mPulse SDK

Version 21.40
June 30th, 2020
Size: 198KB

Android Integration Guide

1 Summary

This document details the process of integrating mPulse SDK with your Android application to send beacons from your application to mPulse.

2 Introduction

The mPulse Android SDK lets you send Custom Metric and Custom Timer beacons to mPulse. For each beacon, you can set the View Group, A/B test, and Custom Dimensions.

The mPulse Android SDK also monitors all network activity performed by the application and its libraries. Each network request can be monitored individually and its performance data will be sent on a beacon.

- Network requests will be automatically instrumented as long as they are sent from java.net.URLConnection.
- If network requests are made through OkHttpClient or the library that uses it (examples are Picassa, Retrofit etc.), the OkHttpClient needs to be configured to use the mPulse interceptor.

The SDK also collects network-related statistics and are sent periodically to a server which can be accessed via portal.

The mPulse Android SDK also allows you to monitor Actions, which are distinct user interactions. Actions can be started at any time via the SDK, and stopped either programmatically or automatically by the SDK once all network activity has quieted down. All network requests during the Action will be included on the Action beacon's Waterfall.

Beacons

App configuration is performed by an App Administrator in mPulse Central. The mPulse Android SDK is not used to manage permissions or to change the configuration of an app.

3 Integration with your Android Application

Before using the mPulse Android SDK, you will need to have a mPulse app and an associated API Key. For information on how to setup the mPulse app and the API Key, go to mPulse setup. Once your app has been configured in mPulse, you can use the mPulse Android SDK.

3.1 Requirements and Dependencies

The SDK supports API 15 and above. The target/compile SDK should be atleast set to API 28 and it is also required by Google for any app updates from November 2019. Google no longer maintains support libraries up until version 28 and has made androidx the default support library in Android Studio. mPulse SDK has migrated from Android Support libraries to androidx-packaged library artifacts. This will not only make our SDK better, but also allows us to use the latest Jetpack features. If you have not migrated to androidx library artifacts, add the following two lines to your gradle.properties file to use mPulse SDK.

```
android.useAndroidX=true
android.enableJetifier=true
```

Androidx also requires minimum build tools version to be set **'com.android.tools.build:gradle:3.2.0'** in the root *build.gradle* file. The minimum gradle version should be set at **gradle-4.6** in the *gradle-wrapper.properties*.

3.2 Include the library

In Android Studio, open the build.gradle file in the app directory (not the one in the root folder) and edit the dependencies sub-section to include .AAR file.

```
dependencies {
        implementation 'com.akamai.android:aka-mpulse:21.40.+'
}
```

Use the latest version of aka-mpulse from jcenter. The project gradle file should have jcenter() by default. If not, add it as shown below.

```
allprojects {
```

```
   repositories {
       jcenter()
    }
}
```

The SDK requires the following permissions for full functionality:

```
<manifest . . . >
     <uses-permission android:name="android.permission.INTERNET" />
     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
</manifest>
```

3.3 Register the SDK

mPulse SDK is initialized automatically and the client does not have to explicitly initialize the SDK. After initialization, the SDK needs to be registered with an API key. Client AndroidManifest.xml will need to be updated in order to complete the SDK integration.

```
<application
....

<!-- Refers to sdk init file in res/xml/ -->
<meta-data
android:name="com.akamai.android.sdk"
android:resource="@xml/akamai_sdk_init" />


....
</application>
```

The SDK uses an xml file to look up the license key to authorize the client app. This information is stored in the file android_sdk_init.xml in the client app's resources folder. The sample file is shown below.

In …./main/res/xml (create if it doesn't exist!) folder, add a new file, android_sdk_init.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<com_akamai_sdk_init>
<!-- SDK license key created on portal→
```

```
<com_akamai_mpulse_license_key></com_akamai_mpulse_license_key>
 </com_akamai_sdk_init>
```

4 API Reference

In this section, we see examples of how to use APIs provided by the SDK.

4.1 Accessing mPulse instance

You can access the instance by calling MPulse.sharedInstance().

