you are in Incognito mode, and your Flash Player version is less than 11.6, this error might occur. We recommend that the player check the browser's version number and advise the user to exit Incognito mode. • Request the license again.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		If the request is successful, you do not need to log or escalate. If the request is unsuccessful, log the content that caused the error. `subErrorId` contains a line error if present. • What the distributor should do: • If retries are unsuccessful on configurations other than Chrome with Flash less than version 11.6, a failure might have occurred in the packaging. • Check whether the issue is specific to certain content and repackage.
3301	`AAXS_AuthenticationFailed`	The server failed to authenticate or authorize the client. • The distributor's software should take any action necessary to re-establish user's credentials or guide the user to acquire access to the content. • The distributor should confirm that distributor's authorization and authentication mechanism is working correctly. If the distributors are not planning to use the authentication or authorization features, they should check whether the policy of the offending content requires authentication and see Diagnosing policy / license discrepancies. For more information about this error code, see *DRM error 3301 causes and resolution*.
3302	`AAXS_RequireSSL`	On Access 4.0 and above, this error is thrown on iOS when the remote key URL does not use HTTPS as the scheme. HTTPS is required. • If distributor is using a version older than Access v4, or the version at least 4 but the platform is not iOS, the distributor's software should log the error. The error is thrown only on iOS. • If the distributor's software is at least Adobe Access version 4, and the platform is iOS, distributors must change the remote key server URL that they are using to HTTPS. If they were only using HTTP, distributors might have to set up an HTTPS server. Otherwise, the distributors

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		need to submit the logged information to Adobe and escalate the issue.
3303	`AAXS_ContentExpired`	The content you are viewing has expired according to the rules set by the content provider. subErrorId contains a client-specific error or line error. • The distributor's software should attempt to reacquire license from the server once to determine whether a new non-expired license is available. If no license is available or the license has expired, allow the user to acquire new license, or inform user that the content cannot be watched.If the content has been packaged with a policy that has a lapsed expiration/end date, the license server logs report a `PolicyEvaluationException` and state that the Policy End Date has lapsed (Server Error code 303). Check the server's log files to verify. If possible, customers should check the policy that they used during packaging to see whether it has expired. The Java command line tool is: `java -jar libs/AdobePolicyManager.jar detail demo.pol` • The distributor should confirm whether license expiration dates are configured as intended. For more information about this error code, see *3303 (Content Expired) with AMS/FMS using a Live Stream?*.
3304	`AAXS_AuthorizationFailed`	For more information about this error code, see *DRM error 3301 causes and resolution*.
3305	`AAXS_ServerConnectionFailed`	The connection to the license or domain servers timed out, either due to network delay or the client being offline. Normally subErrorId contains HTTP return code. • The distributor's software should attempt a network connection to a known good server. If the attempt fails, prompt the user to reconnect to the network. If the attempt is successful, log it. • The distributor should verify that any license and domain servers in use are online and visible from the client's network.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		For more information about this error code, see *DRM 3305 [ServerConnectionFailed] causes and resolution*.
3306	`AAXS_ClientUpdateRequire`	Use a newer version of TVSDK for Android. The current client cannot complete the requested action, but an updated client might be able to complete the request. This can have several causes: • A shared domain was used that is not available on this client. This is likely the case when playback works on Chrome, but not any other browser and vice versa. ***Tip:*** *Chrome uses a different PHDS/PHLS key than the other browsers use. For more information, see https://adobeprimetime.zendesk.com/agent/tickets/2891.* • The application is trying to add multiple DRMSessions when running on an iOS version that is earlier than 5.0. • The metadata has a version of 3 or higher when only version 2 is supported. • The distributor's software should alert the user and abort the operation. If the software has a way of determining whether an upgrade is available, direct the user to that upgrade in the appropriate manner for the platform. • If the issue occurs because of a shared domain, the distributor will need to check with Adobe for an updated runtime or library. For Flash runtime, the distributor can force the upgrade in their application directly. In the case of a library, the distributor will need to obtain an updated library, rebuild their application and deploy it to their users. If the issue occurs because of multiple DRMSessions, the distributor will need to update their application to check the iOS version number prior to adding multiple DRMSessions. Or they can restrict distribution of their application to iOS v5 and above. if the issue occurs because the metadata version is higher than version 2, the issue is probably corrupted

