

Video heartbeat SDK Guide for Android

Contents

- Video heartbeat SDK Guide for Android - Version 1.2.03**
 - Introduction to Video Analytics.....3
 - Integrating Video Analytics4
 - Initialize / Configure Video Tracking Libraries.....5
 - Handle Player Events.....8
 - Implement the Player Delegate.....9
 - Troubleshoot the Integration.....12
 - Get Library Version Information.....13
 - Set Up Video Analytics Reporting on the Server-side.....14
 - Access Video Analytics Reports.....14
 - Copyright.....15

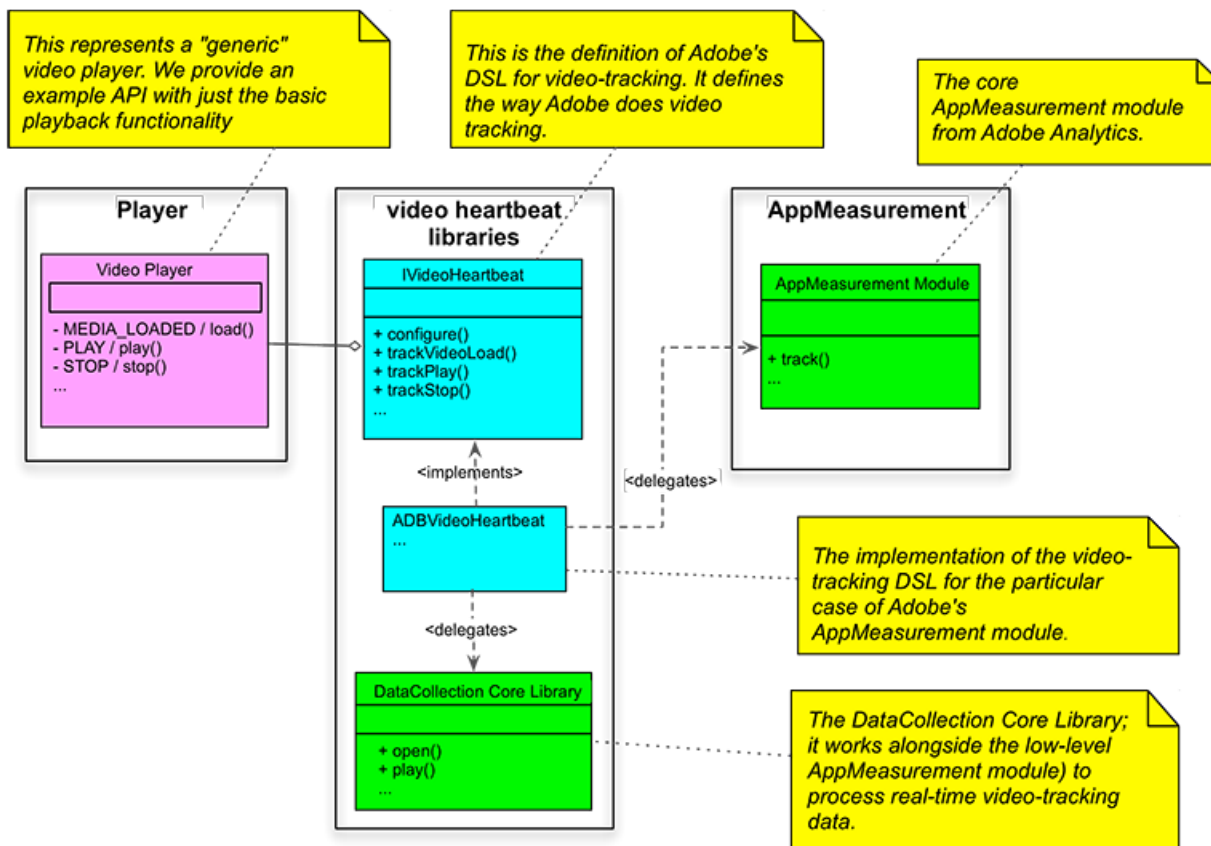
Video heartbeat SDK Guide for Android - Version 1.2.0