```
import com.akamai.mpulse.android.MPulse;
...
MPulse mpulse = MPulse.sharedInstance();
```

4.2 Instrument the network calls

mPulse SDK collects performance data from the app and sends the data back to the mPulse servers via a beacon. Beacons are invisible network requests that contain performance data and other page-load characteristics.

Sending beacons from SDK can be suspended from the portal.

4.2.1 Using SDK with Android HttpsUrlConnection/HttpUrlConnection

By default, after successful initialization the SDK intercepts all HttpUrlConnection and HttpsUrlConnection requests. There is no additional code change required by the application. SDK intercepts these requests by setting a global URLStreamHandlerFactory. All the relevant network statistics related to each request will be captured by the SDK.

4.2.2 Using SDK with Third party HTTP client wrappers

In order to accelerate traffic originating from third-party http clients like Okhttp, Retrofit, and Picasso, an interceptor needs to be added to the request. Sample interceptor classes for all

these libraries have been added in **the wrappers folder under mPulse** in the Akamai_Android_SDKs.zip.

OkHttp

An interceptor for OkHttp needs to be added to the OkHttpClient.Builder as below

```
OkHttpClient client = new OkHttpClient.Builder()
       .addInterceptor(new AkaOkHttpAppInterceptor())
       .build();
Request request = new Request.Builder()
       .url(uri)
       .build();
Response response = client.newCall(request).execute();
```

Retrofit - Version 2

An interceptor for Retrofit 2 needs to be added as the client to retrofit as shown below

```
Retrofit retrofit = new Retrofit.Builder()
       .baseUrl(uri)
       .addConverterFactory(GsonConverterFactory.create())
       .client(AkaRetrofit2Client.getClient())
       .build();
Retrofit2Api client =  retrofit.create(Retrofit2Api.class);
Call<ResponseBody> call = client.get(path);
```

Picasso - version 2.71828 and above

A downloader for Picasso needs to be added as below.

```
Picasso picasso = new Picasso.Builder(getContext())
       .downloader(new OkHttp3Downloader(AkaPicassoDownloader.getClient()))
       .build();
Picasso.setSingletonInstance(picasso);
```

Glide

A downloader for Glide needs to be added as below.
Sample AppGlide module:

```
@GlideModule
@Excludes(OkHttpLibraryGlideModule.class)
public class AppGlide extends AppGlideModule {

   @Override
   public void registerComponents(@NonNull Context context, @NonNull Glide glide, @NonNull
Registry registry) {
       super.registerComponents(context, glide, registry);
       ...
       registry.replace(GlideUrl.class, InputStream.class, new
OkHttpUrlLoader.Factory(AkaGlideClient.getClient()));
   }
}
```

Volley

For volley, an interceptor for OkHttp needs to be added in OkHttpStack class which extends BaseHttpStack, then it should be added to the OkHttpClient.Builder as below:
Sample OkHttpStack class and usage:

```
public class OkHttpStack extends BaseHttpStack {
...
OkHttpClient.Builder clientBuilder = new OkHttpClient.Builder();
clientBuilder.addInterceptor(new AkaOkHttpAppInterceptor());
OkHttpClient client = clientBuilder.build();
...
}

// Volley queue registration
Volley.newRequestQueue(ctx.getApplicationContext(), new OkHttpStack());
```

4.2.3 Network Request Filtering

Network Request Filtering allows developers to selectively monitor the performance of specific network requests in the application. This means you can gather performance data for requests that are important to the user experience, while ignoring requests that are not relevant, such as other library's analytics beacons.

You can modify the state of network request monitoring through either the mPulse UI or via the SDK at runtime. Network request monitoring can be in one of three modes:

- ALL (Blacklist mode) (default): All network requests are instrumented. You can exclude specific URLs.
- MATCH (Whitelist mode): Only those requests matching your criteria will be instrumented. Your Whitelist will control which URLs are included.
- NONE: No network requests are instrumented.

The mPulse Android SDK allows you to control the state of network request monitoring during runtime using the following SDK APIs:
Disable network request beacons (ie. set the mode to NONE):

```
MPulse.sharedInstance().disableNetworkMonitoring();
```

Enable network request beacons (ie. set the mode to ALL, Blacklist mode):

```
MPulse.sharedInstance().enableNetworkMonitoring();
```

Enable Whitelist-based network request filtering (ie. set the mode to MATCH, Whitelist mode):

