Adobe® Primetime

Video heartbeat 1.3.1 for JavaScript SDK Guide

Updated 11 July 2014

Contents

Video heartbeat SDK Guide for JavaScript - Version 1.3.1

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.
   - **VisitorID library** (For Desktop apps only) - Uniquely labels the user of the web page hosting the video player.
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)
   - **Organization ID** (Desktop only) - This is a string value required for instantiating the Visitor component.
   - **Visitor tracking server endpoint** (Desktop only) - The URL of the back-end endpoint providing the unique label for the current user.
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. Desktop applications require an additional library, the VisitorAPI, which is a separate component on the desktop but is built-in to video heartbeats on mobile platforms.
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** - `AppMeasurement.js` - The video heartbeat library uses this library's low-level tracking method to open a viewing session in Adobe Analytics.
  - **video heartbeat library** - `VideoHeartbeat.min.js` - Contains the video heartbeat data collection core logic. This is where all of the real-time video tracking takes place. The video heartbeat library also accesses a subset of the AppMeasurement module APIs, and also delegates some work to the AppMeasurement module.
  - **Visitor API** - `VisitorAPI.js` - Uniquely labels the user of the web page hosting the video player. (This is required only for Desktop apps (ActionScript and JavaScript.)
- **Required Configuration / Initialization Data:**
  - **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
  - **Visitor Tracking Server Endpoint** - The URL of the endpoint that provides the unique "label" for the current user
  - **Organization ID** - A string value required for instantiating the Visitor library
  - **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 Video Analytics libraries must be instantiated and configured at the application level, in the order given here:

1. Instantiate and configure the VisitorAPI library.
2. Create an instance of the AppMeasurement library. This instance must aggregate the VisitorAPI instance.
3. Instantiate and configure the video heartbeat library. The video heartbeat library aggregates the AppMeasurement instance created during the previous step.

The application developer is in charge of properly instantiating all of the required libraries, while managing this dependency chain: VisitorAPI -> AppMeasurement -> video heartbeat.

The ADB global module - In order to keep the JavaScript global namespace as clean as possible, the video heartbeat library defines a specific global-level module that acts as a namespace. This module is called ADB. All

symbols exposed by this library through its public API are made available as properties attached to the ADB module. The set of exposed symbols is further segmented into sub-namespaces. Of particular interest to the application developer is the `va` sub-namespace, which publishes all symbols related to video-tracking APIs. Obtaining a reference to the `VideoHeartbeat()` constructor function can be done as follows:

```
var VideoHeartbeat = ADB.va.VideoHeartbeat
```

1. Instantiate and configure the VisitorAPI library:

   ```
   // Instantiation - Actual values are provided by Adobe
   var visitor = new Visitor("<ORG_ID>, <NAMESPACE>");

   // Configuration
   visitor.trackingServer = "URL_OF_THE_VISITOR_TRACKER_SERVER";
   ```

   Instantiating the Visitor API library requires an organization ID and a namespace (provided in advance by Adobe). These are pure string values.

   There is only one configuration option available for the VisitorAPI component, and that is the URL of the back-end endpoint of the visitor service.

2. Instantiate the AppMeasurement library:

   ```
   // Instantiation
   var appMeasurementObject = new AppMeasurement();
   appMeasurementObject.account = "ACCOUNT_NAME"; // Also known as RSID

   // Use the same value provided in the previous step for Visitor API.
   appMeasurementObject.visitorNamespace = "VISITOR-API_NAMESPACE";
   appMeasurementObject.trackingServer =
     "URL_OF_ADOBE_ANALYTICS_TRACKING_SERVER";

   // Attach the VisitorAPI to the AppMeasurement instance.
   appMeasurementObject.visitor = visitor;
   ```

3. Configure the AppMeasurement library:

   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`
   • `visitorNamespace` (Organization ID)
   • `trackingServer`

   These values are provided in advance by Adobe.

   It is important to properly set up the dependency chain: the AppMeasurement instance aggregates (depends on) the Visitor API library.

4. Instantiate and configure the video heartbeat library.

   Instantiating the video heartbeat library is a two step process, due to its dependency on these two components:

   • The AppMeasurement library
   • An implementation of the of the `PlayerDelegate` abstract class

   It is the responsibility of the video player developer to extend and instantiate the `PlayerDelegate` class.

   Instantiate the video heartbeat component:

   ```
   var VideoHeartbeat = ADB.heartbeat.VideoHeartbeat;

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

```
// We now have all ingredients to instantiate the VideoHeartbeat library.
var videoHeartbeat =
  new VideoHeartbeat(appMeasurementObject, playerDelegate);
```

Configure the newly created video heartbeat instance:

```
var ConfigData = ADB.heartbeat.ConfigData;

// Instantiate the configuration object.
var heartbeatConfig = new ConfigData(<URL_OF_BACK-END_SERVER>,
                                     <JOB_ID>,
                                     <PUBLISHER_ID>);

heartbeatConfig.ovp = <ONLINE_VIDEO_PLATFORM_NAME>; // Such as YouTube, etc.
heartbeatConfig.sdk = <VIDEO_PLAYER_SDK_VERSION>;
heartbeatConfig.channel = <DISTRIBUTION_CHANNEL_NAME>;

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

// Set this to true to activate the debug 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.*

5. 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 `HeartbeatProtocol` also provides a `destroy()` method that allows for the execution of various tear-down operations (including de-allocating internal resources):

```
videoHeartbeat.destroy()
```

