Adobe® Primetime

Video heartbeat SDK Guide for JavaScript - Version 1.4

Contents

Video heartbeat SDK Guide for JavaScript - Version 1.4

This guide contains instructions to instrument a custom video player with the video heartbeat SDK. The video heartbeat SDK enables real-time dashboards and other video reporting capabilities.

This guide is intended for a video integration engineer who has an understanding of the APIs and workflow of the video player being instrumented. Implementing this SDK requires that your video player provides the following:

- An API to subscribe to player events. The video heartbeat SDK requires that you call a set of simple APIs as actions occur in your player.
- An API or class that provides player information, such as video name and playhead location. The video heartbeat SDK requires that you implement an interface that returns current video information.

Integrating Video Heartbeat

Integrating Video Analytics real-time video tracking into a video player requires including video heartbeat libraries in your project, listening for player events, and implementing a player delegate to pass data from the player to the video heartbeats library.

The steps for a developer integrating the video heartbeat SDK are as follows:

1. *Acquire the required libraries* and incorporate them into your project:
   - **Video heartbeat library** - Contains the video heartbeat data collection core logic. This is where all of the real-time video tracking takes place. See *Implement VideoHeartbeat*.
   - **Plug-ins** (Optional) Plug-ins provide support for specific integrations, such as Adobe Analytics. If you are integrating using a plug-in, consult the documentation included with the plug-in or contact the vendor for additional requirements. See *Video Heartbeat Plug-ins*.
2. 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. See *Handle Player Events*.
   - **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. See *Implement the Player Delegate*.
3. Initialize and configure the video tracking libraries on your platform. See *Sample Players* for a working example.
4. Access the reports on the real-time dashboard. For details, see *Access Video Heartbeat Reports*.

Implement VideoHeartbeat

The following sections describe how to instantiate, configure, and destroy the `VideoHeartbeat` object:

- *Instantiation and configuration*
- *Destroy operations for the VideoHeartbeat library*
- *Implementing Without Plug-ins*

Instantiation and configuration

Instantiating the `VideoHeartbeat` library is a two-step process. In order to function properly, the `VideoHeartbeat` instance must be provided with the following:

- A list of required plug-ins (the contents of this list depends on the particular needs of the customer).
- An implementation of the `PlayerDelegate` abstract class. It is the responsibility of the video player developer to extend and instantiate this class. See *Implement the Player Delegate*.

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

// The list of required plug-ins is populated with pre-configured plug-ins.
var plugins = [];
// Instantiate and configure each individual plug-in and add it to the plug-in list.
...
```

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

After the instantiation is complete, you can configure the newly created instance as demonstrated in the following code sample:

```
// Instantiate the configuration object.
var heartbeatConfig = new ConfigData(<URL of the back-end server>,
                                     <the JOB identifier>,
                                     <the publisher identifier>);

heartbeatConfig.ovp = <the name of the online video platform (like youtube etc)>;
heartbeatConfig.sdk = <the version of your video player SDK>;
heartbeatConfig.channel = <the name of the distribution channel>;

// Set this to true to enable the "quietMode"
heartbeatConfig.quietMode = false;

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

videoHeartbeat.configure(heartbeatConfig);
```

The video-heartbeat configuration follows the builder pattern:

- A configuration object is created.
- The configuration object is passed as input parameter to the `config()` method exposed by the `VideoHeartbeat` instance.

There are two types of configuration parameters that can be set on the configuration object:

- *Mandatory configuration parameters*
- *Optional configuration parameters*

Mandatory configuration parameters

This set of parameters are provided as input arguments to the constructor method of the `ConfigData` class. The following table contains a description of these parameters in order:

Parameter	Description
trackingServer	URL of the tracking end-point. This is where all the video-heartbeat calls are being sent. This value is provided by Adobe in advance.
jobId	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	Name of the content publisher. This value is provided by Adobe in advance.

Optional configuration parameters

This set of parameters are provided as publicly accessible instance variables on the `ConfigData` class. Below is a summary of the available properties:

Parameter	Description
channel	Name of the distribution channel. Any string can be provided here. Default value: the empty string.
ovp	Name if the Online Video Platform though which the content gets distributed. Example: YouTube. It can be any arbitrary string. Default value: unknown.
sdk	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 video heartbeat 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 might impact performance. The integration engineer must ensure this flag is set to* `false` *for the production version of the player app. Not setting this flag at all defaults to* `false`*, so the logging mechanism is disabled by default.*

1. Create a `CustomPlayerDelegate` class that extends the `PlayerDelegate` abstract class. See *Implement the Player Delegate*.
2. Instantiate and configure each required plug-in and add it to the plug-in list.
3. Instantiate the `VideoHeartbeat` library.

Destroy operations for the VideoHeartbeat library

The integration engineer is responsible to manually and explicitly manage the lifecycle of the `VideoHeartbeat` instance. As a result, the `IVideoHeartbeat` interface also provides a destroy method which allows for the execution of various tear-down operations (such as removing event some listeners methods and de-allocating internal resources). Calling this method is very simple:

```
videoHeartbeat.destroy();
```

Graceful termination for the VideoHeartbeat library

It is important to understand that all the "track" methods exposed through the video heartbeat public API are non-blocking (i.e. asynchronous). Whenever the integration code calls any of the "track" methods, video heartbeat does very few things synchronously. If player information is needed, the library immediately attempts to obtain the needed information from the player delegate, and all of the computation work is pushed onto a job queue which is processed asynchronously. This means that the calls to the video heartbeat "track" methods return very quickly, even though any resulting data might not yet be sent. This approach helps ensure that the UI is not be blocked when a "track" method is called.

One of the consequences of this design decision is that the end-of-lifecycle for the `VideoHeartbeat` instance requires a little more consideration. Let's assume you use the following code when playback completes:

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

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

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

Technically, there is nothing wrong with this code, and it processes without errors. The problem is that both the `trackComplete()` and `trackVideoUnload()` methods return very quickly after registering their workload as additional tasks to the job queue. However, as demonstrated in the previous code example, the destroy method might execute before the `trackComplete()` and `trackVideoUnload()` are processed by the queue, and these might be canceled before the `COMPLETE` and `UNLOAD` events are sent.

To handle this end-of-lifecycle case, the player delegate class provides an additional callback method, `onVideoUnloaded()`. This method is called by the video heartbeat library after the network call for the `unload` event is sent. This is a signal for the integration layer that the tracking of the current video session is complete and that it is safe to call the `destroy()` method. 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();
}
```

Implement the `onVideoUnloaded()` inside your `PlayerDelegate` implementation.

Implementing Without Plug-ins

When external plug-ins are not required, the second parameter can be an empty plug-in list, a `null` value, or omitted completely when calling the `VideoHeartbeat` constructor method:

```
// Instantiate the VideoHeartbeat library without any plug-in support.
var videoHeartbeat = new VideoHeartbeat(playerDelegate);
```

Handle Player Events

The video player being instrumented must be capable of triggering a series of events through which any subscriber can be informed about what happens inside the video player. 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

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()` (note that if the player is closed before the pause resumes, the corresponding method might not be called)
- `trackSeekStart()` and `trackSeekComplete()` (with an exception: there may be multiple trackSeekStart() calls before a trackSeekComplete())

The `track...Start()` call are not required to be followed by a `track...Complete()` call, as there may be other `track...()` method calls in between. For example, 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.
```

Ad Tracking:

Operation	Method Call	Parameter List
An ad starts	`trackAdStart()`	None
An ad completes	`trackAdComplete()`	None

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()`

Chapter Tracking:

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

QoS Tracking:

Operation	Method Call	Parameter List
The quality of the video changed to a new bit rate (either automatically or manually)	`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 video heartbeat 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);
```

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.

```
var VideoInfo = ADB.va.VideoInfo;

var videoInfo = new VideoInfo();
videoInfo.playerName = <name of video player in use>;
videoInfo.id = <unique video identifier>;
videoInfo.name = <video friendly name>;
videoInfo.length = <duration of video content (in sec)>;
videoInfo.streamType = <one of: ADB.va.ASSET_TYPE_VOD, ADB.va.ASSET_TYPE_LIVE,
ADB.va.ASSET_TYPE_LINEAR>;

// return videoInfo from the PlayerDelegate#getVideoInfo() method.
```

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.

```
var AdInfo = ADB.va.AdInfo;

var adInfo = new AdInfo();
adInfo.id = <unique identifier for the ad content>;
adInfo.name = <ad friendly name>;
adInfo.length = <ad duration (in sec)>;
adInfo.position = <index of the ad inside the current pod (starting from 1)>;
adInfo.cpm = <ad CPM value>;

// return adInfo from the PlayerDelegate#getAdInfo() method.
```

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
`position`	Mandatory	The position of the pod inside the main content (starting with 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:

```
var ChapterInfo = ADB.va.ChapterInfo;

var chapterInfo = new ChapterInfo();
chapterInfo.name = <chapter friendly name>;
chapterInfo.length = <chapter duration (in sec>;
chapterInfo.position = <chapter index (starting from 1)>;
chapterInfo.startTime = <chapter offset (in sec)>;

// return chapterInfo from the PlayerDelegate#getChapterInfo() method.
```

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.

```
var QoSInfo = ADB.va.QoSInfo;

var qosInfo = new QoSInfo();
qosInfo.bitrate = <the stream current bitrate (in bps)>;
qosInfo.fps = <current frame rate>;
qosInfo.droppedFrames = <cumulated number of dropped frames>;

// return qosInfo from the PlayerDelegate#getQoSInfo() method.
```

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 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.