```
MPulse.sharedInstance().enableFilteredNetworkMonitoring();
```

Using any of the above APIs will clear any previously configured Whitelist or Blacklists filters added via `addWhiteListFilter()`, `addBlackListFilter()`, `addUrlWhiteListFilter()` or `addUrlBlackListFilter()`.

When these APIs are used, they will take precedence over the mPulse UI configuration, for the duration of the runtime of the application.

4.2.3.1 Programmatically Extending the Filters

The mPulse Android SDK maintains a Blacklist and Whitelist (for ALL and MATCH modes respectively), and it will use these lists to filter URLs. The mPulse UI allows you to control the Blacklist and Whitelist URLs within the app's configuration dialog. In addition, you can programmatically add URL filters at runtime. These filters will be amended to the app configuration from the mPulse UI.Examples of adding URL filters to the Whitelist and Blacklist:

```
// Adds a RegEx pattern to the list of Whitelist filters
MPulse.sharedInstance().addUrlWhiteListFilter("NameOfFilter", ".*pattern.*");


// Adds a RegEx pattern to the list of Whitelist filters (without a filter name)
```

```
MPulse.sharedInstance().addUrlWhiteListFilter(".*pattern.*");

// Adds a RegEx pattern to the list of Blacklist filters
MPulse.sharedInstance().addUrlBlackListFilter("NameOfFilter", ".*pattern.*");

// Adds a RegEx pattern to the list of Blacklist filters (without a filter name)
MPulse.sharedInstance().addUrlBlackListFilter(".*pattern.*");

// If you wish to only tag a network request with a specific View Group using a pattern
MPulse.sharedInstance().addViewGroupFilter("NameOfFilter", ".*pattern.*", "My View Group");
```

Here's an example of using an advanced filter within your application. The MPFilter class is given direct access to the MPBeacon, and should set result.setMatched(true) when matched.

When in `ALL` (Blacklist) mode, matched means the network request **will not** be monitored.
When in `MATCH` (Whitelist) mode, matched means the network request **will** be monitored.

```
import com.akamai.mpulse.core.filter.MPFilter;
import com.akamai.mpulse.core.filter.MPFilterResult;
import com.akamai.mpulse.core.beacons.MPBeacon;
import com.akamai.mpulse.andriod.beacons.MPApiNetworkRequestBeacon;

// In your code:
MPFilter filter = new MPFilter()
{
  @Override
  public MPFilterResult match(MPBeacon beacon)
  {
    MPFilterResult result = new MPFilterResult();

    // Do something here to either drop or include this beacon
    if (beacon != null && beacon instanceof MPApiNetworkRequestBeacon)
    {
      MPApiNetworkRequestBeacon networkBeacon = (MPApiNetworkRequestBeacon) beacon;
      if (networkBeacon.getUrl().matches(".*pattern.*"))
      {
        result.setViewGroup("Example View Group");
        result.setMatched(true);
      }
    }
    return result;
  }
};
```

For easier interaction with URLs you also have the option to use MPUrlFilter which will pass a String to the match() method:

```
// Imports:
```

```
import com.akamai.mpulse.android.filter.MPUrlFilter;
import com.akamai.mpulse.core.filter.MPFilterResult;

// In your code:
MPUrlFilter filter = new MPUrlFilter()
{
  @Override
  public MPFilterResult match(String url)
  {
    MPFilterResult result = new MPFilterResult();

    if (url != null && !url.equals(""))
    {
      if (url.contains("example.com"))
      {
        result.setViewGroup("example");
        result.setMatched(true);
      }
    }

    return result;
  }
};
```

Now that the filter has been created, you can add it to the Whitelist or Blacklist.
If you wish to include the filter in the Whitelist, setting matched true (MPFilterResult.setMatched(true)) on the MPFilterResult for match(MPBeacon beacon) will tell mPulse to monitor the request, while setting matched to false (MPFilterResult.setMatched(false)) will tell mPulse to ignore the request.

```
MPulse.sharedInstance().getFilterManager().addWhiteListFilter("NameOfFilter", filter);
```

Conversely if you wish to include the filter in the Blacklist, a setting matched to false (MPFilterResult.setMatched(false)) on the MPFilterResult for match(MPBeacon beacon) will tell mPulse to monitor the request, while setting the matched to true (MPFilterResult.setMatched(true)) will tell mPulse to ignore the request.