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:

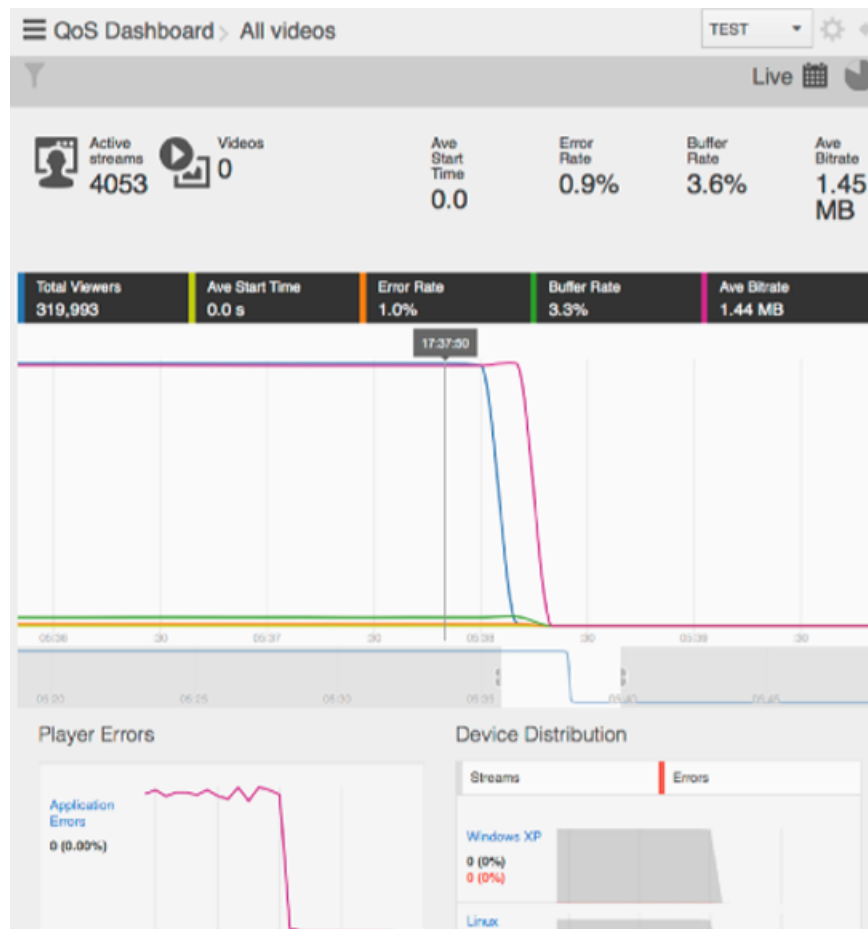


The two main components of the Video Analytics library are:

- **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.

The basic 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` - The video heartbeat library uses this library's low-level tracking method to open a viewing session in Adobe Analytics.
 - **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 uses the AppMeasurement module in the sense that it needs access to a subset of its APIs, and delegates some work to it.
- **Required Configuration / Initialization Data:**
 - **ADBMobileConfig.json** - *It is crucial that the name and the path of the configuration file remain unchanged. The JSON config file MUST be `ADBMobileConfig.json`. The path to this file MUST be `<source root>/assets`.*
 - **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)

The following procedure describes how to initialize and configure the video heartbeat library and its associated libraries. These libraries must be instantiated and configured at the application level, in the order given here:

1. Create an instance of the AppMeasurement library.
2. Create the video heartbeat library which in turn aggregates the AppMeasurement instance created during the previous step.

The application developer is in charge of properly instantiating all of the required libraries.

For the mobile implementations, the lifecycle 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 with the Desktop (AS/JS) clients, where the application developer is required to explicitly resolve the dependency chain between the video heartbeat and AppMeasurement libraries.

