mPulse Mobile SDK

October 21, 2020
Version 21.70

iOS Integration Guide

Summary

This document details integrating the **mPulse mobile SDK** into iOS apps for capturing the app's network traffic and sending it to the mPulse dashboard.

Introduction

The mPulse iOS SDK (version 2) is a library that you can use to send beacons from any iOS application to mPulse.

What types of data can I send?
The mPulse iOS 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 iOS 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.

The mPulse iOS 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.

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

Getting Started

Before using the mPulse iOS 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 the Getting Started Guide. Once your app has been configured in mPulse, you can use the mPulse iOS SDK.

Instrumenting Network Requests

The iOS platform provides several ways to request network resources via HTTP and HTTPS. The mPulse SDK instruments both of the direct download approaches, NSURLSession and NSURLConnection. It also instruments Web pages loaded through UIWebView. WKWebView and SFSafariViewController are run by the OS outside of the app process, and are not handled by the SDK at this time. See our guide on implementing Boomerang tags to track web processes.

Request Class	Request Type	Requires Extra Setup?
NSURLConnection	Individual file	NO
NSURLSession	Individual file	YES for custom sessions. Shared session is automatic.
UIWebView	Web view	NO

Table 1 - URL Request Types

NSURLConnection and UIWebView are automatically monitored once the SDK is installed. Each NSURLSession using a custom configuration requires a configuration call. Each approach is covered in the API Reference section.

Requirements and Dependencies

The mPulse SDK requires iOS 8 or higher.

An mPulse SDK API key is required. This can be created on the mPulse portal.

Framework Size

The mPulse SDK consists of the MPulse framework and a shared AkaCommon framework. These are fat binaries, meaning they contain compiled code for several architectures. Apple separates and delivers the appropriate architecture for the end-user device. Approximate end-user sizes are shown in the arm64 and armv7 columns below.

These frameworks are bitcode-enabled. Apple recompiles the bitcode for each device type into the approximate sizes below. The final size will vary by device type.

	fat framework with bitcode	arm64 without bitcode	armv7 without bitcode
MPulse	18.0 MB	1.5 MB	1.5 MB
AkaCommon (shared)	3.0 MB	452 KB	416 KB
Total download size in app store		1.9 MB	1.9 MB

Table 2 - Framework sizes

Feature Overview

Custom Metrics

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.

You can programmatically increment a Custom Metric using the SDK.

Custom Metrics must be defined in the App dialog before use.

Custom Timers

A Custom Timer can be based on any measurable user-defined duration.

You can programmatically send the value of a Custom Timer using the SDK.

Custom Timers must be defined in the App dialog before use.

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 can programmatically set the View Group using the SDK.

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.

You can programmatically set a Custom Dimension using the SDK.

Custom Dimensions must be defined in the App dialog before use.

Installing the iOS SDK

CocoaPods

CocoaPods is an open source dependency manager for Swift and Objective-C Cocoa projects. Refer to the CocoaPods Getting Started guide if you are unfamiliar with CocoaPods.

1. Create or reuse an existing Podfile. Open it in a text editor.
2. Add MPulse SDK to your Podfile. This will automatically pull in the AkaCommon framework.

```
target 'YOUR_APPLICATION_TARGET_NAME_HERE' do
   use_frameworks!
   pod 'Aka-mPulse'
end
```

3. With the Podfile written, run the CocoaPods install command. This downloads the latest frameworks and creates (or edits) your Xcode workspace.

```
pod install
```

4. Close Xcode and re-open your project via the .xcworkspace file generated by CocoaPods. Always open the project using the workspace, not the project, in order to include the frameworks.

CocoaPods Integration with MAP SDK

The mPulse SDK works seamlessly with Akamai's Mobile App Performance (MAP) SDK. Both share the same AkaCommon framework. CocoaPods takes care of the framework imports. Replace the MAP block in your Podfile with the following.

```
target 'YOUR_APPLICATION_TARGET_NAME_HERE' do
          use_frameworks!
          pod 'Aka-mPulse'
          pod 'Aka-MAP'
       end
```

Refer to the MAP SDK documentation to use its API in your app.

Carthage

Carthage is a less intrusive alternative to CocoaPods. It downloads the latest frameworks but leaves it to you to add them to your app. Carthage is documented here.