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();
```

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 any unwanted side-effects. 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:
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.

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

Ad Tracking:

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

Note: *Ad-tracking APIs are simplified:*

Beginning with v1.3.1/API 1, the `trackAdBreakStart()` and `trackAdBreakComplete()` methods are deprecated. The video heartbeat library no longer requires the integration code to enclose the tracking of ad-segments inside a pair of `trackAdBreakStart()` and `trackAdBreakComplete()` calls. It is sufficient to simply call the `trackAdStart()` and `trackAdComplete()` methods when necessary. With each call to the `trackAdStart()` method, the video heartbeat library will make use of the player delegate implementation to obtain both the ad and the ad-break info from the player delegate.

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

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

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:

```
(function(va) {
    'use strict';
    /**
     * Delegate object for player-specific computations.
     *
     * NOTE: this is an abstract base class designed to be extended.
     *       Not to be instantiated directly.
     */
    function PlayerDelegate() {}
    PlayerDelegate.prototype.getVideoInfo = function() {};
    PlayerDelegate.prototype.getAdBreakInfo = function() {};
    PlayerDelegate.prototype.getAdInfo = function() {};
    PlayerDelegate.prototype.getChapterInfo = function() {};
    PlayerDelegate.prototype.getQoSInfo = function() {};
    PlayerDelegate.prototype.onVideoUnloaded = function() {};
    PlayerDelegate.prototype.onError = function(errorInfo) {};
    // Export symbols.
    va.PlayerDelegate = PlayerDelegate;
})(va);
```

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.

```
(function(va) {
    'use strict';
    /**
     * Container for video related information.
     *
     * @constructor
     */
    function VideoInfo() {
        this.playerName = null;
        this.id = null;
        this.name = null;
        this.length = null;
        this.playhead = null;
        this.streamType = null;
    }
    // Export symbols.
    va.VideoInfo = VideoInfo;
})(va);
```

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 `-1` for the playhead value.*

2. Gather ad content data.

```
(function(va) {
    'use strict';
    /**
     * Container for ad-break related information.
     *
```

```
     * @constructor
     */
    function AdBreakInfo() {
        this.playerName = null;
        this.name = null;
        this.position = null;
        this.startTime = null;
    }
    // Export symbols.
    va.AdBreakInfo = AdBreakInfo;
})(va);
```

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)
`startTime`	Optional	The offset of the ad-break inside the main content (in seconds). Default value: the value of the playhead inside the main content at the moment the call to `trackAdStart()` is made (obtained by the video heartbeat library via the PlayerDelegate implementation).

```
(function(va) {
    'use strict';
    /**
     * Container for ad related information.
     *
     * @constructor
     */
    function AdInfo() {
        this.id = null;
        this.name = null;
        this.length = null;
        this.playhead = null;
        this.position = null;
        this.cpm = null;
    }
    // Export symbols.
    va.AdInfo = AdInfo;
})(va);
```

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

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:

```
(function(va) {
    'use strict';
    /**
     * Information about chapters.
     *
     * @constructor
     */
    function ChapterInfo() {
        this.name = null;
        this.length = null;
        this.position = null;
        this.startTime = null;
    }
    // Export symbols.
    va.ChapterInfo = ChapterInfo;
})(va);
```

Parameter	Required	Description
`name`	Optional	The name of the chapter (opaque string value)
`length`	Mandatory	The duration (in seconds) of the chapter
`position`	Mandatory	The position of the chapter inside the main content (starting from 1)
`startTime`	Mandatory	The offset inside the main content where the chapter starts

4. Gather QoS data.

```
(function(va) {
    'use strict';
    /**
     * Container for QoS related information.
     *
     * @constructor
     */
    function QoSInfo() {
        this.bitrate = null;
        this.fps = null;
        this.droppedFrames = null;
    }
    // Export symbols.
    va.QoSInfo = QoSInfo;
})(va);
```

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.

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.

5. Track Errors

```
(function(va) {
    'use strict';
    /**
     * Container for error related information.
     *
     * @constructor
     */
    function ErrorInfo(message, details) {
        this.message = message;
        this.details = details;
    }
    // Export symbols.
    va.ErrorInfo = ErrorInfo;
})(va);
```

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, the absence of the VisitorAPI values, 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:

   ```
   var ConfigData = ADB.heartbeat.ConfigData;

   // Instantiate the configuration object.
   var heartbeatConfig = new ConfigData();

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

   videoHeartbeat.config(heartbeatConfig);
   ```

   Sample trace line:

   ```
   INFO [media-fork::TimerManager] > #_onTick() > ------------------- (3)
   ```

   Each trace line is made up of the following sections:

   • Level - There are 4 message levels defined: DEBUG, INFO, WARN and ERROR.

     (This info is actually marked as a small icon on the left of the trace message, rather than the text shown in the sample above.)

   • The class name 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:

```
function Version() {}
    /**
     * The current version of the library.
     *
     * This has the following format: $major.$minor.$micro
     */
    Version.getVersion = function() {...};

    /**
     * The major version.
     */
    Version.getMajor = function() {...};

    /**
     * The minor version.
     */
    Version.getMinor = function() {...};
```

```
    /**
     * The micro version.
     */
    Version.getMicro = function() {...};

    // Export symbols.
    va.Version = Version;
```

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

```
js-<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 a new feature or fix is available that requires changes in the public API, 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.

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