

smartclientandroid 2.3.2



smartclip

smartclip's native SDK for Android apps

Version history

Contents

- Introduction
- Features
 - AdView-Settings and configuration
 - Providing necessary information
 - Configuring the environment
 - Event Callback
 - Retrieving ad and display information
 - Separation clips
 - Sequencing
- Implementation guide

Introduction

The smartclip SDK for Android is a native SDK to display instream and out-stream advertising in native Android and FireTV apps. It allows to display various types of video advertisement formats.

Features

Control and display of ads

In the regular instream settings a video player is attached to the SDK to display advertisement. It requires the implementation of an interface, which will serve as connection to the ad server.

It is possible to use one or more ad tags for one ad slot and to configure different types of additional separation clips. In addition a number of additional parameters can be set for the specific environment to pass information to the ad server and to control requested resources.

In an out-stream scenario, based on a standalone video player embedded in regular content, the SDK displays ads and offers a number of settings to control appearance and behaviour of the app.

If the product already provides streams, advertising is embedded in the existing player, providing detailed interfaces for a seamless integration of content and ads. In this scenario the complete playback stack itself remains under the control of the publisher.

Providing necessary information

The interface for the smartclip SDK allows controlling environment and tracking information.

- Overwriting Bundle-ID (for debugging and testing purposes)
- Supplying reporting/tracking information
- Supplying unique advertising id
- Adding metadata on content context

For a full list of possible information settings please refer to the KDoc of the project.

Setting the advertising ID as provided by Google

In case the advertising ID is used inside the app, it should be passed into to the SDK.

Since the field is not validating it is also possible to generate another identifier within the app, which can be used as means of identification and tracking.

The SDK relies on that information and does not try to get a value from the system. This is because using advertising ID requires special attention to the permissions during the app submission process to the Play Store.

Configuring the environment

- Desired bitrate setting
- Desired mime-types
- Set information on network/connection stability

Desired bitrate setting

It is possible to override the default bitrate settings for ad replay by setting a desired bitrate. If the exact value is not present in the ad response, the SDK is going to search for a lower bitrate media file and only if none is available, the SDK will select a media file of higher bitrate. The bitrates should consider the display capabilities of the device and current network settings.

The actually displayed media file can only be selected, if the campaign provides one for the corresponding bitrate.

Configure media file selection based on mime-type

Since the support for different formats varies on certain platforms it is advised to include a list of preferred mime-types, which supports the selection process after parsing of an ad server response. One example is the support of *WebP* formats on certain devices such as the FireTV stick.

Event Callback

During an ad slot or ad session the SDK will inform about a number of state information which can be accessed by registering a listener/delegate to handle callbacks. This will allow the app to interact properly with the advertising content and integrate with the app's control flow.

If there is a registered listener for the *SxEventProvider* it will receive all status transitions that *SxAdInfo*, *SxPublicAd* or *SxPublicAdSlot* goes through. Its main purpose is to aid reacting to media playback states and possible errors.

Retrieving ad and display information

Utilizing *SxEventProvider* the application will get information about the current state of the ad player. It starts with the signal of *ONADTAG_PARSED*, the interpretation of the ad server response, and will iterate through the individual ads that are part of the slot. Callbacks follow the standard VAST events, signalling quartiles and additional information on the type of the ads. Please refer to the KDoc parts to get information on all properties/methods available for the *SxAdInfo*, *SxPublicAd* and *SxPublicAdSlot*.

Separation clips

There are some advanced options available to cover more complex ad break scenarios with multiple ads in one single ad slot. Separation clips can be inserted to signify the beginning or end of an ad break, as well as to separate sponsoring ads from regular advertisement.

Sequencing (instream)

The sequencing component allows to create a schedule for a complete content session. It will automatically create and start ad slots, as well as notify the code if any seek or fast-forward/-backward actions are detected. In that case the SDK returns respective ad slots and allows to reset the mapping. It can be used for various scenarios - even a quite simple preroll- configuration, as it will take care required monitoring and reporting tasks.