1. Instantiate and 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. Thus the configuration steps are as follows:

1. Fill in the JSON config file (ADBMobileConfig.json) with the appropriate values.
2. Place this file into the `assets` folder. This folder must be in the root of your application source tree.
3. Compile and build the final application
4. Deploy and run the bundled application

```
{
  "version" : "1.1",
  "analytics" : {
    "rsids" : "PUT_HERE_YOUR_OWN_ADOBE_ANALYTICS_RSID",
    "server" : "PUT_HERE_THE_ADDRESS_OF_THE_ADOBE_ANALYTICS_SERVER",
    "charset" : "UTF-8",
    "ssl" : false,
    "offlineEnabled" : false,
    "lifecycleTimeout" : 5,
    "batchLimit" : 50,
    "privacyDefault" : "optedin",
    "poi" : []
  },
  "target" : {
    "clientId" : "",
    "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 extend and instantiate the `PlayerDelegate` class.

Instantiate the video heartbeat component:

```
import com.adobe.prime.time.va.adb.VideoHeartbeat;
import com.adobe.prime.time.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 config = new ConfigData(<URL of the back-end server>,
                                   <the JOB identifier>,
                                   <the publisher identifier>);
config.ovp = <the name of the online video platform>;
config.sdk = <the version of your video player SDK>;

// Set this to true to activate the "quietMode"
config.quietMode = false;

// Set this to true to log the URLs of the output HTTP calls.
config.quietMode = true;

// Setting this to true activates the debug tracing
config.debugLogging = false;

videoHeartbeat.config(config);
```

- **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** - This is where all of the video-heartbeat calls are sent. This value is provided by Adobe in advance.
 - **jobId** - 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. This value is provided by Adobe in advance.
 - **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 of the Online Video Platform through which the content gets distributed. 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 selector that allows for the execution of various tear-down operations (including de-allocating internal resources):

```
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.

Playback Tracking:

Operation	Method Call	Parameter List
loading the main video asset	<code>trackVideoLoad</code>	None
unload the main video asset	<code>trackVideoUnload</code>	None
playback start	<code>trackPlay</code>	None
playback stop/pause	<code>trackPause</code>	None
playback complete	<code>trackComplete</code>	None
seek start	<code>trackSeekStart</code>	None
seek complete	<code>trackSeekComplete</code>	None
buffer start	<code>trackBufferStart</code>	None
buffer complete	<code>trackBufferComplete</code>	None

Ad Tracking:

Operation	Method Call	Parameter List
A new ad-break (i.e. pod) starts	<code>trackAdBreakStart</code>	None
An ad-break completes	<code>trackAdBreakComplete</code>	None
An ad starts	<code>trackAdStart</code>	None
An ad completes	<code>trackAdComplete</code>	None

Operation	Method Call	Parameter List
-----------	-------------	----------------

A new chapter starts	trackChapterStart	None
A chapter completes	trackChapterComplete	None

QoS Tracking:

Operation	Method Call	Parameter List
The ABR engine switches the current bit-rate	trackBitrateChange	None

Error Tracking:

Operation	Method Call	Parameter List
error at the player level	trackVideoPlayerError	String <code>errorId</code> - Unique error identifier
error at the application level	trackApplicationError	String <code>errorId</code> - Unique error identifier

Listen for the events issued by the player object and call the corresponding track method exposed by the library's public API.

Implement the Player Delegate

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` abstract class. This is 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 onError(ErrorInfo errorInfo);
}
```

**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 only concern himself with 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 he is actually no longer inside an ad. Obviously, in such a scenario, we are dealing with a synchronisation 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 to activate its internal recovering mechanisms and (if possible) resume the tracking of the playback for the main video.

1. Gather video content data.