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		metadata. They can try rebuilding the metadata and looking at the results. If they continue to see the problem, log the issue and escalate to Adobe. For more information about this error code, see *How to remedy a 3306 DRMErrorEvent Error Code*
3307	`AAXS_InternalFailure`	This generally represents a bug in Adobe Access code and is unexpected, unless there's a known bug, as below. subErrorId contains a client-specific error or line error. • If the browser is Chrome on Windows and Flash version is 11.6 (SWF version 19 or greater), the distributor's software should assume that the user pressed **Deny** on the infobar and treat the same as a 3368. • If 3307 occurs when browser is not Chrome or Flash version is not 11.6, the distributor should escalate to Adobe. ***Important:*** *`3307:1107296344 (FailedToGetBrokerHandle)` might happen with Chrome browser versions 24-28.*
3308	`AAXS_WrongLicenseKey`	This error is throw whenever the license being used contains the wrong key to decrypt the content. subErrorId contains a client-specific error or line error. There seem to only be two ways of generating this bug: • The customer has modified the standard Adobe tooling for generating licenses (for example, the licenser server Java framework). In this case, the license contains a bad key which might not correspond to any content. • The customer has issued multiple licenses with the same license ID. In this case, there are multiple licenses that are available on the client that match the content metadata and the Access code has selected the wrong one for use. • The distributor's software should attempt to reacquire license from the server.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		• If no license is available or license is expired, provide workflow for user to acquire new license, or inform user that the content cannot be watched and log the issue. • If this was a domain bound content (for AIR), provide a way for the user to join the domain. • The distributor should: • Verify that they have not customized the license issuance portions of the Access License server. • Verify that they are issuing unique license IDs for all licenses. • Escalate the issue with Adobe.
3309	`AAXS_CorruptedAdditionalHeader`	This will occur if the header is larger than 65536 bytes. • The distributor's software should Log which piece of content caused the error. • The distributor should confirm that error is reproducible with specific pieces of content. Repackage broken content.
3310	`AAXS_AppIDMismatch`	The Android application does not match the one in use. The correct AIR application or Flash SWF is not being used.
3311	`AAXS_AppVersionMismatch`	Not in use. This issue might still be generated by the version 1.x stack in AIR.
3312	`AAXS_LicenseIntegrity`	To fix this, redownload the license from the server.
3313	`AAXS_WriteMicrosafeFailed`	This issue occurs when the system cannot write to the file system. `subErrorId` contains a client-specific error or line error. On Microsoft Windows, error 3313 might be thrown by Active X or NPAPI plug flash player when the encrypted content has a licenseID or a policyID that is too long. This is because of the maximum path length in Windows. (Pepper plugin does not have this problem.) See watson 3549660 • The distributor's software should prompt the user to confirm that their user directory is not locked nor on a volume that is full or locked. • If the distributor is using AIR, rather than Flash, the issue might be caused by a path length limitation.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		Distributors should shorten the name of their AIR applicaiton to something reasonable. Also, publish contents again with a shorter `licenseID` and a `policyID`.
3314	`AAXS_CorruptedDRMMetadata`	This error often indicates that the content was packaged with test PKI certs, and the player is built with production PKI or vice versa. subErrorId contains a client-specific error or line error. • The distributor's software should log which piece of content caused the error. • The distributor should confirm that the error is reproducible with specific pieces of content. You might have to repackage broken content.
3315	`AAXS_PermissionDenied`	There are known bugs in which this error code is thrown when a 3305 is intended. For more information, see *DRM 3305 [ServerConnectionFailed] causes and resolution*. Remote SWF loaded by AIR is not allowed to access Flash Access functionality. This error code can also be thrown if a security error occurs during network access. Examples include the destination server does not the client to connect by using crossdomain.xml, or the crossdomain.xml is not reachable. For more information, see *DRM error 3315 possible root cause and resolution*.
3316	`AAXS_NOTUSED_MOVED`	Was `ADOBECPSHIM_MinorErr_MissingAdobeCPModule`. Moved to 3344 due to conflict with Flash error code.
3317	`AAXS_LoadAdobeCPFailed`	***Important:*** *This is a rare error and usually does not occur in a production environment.* If the error does occur, you can do one of the following: • If you are using AIR, reinstall it. • If you are using Flash Player, download the `AdobeCP` modules again.
3318	`AAXS_IncompatibleAdobeCPVersion`	Not applicable for Android.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
3319	`AAXS_MissingAdobeCPGetAPI`	Not applicable for Android.
3320	`AAXS_HostAuthenticateFailed`	Not applicable for Android.
3321	`AAXS_I15nFailed`	The process of provisioning the client with keys failed. subErrorId contains a client-specific, server-specific or line error. • The distributor's software should retry the operation at least once. If you are using Google Chrome on Windows, provide an explanation about how to allow plugin access that is not in a sandbox. For more information see *Google Chrome's unsandbox access denied*. • The distributor should complete one of the following tasks: • If the error is consistent across platforms, you should escalate the issue with Adobe. • If the error is confined to Chrome on Windows, guide the user to allow unsandboxed plugin access. Distributors should update their SWF to version 19 or later, and the Chrome-specific 3321 error, a 3368 error is thrown. Error 3368 can be handled more specifically by the distributor's software. This change was introduced in Chrome Stable channel version 26.0.1410.43. ***Tip:*** *Error `3321:1090519056` might happen with Flash Player versions 11.1 to 11.6. We recommend that you upgrade to the latest Flash Player version.* For more information, see *DRM error 3321 Causes & Resolution*.
Global Store corruption errors		
3322	`AAXS_DeviceBindingFailed`	The device does not appear to match the configuration that was present when initialized. subErrorId contains a client-specific or line error. The distributor's software should complete one of the following tasks:

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		• If the device is not using Flash Player, and is using AIR, iOS, and so on, call `DRMManager.resetDRMVouchers()`. If the issue occurs on iOS in a development phase, ask the developer to confirm whether the issue is observed when switching between builds that were downloaded from third-party, pre-release distribution systems (for example, HockeyApp) and a local build from Xcode. Attributes of a previous installation are not entirely overwritten when switching between a build distributed from HockeyApp and a build from Xcode. This situation might trigger the 3322 error. To resolve this issue, the developer should remove the older build from the device before installing the new build. • If the device is using Flash Player, and it is unusable from a 3322 or 3346 error codes, see the instructions from Adobe about how to programmatically reset your DRM license store on *DRM Error 3322/3346/3368 in Chrome (Info-Bar Problems)*. This error is not expected to occur frequently. In corporate environments that uses roaming profiles, if a user was viewing content that is protected by DRM, the chances error 3322 occurring increases as the user logs in from different machines. If possible, distributor should try to get this information from user. If the error occurs frequently, escalate to Adobe. You must notify Adobe whether resetting license store did (or did not) solve the problem and tell Adobe on which browsers the error is occurring. For more information, see the following articles: • *https://forums.adobe.com/message/5520902* • *https://forums.adobe.com/message/5535911* • *https://forums.adobe.com/message/5748618* • *https://forums.adobe.com/message/6061165*
3323	`AAXS_CorruptGlobalStateStore`	Files used by the DRM client have been modified unexpectedly. subErrorId contains a client-specific or line error.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		• The distributor's software should guide the user to reset in the same way as for 3322. • If the GlobalStore is failing at a rate greater than the expected failure rate of the hard drives of your user base, escalate the issue to Adobe.
3324	`AAXS_MachineTokenInvalid`	Reset DRM local storage for this application. Call DRMManager.resetDRM. The license server might not be able to connect to the Certificate Revocation List (CRL) server to refresh its CRL files, or the client machine is requesting a license/authentication that has been revoked by the license server. In the server logs, an error code 111 is MachineTokenInvalid. However, at the client level, error code 111 is translated to error code 3324. The DRM license server administrator should check whether the customer's license server has ever been able to retrieve the Adobe CRL files. If the customer is using Tomcat, the customer can check the `tomcat/temp/` directory to see whether there are 4 CRL files. • If the files are in this directory, double-click the files in Windows Explorer and in the CRL viewer application, determine whether any of the files have expired. • If there are no files in tomcat/temp/, then it can be assumed this license server has never been able to reach the Adobe CRL server due to a firewall/routing issue. For more information, see *Firewall rules*. If the CRL files are not available or have expired, you must confirm whether the license server can be reached. Open a network sniffer on the customer's license server, restart the server, and have a client attempt to request a license from the server. You can observe the network traffic to see whether calls to the following URL endpoints are successful: ***Tip:*** *You can also enter the following CRL URLs in a browser to see whether you can manually download each file.*