Implementation guide

Android

The SDK supports Android platform API level 19 and above, as well as FireTV OS version 5 and above.

Contents of the SDK bundle

The smartclip SDK for Android is shipped in a single *zip* archive with the following contents:

- SDK as *aar* file (*Smartclip-smartclientandroid*)
- apiDocs for the SDK
- Reference implementation
- This documentation

How to work with the reference implementation

It is recommend to start with the reference implementation, which implements all necessary interfaces to start a simple project for both instream and out-stream cases.

For an implementation of an out-stream use case only *SxAdView* and the desired *SxAdSlot* are needed.

If the classes are implemented directly for an instream use case, it is necessary to implement the

interface *SxInstreamVideoPlayerWrapper* to connect individual playback solution to the AdSlots. *SxSequencer* class will than control the whole content and ad playback and is recommended for all standard setups.

Adding the smartclip SDK to a project

To add this SDK to a project extract the *zip* archive's contents, create a new module in Android Studio and import the *aar* file (File → New → New Module → Import .JAR/.AAR Package). Next select the *aar* file from the extracted archive and set a proper "Subproject name" to be used for the smartclip SDK (for example *SxMobileSDK*).

The smartclip SDK is shipped as an *aar* library file. It is developed in Kotlin using Kotlin Coroutines, ExoPlayer and Moshi libraries.

These dependencies should be part of the app's classpath (see next page).

Within the app's build.gradle dependency section add the smartclip SDK dependency (replace *SxMobileSDK* with the module name defined when importing the *aar* file).

```
dependencies {  
    ...  
    implementation project(":SxMobileSDK")  
    ...  
}
```

Also add dependencies for the used libraries:

```
dependencies {  
    ...  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.3.72"  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.2"  
  
    implementation "com.squareup.moshi:moshi-kotlin:1.11.0"  
  
    implementation "com.google.android.exoplayer:exoplayer-core:2.12.2"  
    implementation "com.google.android.exoplayer:exoplayer-ui:2.12.2"  
  
    implementation "io.insert-koin:koin-core:3.1.0"  
    implementation "io.insert-koin:koin-android:3.1.0"  
    ...  
}
```

When targetting SDK 23 or lower you should also add

```
compileOptions {  
    sourceCompatibility 1.8  
    targetCompatibility 1.8  
}
```

to the android configuration block

After that, synchronize the gradle project. smartclip SDK is ready for use.

Activate logs and debug options

For first steps with the SDK activate debug logging. To get detailed information set debug level in the static variable *Log.LogLevel* to *VERBOSE*. By default the log level is set to *WARN*.

The *LogLevel* can be adjusted at any time. To change it for the whole application lifetime it is recommended to set it in *Application*, *Activity* or any *static* context.

```
import tv.smartclip.smartclientandroid.lib.utils

class MyApplication : Application() {

    override fun onCreate() {
        super.onCreate()

        Log.logLevel = Log.LogLevel.WARN

        ....
    }
}
```

Implementing a basic instream ad break with sequencing

For the instream use case start with the *InstreamActivity* (*InstreamExoWrapperActivity* or *AbsVideoPlayerWrapperActivity*) from the *ReferenceApp* package to familiarize with the general setup of playbacks.

SxSequencer is used, which encapsulates most low-level events and in turn uses

SxInstreamVideoPlayerWrapper to control video playback.

The example uses *ExoPlayer* for displaying content video and ads.

Part of the version 2 is a component called *SxAdOverlayContainer*, which manages states like skippable ads and progress as well as buttons to control sound and cancel.

If reusing the component or adding your own overlays you should be aware of the fact that the child views that have been added by the SDK will be reset with any re-run or re-use of the component.

