Adobe® Primetime

Video heartbeat SDK Guide for Android - Version 1.3.1

Updated: 26 August 2014

Contents

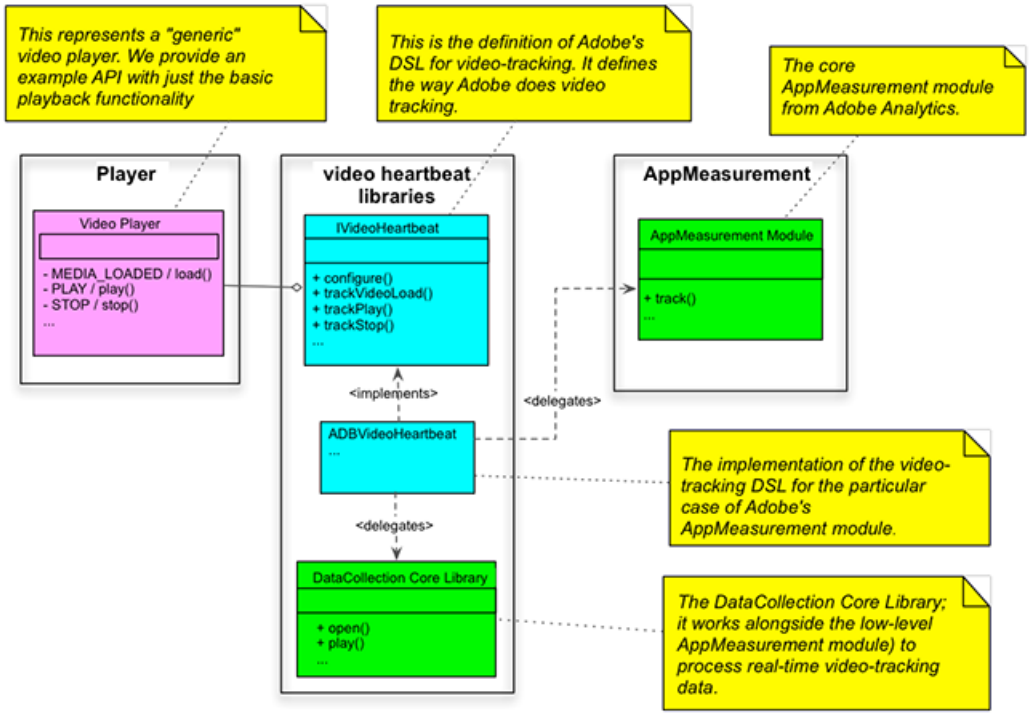
Video heartbeat SDK Guide for Android - Version 1.3.1

Introduction to Video Analytics

The Adobe Video Analytics libraries give video player developers the ability to quickly implement real-time video tracking in their players.

The Adobe Video Analytics video heartbeat library exposes an API that facilitates real-time video tracking in video player applications. The solution documented here is for developers of 3rd-party (non-PSDK-based) video players. For customers interested in activating the built-in video tracking capabilities in a player developed using the Adobe Primetime SDK (on supported platforms), see the *PSDK documentation*.

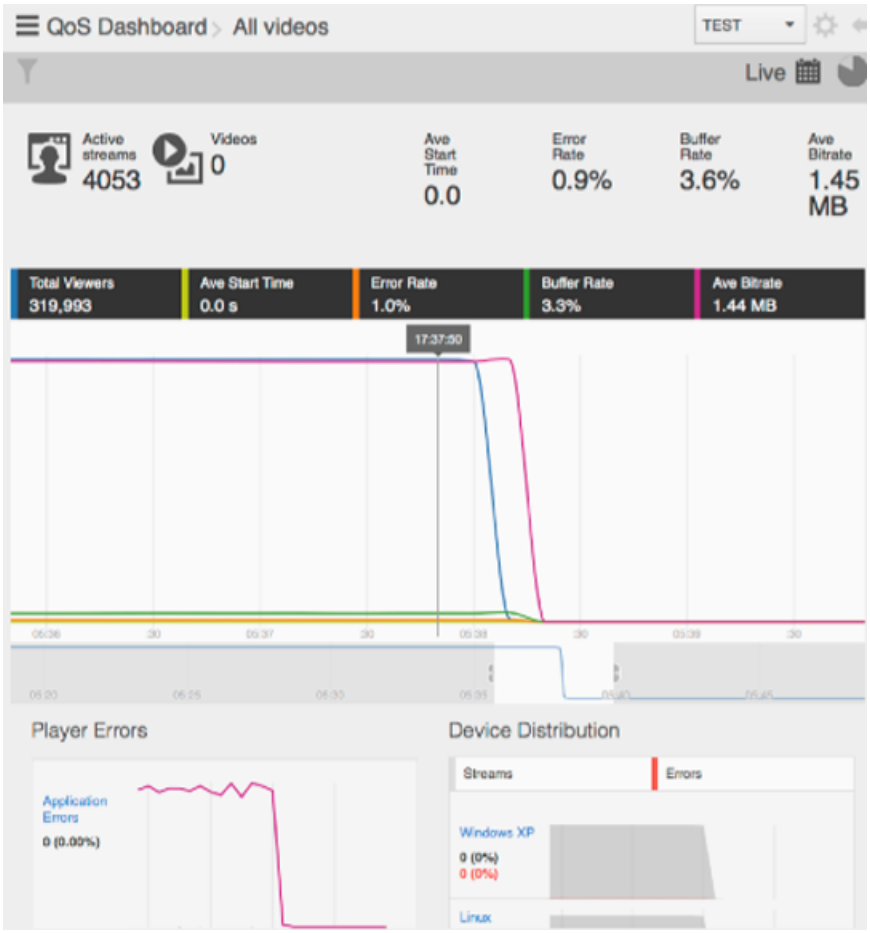
The video heartbeat library mediates between the high-level video player code and lower-level application measurement code, defining an interface that exists solely to solve the problem of real-time video tracking. The following illustration shows this arrangement, in this case featuring a generic video player, video heartbeat code, and the Adobe Analytics AppMeasurement module:



The two main components of the Video Analytics library are:

- `IVideoHeartbeat` - An interface that exposes video tracking methods (`trackPlay()`, `trackAdStart()`, etc.); each `track...()` method corresponds one to one with the video events that occur in the player.
- `PlayerDelegate` - The player delegate is an abstract base class that defines tracking data required by the video heartbeat library. This player delegate must be extended and implemented by the integration engineer to supply tracking data to the video heartbeats library. (Implementation of the player delegate accounts for the bulk of the development effort.)

The ultimate result of adding real-time video tracking capability to your player is the generation of reports on the server-side that display real-time video usage details, such as this sample QOS report:



Integrating Video Analytics

Integrating Video Analytics real-time video tracking into a non-PSDK-based video player requires acquiring video heartbeat libraries, implementing a player delegate to pass data from the player to the video heartbeats library, and handling video heartbeat events in the player.

Before you begin, make sure to be aware of and understand the different types of video tracking available from Adobe, described here: *http://microsite.omniture.com/t2/help/en_US/sc/appmeasurement/hbvideo/*.

The steps for a developer integrating Video Analytics real-time video tracking (video heartbeat) into a non-PSDK-based video player are as follows:

1. Acquire the required libraries from your Adobe representative, and incorporate them into your project:
   - **AppMeasurement library** - The video heartbeat library uses this library's low-level tracking method to open a viewing session in Adobe Analytics.
   - **video heartbeat library** - Contains the video heartbeat data collection core logic. This is where all of the real-time video tracking takes place. The video heartbeat library uses the AppMeasurement module in the sense that it needs access to a subset of its APIs, and delegates some work to it.
2. Acquire video tracking account information from your Adobe representative:
   - **AppMeasurement tracking server endpoint** - The URL of the Adobe Analytics (formerly SiteCatalyst) back-end collection end-point.

- **video heartbeat tracking server endpoint** - The URL of the video heartbeat back-end collection end-point.
- **Account name** - Also known as the Report Suite ID (RSID)

3. Complete the following integration work within your player:
   - **Handle video tracking events** - Listen for the events issued by the player and call the corresponding track method exposed by the video heartbeat library's public API.
   - **Implement the Player Delegate** - The player delegate is a base class that abstracts the different types of data that the video heartbeats library requires from the player (main video, ad content, chapters, and QoS). The video heartbeats library requires this data from the player to perform video-tracking operations. The integration engineer must fill in and return an instance of these data structures, which act as simple containers of the required tracking information. Properly implementing the player delegate is crucial for obtaining a robust video tracking solution.

   Note: *The implementation depends greatly on the specifics of the video player application that the video heartbeat library is being integrated with. The application developer needs to have intimate knowledge about the video player application. The developer must know and understand the mechanisms through which the video player issues notifications about the events that take place during playback. The assumption made here is that the video player is capable of triggering a series of events through which any subscriber can be informed about what happens inside the video player itself.*

4. Initialize and configure the video tracking libraries on your platform. Video Analytics currently employs both the video heartbeats library (for real-time video tracking), as well as the Adobe Analytics AppMeasurement library for certain low-level tasks.
5. Set up reporting on the server side:

   Access the Adobe Analytics Admin Tools to set your reporting parameters. For details, see *Set Up Video Analytics Reporting on the Server-side*.
6. Access the reports on the real-time dashboard. For details, see *Access Video Analytics Reports*.