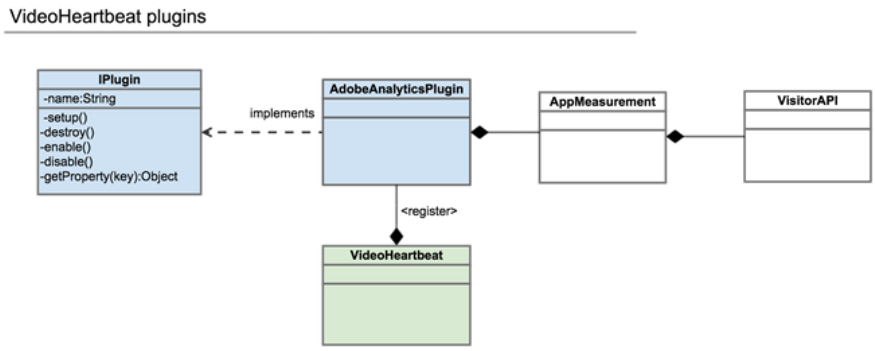
   If your browser of choice allows filtering (like the more recent Chrome versions do) you can filter for the `__job_id` string. The result is a very nice view of all the network calls issued by the heartbeat core engine.

Video Heartbeat Plug-ins

Video heartbeat implements an extensible programatic infrastructure where support for integration with external services is provided via separate modules called plug-ins.

Each plug-in takes the form of a separate library, and the use of a particular plug-in is completely optional depending on the actual needs of the customer. Video heartbeat provides a set of public APIs that allows the integration engineer to register the needed plug-ins at run-time.

For example, Adobe provides a plug-in that supports the integration with Adobe Analytics. Since all the communication with the Analytics data collection server is externalised to this plug-in, the VideoHeartbeat core library manages only the real-time data-collection (the "heartbeats"). The diagram below depicts how these components work together:



From the diagram above we can identify the main players that are involved in the video-tracking process:

- The `IPlugin` interface represents the definition of the "contract" that all plug-ins must implement in order to function as a valid plug-ins inside the video heartbeat library.
- The video heartbeat library is nothing more than a plug-in coordinator. Its main purpose is to take care of the plug-in registration process and to bootstrap and coordinate the lifecycle of all the registered plug-ins.

While this example is focused on the Adobe Analytics plug-in, other plug-ins follow this same workflow.

Note: *The use and presence of plug-ins is completely optional. The integration engineer must decide what plug-ins to instantiate and register with the video heartbeat library.*

Enable and Disable Plug-ins

Plug-in registration is part of the `VideoHeartbeat` instantiation process. The list of plug-ins is provided once when the `VideoHeartbeat` instance is created. The public API of the `VideoHeartbeat` library does not support changing the plug-in list at run-time during the life-cycle of that particular instance. In other words, the integration engineer is not allowed to add/remove plug-ins after the `VideoHeartbeat` instance is created.

However, the `IPlugin` interface allows for the run-time activation/de-activation of a particular plug-in. Below is the relevant code snippet extracted from the `IPlugin` interface definition:

```
/* global core */
(function(core) {
    'use strict';
    /**
     * Interface to be respected by all plug-ins.
     *
     * @interface
     *
```

```
     */
    function IPlugin() {}


    IPlugin.prototype.enable = function() {
        throw new Error("Implementation error: Method must be overridden.");
    };
    IPlugin.prototype.disable = function() {
        throw new Error("Implementation error: Method must be overridden.");
    };

    // Export symbols.
    core.plugin.IPlugin = IPlugin;
})(core);
```

By calling the plug-in's `enable()` and `disable()` public methods, the integration engineer is able to temporarily suspend and re-activate the processing of a particular provided that it was previously registered with the `VideoHeartbeat` library.

All plugins that are passed to the `VideoHeartbeat` constructor are enabled by default. You only need to call `enable()` if you have previously disabled a plug-in.

Available Plug-ins

The following table lists the plug-ins that are currently available for video heartbeat:

Plug-in	Description
Adobe Analytics	The Adobe Analytics plug-in let's you view video heartbeat data in Adobe Analytics reports. See *Measuring Video in Adobe Analytics using Video Heartbeat* for complete integration instructions.

Access Video Heartbeat Reports

The Video Heartbeat SDK provides reports through Primetime Player Monitoring. Additional reports are available based on the plug-ins you have implemented.