To start building the setup for an instream use case, define ad tags and their positions within the content video with the *SxSequencerAdSlots*. A general skip offset can be configured, if needed.

```

fun createAdSlot(position: Double) =
    SxSequencerAdSlot(adTag = <Smartclip-AdTag>,
        position = SxSequencerRelativePosition(position),
        opener = <Opener-URL>,
        forceOpener = <true|false>,
        closer = <Closer-URL>,
        forceCloser = <true|false>,
        bumper = <Bumper-URL>,
        forceBumper = <true|false>,
        skipAdDuration = 5
    )

val adSlots = List<SxSequencerAdSlot> = listOf(
    createAdSlot(0.0),
    createAdSlot(0.33),
    createAdSlot(0.66),
    createAdSlot(1.0))

```

Position definition must be an instance of *SxSequencerPosition*. For example:

- *SxSequencerAbsolutePosition* - defines an absolute position (in milliseconds) within the content video.
- *SxSequencerRelativePosition* - defines a relative position within the content video (0=start; 1=end).

Furthermore a *SxInstreamVideoPlayerWrapper* implementation is required. It will observe and control an existing video playback.

We offer two ways to implement the *SxInstreamVideoPlayerWrapper*:

- using the *InstreamExoWrapper*:
 - Ready to use wrapper based on the ExoPlayer
 - Allows changing of certain features like loading of the video files or react when user seeks through the content video. Also provides possibility to use a custom ExoPlayer instance if needed.
- implementing the *AbsVideoPlayerWrapper*:
 - Abstract class to be used with any other video player than ExoPlayer.
 - Implementation needs to sync observer properties of the wrapper with the video player in and output events.

Keep in mind, that the *onCreate* and *onCreateView* methods can get called again on the same instance after *onDestroy* or *onDestroyView* was called. Because of that, you need to create new instances of the *SxInstreamVideoPlayerWrapper* and the *SxSequencer* every time, *onCreate* or *onCreateView* gets called.

InstreamExoWrapper

```
lateinit var playerWrapper: SxInstreamVideoPlayerWrapper

override fun onCreate(savedInstanceState: Bundle?) {
    ...

    val video: PlayerView = findViewById<PlayerView>(R.id.playerView)
    val contentVideo = "https://url.to.some.content.video"
    playerWrapper = InstreamExoWrapper(video, contentVideo)
}
```

AbsVideoPlayerWrapper

A basic implementation of the *AbsVideoPlayerWrapper* is included in the reference app, which can adjust to the player used in your project.

Setup the sequencer

After implementing the *SxInstreamVideoPlayerWrapper* interface and definition of desired ad slots with the *SxSequencerAdSlot* classes, create a *SxSequencer* instance:

```
lateinit var sequencer: SxSequencer

override fun onCreate(savedInstanceState: Bundle?) {
    ...

    sequencer = SxSequencer(playerWrapper, adSlots)
    playerWrapper.playWhenReady = true
}
```

Lifecycle

Setting *playWhenReady* of the *SxInstreamVideoPlayerWrapper* to *true*, *SxSequencer* will start playback as soon as possible. This property should also be used to pause and resume playback of content or advertisement videos at any time. For example when underlying *Activity* or *Fragment* is paused or resumed, this needs to be forwarded to *SxInstreamVideoPlayerWrapper*:

```
override fun onResume() {
    super.onResume()
    playerWrapper.playWhenReady = true
}

override fun onPause() {
    playerWrapper.playWhenReady = false
    super.onPause()
}
```

Finally it is very important to release the *SxSequencer* every time the *Activity* or *Fragment* is destroyed.

```
override fun onDestroy() {
    sequencer.release()
    super.onDestroy()
}
```

Configuration

All available parameters except the ad tags and their playback position can be configured in the *SxConfiguration* class.

By default the *SxSequencer* uses *SxConfiguration.INSTREAM* as configuration.

To provide your own configuration, just create a copy of the default configuration, set the desired parameters and initialize the *SxSequencer* with that configuration. For example when you want a specific bitrate and no automatic audio focus handling:

```
val configuration = SxConfiguration.INSTREAM.copy(  
    desiredBitrate = 1200,  
    handleAudioFocus = false  
)  
sequencer = SxSequencer(playerWrapper, adSlots, configuration)
```

A more detailed configuration would need additional parameters depending on the desired setup. See KDoc of *SxConfiguration* for more details.