Initialize / Configure Video Tracking Libraries

The following libraries and configuration data are required to complete the integration; obtain these from your Adobe representative:

- **Required Adobe Video Tracking Libraries:**
  - **AppMeasurement library** - `adobeMobileLibrary.jar` - Contains low-level application measurement core logic. It is capable of providing application-level tracking capabilities, as well as some limited video-tracking features (Adobe Analytics Milestone tracking)
  - **video heartbeat library** - `VideoHeartbeat.jar` - Contains the video heartbeat data collection core logic. This is where all of the real-time video tracking takes place. The video heartbeat library accesses a subset of the AppMeasurement module APIs, and also delegates some work to AppMeasurement.
- **Required Configuration / Initialization Data:**
  - **ADBMobileConfig.json** - *It is crucial that the name of the configuration file remain unchanged. The JSON config file MUST be named `ADBMobileConfig.json`. Place this file into the assets/ folder (which stores files to be compiled into the .apk). This folder should usually be located in the root of your application source tree (this may vary or be configurable depending on the build tools used).The path to this file MUST be .*
  - **AppMeasurement Tracking Server Endpoint** - The URL of the endpoint where all of the AppMeasurement tracking calls are sent
  - **video heartbeat Tracking Server Endpoint** - The URL of the endpoint where all of the video heartbeat tracking calls are sent

- **Job ID** - This is the processing job identifier. It is an indicator for the back-end endpoint about what kind of processing should be applied for the video tracking calls.
- **Publisher Name** - This is the name of the content publisher.
- **Account Name** - Also known as the Report Suite ID (RSID)

For mobile implementations, the life cycle of the AppMeasurement library is not in the hands of the application developer. The AppMeasurement library is instantiated automatically at application load time, and is made available as a globally accessible singleton. So, the application developer is only concerned with the instantiation of the video heartbeat library, which obtains a reference to the AppMeasurement instance behind the scenes. (This is in contrast to Desktop clients, in which the application developer is required to explicitly resolve the dependency chain between the video heartbeat and AppMeasurement libraries.)

1. Configure the AppMeasurement library:

   Most of the configuration options are captured inside a JSON-formatted configuration file. (A limited number of options are available at run-time through API calls.) The config file must be bundled with the application itself as a resource file. This resource file is scanned only once at application load time. Once these values are read from the configuration file, they remain constant throughout the lifecycle of the whole application. AppMeasurement is configured when the following steps are complete:

   1. Fill in the JSON config file (`ADBMobileConfig.json`) with the appropriate values.
   2. Place this file into the `assets` folder. This folder should usually be located in the root of your application source tree (this may vary or be configurable depending on the build tools used)
   3. Compile and build the final application
   4. Deploy and run the bundled application

   Note: *Pay attention to the name and path of the configuration file. The JSON config file must be named* `ADBMobileConfig.json` *and it must be placed inside the assets/ folder which gets compiled into the .apk.*

```
{
    "version" : "1.1",
    "analytics" : {
        "rsids" : "YOUR_OWN_ADOBE_ANALYTICS_RSID",
        "server" : "ADDRESS_OF_THE_ADOBE_ANALYTICS_SERVER",
        "charset" : "UTF-8",
        "ssl" : false,
        "offlineEnabled" : false,
        "lifecycleTimeout" : 5,
        "batchLimit" : 50,
        "privacyDefault" : "optedin",
        "poi" : []
    },
    "target" : {
        "clientCode" : "",
        "timeout" : 5
    },
    "audienceManager" : {
        "server" : ""
    }
}
```

   There are many configuration options available on the AppMeasurement instance that are not shown here. (See the *Adobe Analytics Developer page* for more configuration options.) The options shown in the sample code above are required:

   - `account`
   - `trackingServer`

   These values are provided in advance by Adobe.

2. Instantiate and configure the video heartbeat library.

It is the responsibility of the video player developer to implement and instantiate a subclass of the PlayerDelegate.