To install the mPulse SDK using Carthage:

1. Create or reuse an existing Cartfile. Open it in a text editor.
2. Add mPulse and AkaCommon to your Cartfile. AkaCommon is a required dependency.

```
binary "https://downloads.pvoc-anaina.com/ios/akaCommon/aka-common-ios.json"
binary "https://downloads.pvoc-anaina.com/ios/mPulse/aka-mpulse-ios.json"
```

3. With the Cartfile written, run the Carthage install command. This downloads the latest frameworks into *<your project>/Carthage/Builds/iOS/*.

```
carthage update
```

4. Add the frameworks to Xcode:
    a. Open Xcode to your app target's "General" tab.
    b. Open a finder window to *<project>/Carthage/Builds/iOS*.
    c. Drag both **AkaCommon.framework** and **Aka-mPulse.framework** from the Carthage folder into your Xcode "Frameworks, Libraries, and Embedded Content" section.
    d. For both frameworks, select "Do Not Embed." Embedding will be performed by Carthage in the following build phase.
5. Add the embed frameworks script in Xcode:
    a. Open Xcode to your app target's "Build Phases" tab.
    b. Create a new Run Script build phase. Optionally, name it "Carthage Run Script."
    c. Add the following shell command.

```
/usr/local/bin/carthage copy-frameworks
```

    d. Add both frameworks to the run script's "Input Files" section:

```
$(SRCROOT)/Carthage/Build/iOS/AkaCommon.framework
$(SRCROOT)/Carthage/Build/iOS/Aka-mPulse.framework
```

6. Build and run your app. The SDK will print its version number to the XCode console as confirmation that it is installed correctly.

Carthage Integration with MAP SDK

The mPulse SDK works seamlessly with Akamai's Mobile App Performance (MAP) SDK. Both share the same AkaCommon framework. To install mPulse SDK with MAP SDK using Carthage, follow the Carthage steps above but include the AkaCommon framework only once.

Cartfile:

```
binary "https://downloads.pvoc-anaina.com/ios/akaCommon/aka-common-ios.json"
binary "https://downloads.pvoc-anaina.com/ios/mPulse/aka-mpulse-ios.json"
binary "https://downloads.pvoc-anaina.com/ios/MAP/aka-map-ios.json"
```

The Carthage run script build phase has the same three entries:

```
$(SRCROOT)/Carthage/Build/iOS/AkaCommon.framework
$(SRCROOT)/Carthage/Build/iOS/Aka-mPulse.framework
$(SRCROOT)/Carthage/Build/iOS/AkaMap.framework
```

Adding Frameworks Manually

Manual integration requires the Aka-Common and mPulse frameworks. Direct links to the latest versions are located in README.txt in the documentation bundle, *Package.zip*. The package can be obtained from Akamai's mPulse configuration portal, or by contacting your Services Team member.

Because these are fat frameworks (combined simulator and device), the simulator binaries must be stripped out of the frameworks before app store submission. That step is included here.

1. Download both frameworks from direct links. Ensure versions match for both Aka-Common and mPulse. For example, both must be 21.11.x. The minor version may differ.
2. Unzip the frameworks into your project workspace.
3. Add the frameworks to Xcode:
    a. Open Xcode to your app target's "General" tab.
    b. Open a finder window to *<project>/<frameworks location>*.
    c. Drag both **AkaCommon.framework** and **Aka-mPulse.framework** from that folder into your Xcode "Frameworks, Libraries, and Embedded Content" section.
    d. For both frameworks, select "Embed & Sign." Alternatively, you may create a Build Phase for "Copy Frameworks" and add both frameworks there.
4. Add the script to strip fat frameworks for Release builds.

    Note: This script runs only in Release builds. It removes simulator architectures in all frameworks that are part of the build, not just the Akamai frameworks.

    a. Open your app target's settings and go to Build Phases.
    b. Click "+" and add a "New Run Script Phase."

c. Drag this phase to be last; it must run after the Copy or Embed Frameworks phase.
d. Click the new phase's name and rename it to "strip frameworks."
e. Click to expand the strip frameworks phase.
f. Leave the default shell of "/bin/sh/".
g. Add the following text to the script area. The strip_frameworks.sh path is relative to your .xcodeproj file and may need modification depending on where you unzipped it (/frameworks/mPulse/ in this example). The strip_frameworks.sh file is bundled alongside the mPulse .framework file.

```
./frameworks/mPulse/strip_frameworks.sh
```

In this example, the folder structure is as follows:
/myproject/myproject.xcodeproj
/myproject/frameworks/mPulse/strip_frameworks.sh
/myproject/frameworks/mPulse/Aka-mPulse.framework

5. If your app is Swift,
   a. define or add to an existing bridging header file.

```
Swift:

#ifndef Demo_Bridging_Header_h
#define Demo_Bridging_Header_h

#import <AkaCommon/AkaCommon.h>
#import <Aka-mPulse/MPulse.h>
// #import <AkaMap/AkaMap.h>    // if also including MAP SDK

#endif /* Demo_Bridging_Header_h */
```

   b. Add the bridging header path/filename to your target's Settings, *Objective-C Bridging Header*.