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		• *crl2.adobe.com/Adobe/FlashAccessIndividualizationCA.crl* • *crl2.adobe.com/Adobe/FlashAccessIntermediateCA.crl* • *crl2.adobe.com/Adobe/FlashAccessRootCA.crl* • *crl3.adobe.com/AdobeSystemsIncorporated FlashAccessRuntime/LatestCRL.crl* If the firewall rules are open and there are no current 3324 errors, there might have been a temporary network issue. Check the customer's server logs, which are probably in the `/tomcat/logs/` directory, to determine whether an error occurred when the license server tried to fetch the Certificate Revocation Lists. ***Important:*** *An error might occur when a large number (or a burst) of clients report a 3324 error to a temporary network issue when renewing a CRL file. When the network issue was resolved, the 3324 issues were also resolved.* If all 4 of the CRL files exist in the `tomcat/temp/` directory, and clients are still getting 3324 error codes, there might be file access issues to the CRL files. To resolve this issue, you might want to review the logs and purge the existing CRL files. If there are no server issues, prompt the user to reset in as described in 3322.
Server Store corruption errors		
3325	`AAXS_CorruptServerStateStore`	Files used by the DRM client have been modified unexpectedly. `subErrorId` contains a client-specific or line error. • The distributor's software should retry the operation again, because AdobeCP has deleted the offending server store internally, and a retry should succeed. If retry fails, log the issue. • If retries are failing at a rate greater than the expected failure rate of the hard drives of your user base, escalate the issue to Adobe.
3326	`AAXS_StoreTamperingDetected`	Call `DRMManager.resetDRM`. The License store has been tampered/corrupted and can no longer be used.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		The distributor's software should guide the user to reset in the same way as described in 3322.
3327	`AAXS_ClockTamperingDetected`	Fix the clock or acquire `Authn/Lic/Domain` license again.
Authentication/License/Domain server errors		
3328	`AAXS_ServerErrorTryAgain`	This is a server-side error where the server was unable to complete the request from the client. This error can occur when, for example, the server is busy, HTTP/500, the server does not have the needed key to decrypt the request, and so on. On the client, there is no way to determine what went wrong. The customer must review the Adobe Access server logs, which are usually called `AdobeFlashAccess.log`, to determine what went wrong. There is always a descriptive stack trace in the log to indicate the problem. `subErrorId` contains a server-specific or line error. The distributor should look at server logs to identify which server is sending this error. For 3328 errors that has a sub-error code 101, the server cannot decrypt the request. The customer must validate that the license / transport server certificates that are installed on the license server match and correspond with the certificates that is used during packaging. In addition, if customers are using the Reference Implementation, they must ensure that there are no typos in the `flashaccess-refimpl.properties` file where the primary and additional certificates are specified.
3329	`AAXS_ApplicationSpecificError`	The application-specific sub error code is not known to Flash Access. `subErrorId` contains a server-specific error from the publishers customized license server. The server returned an error in the application-specific namespace.
3330	`AAXS_NeedAuthentication`	This error occurs when the content is configured to ask clients to authenticate before getting the licenses.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		• The distributor's software should authenticate the user and then acquire the license again. If your service does not intend to use authentication, log the identify of the content that is causing this error. • This error should not require an escalation, unless the content is not supposed to be configured to require authentication. In this case, repackage the offending content with proper policy. If the content is packaged correctly, see Diagnosing policy / license discrepancies.
License Enforcement errors that aren't covered above		
3331	`AAXS_ContentNotYetValid`	The acquired license is not yet valid. To resolve this issue, check whether the client clock is not set correctly. To set the client clock, repackage the content or modify the license server configuration.
3332	`AAXS_CachedLicenseExpired`	Reacquire license from the server.
3333	`AAXS_PlaybackWindowExpired`	You must notify users that they cannot play this content till the policy expires.
3334	`AAXS_InvalidDRMPlatform`	This platform is not allowed to playback the content because, for example, the content provider has configured Adobe Access to deny content to Adobe Access on a platform or a shared domain-bound license is bound to a shared domain token that is meant for a different partition. CDM might throw this error if content was not packaged by using an appropriate (CDM feature gated) packager certification. If the content is packaged with an incorrect PHDS/PHLS certificate, the content might work in Chrome but not other browsers (or vice versa). ***Tip:*** *This is because Chrome uses different PHDS/PHLS certificates.* To confirm which certificate is being used, dump the details of the content metadata and look for the *recipient*