Instantiate the video heartbeat component:

```
import com.adobe.primetime.va.adb.VideoHeartbeat;
import com.adobe.primetime.va.IPlayerDelegate;

// Here, the CustomPlayerDelegate class extends the PlayerDelegate abstract class.
CustomPlayerDelegate playerDelegate = new CustomPlayerDelegate();

VideoHeartbeat videoHeartbeat = new VideoHeartbeat(playerDelegate);
```

Configure the newly created video heartbeat instance:

```
ConfigData heartbeatConfig = new ConfigData("URL of the back-end server",
                                            "The Job identifier",
                                            "The publisher identifier");
heartbeatConfig.ovp = "The name of the online video platform";
heartbeatConfig.sdk = "The version of your video player SDK";

// Set this to true only if you want to
//   activate "quietMode" (no network calls)
heartbeatConfig.quietMode = false;

// Set this to true to activate debug message tracing
heartbeatConfig.debugLogging = false;

videoHeartbeat.configure(heartbeatConfig);
```

- **Mandatory configuration parameters.** This set of parameters are provided as input arguments to the constructor method of the ConfigData class. Below is a description of these parameters in order:
  - **URL of the tracking end-point** - `heartbeats.omtrdc.net` - This is where all of the video-heartbeat calls are sent.
  - **jobId** - `j2` - This is the processing job identifier. It is an indicator for the back-end end-point about what kind of processing should be applied for the video-tracking calls.
  - **publisher** - This is the name of the content publisher. This value is provided by Adobe in advance.

Optional configuration parameters (provided as publicly accessible instance variables on the ConfigData class):

- **`channel`** - The name of the distribution channel. Any string can be provided here. Default value: **the empty string**.
- **ovp** - The name if the Online Video Platform though which the content is distributed. (For example, YouTube). It can be any arbitrary string. Default value: **unknown**.
- **sdk** - The version string of the video-player app. It can be any arbitrary string. Default value: **unknown**.
- **debugLogging** - Activates the tracing and logging infrastructure inside the VideoHeartbeat library. Default value: **false**.
- **`quietMode`** - Activates the "quiet mode" of operation, where all output HTTP calls are suppressed. Default value: **false**.

Note: *Setting the `debugLogging` flag to `true` activates extensive tracing messaging, which may impact performance, and may significantly increase the attack surface. While tracing is useful during development and debugging efforts, the application developer must set this flag to `false` for the production version of the player app. The logging mechanism is disabled by default.*

3. Tear down the video heartbeat instance.

   The integration engineer is responsible for manually and explicitly managing the lifecycle of the video heartbeat instance. As a result, the `IVideoHeartbeat interface` also provides a `destroy()` method that allows for the execution of various tear-down operations (including de-allocating internal resources):

   ```
   videoHeartbeat.destroy()
   ```

   I

Note: *Graceful Termination for the VideoHeartbeat instance:*

It is important to understand that all the "track" methods exposed through the VideoHeartbeat public API are non-blocking (i.e., asynchronous). Whenever the integration code calls any of the `track*()` *methods, the VideoHeartbeat instance does very few things synchronously. The library will rush to obtain the information it needs from the player delegate (if necessary) and all of the computation work is pushed onto a job-queue that is consumed asynchronously. This means that the calls to the Videoheartbeat* `track*()` *methods are practically instantaneous. This approach ensures that the UI will not be blocked when a* `track*()` *method is called.*

This design also means that the end-of-lifecycle for the VideoHeartbeat instance requires a little extra consideration to ensure a graceful termination. For example, the following code sample works fine, technically, however it could present a timing issue:

```
// The playback of the main video has completed.
videoHeartbeat.trackComplete();

// The main video asset was unloaded.
videoHeartbeat.trackVideoUnload();

// Terminate the VideoHeartbeat instance.
videoHeartbeat.destroy();
videoHeartbeat = null;
```

The problem with this approach is that both the `trackComplete()` *and* `trackVideoUnload()` *methods return very quickly after registering their workload as additional tasks to the job-queue being consumed asynchronously. The purpose of the* `trackComplete()` *method is to sent the* `COMPLETE` *event over the network while the* `trackVideoUnload()` *wants to send the* `UNLOAD` *event. However, the immediate invocation of the* `destroy()` *method will cancel all the pending tasks inside the job-queue. The net result is that the VideoHeartbeat instance will no longer be able to send either the* `COMPLETE` *or the* `UNLOAD` *events.*

To handle this end-of-lifecycle corner case, the player delegate class includes the `onVideoUnloaded()` *callback method. This method is called by the VideoHeartbeat instance after the network call for the* `UNLOAD` *event is sent over the wire. This signals the integration layer that the tracking of the current video session is complete, and that it is safe to call the* `destroy()` *method without losing any data. The code sample below demonstrates the graceful termination of a VideoHeartbeat instance:*

```
// The playback of the main video has completed.
videoHeartbeat.trackComplete();

// The main video asset was unloaded.
videoHeartbeat.trackVideoUnload();

// Inside the custom implementation of the PlayerDelegate:
// -------------------------------------------------------
public class CustomPlayerDelegate extends PlayerDelegate {
    [...]

    @Override
    public function onVideoUnloaded() {
        // It is now safe to terminate the VideoHeartbeat instance.
        videoHeartbeat.destroy();
    }
}
```

Handle Player Events

The assumption made here is that the video player you are working with is capable of triggering a series of events through which any subscriber can be informed about what happens inside the video player itself. The following tables present the one-to-one correspondence between player events and the associated function call exposed by the public API of the video heartbeats library.