```
MPulse.sharedInstance().getFilterManager().addBlackListFilter("NameOfFilter", filter);
```

Should a viewGroup already be configured via the SDK (using MPulse.sharedInstance().setViewGroup(String viewGroup)) the viewGroup set by MPFilterResult via MPFilterResult.setViewGroup("myviewgroup") will override it. A value of null or "" (empty string) will not change the viewGroup on the beacon.

4.3 Send a Custom Timer

A Custom Timer can be based on any measurable user-defined duration. Custom Timers must be defined in the App dialog before use.

The mPulse Android SDK can be used to send a Custom Timer. You can track the time it took for an action to occur, such as an image upload or an attachment file download, using Custom Timers.

At the start of your action, call startTimer() by giving it a timerName. startTimer() will return a unique Timer ID (String) and will keep track of the start time:

```
String timerID = MPulse.sharedInstance().startTimer("MyTimer");
// -> "MyTimer-d4d67062-7064-42b5-85ed-4e69d8824ef9"
```

At the end of your action, call stopTimer() by passing in the Timer ID. mPulse stops the timer and sends a beacon to the server:

```
MPulse.sharedInstance().stopTimer(timerID);
```

You may also directly specify a timer name and value using sendTimer(name, value):

```
// value is in milliseconds
MPulse.sharedInstance().sendTimer("MyTimer", 4);
```

The value passed to sendTimer() in Android is a long (in milliseconds).

By default, the View Group, A/B Test and Custom Dimensions for the timer will be copied at the time startTimer() was called. If you wish to use the View Group, A/B Test and Custom Dimensions copied when stopTimer() was called, you can specify true for the updateDimensions parameter:

```
// stops the timer and updates dimensions
MPulse.sharedInstance().stopTimer(timerID, true);
```

If you wish to cancel a timer (and not send a beacon to mPulse), you can call cancelTimer():

```
// cancel the timer
MPulse.sharedInstance().cancelTimer(timerID);
```

Both startTimer() and sendTimer() accept a final parameter of MPulseMetricTimerOptions, which controls the behavior of Custom Timers while an Action is ongoing. See the Actions documentation for details:

```
MPulseMetricTimerOptions options = new MPulseMetricTimerOptions();

// include on the Action beacon (instead of sending a separate beacon)
options.duringAction =
MPulseMetricTimerOptions.DuringAction.INCLUDE_ON_ACTION_BEACON;

// if the same Custom Timer was used twice on this Action, SUM the results
options.onActionDuplicate = MPulseMetricTimerOptions.OnActionDuplicate.SUM;

String timerID = MPulse.sharedInstance().startTimer("MyTimer", options);

// or

MPulse.sharedInstance().sendTimer("MyTimer", 100, options);
```

4.4 Send a Custom Metric

Custom Metrics are user-defined counts that refer to a business goal, or to a Key Performance Indicator (KPI) such as revenue, conversion, orders per minute, widgets sold, etc. The value or meaning of a Custom Metric is defined by the App Administrator. Custom Metrics must be defined in the App dialog before use.

You may increment a Custom Metric by using sendMetric():

```
MPulse.sharedInstance().sendMetric("MyMetric", 23);
```

sendMetric() accepts a final parameter of MPulseMetricTimerOptions, which controls the behavior of Custom Metrics while an Action is ongoing. See the Actions documentation for details:

```
MPulseMetricTimerOptions options = new MPulseMetricTimerOptions();

// include on the Action beacon (instead of sending a separate beacon)
options.duringAction =
MPulseMetricTimerOptions.DuringAction.INCLUDE_ON_ACTION_BEACON;

// if the same Custom Metric was used twice on this Action, SUM the results
options.onActionDuplicate = MPulseMetricTimerOptions.OnActionDuplicate.SUM;

MPulse.sharedInstance().sendMetric("MyMetric", 23, options);
```

4.5 Set View Groups

A View Group (also known as a Page Group in a web app) allows for measurement across views that belong together. Grouping views in this way helps you capture and summarize the performance characteristics across the entire group.

For mobile apps, Launch may make up one View Group, while the Login view may make up a second, and Product views a third group. Search Results and Checkout views may also have their own groups.

You may get, set, and reset the View Group. Once set, the View Group will be associated with every subsequent beacon.

View Group strings are limited to 100 characters, and can include any of the following:

- 0-9
- a-z
- A-Z
- (space)
- _ (underbar)
- - (dash)

Set a View Group using setViewGroup():

```
MPulse.sharedInstance().setViewGroup("MyViewGroup");
```

Reset the View Group using resetViewGroup():

```
MPulse.sharedInstance().resetViewGroup();
```

Get the current View Group using getViewGroup():