Click-Through

We support three different Click-Through setups:

- **Direct:** The landing page is opened directly when a user taps an ad.
 - *clickType* = *CLICKABLE_ON_FULL_AREA*
- **Dialog** (*default*): A user needs to confirm a dialog before the landing page is opened.
 - *clickType* = *CLICKABLE_WITH_CONFIRMATION_DIALOG*
 - *clickThroughListener* = *null*
 - optional:
 - *clickThroughDialogTitle*
 - *clickThroughDialogMessage*
 - *clickThroughDialogPositiveAnswer*
 - *clickThroughDialogNegativeAnswer*
- **Custom:** When a user taps an ad, your registered click-through listener is called and the ad is paused. Using the provided parameter of the listener will open the landing page or continue playback.
 - *clickType* = *CLICKABLE_WITH_CONFIRMATION_DIALOG*
 - *clickThroughListeners* set

The implementation of the *Direct* or *Dialog* setups is straight forward. Please see KDoc of *SxConfiguration* for more details. Because the implementation for the *Custom* setup is a little bit more complex, it is shown below.

For the *Custom* setup you only need to set the *clickType* parameter to *CLICKABLE_WITH_CONFIRMATION_DIALOG* and the *clickThroughListener* parameter to your callback. The *clickThroughListener* parameter expects a lambda of the type *(listener: (Boolean) -> Unit) -> Boolean* and is used in the following way:

- the lambda is invoked when a user taps the ad
- you need to return a Boolean in this lambda to signal how the click-through should be

handled:

- *false* to let the SDK handle the click-through. It will ask the user using a dialog, if he wants to open the landing page or not. The playback is automatically resumed, after the user returns from the browser or dialog. This is technically the same as setting the *clickThroughListener* to *null* (see *Dialog* setup).
- *true* if you want to show your own dialog or do what ever you want. The SDK will also pause the playback, but you are responsible to signal the users decision to the SDK. To do so call the provided parameter *listener* of your *clickThroughListener* lambda. *listener* is also a lambda that expects one Boolean parameter:
 - *true* if the user accepts the click-through. This will open the landing page with the default browser and resumes the ad playback when the user returns to the ad in some way.
 - *false* if the user regrets the click-through. This will just resume the playback of the ad.

```
val config = SxConfiguration.INSTREAM.copy(clickThroughListener = { listener ->
    AlertDialog.Builder(context)
        .setMessage("Do you want to open the landing page?")
        // Call `listener` with the proper parameter to signal the users
decision.
        .setPositiveButton("Yes") { _, _ -> listener(true) }
        .setNegativeButton("No") { _, _ -> listener(false) }
        .setOnDismissListener { listener(false) }
        .show()
    true // You want to handle the click-through.
})
```

The example above shows a minimal implementation of the *Custom* click-through setup. Keep in mind to call *listener* for all possible ways the user can go from here (positive, negative, dismiss).

Unfortunately we can not provide more information, like the URL of the landing page, at this point.

Seeking

The shown example configures four scheduled ad breaks, one pre-roll, two mid-rolls and a post-roll (as showcased in the *ReferenceApp*).

```
private lateinit var latestCreatedAdSlots: List<SxSequencerAdSlot>

protected open fun createAdSlots(): List<SxSequencerAdSlot> {
    latestCreatedAdSlots = listOf(
        createAdSlot(0.0),
        createAdSlot(0.33),
        createAdSlot(0.66),
        createAdSlot(1.0)
    )
    return latestCreatedAdSlots
}
```

Overwrite the method *onContentVideoSought* of *SxInstreamVideoPlayerWrapper* interface to enhance the implementation, which is called directly after the video was sought. In the following example the created *SxSequencerAdSlots* are stored in the *latestCreatedAdSlots* property directly after they were created.