Table 1: Playback Tracking:

Operation	Method Call	Parameter List
Load the main video asset	`trackVideoLoad()`	None
Unload the main video asset	`trackVideoUnload()`	None
Playback start	`trackPlay()`	None
Playback stop/pause	`trackPause()`	None
Playback complete	`trackComplete()`	None
Seek start	`trackSeekStart()`	None
Seek complete	`trackSeekComplete()`	None
Buffer start	`trackBufferStart()`	None
Buffer complete	`trackBufferComplete()`	None

Note: *Each tracking session begins with a* `trackVideoLoad()` *and ends with a* `trackVideoUnload()`*. You should call* `trackComplete()` *before calling* `trackVideoUnload()` *if the video was completed (played to the end). If the user exited the video before its completion (e.g., switched to another video in a playlist), you should not call* `trackComplete()`*.*

Note: Methods to be called in pairs *-- The following methods must be called in pairs (that is, each* `track...Start()` *must have a corresponding* `track...Complete()`*):*

- `trackBufferStart()` *and trackBufferComplete()*
- `trackPause()` *and trackPlay() (unless the user never resumes from a pause)*
- `trackSeekStart()` *and* `trackSeekComplete()` *(with an exception: there may be multiple* `trackSeekStart()` *calls before a* `trackSeekComplete()`*)*

A `track...Start()` *does not *need* to be followed by a* `track...Complete()` *call (there may be other track method calls in between). For instance, the following sequence of track method calls is valid and describes a user who is seeking through the stream while paused, and resumes playback after two seeks:*

```
trackPause(); // Signals that the user paused the playback.

trackSeekStart(); // Signals that the user started a seek operation.

trackSeekStart(); // Signals that the user started another seek operation (before the first
 one was completed).

trackSeekComplete(); // Signals that the second seek operation has completed.

trackPlay(); // Signals that the user resumed playback.
```

Table 2: Ad Tracking:

Operation	Method Call	Parameter List
An ad starts	`trackAdStart()`	None
An ad completes	`trackAdComplete()`	None
DEPRECATED - A new ad-break (i.e., pod) starts	*`trackAdBreakStart`*	None
DEPRECATED - An ad-break completes	*`trackAdBreakComplete`*	None

Note: *Ad-tracking APIs are simplified:*

Beginning with v1.3.1/API 1, the `trackAdBreakStart()` and `trackAdBreakComplete()` methods have been deprecated. The video heartbeat library no longer requires the integration code to signal the beginning and end of an ad-break. It is sufficient to call the `trackAdStart()` and `trackAdComplete()` methods to signal the beginning and the completion of ads.

The player delegate implementation still needs to provide both the `adInfo` and the `adBreakInfo`. With each call to the `trackAdStart()` method, the video heartbeat library will make use of the player delegate to obtain these pieces of information.

Note: *The `trackAdStart()` and `trackAdComplete()` methods are the only track methods required in order to signal the beginning and completion of an ad. You do not need to (and should not) call any additional track methods to signal the transition from ad to content or vice-versa. For instance, you should not signal the pause of the main video (via `trackPause()`) when an ad starts. This is handled automatically inside the VideoHeartbeat library when you call `trackAdStart()`.*

Table 3: Chapter Tracking:

Operation	Method Call	Parameter List
A new chapter starts	`trackChapterStart()`	None
A chapter completes	`trackChapterComplete()`	None

Table 4: QoS Tracking:

Operation	Method Call	Parameter List
The quality of the video changed to a new bit rate (either automatically or manually)	`trackBitrateChange()`	None

Table 5: Error Tracking:

Operation	Method Call	Parameter List
Error at the player level	`trackVideoPlayerError()`	`String errorId` - Unique error identifier

Error at the application level	`trackApplicationError()`	`String errorId` - Unique error identifier

Implement the Player Delegate

The Player Delegate is where the integration engineer queries the video player APIs to gather the data required by the video heartbeat library.

Almost all of the information that the video heartbeat library requires from the video player is provided by the integration engineer through an implementation of the extended `PlayerDelegate` base class. This is usually the only location where the integration engineer queries the video player APIs to gather the data required by the video heartbeat library.

