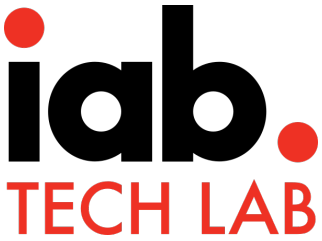

Open Measurement SDK

Migrating to OM SDK 1.3

November 2019

Table of Contents

Objective **3**

Background **3**

OM SDK 1.3 Features 3

Semantic Versioning and Deprecation Policy 3

Compatibility between Native and JavaScript layers of OM SDK 4

Suggested Migration Plan **4**

Migrating the Native Layer of Integrations **5**

Activating OM SDK 5

Ad Session Context: Declare Session Type and Content URL 6

Ad Session Configuration: Declare Creative and Impression Types 7

Ad Obstructions: Declare Obstruction Types 9

Signal "loaded" Event 10

Migrate from Video events to Media events 11

Migrating the JavaScript Layer of Integrations **12**

Ad Session Start: Declare Creative and Impression Types 12

Signal "loaded" Event 13

Migrate from Video events to Media events 14

Migrating Verification Scripts **14**

Determining Creative Type and Measurement Methodology 14

Handling Media Sessions 15

Handling JavaScript Sessions 15

Objective

Provide instructions for developers migrating their apps, SDKs, and scripts from OM SDK 1.2 to 1.3.

Background

Open Measurement SDK 1.2 was released to integrators and measurement services in July 2018. Since that release, more than 30 integrations have become certified. These integrations provide inventory measured by several vendors.

Changes in OM SDK 1.3 support some key use cases for OMID 1.3 while allowing scripts using OMID 1.2 to run correctly. Integrations (apps and SDKs) using OM SDK require code changes.

OM SDK 1.3 Features

Summary of the new features added in OM SDK 1.3:

- **Better classification**: new **ad session types and creative types** more clearly defined for robust creative measurement
- **Improved transparency:** new **Begin to Render** impression definition with the capability to use different impression types in a transparent manner
- **Simplified integration:** OM SDK **activation method** made easier to use
- **Brand safety support:** added capability to declare the **content URL** in which the ad is being shown to the user
- **Audio ad support:** new creative type added for audio and new rules to support audio measurement
- **Friendly obstruction support:** Capability to declare **friendly obstructions** and the reason for the obstruction in the verification data

Semantic Versioning and Deprecation Policy

Open Measurement SDK follows semantic versioning for the OMID JavaScript session and verification interfaces. This means that verification scripts and session scripts for a major version (1.x) will run on any integration also running OM SDK 1.x. (Of course, features introduced in later minor versions of OM SDK will not be available when running on an integration using an earlier minor version.)

The native-layer integration APIs for Open Measurement SDK don't follow semantic versioning. Instead, integration APIs are added or deprecated at minor releases and phased out gradually. This gradual deprecation gives integrators time to import OM SDK with the new API, develop against it, and test their changes.

The semantic versioning process takes at least 4 OM SDK patch releases (about 2 months):

- First, a comment will be added to the source code saying that the function is deprecated and will be removed on a given date. Release notes for OM SDK will include the same information. This stage will last 2 patch releases (1 month). It will apply to OM SDK version 1.3.0 and 1.3.1.
- Second, the deprecated function will be marked for the compiler (`@deprecated in Java`, `__attribute((deprecated))` in Objective-C). This stage also lasts for 2 patch releases. It will apply to OM SDK 1.3.2 and 1.3.3.
- Finally, the deprecated function is removed from OM SDK implementation. It will apply to OM SDK 1.3.4 and later.

Compatibility between Native and JavaScript layers of OM SDK

Apps that embed OM SDK can be out in the field for a long time. Users can be slow to download new versions of apps, and the app developers may withhold app updates. These delays result in a mix of versions for an app by a single integrator. Therefore, the OM SDK JavaScript service needs to be compatible with both old and current versions of the native layer, but the reverse of that is not required.