In the *onContentVideoSought* callback they now can be used to restore ad slots that were removed before. The following example shows how to restore a removed ad slots, after a user seeks back before the original trigger position of this ad slot.

```
override suspend fun onContentVideoSought(
    remainingAdSlots: List<SxSequencerAdSlot>,
    removedAdSlots: List<SxSequencerAdSlot>) : List<SxSequencerAdSlot> {

    val adSlotsToReturn = remainingAdSlots.toMutableList()
    val relativePosition = (exoPlayer?.currentPosition ?: 0) /
        (exoPlayer?.duration ?: -1).toDouble()

    /** Read ad slots, if they were removed before. This is the case when user
    seeks forward, which will remove skipped ad slots, and seek the video backward
    after that, which will insert the removed ad slots again.
    The following will restore any removed ad slot for this use case:
    1. User starts a content video with ad slots at the relative positions 0.0, 0.5
    and 1.0.
    2. After the pre-roll finished user seeks video to 2/3 of the complete content
    video duration.
    3. Because the ad slot with the relative position of 0.5 was jumped over, the
    slot is removed from the list of remaining ad slots.
    4. Now the user seeks the content video back to 1/3 of its duration.
    5. Because the ad slot with position 0.5 is now in front of the current content
    video position, the ad slot is restored to the list of remaining ad slots.

    (The following code only shows step 5. The other steps are automatically
    processed by the SxSequencer.) */
    latestCreatedAdSlots.asReversed().forEach { adSlot ->
        if (relativePosition < adSlot.position.value as Double
            && remainingAdSlots.none {
                (it.position as SxSequencerRelativePosition).value
                    == adSlot.position.value as Double
            }) {
            adSlotsToReturn.add(adSlot)
        }
    }
    return adSlotsToReturn
}
```

Instead of restoring removed ad slots it is also possible to add a completely new *SxSequencerAdSlot*. The only requirement is, that the new ad slot's trigger position is in front of the current content video position.

By default the *onContentVideoSought* callback will just return the *remainingAdSlots*. Any other behaviour like the described above needs to be implemented manually.

For more detailed information on the active ad slots the *SxSequencer* implements the *SxEventProvider* interface, which can be used to subscribe to changes of the events *SxAdInfo*, *SxPublicAd* and *SxPublicAdSlot*.

Implementing a basic out-stream ad

In out-stream use cases the smartclip SDK takes care of most processes around the configuration and rendering of advertisements.

It is possible to insert the ad placement directly in an XML layout file, although all configuration options are available for dynamic creation as well (see *SxConfiguration* for more details).

```
<tv.smartclip.smartclientandroid.lib.SxAdView
    android:id="@+id/advertisement"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:adTag="@string/ad_tag_5sec"
    app:title="@string/single_page_ad_title"
    app:titleStyle="@style/SinglePage.Title"
    app:showMuteToggleButton="true"
    app:muteIconStyle="@style/SinglePage.MuteIcon.Mute"
    app:unmuteIconStyle="@style/SinglePage.MuteIcon.Unmute"
    app:allowAdSkipping="true"
    app:skipButtonText="@string/single_page_skip_button"
    app:skipButtonStyle="@style/SinglePage.SkipButton"
    app:initialMuted="true"
    app:onEndAction="repeatButton"
    app:showProgressBar="true"
    app:progressBarStyle="@style/SinglePage.ProgressBar"
    app:clickThroughDialogMessage="@string/single_page_click_through_message"
    app:clickThroughDialogTitle="@string/single_page_click_through_title" />
```

The corresponding activity or fragment just needs to inflate the created XML. It is also necessary to connect the advertisement with the Android lifecycle with *onResume* and *onPause* and to call the corresponding shutdown method *release* on the *AdView*, once the activity or fragment is stopped or the view is not needed any more.

```

import kotlinx.android.synthetic.main.fragment_single_page.*

class SinglePage : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?) =
        inflater.inflate(R.layout.fragment_single_page, container, false)!!

    override fun onResume() {
        super.onResume()
        advertisement.onResume()
    }

    override fun onPause() {
        advertisement.onPause()
        super.onPause()
    }

    override fun onDestroyView() {
        advertisement.release()
        super.onDestroyView()
    }
}

```

For the different parameters and functions please check KDoc or the section for configuring out-stream placements in this document.