PlayerDelegate Abstract Class:

```
public class PlayerDelegate {
    public VideoInfo getVideoInfo();
    public AdBreakInfo getAdBreakInfo();
    public AdInfo getAdInfo();
    public ChapterInfo getChapterInfo();
    public QoSInfo getQoSInfo();


    public void onVideoUnloaded();
    public void onError(ErrorInfo errorInfo);
}
```

The methods declared by the PlayerDelegate base class match the basic abstractions based on which the video heartbeat library performs its video-analytics operations: main video, ad content, chapters, and QoS information. The integration engineer is required to fill in each of the applicable methods, and return an instance of a specific data structure (i.e., class) which acts as a simple container for the required tracking information.

Note:

The integration engineer has no control over either the sequence or the exact moment when the methods inside the player delegate are called upon by the video heartbeat code. The integration code should not make any assumptions about the order or timing in which these calls are being made. The integration engineer should make sure the player delegate is providing the most accurate information possible at any moment in time. If that is not possible, the integration code should just return `NULL` to any of the `get...()` methods defined by the player delegate.

For example: Let's assume a situation in which the video heartbeat library makes a call to the `getAdInfo()` method. However, at the level of the integration code, the player decides that it is actually no longer inside an ad. Obviously, in such a scenario, we are dealing with a synchronization issue between the player and the video heartbeat library. At this point, the integration engineer should just return `NULL`. This would allow the video heartbeat library to activate its internal recovering mechanisms and (if possible) resume the tracking of the playback for the main video.

1. Gather video content data.

   VideoInfo Class:

   ```
   public final class VideoInfo {
       public String playerName;
       public String id;
       public String name;
       public Double length;
       public Double playhead;
       public String streamType;
   }
   ```

Parameter	Required	Description
`playerName`	Mandatory	The name of the video player that is playing back the main content
`id`	Mandatory	The ID of the video asset
`name`	Optional	The name of the video asset
`length`	Mandatory	The duration of the video asset (in seconds). For Live and Linear streams this should be set to -1.
`playhead`	Mandatory	The current playhead (in seconds) inside the video asset (excluding ad content)
`streamType`	Mandatory	The type of the video asset. Use one of the constants defined in the `AssetType` class: • `ASSET_TYPE_VOD` • `ASSET_TYPE_LIVE` • `ASSET_TYPE_LINEAR`

2. Gather ad content data.

   The method which triggers ad-related tracking workflows inside the VideoHeartbeat library is:

   • `trackAdStart()` - Instructs the video heartbeat library that an ad has started.

   Calling `trackAdStart()` translates into the following internal operations in the video heartbeat library:

   • Interrogates the player delegate for the latest video, ad-break, and ad information
   • Switches into ad-tracking mode, pausing video-tracking during ad playback (until a `trackAdComplete()` call or a `trackSeekComplete()` signaling a seek outside the ad)

   The required ad-break information is defined in the `AdBreakInfo` class.

   AdBreakInfo Class:

```
public final class AdBreakInfo {
    public String playerName;
    public String name;
    public Long position;
    public Double startTime;
}
```

Parameter	Required	Description
`playerName`	Mandatory	The name of the video player responsible for playing back the current ad break. This may be the same as the video player responsible for playing back the main content.
`name`	Optional	The name of the ad break
`position`	Mandatory	The position (index) of the pod inside the main content (starting with 1)

`startTime`	Optional	The offset of the ad-break inside the main content (in seconds). Defaults to the playhead inside the main content at the moment of the `trackAdStart()` call.

The required ad information is defined in the `AdInfo` class.

AdInfo Class:

```
public final class AdInfo {
    public String id;
    public String name;
    public Double length;
    public Double playhead;
    public Long position;
    public String cpm;
}
```

Parameter	Required	Description
`id`	Mandatory	The ID of the ad asset
`name`	Optional	The name of the ad asset
`length`	Mandatory	The duration (in seconds) of the ad asset
`playhead`	Mandatory	The playhead value (in seconds) inside the ad asset (how much of the ad has played)
`position`	Mandatory	The position (index) of the ad inside the parent ad-break (starting from 1)
`cpm`	Optional	The CPM value associated with this ad

3. Gather chapter data.

   When the integration code makes a call to the `trackChapterStart()` method, the video heartbeat library will react with the following operation: it triggers a call into the "player delegate" to obtain the required information about the chapter which is about to start.