- Backwards compatibility of the service is guaranteed: OM SDK JavaScript version X will run on all versions of OM SDK native (iOS/Android) <= X.
- Forwards compatibility of the service is not guaranteed: OM SDK JavaScript version Y may not run on any OM SDK native > Y.

Given this policy, an integration must deploy new versions of the JavaScript service (omsdk-v1.js) **before** they deploy new versions of the native SDKs.

Suggested Migration Plan

The following four step plan is a model for a smooth transition:

1. Start with the OM SDK JavaScript service, which is backward compatible with the installed base of apps using OM SDK 1.1.x and 1.2.x.
    a. Import the OM SDK JavaScript service (omsdk-v1.js) 1.3.x into your build system.
    b. Test your existing (1.2.x) integration against the 1.3.x service. It should continue to work without any changes.
    c. Deploy the OM SDK 1.3.x JavaScript service. It should continue to work without new errors.
2. Next rebuild your integration using the new OM SDK library code with a release of OM SDK 1.3.0 or 1.3.1.
    a. Import the Android, iOS, and JavaScript libraries for OM SDK 1.3.0 or 1.3.1 into your build system. Your existing OM integration should work without changing source code.
    b. Test your software against OM SDK 1.3.0 or 1.3.1. It should function the same.
    c. Deploy your software linked against OM SDK 1.3.0 or 1.3.1. It should function the same.

3. Then port your integration to use the new OM SDK 1.3 API.
    a. Still using OM SDK 1.3.0 or 1.3.1, migrate your source code to use the new API.
    b. Test your migrated software. It should function the same, though the OMID events will include additional data.
    c. Deploy your migrated software.
4. As OM SDK releases incremental versions (1.3.2, 1.3.3, etc.), import them into your build system, test your software, and deploy. You should not need any source code changes to your already migrated software.

Migrating the Native Layer of Integrations

The instructions in the following sections provide specific code updates in the native layer app for OM SDK 1.3 integration in both Android and iOS.

Activating OM SDK

The version check is no longer needed when initializing OM SDK.

Android

Change:
```
String omsdkVersion = Omid.getVersion();
if (Omid.isCompatibleWithOmidApiVersion(omsdkVersion) {
    Omid.activateWithOmidApiVersion(omsdkVersion,
applicationContext);
}
```

To:
```
Omid.activate(applicationContext);
```

iOS

Change:
```
if ([OMIDSDK
isCompatibleWithOMIDAPIVersion:OMIDSDKAPIVersionString]) {
    [[OMIDSDK sharedInstance]
        activateWithOMIDAPIVersion:OMIDSDKAPIVersionString
        error:&error];
}
```

To:
```
[[OMIDSDK sharedInstance] activate]
```

Ad Session Context: Declare Session Type and Content URL

When starting an OMID ad session, the "ad session context" object describes the environment in which the ad will be shown. OM SDK uses the ad session context to specify the execution environment where OM SDK JavaScript code and verification scripts run. In OM SDK 1.2, there are two ways to specify that JS execution environment: either by passing a webview object owned by the integration (which creates an "html" ad session), or by having OM SDK create a private JavaScript execution context or webview (a "native" ad session).

However, there was some ambiguity for measurement providers: HTML ads were always in "html" ad sessions, but native ads could be in either "html" or "native" ad sessions, depending on whether or not the integration used a webview for running business logic. (For example, an ad integration might use a native video player but do its VAST parsing in JavaScript.) So OM SDK 1.3 introduces a new "javascript" ad session to resolve that ambiguity.

In addition to that, there is a new context parameter for "content URL". This is the deep-link URL for the app screen that is displaying the ad. This can be an Android deep link or iOS universal link. If the content URL is not known, pass null for the parameter.

Android

- If the integration called `AdSessionContext.createNativeAdSessionContext()`, simply add the `contentUrl` parameter.

- If the integration called `AdSessionContext.createHtmlAdSessionContext()`, and it renders the ad in HTML, it should continue to call that method. Just add the `contentUrl` parameter, and make sure it calls `AdSession.registerAdView()` with the same webview that it passed to AdSessionContext.