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		certificates. For more information, see *https://adobeprimetime.zendesk.com/agent/tickets/2891*.
3335	`AAXS_InvalidDRMVersion`	Upgrade to the latest version of the TVSDK for Android. To resolve this issue, complete one of the following tasks: • Upgrade AIR • For Flash Player, upgrade the AdobeCP module and retry playback.
3336	`AAXS_InvalidRuntimePlatform`	This platform is not allowed to playback the content because, for example, the content provider has configured Access to deny content to FP/AIR on a platform.
3337	`AAXS_InvalidRuntimeVersion`	Upgrade to the latest version of TVSDK for Android. This occurs if the content or the server is configured to deny playback to a particular version of the Flash or AIR runtimes. • If the user is on an operating system on which Flash can be upgraded, the distributor's software should prompt the user to upgrade Flash and try again. Otherwise advise the user to use a different machine. • If error 3337s is suspected, identify whether it is occurring for specific content and repackage that content. If content is packaged correctly see Diagnosing policy / license discrepancies
3338	`AAXS_UnknownConnectionType`	Unable to detect the connection type, and the policy requires you to turn on Output Protection. This issue is expected only if the content is packaged to require digital or analog output protection. An issue in versions of Flash Player older than version 11.8.800.168 caused error 3338 to occasionally occur on content for which the policy indicated that content protection is `USE IF AVAILABLE`. This issue is fixed in version 11.8.800.168 and later. • The distributor's software select a variant of the content that does not require output protection (for example SD variant of an HD stream). If error 3338 is occurring on `USE_IF_AVAILABLE` content, check for player version number. If the player

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		version is less than 11.8.800.168, advise the user to upgrade Flash Player. If error 3338 is occurring on versions above 11.8.800.168, log which content caused the error. • The distributor should check which content is causing this error and validate that the content's policy is setting `NO_PROTECTION` or `USE_IF_AVAILABLE` for analog and digital outputs. If content is inadvertently packaged with `NO_OUTPUT` or `REQUIRED`, repackage the content. If content is packaged correctly see Diagnosing policy / license discrepancies. Otherwise escalate to Adobe. For more information, see *Getting unexpected 3338 errors when your DRM policy is set to USE_IF_AVAILABLE?*
3339	`AAXS_NoAnalogPlaybackAllowed`	Unable to play back on analog device. To resolve the issue, connect a digital device.
3340	`AAXS_NoAnalogProtectionAvail`	Unable to play back content because the connected analog external display device (monitor/TV) does not have the correct capabilities (for example, the device does not have Macrovision or ACP).
3341	`AAXS_NoDigitalPlaybackAllowed`	Unable to play back content on a digital device. ***Important:*** *This issue should not happen in a production environment, because content publishers should not disallow digital playback.*
3342	`AAXS_NoDigitalProtectionAvail`	The connected digital external display device (monitor/TV) does not have the correct capabilities. For example, the device does not have HDCP.

3343	`AAXS_IntegrityVerificationFailed`	Not applicable for Android.
		This error is currently known to happen initially after a new version of Flash is released. It occurs because Flash upgraded while Flash was open, which puts Flash in a bad state until browser restarts.
		• The distributor's software should complete the following tasks:
		  • Recommend that the user close or quit all browsers and then reopen.
		  • Check if the version of Flash is current.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		If the version is not current, advise the customer to upgrade, close all tabs in their browser, and reopen. • If error appears to occur after a successful browser restart, escalate to Adobe. When a new version is released, we recommend that you contact Adobe Support to see whether the background updates issue has been fixed.
3344	`AAXS_MissingAdobeCPModule`	Not applicable for Android.
3345	`AAXS_DRMNoAccessError`	Not applicable for Android. This error occurs when part of Flash or AIR was not installed correctly. The distributor's software should do one of the following: • Ask the user to uninstall and reinstall AIR. • For Flash Player, call `System.update`.
3346	`AAXS_MigrationFailed`	• The distributor's software should do one of the following: • If AIR, call `DRMManager.resetDRMVouchers()` • If Flash has is unusable because of errors 3322 or 3346 error code, users should go to *http://forums.adobe.com/message/5535907#5535907* and follow the Adobe article's instructions to programmatically reset their DRM license store. • If this error occurs frequently, the distributor should provide the details about the frequency player version and the browser version to Adobe. For more information, see the following forum articles: • *DRM Error 3322/3346/3368 in Chrome (Info-Bar Problems)* • *3322 or 3346 error after hardware change*
3347	`AAXS_InsufficientDeviceCapabilites`	The primary meaning of this error is that the license has a constraint which the clients' DRM certificate indicates it cannot satisfy. The following "hardware capabilities" are defined when the clients DRM certificate is issued: • **Non-User Accessible Bus**. If **true**, the decrypted media never flows across a bus or into main memory where an application can access to it.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		If **false**, content might be accessible to the application after decryption. • **Hardware Root of Trust**. If **true**, all software that is loaded at boot time on the device was validated against a key or digest that is only available in hardware. Both of these constraints are checked on the client side when the license is opened against the DRM certificate for the client and failure is immediate. These constraints can also be checked on the server side prior to issuing the license. The secondary meaning of this error is that the license has the “Jailbreak Enforcement" policy set and a jailbreak has been detected on the device. This check is done periodically on the client side and cannot be checked on the server side. The distributors can update the policies and remove the restrictions. For device capability policies, issue the policy update command with the `-devCapabilitiesV1` flag and no arguments. For jailbreak enforcement, set `policy.enforceJailbreak=false`.
3348	`AAXS_HardStopIntervalExpired`	Hard stop interval expired.
3349	`AAXS_ServerVersionTooHigh`	The server is running at a version that is higher than the highest version that is supported by client.
3350	`AAXS_ServerVersionTooLow`	The server is running at a version that is lower than the minimum version that is supported by client.
3351	`AAXS_DomainTokenInvalid`	Domain token was invalid. To resolve this issue, register with the domain again.
3352	`AAXS_DomainTokenTooOld`	The domain token is older than the token that is required by the license. To resolve the issue, register with the domain again.
3353	`AAXS_DomainTokenTooNew`	The domain token is newer than the token that is required by the license.
3354	`AAXS_DomainTokenExpired`	Domain token has expired.
3355	`AAXS_DomainJoinFailed`	Domain join failed.
3356	`AAXS_NoCorrespondingRoot`	A root license for a V3 leaf license was not found.
3357	`AAXS_NoValidEmbeddedLicense`	No valid embedded license was found.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
3358	`AAXS_NoACPProtectionAvail`	Cannot play back because the connected analog device does not have ACP protection.
3359	`AAXS_NoCGMSAProtectionAvail`	Cannot play back because connected analog device does not have CGMS-A protection.
3360	`AAXS_DomainRegistrationRequired`	Content requires domain registration.
3361	`AAXS_NotRegisteredToDomain`	Machine is not registered to the domain for the specified metadata.
3362	`AAXS_OperationTimeoutError`	Asynchronous operation took longer than `maxOperationTimeout`. Only returned by iOS DRMNative Framework.
3363	`AAXS_UnsupportedIOSPlaylistError`	The M3U8 playlist passed in had unsupported content. Only returned by iOS DRMNative Framework.
3364	`AAXS_NoDeviceId`	The framework requested the device ID, but the returned value was empty. The user should not select the **Allow identifiers for protected content** check box in Chrome settings.
3365	`AAXS_IncognitoModeNotAllowed`	This browser/platform combination does not allow DRM-protected playback in Incognito mode. The distributor's software should advise the user to exit Incognito mode or use a different browser. For more information, see *DRM error 3365 cause and resolution*.
3366	`AAXS_BadParameter`	The host runtime called the Access library with a bad parameter.
3367	`AAXS_BadSignature`	m3u8 manifest signing failed. Only returned by iOS DRMNative Framework or AVE.
3368	`AAXS_UserSettingsNoAccess`	The user cancelled the operation or has entered settings that disallow access to the system. This error is only thrown when the SWF version is 19 or later. For backward compatibility, 3321 is thrown when the SWF is version 18 or earlier. The distributor's software should guide the user to an explanation of how to allow unsandboxed plugin access. For more information, see *Google Chrome's unsandbox access denied* and *DRM Error 3322/3346/3368 in Chrome (Info-Bar Problems)*.