Integrating with your iOS Application

Import SDK Frameworks

See the previous section for including frameworks with CocoaPods, Carthage, or manually.

Initialize SDKs

mPulse is initialized by calling the AkaCommon library's *configure* method. This detects the linked mPulse framework and passes it the API key specified in your Info.plist file.

```
Objective-C:
#import <AkaCommon/AkaCommon.h>

[AkaCommon configure];

Swift:
AkaCommon.configure
```

There is also an option to configure the library with a custom plist file. To do so, create a plist file and specify the configuration (in the same format as you would in Info.plist file), then pass the path to this file as instructed in the below snippet.

```
Objective-C:
#import <AkaCommon/AkaCommon.h>

NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Aka-Info" ofType:@"plist"];
if (filePath) {
  [AkaCommon configureWithPlist:filePath];
}

Swift:
if let filePath = Bundle.main.path(forResource:"Aka-Info", ofType:"plist") {
  AkaCommon.configure(withPlist: filePath)
}
```

To make optional API calls from AkaCommon, send them to the shared instance.

```
Objective-C:
AkaCommon *akaCommon = [AkaCommon shared];

Swift:
let akaCommon = AkaCommon.shared()
```

Using MPulse APIs

Calls to the framework are made to the MPulse shared instance.

```
Objective-C:
#import <Aka-mPulse/MPulse.h>

MPulse *mpulse = [MPulse sharedInstance];

Swift:
// Your bridging header already contains #import <Aka-mPulse/MPulse.h>

let mpulse = MPulse.sharedInstance()
```

Specifying an API Key

The mPulse API key is specified in your app's Info.plist file. An API key can be obtained from the mPulse Web portal.

Key	*Type*	*Example*
com.akamai *> mpulse* *> api_key*	*Dictionary* *Dictionary* *String*	*{:}* *{:}* *123456ABCDEF*

API Reference

This section discusses mPulse features and provides example API usage.

Enabling or Disabling the SDK at Runtime

Once configure is called, the mPulse iOS SDK is enabled.

You can disable the mPulse iOS SDK at runtime by calling disable:

```
Objective-C:
[AkaCommon configure];

// later:
[[MPulse sharedInstance] disable];

Swift:
AkaCommon.configure

// later:
MPulse.sharedInstance()?.disable()
```

Once disabled, mPulse will no longer send beacons.

mPulse can be re-enabled by calling enable:

```
Objective-C:
[[MPulse sharedInstance] enable];

Swift:
MPulse.sharedInstance()?.enable()
```

Network Request Instrumentation

Automatic Network Request Instrumentation

Once included in the project, mPulse will automatically instrument all network requests made on shared sessions. This includes the shared NSURLSession and all NSURLConnection requests.

- URLConnection
- URLSession - shared session configuration

The following libraries (which use the above classes) will be instrumented whenever they used shared URLSessions/URLConnections. Where these libraries use private URLSessions, those sessions must have mPulse added to them, as described in the manual instrumentation section below.

- AFNetworking
- SDWebImage
- AlamoFire
- KingFisher

Waterfall is the recommended implementation for capturing beacons. Your app does this by defining Actions. Beacons captured inside of an Action automatically become part of that Action's Waterfall.

Manual Network Request Instrumentation

For custom URLSession configurations, mPulse must first be added to that configuration. All URLSessions generated from that configuration will then be instrumented.

```
Objective-C:
NSURLSessionConfiguration *sessionConfig = [NSURLSessionConfiguration defaultConfiguration];

[[AkaCommon shared] interceptSessionsWithConfiguration:sessionConfig];

// Now create a session seen by mPulse.
NSURLSession *mySession = [NSURLSession sessionWithConfiguration:sessionConfig delegate:nil
delegateQueue:nil];

Swift:
let sessionConfig = URLSessionConfiguration.default
AkaCommon.shared().interceptSession(with: sessionConfig)

// Now create a session seen by mPulse.
let mySession = URLSession.init(configuration: sessionConfig)
```

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 iOS SDK allows you to control the state of network request monitoring during runtime using the following SDK APIs:

Disable network request beacons (i.e., set the mode to NONE):

```
Objective-C:
[[MPulse sharedInstance] disableNetworkMonitoring];

Swift:
MPulse.sharedInstance()?.disableNetworkMonitoring()
```

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

```
Objective-C:
[[MPulse sharedInstance] enableNetworkMonitoring];

Swift:
MPulse.sharedInstance()?.enableNetworkMonitoring()
```

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

```
Objective-C:
[[MPulse sharedInstance] enableFilteredNetworkMonitoring];

Swift:
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.

Programmatically Extending the Filters

The mPulse iOS 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.

If you wish to add to the Blacklist for ALL mode, you can use the example below. Returning a MPFilterResult with setMatched:YES will result in the network request not being monitored.
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.