- If the integration called `AdSessionContext.createHtmlAdSessionContext()` for a native (non-HTML) ad, it must change to a "javascript" session. Call `AdSessionContext.createJavaScriptAdSessionContext()` with the same parameters, and make sure `AdSession.registerAdView()` specifies the native view that renders the ad.

iOS

- If the integration called `-[OMIDAdSessionContext initWithPartner:script:resources:customReferenceIdentifier:error:]` to create a native session, call the new version that adds a `contentUrl` parameter..

iOS (continued)

- If the integration called –`[OMIDAdSessionContext initWithPartner:webView:customReferenceIdentifier:error:]`, and it renders an ad in HTML, few changes are needed. Just add the `contentUrl` parameter, and make sure it calls –`[OMIDAdSession setMainAdView:]` with the same webview that it passed to OMIDAdSessionContext.

- If the integration called –`[OMIDAdSessionContext initWithPartner:webView:customReferenceIdentifier:error:]` for a native (non-HTML) ad, it must change to a "javascript" session. Call –`[OMIDAdSessionContext initWithPartner:javaScriptWebView:contentUrl:customReferenceIdentifier:error:]` with the same parameters, and make sure –`[OMIDAdSession setMainAdView:]` specifies the native view that renders the ad.

Ad Session Configuration: Declare Creative and Impression Types

When starting an OMID ad session, the "ad session configuration" object includes information about the ad that is about to be shown. OM SDK 1.3 adds parameters to explicitly declare the creative type of the ad session and how it will signal impressions. The old constructor without the new parameters is being deprecated.

In earlier versions, the creative type was implied by the "media events owner" parameter, so for many integrations, it may be possible to set creativeType to display when the media events owner is "none", and to video otherwise – this is shown in the example below. An alternative (not shown) for integrations that set "impression owner" to JavaScript is to add code to their JS session script to set the creative type as well.

The new parameter for impression type doesn't have precedent in older versions of OM SDK, so the sample code below leaves it as an exercise for the integrator. If the integration has set the "impression owner" to JavaScript, it can set the impression type in their JS session script rather than in the native layer.

Android

Change:

```
AdSessionConfiguration mySessionConfig =
createAdSessionConfiguration(
    impressionOwner, mediaEventsOwner,
    isolateVerificationScripts);
```

To:

```
boolean isHtmlAdSession = ...;
ImpressionType impressionType = ...;
CreativeType creativeType;
// NOTE: If you want to let the JS session script declare
creative type,
// omit the `if` statement and set creativeType to
// CreativeType.DEFINED_BY_JAVASCRIPT.
if (mediaEventsOwner == Owner.NONE) {
```

```
  creativeType = isHtmlAdSession
      ? CreativeType.HTML_DISPLAY :
CreativeType.NATIVE_DISPLAY;
} else {
  creativeType = CreativeType.VIDEO;
}
AdSessionConfiguration mySessionConfig =
createAdSessionConfiguration(
    creativeType, impressionType, impressionOwner,
mediaEventsOwner,
    isolateVerificationScripts);
```

iOS

Change:

```
OMIDAdSessionConfiguration mySessionConfig =
    [[OMIDAdSessionConfiguration alloc]
           initWithImpressionOwner:impressionOwner
                  videoEventsOwner:videoEventsOwner
        isolateVerificationScripts:isolateVerificationScripts
                             error:&error];
```

To:

```
BOOL isHtmlAdSession = ...;
OMIDImpressionType impressionType = ...;
OMIDCreativeType creativeType;
// NOTE: If you want to let the JS session script declare
// creative type,
// omit the `if` statement and set creativeType to
// OMIDCreativeTypeDefinedByJavaScript.
if (mediaEventsOwner == OMIDNoneOwner) {
  creativeType = isHtmlAdSession
      ? OMIDCreativeTypeHtmlDisplay :
OMIDCreativeTypeNativeDisplay;
} else {
  creativeType = OMIDCreativeTypeVideo;
}
OMIDAdSessionConfiguration mySessionConfig =
    [[OMIDAdSessionConfiguration alloc]
             initWithCreativeType:creativeType
                   impressionType:impressionType
                  impressionOwner:impressionOwner
                 videoEventsOwner:videoEventsOwner
        isolateVerificationScripts:isolateVerificationScripts
                             error:&error];
```