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
3369	`AAXS_InterfaceNotAvailable`	A required browser interface is not available. This issue occurs only on Pepper. There could be a mismatch between the Flash plugin and the browser version. The distributer's software should guide the user to ensure that they have the latest version of the browser installed. If the incidences of this error are increasing, and corresponds to a browser update being released, escalate to Adobe.
3370	`AAXS_ContentIdSettingsNoAccess`	The user has disabled the **Allow identifiers for protected content** setting. ***Tip:*** *This error appeared with Pepper versions 13.0.0.x or greater.* The distributor's software should guide the user to enable the **Allow identifiers for protected content** setting. The distributor's operations team should guide the user to enable the **Allow identifiers for protected content** setting. For more information, see *https://forums.adobe.com/message/6518323#6518323*.
3371	`AAXS_NoOPConstraintInPixelConstraints`	Malformed resolution based on output protection constraints in the license. The distributor's software should display an error message. Ask user to report the problem to the distributor with a content title. The distributor should repackage content with a valid policy.
3372	`AAXS_ResolutionLargerThanMaxResolution`	The content's resolution is larger than the maximum resolution that is specified in the output-protection constraint. If the distributor's operations team sees this error in their logs, they should review the resolution-based

Value for NATIVE_-ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
		output protection policy, and if necessary, repackage the content.
3373	AAXS_MinorErr_DisplayResolutionLargerThanConstrain	The content's resolution is larger than the resolution that is specified by the currently active output-protection constraint. If the distributor's operations team sees this error in their logs, they should review the resolution-based output protection policy, and if necessary, repackage the content.
3374	AAXS_MinorErr_ClientCommProcessFailed	Failed during client-side communication processing, for example, request generation, response processing, bad auth token, and so on. If the distributor's operations team sees this error in their logs, they should review the resolution-based output protection policy, and if necessary, repackage content.

NATIVE_ERROR: Video playback values

The Video Encoder interface of the AVE returns these video playback notifications in the `NATIVE_ERROR` metadata object.

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Description
-1	END_OF_PERIOD	End of period.
0	SUCCESS	Operation successful.
1	ASYNC_OPERATION_IN_PROGRESS	Asynchronous operation. The operation request has been made. Success/failure information will be available later.
2	EOF	Operation not possible due to end of file (EOF) condition.
3	DECODER_FAILED	The decoder failed at runtime.
4	DEVICE_OPEN_ERROR	Failed to open hardware decoder.
5	FILE_NOT_FOUND	Resource cannot be located.
6	GENERIC_ERROR	Generic error.
7	IRRECOVERABLE_ERROR	An error condition that the Video Engine cannot recover from.