```
String viewGroup = MPulse.sharedInstance().getViewGroup();
```

In addition, for each network request beacon, you can override the global View Group for that beacon by creating a filter and calling setViewGroup() on the MPFilterResult.

You can call setViewGroup() even if the result is unmatched. For example, in ALL (Blacklist) mode, you can call setMatched(false) so the network request is still monitored, and call setViewGroup() to set that request's View Group. If multiple filters set the View Group, the result is undefined (the last filter will take precedence).

See the Network Request Filtering section for details.

4.6 Set A/B Test

You may get, set, and reset the A/B Test. Once set, the A/B Test will be associated with every subsequent beacon.

Set a A/B Test using setABTest():

```
MPulse.sharedInstance().setABTest("A");
```

Reset the A/B Test using resetABTest():

```
MPulse.sharedInstance().resetABTest();
```

Get the current A/B Test using getABTest():

```
String abTest = MPulse.sharedInstance().getABTest();
```

4.7 Set Custom Dimensions

In addition to the out-of-the-box dimensions already provided within mPulse, App Admins can define additional Custom Dimensions for the given app. For example, a Custom Dimension to track Premium Users versus Free Users.Custom Dimensions must be defined in the App dialog before use.

You may get, set, and reset Custom Dimensions. Once set, the Custom Dimensions will be associated with every subsequent beacon.

Set or reset a Custom Dimension using setDimension():

```
MPulse.sharedInstance().setDimension("MyDimension", "new value");
```

Reset the Custom Dimension using resetDimension():

```
MPulse.sharedInstance().resetDimension("MyDimension");
```

Reset all Custom Dimensions using resetAllDimensions():

```
MPulse.sharedInstance().resetAllDimensions();
```

Get a list of all Custom Dimensions using getDimensions():

```
String[] dimensions = MPulse.sharedInstance().getDimensions();
```

4.8 Global Settings

The MPulseSettings class can be used to configure multiple SDK settings at once. MPulseSettings can be given to initializeWithAPIKey() to apply settings at startup, or later to updateSettings() to update multiple settings at once.

MPulseSettings has getters and setters for configuring the View Group, A/B Test, Custom Dimensions, Network Filters and Action settings.

When using MPulseSettings, any items that have not been set will not be changed:

```
MPulseSettings settings = new MPulseSettings();
settings.setViewGroup("Default");
settings.setABTest("A");
settings.setActionMaxResources(100);
settings.setCustomDimension("MyDimension", "A");
settings.enableNetworkMonitoring();
// settings.enableFilteredNetworkMonitoring();
// settings.disableNetworkMonitoring();
settings.setNetworkRequestFilterOptions(NetworkRequestFilterOptions.NONE);

// See MPulseSettings for a full list of options

// configure settings at init
MPulse.sharedInstance().initializeWithAPIKey(MPULSE_API_KEY,
getApplicationContext(), settings);

// change settings later
MPulse.sharedInstance().updateSettings(settings);
```

4.9 Actions

The mPulse Android SDK allows you to monitor Actions, which are distinct user interactions.

This features is available from mPulse Android SDK version 2.6.0+.

Actions can be started at any time by calling startAction(), and can be stopped by either calling stopAction(), or, by having the SDK automatically stop the Action once all network activity has finished.

The two Action modes are called Wait mode and Timeout mode:

- Wait mode will wait for stopAction() to be called, and the duration of the action will be from startAction() to stopAction()
    - This mode is best when you want to define the Action's start and end times in your application
- Timeout mode (default) will monitor background network activity to determine when the Action has ended, and will set the end timestamp when the final network request is complete
    - This mode is best when the Action triggers multiple network requests. The SDK will monitor all of those requests automatically.
    - By default, the mPulse SDK will wait for 1,000ms after the final network request to see if any additional network requests are started (and if so, wait for those requests to complete)
    - This mode should only be used when the Action triggers network activity

All network requests during the Action will be included on the Action beacon's Waterfall. There can only be a single Action ongoing at a time.

4.9.1 Starting an Action

To start an Action, you call the startAction() API:

```
// start an Action without a name
MPulse.sharedInstance().startAction();

// or, start an Action with a specific name
// Action name must match format: ^[a-zA-Z0-9-%:$#@!()&\[\]><* ]{0,50}$
MPulse.sharedInstance().startAction("MyAction");