   ChapterInfo Class:

```
public final class ChapterInfo {
    public String name;
    public Double length;
    public Long position;
    public Double startTime;
}
```

Parameter	Required	Description
`name`	Optional	The name of the chapter
`length`	Mandatory	The duration (in seconds) of the chapter
`position`	Mandatory	The position (index) of the chapter inside the main content (starting from 1)
`startTime`	Mandatory	The offset of the chapter relative to the main content (in seconds)

4. Gather QoS data.

   There is a single method defined by the `IVideoHeartbeat` interface which explicitly triggers QoS-related video-tracking workflows inside the video heartbeat library: `trackBitrateChange()`. This will cause an immediate (i.e., out-of-band) HTTP heartbeat call to be sent over the network with the new values of the QoS info (which automatically include the new bitrate value). In response to the `trackBitrateChange()` call, the video heartbeat library will regularly invoke the `getQoSInfo()` method on the player delegate to ensure that the video heartbeat uses the most up-to-date information for its output tracking reports.

   QoSInfo Class:

```
public final class QoSInfo {
    public Long bitrate;
    public Double fps;
    public Long droppedFrames;
}
```

Parameter	Required	Description
`bitrate`	Mandatory	The bitrate value (expressed in bps)
`fps`	Mandatory	The frame rate (it will be truncated to its integer part)
`droppedFrames`	Mandatory	Cumulated value of the number of frames dropped by the video rendering engine. NOTE: This is not a "per-second" metric.

Note: *The QoS information is appended to all out-bound HTTP calls issued by the video heartbeat library. So, the QoS-related video-tracking workflows are triggered implicitly by all of the `track...()` methods defined by the `IVideoHeartbeat` interface. The video heartbeat library implements a "data pull" model, in which the most recent QoS data is taken when needed from the player itself through the invocation of the `getQoSInfo()` method on the player delegate.*

5. Track Errors

   Two methods are defined by the `IVideoHeartbeat` interface that allow for the tracking of ERROR events:

   - `trackVideoPlayerError()` - This method is used to track an error event that takes place inside the video player software stack. Examples of such errors include: playback errors, segment download errors, video decoding errors, DRM-related errors, etc.
   - `trackApplicationError()` - This method is used to track an error event that takes place outside the context of pure video playback. For example: an error triggered by an external application plugin (such as an integration with a social network).

   These two tracking methods are the only exceptions to the rule that says "all information is passed to the video heartbeat library via the player delegate." The error tracking methods accept a single input parameter: a String value representing the error code. This is because, most of the time, when an ERROR event gets triggered, the error information (such as the error code and description) is provided as an attachment to the error event object itself; the information needed by the video heartbeat library is readily available in the context of the event-handling code.

6. Monitor the **Error State** of the video heartbeat library.

The integration engineer is informed via the `onError()` callback that the video heartbeat library is no longer able to continue with the real-time video-tracking process. The details about the error are provided via the `ErrorInfo` object passed by the library as an input argument to the `onError()` callback method.

```
public final class ErrorInfo {
    public String getMessage();
    public String getDetails();
}
```

Method	Description
`getMessage()`	Returns a generic description of the problem category. For example: "Invalid configuration."
`getDetails()`	Returns a description of the exact root cause of the problem. For example: "URL of the tracking end-point cannot be NULL or empty string."

The player delegate also provides a way to signal back to the application layer the error states occurring inside the library itself. The video heartbeat library will switch into the ERROR State when it determines that it no longer has enough relevant information to continue with a meaningful video-tracking session. Examples of such situations include:

- Invalid configuration - For example: the absence of the RSID, an invalid URL for the tracking end-point, etc.
- Invalid information provided via the player delegate. For example: the ID of the main video is NULL or the empty string.
- Internal exception triggered inside the video heartbeat library.

Note: *While in the ERROR State, all tracking activities inside the video heartbeat library are suspended. Calling any of the `track...()` methods (except `trackVideoLoad()`) will have no effect (other than a warning message being sent to the tracing console). Getting out of the ERROR State is only possible by starting a new tracking session via a call to the `trackVideoLoad()` method (i.e., reload the main content).*

Note: *If you provide invalid configuration information, the integration engineer needs to properly re-configure the library via a call to the `configure()` method. Otherwise, calling `trackVideoLoad()` again will just cause the video heartbeat library to return to the ERROR State, because the cause of the error remains.*

Troubleshoot the Integration

To troubleshoot the video heartbeat library integration you can leverage the extensive tracing and logging mechanism that is integrated throughout the video tracking stack. Both the video heartbeat library and the AppMeasurement library are equipped with this tracing and logging infrastructure.