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

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Description
36	PERIOD_NOT_LOADED	An operation was attempted on a HOLD period or a period that has not yet been loaded.
37	INVALID_REPLACE_DURATION	The replace duration specified is invalid or extends past the end of the stream.
38	CALLED_FROM_WRONG_THREAD	API can't be called from the wrong thread. Mostly, for API elements that should be called from Main thread only.
39	FRAGMENT_READ_ERROR	Fragment read error. No failover present. Engine will try to read the next fragment.
40	ABORTED	The operation was aborted by an explicit Abort or Destroy call.
41	UNSUPPORTED_HLS_VERSION	Cannot play this version of HLS media.
42	CANNOT_FAIL_OVER	Cannot fail over.
43	HTTP_TIME_OUT	HTTP download has timed out.
44	NETWORK_DOWN	The user's network connection is down. Playback could stop any moment and will resume when the connection is available.
45	NO_USABLE_BITRATE_PROFILE	No usable bit rate profile found in the stream.
46	BAD_MANIFEST_SIGNATURE	The manifest has a bad signature. It failed the manifest signing test.
47	CANNOT_LOAD_PLAYLIST	Cannot load a playlist.
48	REPLACEMENT_FAILED	Replacement specified in an Insert API could not succeed. This means that the insertion succeeded but replacement did not. Replacement could fail if the manifest to be replaced has been removed from the timeline.
49	SWITCH_TO_ASYMMETRIC_PROFILE	DRM is switching to an asymmetric profile. All the profiles are expected to be aligned in duration. If not, this warning will be thrown, and there may be jumps in the playback.
50	LIVE_WINDOW_MOVED_BACKWARD	Live window is expected to move forward only. If not, this warning will be thrown, and the window will not be

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Description
		read. Because of that, there may be jumps (or stop / long pause) in the playback.
51	`CURRENT_PERIOD_EXPIRED`	Live window moved beyond the current period.
52	`CONTENT_LENGTH_MISMATCH`	The content-length reported by the HTTP server did not match the actual media size.
53	`PERIOD_HOLD`	The media reader is unable to read further because it has reached the time set by setHoldAt API.
54	`LIVE_HOLD`	The media reader is unable to load segments because it has reached the end of the live window. Segment loading will resume when the server ads new media to the live window. This state is usually reached if: • The `bufferTime` is too high (equal to or higher than the live window duration). • A combination of one or more of insert/erase API replaced more media than it added. • The next period is a live period with a pending media replacement (due to InsertBy API call)
55	`BAD_MEDIA_INTERLEAVING`	The audio and video interleaving in the media is not done properly. This is a packaging error. The warning is dispatched when the difference exceeds two seconds.
56	`DRM_NOT_AVAILABLE`	
57	`PLAYBACK_NOT_AUTHORIZED`	HLS playback has not been enabled in the Flash Player. See AuthorizedFeatures.enableHLSPlayback.
58	`BAD_MEDIA_SAMPLE_FOUND`	The decoder received a bad sample that cannot be decoded. This is usually not a fatal error but indicates that there may be glitches in the audio/video. Too many instances of this error indicate a bad encoding or bad file.
59	`RANGE_SPANS_READ_HEAD`	After playback has started, the Insert/Replace range should not contain the read head.

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Description
60	POSTROLL_WITH_LIVE_NOT_ALLOWED	Post-roll insertions are not allowed on a live media. They are, however, allowed after the server marks the media as complete.
61	INTERNAL_ERROR	A very rare issue that should never happen.
62	SPS_PPS_FOUND_OUTSIDE_AVCC	The stream does not follow the packaging recommendation of always putting H264 SPS/PPS in an AVCC. Seek / playback issues might be seen.
63	PARTIAL_REPLACEMENT	Replacement specified in an Insert API was only partially done. This happens when replaceDuration spans over the timeline duration.
64	RENDITION_M3U8_ERROR	Rendition playlist had an error loading. This is only for AVE, not for FlashPlayer.
65	NULL_OPERATION	Operation does not do anything.
66	SEGMENT_SKIPPED_ON_FAILURE	Segment cannot be played and is skipped on failure.
67	INCOMPATIBLE_RENDER_MODE	Incompatible render mode.
68	PROTOCOL_NOT_SUPPORTED	The Web protocol used in the URL is not supported.
69	PARSE_ERROR_INCOMPATIBLE_VERSION	Error while parsing media file.
70	MANIFEST_FILE_UNEXPECTEDLY_CHANGED	Manifest file was changed in an unexpected manner.
71	CANNOT_SPLIT_TIMELINE	Cannot perform a split operation on a timeline.
72	CANNOT_ERASE_TIMELINE	Cannot perform an erase operation on a timeline.
73	DID_NOT_GET_NEXT_FRAGMENT	Did not get the next fragment.
74	NO_TIMELINE	No timeline present in an internal data structure.
75	LISTENER_NOT_FOUND	No listener found in an internal data structure.
76	AUDIO_START_ERROR	Unable to start audio.
77	NO_AUDIO_SINK	No audio sink present in an internal data structure.
78	FILE_OPEN_ERROR	Unable to open file.
79	FILE_WRITE_ERROR	Unable to write to a file.

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Description
80	FILE_READ_ERROR	Unable to read from a file.
81	ID3PARSE_ERROR	There was an error parsing ID3 data.
82	SECURITY_ERROR	Loading the content failed because of security restrictions.
83	TIMELINE_TOO_SHORT	The timeline duration is too short. If this is a live stream, frequent buffering may happen.
84	AUDIO_ONLY_STREAM_START	The stream has been switched to an audio-only stream.
85	AUDIO_ONLY_STREAM_END	The stream has been switched from audio-only to a stream with video.
87	KEY_NOT_FOUND	Key cannot be found.
88	INVALID_KEY	The key is invalid.
89	KEY_SERVER_NOT_FOUND	Key server does not return a key.
90	MAIN_MANIFEST_UPDATE_TO_BE_HANDLED	Cannot handle main manifest update.
91	UNREPORTED_TIME_DISCONTINUITY_FOUND	Unreported time (PTS) discontinuity found.
92	UNMATCHED_AV_DISCONTINUITY_FOUND	Unmatched Audio and Video discontinuity found.
93	TRICKPLAY_ENDED_DUE_TO_ERROR	There was an error while playing media in *trick play* mode. Trick play mode is ended and the stream is paused. Call `Play()` to play the media in normal mode.
95	LIVE_WINDOW_MOVED_AHEAD	The player is out of the live window and must seek forward to catch up.