// or, start an Action with a name or other settings
MPulseSettings settings = new MPulseSettings();
settings.setActionName("MyAction");
settings.setActionTimeout(2000);
settings.setActionMaxResources(200);
```

```
settings.setTimeoutForStop();
MPulse.sharedInstance().startAction(settings);
```

Starting an Action while an existing Action is ongoing will abort the previous Action.

4.9.2 Wait Mode

In Wait mode, the mPulse Android SDK will wait for stopAction() to be called. The duration of the Action will be from startAction() to stopAction().

Any network activity that occurred during the Action will be included on the Action's Waterfall. Example:

```
// start an Action
MPulse.sharedInstance().startAction("MyAction");

// ... do stuff ...

// stop the Action
MPulse.sharedInstance().stopAction();
```

4.9.3 Timeout Mode

In *Timeout* mode, the mPulse Android SDK will monitor all network activity. Once network requests begin to start, the Action will remain active until all network requests have finished.

After the last network request has finished, the SDK will wait the Action Timeout (default 1,000ms) to see if any additional network activity was triggered by the last request. If not, the duration of the Action will be set to the end of the final network request. If a new activity is triggered during the waiting period, the SDK will wait until all network activity has finished again.

4.9.4 Action Settings

You can set global Action settings that will apply to all Actions, as well as overriding the global Action settings when starting a new Action.

Settings:

- Action Collection Behavior: Whether to use Wait or Timeout mode
- Action Timeout: How long, in Timeout mode, the SDK waits after the last network request has finished to ensure no additional network requests were started

- Action Max Resources: The maximum number of network requests that will be included on an Action's Waterfall. In Timeout mode, this does not limit how many network requests will be waited for, just how many network requests will be reported in the Waterfall.

Configuring global Action settings:

```
// Timeout in milliseconds
MPulse.sharedInstance().setActionTimeout(2000);

// Maximum number of resources on an Action's Waterfall
MPulse.sharedInstance().setActionMaxResources(200);

// Wait mode
MPulse.sharedInstance().setActionCollectionBehavior(MPulseSettings.ActionCollection
Behavior.WAIT);
```

You can overwrite the global Action settings when starting an Action:

```
// start an Action with a name and other settings, overwriting the global Action
settings
MPulseSettings settings = new MPulseSettings();
settings.setActionName("MyAction");
settings.setActionTimeout(3000);
settings.setActionMaxResources(300);
settings.setTimeoutForStop();
MPulse.sharedInstance().startAction(settings);
```

4.9.5 Custom Metrics and Custom Timers during Actions

Custom Metrics and Custom Timers that occur while an Action is ongoing can be included on that Action beacon. Otherwise, Custom Timers and Custom Metrics will generate their own beacon.

The benefit of including Custom Timers and Custom Metrics on the Action beacon is that it ensures the context is kept – the Timer and Metric are linked directly to the Action.

When starting a Custom Timer or sending a Custom Metric, you can decide the what to do if an Action is ongoing. The MPulseMetricTimerOptions class configures this:

- MPulseMetricTimerOptions.DurationAction: What to do with a Custom Timer or Custom Metric when an Action is happening
    - SEND_DIRECT_BEACON: Send the Custom Timer/Custom Metric as a Custom Timer/Custom Metric beacon
    - INCLUDE_ON_ACTION_BEACON: (default) Include the Custom Timer/Custom Metric on the Action beacon

If the Custom Timer / Custom Metric is set to INCLUDE_ON_ACTION_BEACON, you should also decide what happens when a Custom Timer / Custom Metric is repeated during the same Action. An Action can only include a single named Custom Timer / Custom Metric per Action (Timer1 and Timer2 can both be included, but Timer1 can't be included twice).

There are 4 options for what happens when a Custom Timer / Custom Metric occurs repeatedly during the same Action:

- MPulseMetricTimerOptions.OnActionDuplicate: What to do with a Custom Timer or Custom Metric when it is repeated during the same Action
    - OVERWRITE: Overwrite the old value
    - IGNORE: Ignore the new value
    - SUM: Add the two values together
    - SEND_DIRECT_BEACON: (default) Convert the new value to an individual Custom Timer / Custom Metric beacon

Example usage:

```
MPulseMetricTimerOptions options = new MPulseMetricTimerOptions();

// include on the Action beacon (instead of sending a separate beacon)
options.duringAction =
MPulseMetricTimerOptions.DuringAction.INCLUDE_ON_ACTION_BEACON;