To subscribe to the stream of events during an ad session use the var *SxAdView.addInfoListener*, which expects a lambda in the form of

```
(suspend (SxAdInfo) -> Unit)?
```

Configuring with *SxConfiguration*

Using *SxConfiguration* there are multiple options to create a sequencer or a single ad slot. Note that the configuration object controls all settings, macros and additional native tracking information for both instream and out-stream scenarios. That also means not all options may apply to a specific scenario (e.g. values in *SxPlacement* or the out-stream formatting options).

When in doubt use the default configurations by not providing a different one. They are listed in the constants of *SxConfiguration.INSTREAM* and *SxConfiguration.OUTSTREAM* and must be applied like this:

```

val config = SxConfiguration.INSTREAM.copy(
    showMuteToggleButton = false
)

// for Instream
val sequencer = SxSequencer(this, controlOverlay, adSlots, config)

// or for Outstream
val adView = object : SxAdView(context, null, 0) {
    override val configuration = config
}

```

The following sections will outline some common use cases. Please refer to KDoc on *SxConfiguration* for an overview of all parameters.

Setting a desired bitrate and desired mime-types

The SDK will try to select a media file of the desired bitrate from the available media files in the server's ad response. If the exact value is not present in the ad response, the SDK is going to search for a lower bitrate media file and only if none is available, the SDK will select a media file of higher bitrate.

In order to restrict the selection to certain mime-types, those should be set as environment variable by handing the list of desired mime-types to *SxMacros* as an array.

```

SxConfiguration.INSTREAM.copy(
    desiredBitrate = 1200
    desiredMimetypes = listOf("video/mp4", "video/webm")
)

```

Passing information

Additional information is also considered part of the configuration and has to be defined at initialization of the *SxSequencer* or *SxAdView*. There is a number of information, according to the VAST standard, that should be available for reporting or ad requests.

```

SxConfiguration.INSTREAM.copy(
    initialMuted = false,
    placementType = SxPlacementType.Companion.UNDEFINED,
    breakPosition = SxBreakPosition.PRE_ROLL,
    contentId = "Some identifier"
)

```

The full list of supported information is compiled in the following table for all supported macros and configuration values (if default is empty the native SDK does not set a value here). The definition of expected values is in line with standard VAST 4.1, section 6.

Value	Effect	Default
adCategories	Content categories	
appName	Freely specified app name	
appBundle	VAST 4.1 standard-identifier	package name from

apiFrameworks	Player's feature support	Context
blockedCategories	Blocked content categories	7
breakPosition	Position of the ad break in relation to content	0 Instream (4 for out- stream)
contentId	Customer-specific content identifier	
contentUri	Customer-specific content resource identifier	
deviceIp	Device IP address	
domain	Domain information	
extension	Defines the position of an advertisement break inside an instream configuration.	OTHER
ifa	Advertising Identifier	
ifaType	rida-Roku ID, aaid-Android ID, idfa-Apple ID	
inventoryState	List of options indicating attributes of the inventory. Possible values: <i>skippable</i> to indicate ads can be skipped, <i>autoplayed</i> to indicate ads are autoplayed with audio unmuted <i>mautoplayed</i> to indicate ads are autoplayed with audio muted <i>optin</i> to indicate the user takes an explicit action to knowingly start playback of ads	
latLong	user position as lat long floats	
limitAdTracking	Did the user restrict the use of advertising?	0
mediaPlayhead	Current playhead of content stream	
placementType	One of the <i>SxPlacementTypes</i>	
playerCapabilities	List of player capabilities as string list	
regulations	Privacy regulations that apply	
verificationVendors	List of verification vendors	

Define skip options for your app

The SDK supports options to skip ads. Sometimes this information is part of the VAST document returned from the ad server.