NATIVE_ERROR: Crypto values

The crypto module of the Adobe video engine returns these notifications in the `NATIVE_ERROR` metadata object.

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
300	CRYPTO_ALGORITHM_NOT_SUPPORTED	Algorithm being used is not supported.
301	CRYPTO_ERROR_CORRUPTED_DATA	Data is corrupted.
302	CRYPTO_ERROR_BUFFER_TOO_SMALL	Buffer too small.
303	CRYPTO_ERROR_BAD_CERTIFICATE	Bad certificate.
304	CRYPTO_ERROR_DIGEST_UPDATE	Digest update.

Value for NATIVE_ERROR_CODE metadata key	Value for NATIVE_ERROR_NAME metadata key	Meaning
305	`CRYPTO_ERROR_DIGEST_FINISH`	Digest finish.
306	`CRYPTO_ERROR_BAD_PARAMETER`	Bad parameter.

Primetime player events summary

Your application can monitor the activity in your player and the changing status of the player by listening for the events that are dispatched by TVSDK.

Events

TVSDK notifies you when events, to which your application must respond, occur. Each event corresponds to a listener class, with a callback method that you must implement.

Tip: *The event codes are the constants of the* `MediaPlayerEvent` *enum.*

`AdBreakCompletedEventListener`

Meaning	The playback of the ad break is complete.
Callback to implement	`onAdBreakCompleted(AdBreakPlaybackEvent event)`
Event code	`AD_BREAK_COMPLETE`

`AdBreakSkippedEventListener`

Meaning	An ad break was skipped during playback.
Callback to implement	`onAdBreakSkipped(AdBreakPlaybackEvent event)`
Event code	`AD_BREAK_SKIPPED`

`AdBreakStartedEventListener`

Meaning	The playback of ad break has started.
Callback to implement	`onAdBreakStarted(AdBreakPlaybackEvent event)`
Event code	`AD_BREAK_START`

`AdClickedEventListener`

Meaning	An ad was clicked during playback.
Callback to implement	`onAdClicked(AdClickEvent event)`
Event code	`AD_CLICK`

`AdCompletedEventListener`

Meaning	The playback of the ad is complete.
Callback to implement	`onAdCompleted(AdPlaybackEvent event)`

Event code	`AD_COMPLETE`

`AdProgressEventListener`

Meaning	Reporting progress during playback.
Callback to implement	`onAdProgress(AdPlaybackEvent event)`
Event code	`AD_PROGRESS`

`AdResolutionCompleteEventListener`

Meaning	Primetime ad decisioningad resolution is complete. This event is only applicable to VOD content.
Callback to implement	`onAdResolutionComplete()`
Event code	`AD_RESOLUTION_COMPLETE`

`AdStartedEventListener`

Meaning	The playback of the ad has started.
Callback to implement	`onAdStarted(AdPlaybackEvent event)`
Event code	`AD_START`

`AudioUpdatedEventListener`

Meaning	A new audio track has been detected.
Callback to implement	`onAudioUpdated(MediaPlayerItemEvent event)`
Event code	`AUDIO_TRACK_UPDATED`

`BufferingBeginEventListener`

Meaning	The player has started buffering.
Callback to implement	`onBufferingBegin(BufferEvent event)`
Event code	`BUFFERING_BEGIN`

`BufferingEndEventListener`

Meaning	The player has stopped buffering.
Callback to implement	`onBufferingEnd(BufferEvent event)`
Event code	`BUFFERING_END`

`BufferPreparedEventListener`

Meaning	The buffer is prepared.
Callback to implement	`onBufferPrepared()`
Event code	`BUFFER_PREPARED`

`CaptionsUpdatedEventListener`

Meaning	A new caption track has been detected.
Callback to implement	`onCaptionsUpdated(MediaPlayerItemEvent event)`
Event code	`CAPTIONS_UPDATED`

`DRMMetadataInfoEventListener`

Meaning	A new DRM metadata has been detected in the media stream.
Callback to implement	`onDRMMetadataInfo(DRMMetadataInfoEvent event)`
Event code	`DRM_METADATA`

`ItemCreatedEventListener`

Meaning	A new media player item has been created.
Callback to implement	`onItemCreated(MediaPlayerItemEvent event)`
Event code	`ITEM_CREATED`

`ItemLoadCompleteEventListener`

Meaning	New load information has been created for the current item.
Callback to implement	`onLoadComplete(MediaPlayerItemEvent event)`
Event code	`ITEM_UPDATED`

`LoadInformationEventListener`

Meaning	A new segment has been loaded.
Callback to implement	`onLoadInformation(LoadInformationEvent event)`
Event code	`LOAD_INFORMATION_AVAILABLE`

`MainManifestUpdatedEventListener`

Meaning	The main manifest or playlist has been updated.
Callback to implement	`onMainManifestUpdated(MediaPlayerItemEvent event)`
Event code	`MANIFEST_UPDATED`

`NotificationEventListener`

Meaning	The operation has failed.
Callback to implement	`onNotification(NotificationEvent event)`
Event code	`OPERATION_FAILED`

`PlaybackRangeUpdatedEventListener`

Meaning	The playback range has been updated.
Callback to implement	`onPlaybackRangeUpdated(MediaPlayerItemEvent event)`

Event code	`PLAYBACK_RANGE_UPDATED`

`PlaybackRatePlayingEventListener`

Meaning	A new playback rate is visible on the screen.
Callback to implement	`onRatePlaying(PlaybackRateEvent event)`
Event code	`RATE_PLAYING`