// if the same Custom Timer was used twice on this Action, SUM the results
options.onActionDuplicate = MPulseMetricTimerOptions.OnActionDuplicate.SUM;

String timerID = MPulse.sharedInstance().startTimer("MyTimer", options);
```

4.9.6 Cancelling an action

From version 21.21 SDK allows the user to cancel an action even before stopping it. When action is cancelled, the beacon won't be sent.

Example usage:

```
...
MPulse.sharedInstance().cancelAction();
...
```

4.9.7 Debugging APIs

The SDK provides APIs for developers to debug requests made through the SDK. There are two logging levels supported, viz. DEBUG and INFO. INFO is the default logging mode and contains logging with minimal output. DEBUG, on the other hand, is the enhanced logging mode. The logging level can be changed at runtime and is not persisted through multiple app sessions. All the APIs are supported through Logger.java class.

```
import com.akamai.android.sdk.Logger;


public enum LEVEL {
   /**
    * The default logging mode. This is the production level with minimal output.
    */
   INFO,
   /**
    * The enhanced logging mode for DEBUGGING purposes only.
    */
   DEBUG
}

/**
*
* @param level defines the SDK logging level.
*                The default level is LEVEL.INFO. This is the production level with minimal
output.
*                LEVEL.DEBUG is enhanced logging mode for DEBUGGING purposes only.
* Also see, {@link Logger.LEVEL}
*/
public static void setLevel(LEVEL level)
```

5. Mobile App Performance (MAP) Integration

When including mPulse with MAP you need to exclude akaCommon either in MAP or in mPulse. aka-common is a library used by both SDKs.

```
dependencies {
        implementation 'com.akamai.android:aka-mpulse:21.40.xx'
        implementation 'com.akamai.android:aka-map:21.40.xx'
}
```

When using both mPulse and MAP, the clients that are using okhttp wrappers should be using AkaOkHttpInterceptor or AkaOkHttpAppInterceptor under MAP/wrappers directory.

6 Appendix

6.1 Upgrading from a previous SDK version to 20.32 and above

Please follow the 20.3.2 upgrade guide packaged with the zip file.
mPulse SDK - Android v20.32+ Upgrade Guide.pdf

6.2 SDK Debug logs

For us to debug an SDK issue, we need logs. Retrieve us the logs associated with the following Logcat tag

```
adb logcat -s 'AkaSdKLogger'
```

If you are using both mPulse and MAP, use the following to verify the logs

mPulse

```
adb logcat -s 'AkaSdKLogger-mpulse'
```

MAP

```
adb logcat -s 'AkaSdKLogger-map'
```

common

```
adb logcat -s 'AkaSdKLogger-common'
```

6.3 Proguard

Proguard rules are already applied for mPulse and customer does not have to apply one. The one marked with *@PublicApi* are the classes and methods that are available for customer use. The one marked with *@AkamaiInternal* is not for customer use. If you make any calls to those methods, you will receive a lint error.

6.4 Troubleshooting guide

You need to set the log level to debug to see the debug logs associated with mPulse-sdk

```
Logger.setLevel(Logger.LEVEL.DEBUG);
```

6.4.1 Success

SDK discovered

```
D/AkaSDKLogger: Initialized com.akamai.android.aka_common, version
I/AkaSDKLogger: MPulseInternal mPulse initialized.
D/AkaSDKLogger: ComponentContainer: Initialized com.akamai.mpulse.android,
version 20.32.47
```

Interception started:

```
D/AkaSDKLogger: AkaCommon: No external URL handler to handle http/s stream
```

OR

```
D/AkaSDKLogger: External Url Stream handler to handle http/s stream::
com.akamai.android.sdk.AkaMap
```

Config requested

```
D/AkaSDKLogger: MPConfig Config request to url:
'https://c.go-mpulse.net/api/config.json?key=...' was successful.
D/AkaSDKLogger: MPConfig Response received: {"h.key":"
```

Analytics success

```
D/AkaSDKLogger: MPBatchTransport Successfully sent 1 record(s) to the server.
```

6.3.2 Failures

Wrong library