1. Log in at *http://rtd.adobeprimetime.com*.
2. Select a video from the overview list.
   The real-time report for the selected video is displayed.

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.

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.

Transitioning from Version 1.3.x

This section targets the integration engineer who already has a successful integration with video heartbeat library version 1.3.x. This section highlights the main differences between version 1.3 and version 1.4, and lists the places where changes in the integration code are required in order to upgrade to version 1.4.

Complete the following tasks to transition from Version 1.3.x:

Add the Plug-in Instantiation/Configuration Code for Adobe Analytics

The biggest change in version 1.4 is the new plug-in infrastructure. The new approach breaks the previous monolithic implementation of the video heartbeat library into multiple, independent plug-ins. The immediate consequence of this fact is that the integration engineer is now tasked with the additional work of putting all these pieces together. To summarise these changes:

- There is a new library required for the Adobe Analytics integration, the `AdobeAnalyticsPlugin` library.
- There is no longer a direct dependency between the `VideoHeartbeat` and the `AppMeasurement` libraries. In practical terms, this translates into the fact that a reference to the `AppMeasurement` instance is no longer passed to the `VideoHeartbeat` library, but to the `AdobeAnalyticsPlugin` instead.
- The integration engineer must register the Adobe Analytics plug-in with the `VideoHeartbeat` library. Without this registration step, the code inside the plug-in does not execute.

All of the points enumerated above are illustrated in the code-snippet below:

```
// Setup the Visitor API instance.
var visitor = new Visitor(<org id>, <namespace>);
visitor.trackingServer = <tracking server URL>;

// Setup the AppMeasurement instance.
var appMeasurement = new AppMeasurement();
appMeasurement.account = <account name (a.k.a rsid list)>;
appMeasurement.visitorNamespace = <visitor-api namespace>;
appMeasurement.trackingServer = <URL of the back-end server>;
appMeasurement.visitor = visitor;

var VideoHeartbeat = ADB.va.VideoHeartbeat;
var AdobeAnalyticsPlugin = ADB.va.plugins.AdobeAnalyticsPlugin
// Setup the AdobeAnalytics plug-in (pass the AppMeasurement instance to it).
var aaPlugin = new AdobeAnalyticsPlugin(appMeasurement);

// Set up the VideoHeartbeat library and register the AdobeAnalytics plug-in.
var plugins = [aaPlugin];
var playerDelegate = new CustomVideoPlayerDelegate();
var videoHeartbeat = new VideoHeartbeat(playerDelegate, plugins);
```

The code-snippet above is no more than a compilation of code-snippets that already exist in your implementation. However, it may be useful to look at it in one single piece: each setup/configuration phase is crucial for a successful integration. Each phase builds on top of the previous phase and the component dependency chain becomes obvious: VisitorAPI -> AppMeasurement -> AdobeAnalyticsPlugin -> VideoHeartbeat.

1. Import the `AdobeAnalyticsPlugin` and create an instance of `AdobeAnalyticsPlugin`.
2. Pass your `AppMeasurement` instance to the plug-in instead of to the `VideoHeartbeat` library.

Remove AppMeasurement from VideoHeartbeat Instantiation/Configuration Code

In version 1.3.x, the `VideoHeartbeat` code, due its monolithic architecture, was making direct references to symbols defined by the `AppMeasurement` library. With the new plug-in architecture, the entire interaction with the `AppMeasurement` APIs is now externalised to the `AdobeAnalyticsPlugin`.

1. Remove the `AppMeasurement` object from the `VideoHeartbeat` constructor.
2. Pass the `AdobeAnalyticsPlugin` instance created in *Add the Plug-in Instantiation/Configuration Code for Adobe Analytics* to the `VideoHeartbeat` constructor.

Remove the *playhead* Property from AdInfo

Starting with v1.4, the `playhead` property is no longer part of the `AdInfo` class. Since ad reports focus on time played and not specific fallout points, reporting the specific `playhead` location inside of an ad brought little reporting value. It is also a particularly difficult value to obtain since many players do not offer enough visibility into their timeline structure to allow its computation.

Remove all references to the `playhead` property from `AdInfo` objects.

Remove trackAdBreakStart and trackAdBreakComplete

The `trackAdBreakStart()` and `trackAdBreakComplete()` methods were deprecated in version 1.3.1. These methods are completely removed in version 1.4. As a result, upgrading to version 1.4 requires the integration engineer to remove any calls to the `trackAdBreakStart()` and `trackAdBreakComplete()` methods.

Remove all calls to `trackAdBreakStart()` and `trackAdBreakComplete()`.

Sample Players

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

Copyright

Video heartbeat SDK Guide for JavaScript

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