```
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 (opaque string value)
length	Mandatory for VOD	The duration (in seconds) of the video asset (if available, see notes below)
playhead	Mandatory	The playhead (in seconds) value inside the video asset (excluding ad content) at the moment this method was called
streamType	Mandatory	The type of the video asset (one of the values defined in the AssetType class: VOD, LIVE or LINEAR)



Note: There are situations where the length value for the main asset is unavailable to the player. This is true for LINEAR/LIVE assets. In such cases, the integration engineer should pass NULL for the playhead value.



Note: A value of NULL for the length property is not allowed in the context of VOD events (it causes problems with reporting). If the integration engineer passes NULL for the length property when the type is VOD, the video heartbeat library will switch into the ERROR state and will suspend all tracking activities until a new call to the `trackVideoLoad()` method is made.

2. Gather ad content data.

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

Parameter	Required	Description
playerName	Mandatory	The name of the video player responsible for playing back the current advertisement break

name	Optional	The name of the ad break (opaque string value)
position	Mandatory	The position of the pod inside the main content (starting with 0)

```
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 (opaque string value)
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)
cpm	Optional	The CPM value associated with this ad



Note: The integration engineer is required to clearly mark ad-breaks with explicit calls to the `trackAdBreakStart()` and `adBreakComplete()` methods. Calling the `trackAdStart()` method outside the context of an ad-break (i.e., before calling `trackAdBreakStart()` or after calling `trackAdBreakComplete()`) will cause the video heartbeat library to switch into the **ERROR** state. Recovery is possible only by restarting the tracking session via a call to `trackVideoLoad()` method.

3. Gather chapter data.



Note: The chapter tracking APIs are not yet implemented in the 1.2 version of the video heartbeat library. Calling any of the chapter-related methods will trigger a run-time exception. They are included here so that developers can get positioned for this feature prior to its release in a future version.

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

4. Gather QoS data.

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



Note: QoS data is appended to all out-bound HTTP calls issued by the video heartbeat library. So, QoS-related video-tracking workflows are triggered implicitly by all the `track...()` methods defined by the `IVideoHeartbeat` interface. The video heartbeat library addresses this situation by defining a

trackQoSUpdateInfo() method through which the integration engineer can "inject" the latest QoS values provided by the player into the video heartbeat library at regular intervals.

5. Track Errors

```
public final class ErrorInfo {
    public ErrorInfo(String message, String details);

    public String getMessage();
    public String getDetails();
}
```

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

In addition to the methods allowing the integration engineer to provide information back to the video heartbeat library, 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 - 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.config(configData);
```

But wait, what if I am using the release version of the libraries?

Sample trace line:

```
ADBMobile Debug: [2014-02-14 at 15:34:43][ADBHeartbeatTimerManager] #_onTick() >
----- ( 2 )
```

Each trace line is made up of the following sections:

- **Level** - There are 3 message levels defined: DEBUG, WARN and 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.

If your browser of choice allows filtering (like the more recent Chrome versions do) you can filter for the `__job_id` string. The result is a very nice view of all the network calls issued by the heartbeat 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 {  
    /**  
     * The current version of the library.  
     *  
     * This has the following format: $platform-$major.$minor.$micro  
     */  
    public static String getVersion();  
  
    /**  
     * The major version.  
     */  
    public static String getMajor();  
  
    /**  
     * The minor version.  
     */  
    public static String getMinor();  
  
    /**  
     * The micro version.  
     */  
    public static String getMicro();  
}
```

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>.<patch>
```

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 a new feature or fix is available that requires changes in the public API, this number will be increased.
- **patch** - This is increased frequently. Each bug fix that makes it into the GIT repository will automatically increase this number.



Note: If either the major or the minor element changes, it means that the API has changed and a visit to the (updated) doc pages is required.

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 granularity 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](#)

Copyright

© 2014 Adobe Systems Incorporated. All rights reserved.

Video heartbeat SDK Guide

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

All other trademarks are the property of their respective owners.

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