Ad Obstructions: Declare Obstruction Types

OM SDK does a check of full native-layer view hierarchy to consider views that overlap the ad view. Some apps need to place overlay views on top of the ad as part of the ad experience (e.g. video controls, close button, gesture tracking, etc.). Such views don't block visibility of the ad, so the OM SDK integrator can declare those overlays as "friendly obstructions", and OM SDK will exclude them from visibility calculations.

To discourage abuse of friendly obstructions, OM SDK 1.3 requires that integrators provide additional information that describes the purpose of each friendly obstruction. The supported purposes:

- VIDEO_CONTROLS: Views related to interacting with a video (e.g. play/pause buttons)
- CLOSE_AD: Views relating to closing an ad (e.g. close button)
- NOT_VISIBLE: Views that are not visibly obstructing the ad but may seem so due to technical limitations
- OTHER: Views that are obstructing for any purpose not already described.

Android

Change:

```
adSession.addFriendlyObstruction(obstructingView);
```

To:

```
try {
  adSession.addFriendlyObstruction(
      obstructingView,
      <enum representing purpose of obstruction>,
      null);
} catch (RuntimeException ex) {
  // Handle illegal argument or state error
}
```

iOS

Change:

```
[adSession addFriendlyObstruction:obstructingView];
```

To:

```
if (![adSession addFriendlyObstruction:obstructingView
        purpose:<enum representing purpose of obstruction>
        detailedReason:nil
        error:&error]) {
    // Handle illegal argument or state error
}
```

Signal "loaded" Event

OM SDK 1.2 included a "loaded" event for video creatives. OM SDK 1.3 expands the domain of "loaded" events to support display creatives as well. Integrations that show display ads must trigger an OMID "loaded" event before they signal an impression.
The "loaded" event is sent via an AdEvents object. This is a change from OM SDK 1.2, which used a VideoEvents object to send the "loaded" event. Similarly, the integration declares the layer from which the "loaded" event will be sent via the "impression owner" parameter to the Ad Session Configuration; in OM SDK 1.2 it was the "video events owner" that determined the layer.

Android

For video sessions that send events from the native layer,

Change:
```
VideoEvents videoEvents =
VideoEvents.createVideoEvents(adSession);
// ...
videoEvents.loaded(vastProperties);
```

To:
```
AdEvents adEvents = AdEvents.createAdEvents(adSession);
// ...
adEvents.loaded(vastProperties);
```

For display sessions that send impressions from the native layer, precede your impression event with a loaded event. Ideally this occurs early in the ad session when the creative is loaded, but it can be sent just before the impression:

```
AdEvents adEvents = AdEvents.createAdEvents(adSession);
// ...
adEvents.loaded();
// ...
adEvents.impressionOccurred();
```

iOS

For video sessions that send events from the native layer,

Change:
```
OMIDVideoEvents videoEvents =
  [[OMIDVideoEvents alloc] initWithAdSession:adSession
  error:&error];
  // ...
  [videoEvents loadedWithVastProperties:vastProperties
  error:&error];
```

To:
```
OMIDAdEvents adEvents =
  [[OMIDAdEvents alloc] initWithAdSession:adSession
  error:&error];
  // ...
  [adEvents loadedWithVastProperties:vastProperties
  error:&error];
```

For display sessions that send impressions from the native layer, precede your impression event with a loaded event. Ideally this would occur as early in your ad session as the creative is loaded, but it is OK to send it just before the impression:

```
OMIDAdEvents adEvents =
  [[OMIDAdEvents alloc] initWithAdSession:adSession error:&error];
// ...
[adEvents loadedWithError:&error];
// ...
[adEvents impressionOccurredWithError:&error];
```

Migrate from Video events to Media events

OM SDK 1.3 supports audio (non visual) ad sessions. Video ad sessions will use a new MediaEvents class to generate events instead of the VideoEvents class, but the semantics of the events are unchanged.