1. Activate tracing at runtime.

   Tracing can be activated at run-time via the `debugLogging` configuration flag:

   ```
   // Activate tracing/logging.
   configData.debugLogging = true;

   // Apply the configuration.
   videoHeartbeat.configure(configData);
   ```

   Sample trace line:

   ```
   D/[1393408109363][com.adobe.primetime.va.adb.heartbeat.realtime.clock.TimerManager](10488):
    #_onTick() ------------------- ( 2 )
   ```

   Each trace line is made up of the following sections:

- Level - There are 3 message levels defined: D (debug), W (warn) and E (error) .
- Current timestamp - The current CPU time (time-zoned for GMT)
- The fully qualified name of the class that issued the trace message
- The method in which the trace message originated (it starts with the '#' symbol)
- The actual trace message (following the '-' symbol)

2. Look at the network calls that are issued by the video heartbeat module.

   In the debug console one can look for all the debug messages that contain the `__job_id` string. The result is a view of all the network calls issued by the VideoHeartbeat core engine.

Get Library Version Information

The video heartbeat library provides an interface called Version that contains all version-related information.

Version Class:

```
public final class Version {
    public static String getVersion();
    public static String getMajor();
    public static String getMinor();
    public static String getMicro();
    public static String getPatch();
    public static String getBuild();
    public static String getApiLevel();
}
```

The public API exposed by the Version class defines a read-only version property which provides a string with the following format:

```
android-<major>.<minor>.<micro>.<patch>-<build.no>
```

Here are the rules that govern the elements in the version string:

- **`major`** - When there are major architectural changes that significantly affect the client-backend protocol, this number will be increased.
- **`minor`** - When significant new functionality is added (usually with API changes), this number will be increased.
- **`micro`** - When new functionality or a fix is available (may introduce API changes)
- **`patch`** - This is increased frequently. Bug fixes automatically increase this number.
- **`build.no`** - This is a string consisting of the first 7 hex digits identifying the Git commit from which the release package was generated

Determine version-related information on your copy of the video heartbeat library by using this API.

In addition to the library version number, the video heartbeat library exposes an API level number. This is an integer number identifying the API version in use. While the library version number tracks the addition of new features, the API level tracks the modifications of the library's public API. So, when the API level increases, the integration engineer should consult the documentation and make appropriate updates inside the integration code.

Set Up Video Analytics Reporting on the Server-side

Adobe Analytics Video Essentials requires set up on the server-side.

If you are only using the built-in Primetime player monitoring aspect of Video Analytics, you can skip this section. Reporting set up is only necessary for customers who are leveraging Adobe Analytics Video Essentials (which provides video engagement metrics). Your Adobe representative will handle most aspects of the server-side setup for Adobe Analytics reporting, but you can also see detailed documentation on the process here: *Analytics Help and Reference - Report Suite Manager*.

These are the main tasks to be accomplished for server-side setup:

- Analytics setup - Enable conversion level for RSID
- Analytics setup - Enable video tracking

The following procedure describes how to complete server-side setup:

1. Analytics Setup - Enable conversion level for RSID:
   a) Access **Admin Tools**
   b) Select **Report Suites**
   c) Select the RSID to set up
   d) Select **Edit Settings** -> **General** -> **General Account Settings**
   e) Choose **Enabled, no Shopping Cart** in the **Conversion Level** combo box
   f) Click **Save**.
2. Analytics Setup - Enable Video Tracking
   a) Access **Admin Tools**
   b) Select **Report Suites**
   c) Select the RSID to set up
   d) Select **Edit Settings** -> **Video Management** -> **Video Reporting**
   e) Click on the **Yes, start tracking** green button

Access Video Analytics Reports

Access Video Analytics reports on Adobe Analytics and on the Primetime Player Monitoring.

Video Analytics reports are routed to two different reporting platforms:

- Adobe Analytics
- Primetime player monitoring

1. Access Adobe Analytics:
   a) Select the video-tracking enabled RSID.
   b) Navigate to Video -> Video Engagement -> Video Overview.

      This brings up the overview report.

   c) Select a video clip

      This displays the minute level granularitgy drop-off report.

   For more information on Adobe Analytics setup, see *Adobe Analytics Documentation Home*.
2. Access Primetime player monitoring:
   a) Navigate to `http://rtd.adobeprimetime.com`.
   b) Log in with your credentials.

      An overview report is displayed, that shows all running videos in a list.

   c) Select a video from the overview list.

      The real-time report for the selected video is displayed.

   To view examples of video reports, see *Video Reports*

Sample Player

For an example of a real-time video tracking solution, see the sample application in the `samples` folder of your SDK installation.

Copyright

© 2014 Adobe Systems Incorporated. All rights reserved.

Video heartbeat SDK Guide for Android

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

All other trademarks are the property of their respective owners.

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