```
ERROR: Failed to resolve: com.akamai.android:aka-mpulse:20.32.47
```

Solution: Integration step is wrong. Add jcenter() in the project build.gradle

Missing config file reference

`E/AkaSDKLogger: Couldn't find resource file for meta-data key com.akamai.android.sdk!`

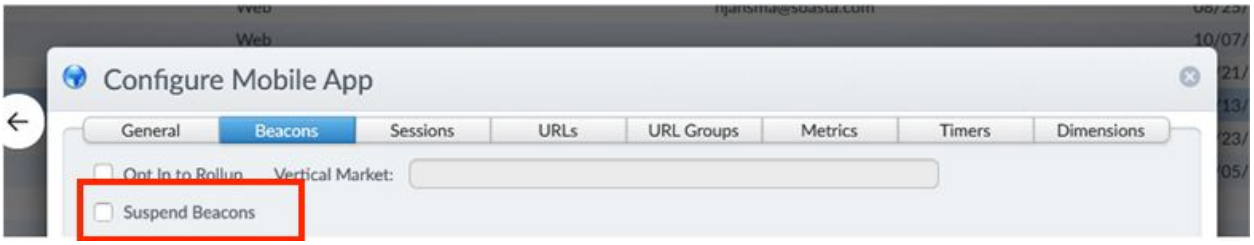
Solution: Check *akamai_sdk_init.xml* file is in **res/xml** and reference it in the AndroidManifest.xml

```
<meta-data
            android:name="com.akamai.android.sdk"
            android:resource="@xml/akamai_sdk_init" />
```

6.5 Suspend Beacons

The SDK provides options to turn ON and OFF the data transfers. This is in case, for whatever reason, we decide to turn off transfer.

This can be done via Soasta Portal.



- Checking ***Suspend Beacons*** from the mPulse App Editor will send a `204 No Content` for config.json so no beacons will send.
- Selecting Suspend Beacons will disable beacons for the mobile app.
- Note: it takes **~15 mins** for the config to update the app, and the app needs to restart.

7 Release Notes

The mPulse Android SDK can be found on JCenter.

21.40 (2020-06-30)

Changes:

- Bug fixes

21.30 (2020-05-29)

Changes:

- Bug fixes

21.21 (2020-04-27)

Changes:

- Bug fixes.
- Add support for cancelling action.

21.13 (2020-04-07)

Changes:

- Bug fixes

21.12 (2020-02-14)

Changes:

- Bug fixes

21.11 (2020-01-21)

Changes:

- Bug fixes

20.41 (2020-01-09)

Changes:

- Bug fixes

20.33 (2019-11-20)

- Moved back from R8 to proguard
- Migrated to androidx libraries
- Bug Fixes on whitelist Filters.

20.32.0 (2019-10-14)

Breaking Changes:

- Removed plugin application
- Group id has changed from com.soasta.android to com.akamai.android
- Artifact id has changed from mpulse-android to aka-mpulse
- The library import path has been changed from com.soasta.* to com.akamai.*
- Exclusion of instrumentation for a particular library is not supported in this version.

Changes:

- Moved to joint-SDK architecture.
- Removed aspectj interception mechanism and introduced akaCommon for interception.

2.6.3 (2019-08-07)

New Features:

- Optional query string obfuscation
- Adds support for older versions of Retrofit (e.g. 2.3.0)

2.6.2 (2019-07-08)

New Features:

- Capture Android Version Code in Site Version
- Improved OkHttp3 compatibility

2.6.0 (2019-04-10)

New Features:

- Adds support for Action Beacons
- Action Beacons: Custom Timers and Custom Metrics are now included on the Action Beacon
- Action Beacons: Configurable limit for the maximum number of resources to include per Action Beacon
- Logging: Adds support for MPLog.setTrace(true) to enable trace-level logging

Bug Fixes:

- Improves OkHttp3 monitoring by hooking into onResponse() and onFailure()
- Cleans up use of deprecated Apache HttpClient libraries
- Includes <uses-library> for org.apache.http.legacy for apps that need it
- Network Request Filters are now applied in the order they were added (previously, no order was maintained)
- Fixes SecurityException if ACCESS_NETWORK_STATE is missing

2.6.0-beta1 (2019-01-07)

New Features:

- Adds support for Action Beacons

2.5.1 (2019-01-28)

New Features:

- Additional support for monitoring OkHttp3 requests

Bug Fixes:

- Fixes an exception in OkHttp3Aspect

2.5.0 (2018-11-02)

New Features:

- Support for Java8
- Support for GradleAspectJ-Android and AspectJX Aspect plugins