Android

For video sessions that send events from the native layer, change

Change:
```
VideoEvents videoEvents =
VideoEvents.createVideoEvents(adSession);
// ...
videoEvents.someEvent();
```

To:
```
MediaEvents mediaEvents =
MediaEvents.createMediaEvents(adSession);
// ...
mediaEvents.someEvent();
```

iOS

For video sessions that send events from the native layer,

Change:
```
OMIDVideoEvents videoEvents =
  [[OMIDVideoEvents alloc] initWithAdSession:adSession
  error:&error];
  // ...
  [videoEvents someEvent];
```

To:
```
OMIDMediaEvents mediaEvents =
  [[OMIDMediaEvents alloc] initWithAdSession:adSession
  error:&error];
  // ...
  [mediaEvents someEvent];
```

Migrating the JavaScript Layer of Integrations

Integrations using "html" or "javascript" ad sessions must update their OMID session scripts for version 1.3. Note that if an integration's session script can run in apps using older versions of their OM SDK native-layer integration, the integrator needs to make their session scripts backward compatible, so that it can run on both their 1.2.x and 1.3.x install base.

Ad Session Start: Declare Creative and Impression Types

The native layer of an integration can let its JavaScript layer declare the creative type for a session by passing the special DEFINED_BY_JAVASCRIPT value for the creativeType parameter in AdSessionConfiguration. If it does that, it must set the creative type in its session script handler for the "startSession" event.

Similarly, the native layer can specify "DEFINED_BY_JAVASCRIPT" for the impression type, and the session script must set that value in its "startSession" event handler.

To support backwards compatibility, the session script needs to check that it can set these values. Otherwise OM SDK will throw a JavaScript exception.

The event handler could look like this:

```
const myCreativeType = "video";
const myImpressionType = "beginToRender";
// ...
adSession.registerSessionObserver((event) => {
  if (event.type === "sessionStart") {
    // ...
    if (event.data.creativeType === "definedByJavaScript") {
      adSession.setCreativeType(myCreativeType);
    }
    if (event.data.impressionType === "definedByJavaScript") {
      adSession.setCreativeType(myImpressionType);
    }
    // ...
  } else if (event.type === "sessionError") {
    // ... handle error
  } else if (event.type === "sessionFinish") {
    // ... clean up
  }
});
```

Signal "loaded" Event

If the native integration sets "impressionOwner" configuration property to JavaScript, the session script must trigger a "loaded" event for both video and display ad sessions. The event should occur as soon as possible after the creative has loaded and the creative type is known.

If the native layer sets the creative and impression types, the session script can trigger the "loaded" event at any time, including before it receives the "sessionStart" event. But if the native layer sets the creative type or impression type to DEFINED_BY_JAVASCRIPT, the session script must wait until after receiving the "sessionStart" event and specifying the creative or impression types.

Note that in OM SDK 1.2, the integration triggered the "loaded" event only for video sessions, and it was under the control of the "videoEventsOwner" configuration property. For easier backwards compatibility, a session script using the 1.3 session client library can send a "loaded" event for all video or display sessions, regardless if the native layer is using OM SDK 1.2.x or 1.3.x. The event will be ignored in a display session when the native layer is using 1.2.x.