`PlaybackRateSelectedEventListener`

Meaning	The MediaPlayer's rate attribute has been set.
Callback to implement	`onRateSelected(PlaybackRateEvent event)`
Event code	`RATE_SELECTED`

`PlayStartEventListener`

Meaning	The playback has started.
Callback to implement	`onPlayStart()`
Event code	`PLAY_START`

`ProfileChangeEventListener`

Meaning	The MediaPlayer's current profile has changed.
Callback to implement	`onProfileChanged(ProfileEvent event)`
Event code	`PROFILE_CHANGED`

`ReservationReachedEventListener`

Meaning	Playback reached a timeline reservation.
Callback to implement	`onReservationReached(ReservationEvent event)`
Event code	`RESERVATION_REACHED`

`SeekBeginEventListener`

Meaning	Seek operation started.
Callback to implement	`onSeekBegin(SeekEvent event)`
Event code	`SEEK_BEGIN`

`SeekEndEventListener`

Meaning	The seek operation has finished.
Callback to implement	`onSeekEnd(SeekEvent event)`
Event code	`SEEK_END`

SeekPositionAdjustedEventListener

Meaning	The seek position has been adjusted because of internal playback rules or external business rules.
Callback to implement	`onPositionAdjusted(SeekEvent event)`
Event code	`SEEK_POSITION_ADJUSTED`

SizeAvailableEventListener

Meaning	The size of the media is available.
Callback to implement	`onSizeAvailable(SizeAvailableEvent event)`
Event code	`SIZE_AVAILABLE`

StatusChangeEventListener

Meaning	The MediaPlayer state has changed.
Callback to implement	`onStatusChanged(MediaPlayerStatusChangeEvent event)`
Event code	`STATUS_CHANGED`

TimeChangeEventListener

Meaning	The playhead has changed.
Callback to implement	`onTimeChanged(TimeChangeEvent event)`
Event code	`TIME_CHANGED`

TimedEventEventListener

Meaning	The operation is complete with the time taken for the operation.
Callback to implement	`onTimedEvent(TimedEventEvent event)`
Event code	`TIMED_EVENT`

TimelineMetadataAddedInBackgroundEventListener

Meaning	A new timed metadata hsa been added to an item in background.
Callback to implement	`onTimedMetadata(TimedMetadataEvent event)`
Event code	`TIMED_METADATA_ADDED_IN_BACKGROUND`

TimedMetadataEventListener

Meaning	A new timed metadata was detected in the media stream.
Callback to implement	`onTimedMetadata(TimedMetadataEvent event)`
Event code	`TIMED_METADATA_AVAILABLE`

`TimelineUpdatedEventListener`

Meaning	The timeline has been modified. Ads might have been added to or removed from the timeline.
Callback to implement	`onTimelineUpdated(TimelineEvent event)`
Event code	`TIMELINE_UPDATED`

Billing metrics

To accommodate customers who want to pay only for what they use, rather than a fixed rate regardless of actual use, Adobe collects usage metrics and uses these metrics to determine how much to bill the customers.

Every time the player generates a stream start event, TVSDK starts to send HTTP messages periodically to Adobe's billing system. The period, known as billable duration, can be different for standard VOD, pro VOD (mid-roll ads enabled), and live content. The default duration for each content type is 30 minutes, but your contract with Adobe determines the actual values.

The messages contain the following information:

- Content type (live, linear, or VOD)
- Content URL
- Whether ads are enabled
- Whether mid-roll ads are enabled (VOD only)
- Whether the stream is protected by DRM
- The TVSDK version and platform

Adobe preconfigures this arrangement, but you can work with your Adobe Enablement representative to change the arrangement, work with your Adobe Enablement representative.

To monitor the statistics that TVSDK sends to Adobe, obtain the URL from your Adobe Enablement representative, and use a network capture tool, like Charles, to see the data.

Configure billing metrics

If you use the default configuration, there is nothing else you need to do to enable or configure billing. If you obtained different configuration parameters from your Adobe Enablement representative, use the `BillingMetricsConfiguration` class to set these parameters up before initializing the media player.

Tip: *Most customers should use the default configuration.*

Important: *The configuration you set remains in effect for the life of the media player. Once you initialize the media player, you cannot change the configuration.*

To configure billing metrics:

1. Enter the following code sample.

```
MediaPlayerItemConfig config = new MediaPlayerItemConfig();
BillingMetricsConfiguration billingConfig = new BillingMetricsConfiguration();
billingConfig.setEnabled(true);
billingConfig.setProVODBillableDurationMinutes(60);
billingConfig.setStdVODBillableDurationMinutes(30);
billingConfig.setLiveBillableDurationMinutes(15);
```

```
   config.setBillingMetricsConfiguration(billingConfig);
   mediaPlayer.replaceCurrentResource(mediaResource, config);
```

Transmit billing metrics

TVSDK sends billing metrics to Adobe in an XML format.

If you use a network capture tool to monitor the statistics TVSDK transmits to Adobe, you should see units like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
    <sc_xml_ver>1.0</sc_xml_ver>
    <reportSuiteID>primesample2</reportSuiteID>
    <visitorID>947310128cb56f41</visitorID>
    <pageURL>http://. . ..m3u8</pageURL>
    <timestamp>2016-4-7T10:1:4</timestamp>
    <contextData>
        <billingMetrics>
            <publisherID>com.adobe.primetime.reference.PrimetimeReference</publisherID>
            <contentType>vod</contentType>
            <adsEnabled>true</adsEnabled>
            <midrollEnabled>true</midrollEnabled>
            <platform>Mac OSX 10.11.5</platform>
            <tvsdkVersion>2,4,0,1559</tvsdkVersion>
            <contentURL>http://. . ..m3u8</contentURL>
        </billingMetrics>
    </contextData>
</request>
```

The boolean properties `drmProtected`, `adsEnabled`, and `midrollEnabled` appear only if they are true.