A general skip option for all ads (which do not bring their own) can be set via *skipAdDuration* in the creation of the *SxAdSlot* (Outstream) or *SxSequencerAdSlot* (Instream).

The value of *skipAdDuration* denotes the time period in seconds that will pass before the skip button appears.

Any skip offset defined by the ad server will overwrite the general skip offset defined programmatically in this setting.

It is possible to change the appearance of the button and the text by using the corresponding resource settings *skipButtonText* and *skipButtonStyle*.

Using separation clips

It is possible, but not required, to specify separation clips which are played back at fixed places within an ad slot. The definition of these separation clips and handling of the video files is done in the source code of the Android app.

Note that handling of these configurations requires to first define a *SxAdSlotDelegate* and set the configuration in *SxAdSlotController* or use the simpler interface of *SxSequencerAdSlot* object first and then add Opener, Closer or Bumper to their initialisation.

```
val slot = SxAdSlot(getString(R.string.ad_tag_5sec),
                    SxSequencerRelativePosition(0.0),
                    getString(R.string.default_opener_url),
                    getString(R.string.default_closer_url),
                    getString(R.string.default_bumper_url))
```

For the out-stream use case, define the separation clips via *loadAd* method of *SxAdView*.

Configuring out-stream placements

To configure out-stream placements there is a number of options available in the *SxConfiguration* object. There is also a preset for out-stream, which is available as a static object.

Value	Effect	Default
onEndBehaviour	Define what happens after the last ad has been played, NOTHING (last frame stays), REPEAT (a repeat button is offered), COLLAPSE (Remove view into invisibility)	REPEAT
onEndBehaviorAfterSkip	Similar to above, but for users who skipped the last ad	REPEAT
initialMuted	Defines the playback configuration	true
skipButtonText	Button text, in case skip ad is offered	skip
skipButtonStyle	StyleResource	
repeatButtonStyle	StyleResource	
unmuteIconStyle	StyleResource	
muteIconStyle	StyleResource	
showPlaybackProgress	Whether a progress bar should be displayed	
progressBarStyle	StyleResource	
title	A title to display the advertising	Advertisement
titleStyle	StyleResource	

Events

The events can be classified as informative events or as behavioural events that need to be considered in regard to functioning of the application and general user experience.

Behavioural events

These events may require attention when it comes to displaying or stopping a content video. They are expected in the order of the following list:

- ON_AD_MANIFEST_LOADED (ad server response has been parsed, information on the upcoming slot is complete)
- ON_AD_SLOT_STARTED (ad slot has started playing)
- ON_AD_STARTED (an ad started to play)
- ON_AD_FINISHED (an ad finished playback)
- ON_AD_SLOT_FINISHED (the last item has been played back)
- ON_AD_SLOT_COMPLETE (ad slot has reached its end)

Informative events

These events mainly cover standard VAST events or information on states that have been changed by either player or user.

- ON_AD_SCHEDULED
- ON_AD_PLAYBACK_START
- ON_AD_PLAYING
- ON_AD_FIRST_QUARTILE
- ON_AD_SECOND_QUARTILE
- ON_AD_MID_POINT
- ON_AD_THIRD_QUARTILE
- ON_AD_IMPRESSION
- ON_AD_PLAYBACK_FINISHED
- ON_AD`LINEARITY`CHANGED
- ON_AD_PAUSED
- ON_AD_SKIPPED

Error events

Fired in case of errors

- ON_AD_ERROR

Whenever the SDK encounters a problem during loading, rendering there will be a feedback to the publisher, based on the *ON_AD_ERROR* event.

Tracking as defined in the VAST document will be fired.

Since the instream scenario relies on the publishers' implementation, it is important to implement the interface method *reportPlayerError*.

This will notify the SDK of any playback issues that happen in the scope of the player. Of course that results in an *ON_AD_ERROR* event that will be reported back to the publisher.

An event of this kind does not signal the end of an ad slot or ad replay. It might trigger other behaviour, such as fallback to buffet ads or new playback commands.