The code to handle creative loading could look like the following. See the notes in the code for places to simplify for different integration scenarios.

```
let isCreativeLoaded = false;
// NOTE: You can skip defining vastProperties if this script only
// supports display sessions.
let vastProperties = null;
// NOTE: You need the following flag only when the native layer
sets
// impression or creative type to DEFINED_BY_JAVASCRIPT.
let isSessionStarted = false;
let didSendLoadedEvent = false;
const adSession = new AdSession(...);
const adEvents = new AdEvents(adSession);

function sendLoadedEvent() {
  if (didSendLoadedEvent) return;
  if (!isSessionStarted) return;
  // NOTE: If this script only needs to support display sessions,
  // you can call adEvents.loaded() instead.
  adEvents.loaded(vastProperties);
  didSendLoadedEvent = true;
}

// Call this function when the creative has loaded.
function onCreativeLoaded(creativeData) {
  isCreativeLoaded = true;
  // NOTE: You must not construct vastProperties for display
creatives.
  vastProperties = new VastProperties(... fields of creativeData
...);
```

```
  sendLoadedEvent();
}

// ...
adSession.registerSessionObserver((event) => {
  if (event.type === "sessionStart") {
    // ...
    // NOTE: If needed, call setCreativeType or setImpressionType
first.
    if (isCreativeLoaded) {
      sendLoadedEvent();
    }
    // ...
  } else ...
});
```

Migrate from Video events to Media events

OM SDK 1.3 supports audio (non visual) ad sessions. Video ad sessions will use a new MediaEvents class to generate events instead of the VideoEvents class, but the semantics of the events are unchanged.

If your session script sends video events, change

Change:
```
const videoEvents = new VideoEvents(adSession);
// ...
videoEvents.someEvent();
```

To:
```
const mediaEvents = new MediaEvents(adSession);
// ...
mediaEvents.someEvent();
```

Migrating Verification Scripts

Per the semantic versioning policy, verification scripts using only OMID 1.2 features do not need any modification to run on integrations using OM SDK 1.3. For a verification script, the changes in OM SDK 1.3 are purely additive – they will receive the same data as in 1.2, plus some additional data that 1.3 provides. Nonetheless, verification providers are encouraged to use OMID 1.3 data when present.

Determining Creative Type and Measurement Methodology

In OMID 1.2, the only standard way to determine creative type and measurement methodology (whether to use video or display techniques) was to examine the "mediaType" property of the "impression" event.. Since the criterion for the impression event is under the control of the integration, it could fire at different times in different integrations, such as when the creative loaded, or became visible, or even when it became viewable.

OMID 1.3 defines a "loaded" event for display, video and audio ad sessions, and it includes a "creativeType" attribute on that event. Verification scripts can use that to determine measurement methodology rather than waiting for the "impression" event.

However, verification scripts will still need to maintain backward compatibility with OMID 1.2, because apps that don't upgrade to OM SDK 1.3 will be in the field for some time.

The check is straightforward:

- If the "sessionStart" event includes a "creativeType" attribute, use OMID 1.3 techniques, and check the "creativeType" on the "loaded" event. (You should not use the "creativeType" value on the "sessionStart" event, because it could have the value "definedByJavaScript". That's a signal for session scripts to update the creative type before triggering a "loaded" event.)
- if the "sessionStart" event lacks a "creativeType" attribute, fall back to OMID 1.2 techniques, and check the "mediaType" on the "impression" event.

Handling Media Sessions

OMID 1.3 adds support for audio ads in a backward-compatible fashion with video ads.

- The same event types (e.g. "start", "firstQuartile", "pause", "complete", etc.) are sent for both audio sessions and video sessions.
- Verification scripts running on an OM SDK 1.3 implementation can register a "media" event handler, and they will receive all audio and video events. This is the same as registering a "video" event handler on either OM SDK 1.2 or 1.3.
- Error events (event.type = "error") can have event.data.errorType = "media" in addition to errorType = "video".

Audio ad sessions do not send ad geometry or visibility information. If a verification script resource is attached to an audio ad (e.g. in DAAST), then it should be prepared to handle an audio ad events without "adView" data. Otherwise (i.e. it is only attached to video or display ads), no changes are needed.

Handling JavaScript Sessions

OMID 1.3 declares a new ad session type of "javascript". This is largely the same as an ad session type of "html", except that the webview does not have the creative. The integration chooses a "javascript" session when it renders the ad creative in the native layer but runs session logic in a webview.

When the "sessionStart" event has data.context.adSessionType = "javascript", the verification script should not expect events to include creative measurement attributes like "measuringSlotElement" or "measuringVideoElement", nor should it try to search for the creative in the webview DOM.