```
Objective-C:
[[MPulse sharedInstance] addURLBlackListFilter:@"Example.com Filter" filter:^MPFilterResult
*(NSString *url) {
  // By default MPFilterResults are set to not matched.
  MPFilterResult *result = [[MPFilterResult alloc] init];

  if ([url isEqual:@"http://example.com"])
  {
    [result setMatched:YES];
  }

  return result;
}];

Swift:
MPulse.sharedInstance()?.addURLBlackListFilter("MyFilter") { (url) -> MPFilterResult? in
  // By default MPFilterResults are set to not matched.
  let result = MPFilterResult.init()

  if let url = url,
    url.contains("example.com") {
    result.matched = true
  }

  return result
}
```

If you wish to add to the Whitelist for MATCH mode, you can use the example below. Returning a MPFilterResult with setMatched:YES will result in the network request being monitored.

```
Objective-C:
[[MPulse sharedInstance] addURLWhiteListFilter:@"Example.com Filter" filter:^MPFilterResult
*(NSString *url) {
  // By default Filter results are set to not matched.
  MPFilterResult *result = [[MPFilterResult alloc] init];

  if ([url containsString:@"example.com"])
  {
    [result setMatched:YES];
    [result setViewGroup:@"My View Group"];
  }

  return result;
}];

Swift:
MPulse.sharedInstance().addURLWhiteListFilter("MyFilter") { (url) -> MPFilterResult? in
  // By default MPFilterResults are set to not matched.
  let result = MPFilterResult.init()

  if let url = url,
    url.contains("example.com") {
    result.matched = true
    result.viewGroup = "My View Group"
  }

  return result
}
```

Send a Custom Timer

The mPulse iOS 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: and give it a Timer Name. startTimer: will return a unique Timer ID (NSString) and will keep track of the start time:

```
Objective-C:
NSString *timerID = [[MPulse sharedInstance] startTimer:@"TimerName"];

Swift:
let timerID = MPulse.sharedInstance()?.startTimer("TimerName")
```

At the “end” of your action, call stopTimer: by passing in the timerID. mPulse stops the timer and sends a beacon to the server:

```
Objective-C:
[[MPulse sharedInstance] stopTimer:timerID];

Swift:
MPulse.sharedInstance()?.stopTimer(timerID)
```

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

```
Objective-C:
// value is NSTimeInterval
[[MPulse sharedInstance] sendTimer:@"TimerName" value:4];

Swift:
MPulse.sharedInstance()?.sendTimer("TimerName", value: 4)
```

The value passed to sendTimer: is a NSTimeInterval (in seconds.milliseconds) in iOS.

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:

```
Objective-C:
// stops the timer and updates dimensions
[[MPulse sharedInstance] stopTimer:timerID updateDimensions:true];

Swift:
MPulse.sharedInstance()?.stopTimer(timerID, updateDimensions: true)
```

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

```
Objective-C:
[[MPulse sharedInstance] cancelTimer:timerID];

Swift:
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:

```
Objective-C:
MPulseMetricTimerOptions* options;
```

```
// include on the Action beacon (instead of sending a separate beacon)
options.duringAction = MPulseDataDuringActionIncludeOnActionBeacon;

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

NSString *timerID = [[MPulse sharedInstance] startTimer:@"TimerName" withOptions:options];

// or

[[MPulse sharedInstance] sendTimer:@"TimerName" value:10 withOptions:options];

Swift:
let options = MPulseMetricTimerOptions()

// include on the Action beacon (instead of sending a separate beacon)
options.duringAction = MPulseDataDuringAction.includeOnAction

// if the same Custom Timer was used twice on this Action, SUM the results
options.onActionDuplicate = MPulseDataOnDuplicate.sum

let timerID = MPulse.sharedInstance()?.startTimer("TimerName", with: options)

// or

MPulse.sharedInstance()?.sendTimer("TimerName", value: 10, with: options)
```

Send a Custom Metric

You may increment a Custom Metric by using sendMetric::

```
Objective-C:
[[MPulse sharedInstance] sendMetric:@"MyMetric" value:[NSNumber numberWithInt:23]];

Swift:
MPulse.sharedInstance()?.sendMetric("MyMetric", value: 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:

```
Objective-C:
MPulseMetricTimerOptions* options;

// include on the Action beacon (instead of sending a separate beacon)
options.duringAction = MPulseDataDuringActionIncludeOnActionBeacon;
```

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

[[MPulse sharedInstance] sendMetric:@"MetricName" value:[NSNumber numberWithInt:10]
withOptions:options];

Swift:
let options = MPulseMetricTimerOptions()

// include on the Action beacon (instead of sending a separate beacon)
options.duringAction = MPulseDataDuringAction.includeOnAction

// if the same Custom Metric was used twice on this Action, SUM the results
options.onActionDuplicate = MPulseDataOnDuplicate.sum

MPulse.sharedInstance()?.sendMetric("MetricName", value: 10, with: options)
```

Set View 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::

```
Objective-C:
[[MPulse sharedInstance] setViewGroup:@"MyViewGroup"];

Swift:
MPulse.sharedInstance()?.setViewGroup("MyViewGroup")
```

Reset the View Group using resetViewGroup::

```
Objective-C:
[[MPulse sharedInstance] resetViewGroup];
```

```
Swift:
MPulse.sharedInstance()?.resetViewGroup()
```

Get the current View Group using getViewGroup::

```
Objective-C:
NSString* viewGroup = [[MPulse sharedInstance] getViewGroup];

Swift:
let 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:NO 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.

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::

```
Objective-C:
[[MPulse sharedInstance] setABTest:@"A"];

Swift:
MPulse.sharedInstance()?.setABTest("A")
```

Reset the A/B Test using resetABTest::

```
Objective-C:
[[MPulse sharedInstance] resetABTest];

Swift:
MPulse.sharedInstance()?.resetABTest()
```

Get the current A/B Test using getABTest::

```
Objective-C:
NSString* abTest = [[MPulse sharedInstance] getABTest];

Swift:
let abTest = MPulse.sharedInstance()?.getABTest()
```

Set Custom Dimensions

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

Set or reset a Custom Dimension using setDimension::

```
Objective-C:
[[MPulse sharedInstance] setDimension:@"My Dimension" value:@"new value"];

Swift:
MPulse.sharedInstance()?.setDimension("My Dimension", value: "new value")
```

Reset the Custom Dimension using resetDimension::

```
Objective-C:
[[MPulse sharedInstance] resetDimension:@"My Dimension"];

Swift:
MPulse.sharedInstance()?.resetDimension("My Dimension")
```

Reset all Custom Dimensions using resetAllDimensions::

```
Objective-C:
[[MPulse sharedInstance] resetAllDimensions];

Swift:
MPulse.sharedInstance()?.resetAllDimensions()
```

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:

```
Objective-C:
```

```
MPulseSettings* settings;
[settings setViewGroup:@"Default"];
[settings setAbTest:@"A"];
[settings setMaxActionResources:100];

// configure settings at init
[MPulse initializeWithAPIKey:MPULSE_API_KEY withSettings:settings];

// change settings later
[[MPulse sharedInstance] updateSettings:settings];

Swift:
let settings = MPulseSettings()
settings.viewGroup = "Default"
settings.abTest = "A"
settings.maxActionResources = 100

// change settings later
MPulse.sharedInstance()?.update(settings)
```

Actions

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

This feature is available with mPulse iOS SDK version 2.6.0+.

Actions can be started at any time by calling startAction:, and can be stopped either by 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.

Starting an Action

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

```
Objective-C:
// 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] startActionWithName:@"MyAction"];

// or, start an Action with a name or other settings
MPulseSettings* settings;
[settings setActionName:@"MyAction"];
[settings setActionTimeout:[NSNumber numberWithInt:500]];
[settings setMaxActionResources:100];
[settings timeoutToStop]; // Timeout mode
[settings waitForStop]; // or, Wait mode

[[MPulse sharedInstance] startActionWithSettings:settings];

Swift:
// start an Action without a name
MPulse.sharedInstance()?.startAction()

// or, start an Action with a specific name
MPulse.sharedInstance()?.startAction(withName: "MyAction")

// or, start an Action with a name or other settings
let settings = MPulseSettings()
settings.actionName = "MyAction"
settings.actionTimeout = 500
settings.maxActionResources = 100
settings.timeoutToStop() // Timeout mode
settings.waitForStop() // or, Wait mode

MPulse.sharedInstance()?.startAction(with: settings)
```

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

Wait Mode

In Wait mode, the mPulse iOS 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:

```
Objective-C:
// start an Action
[[MPulse sharedInstance] startActionWithName:@"MyAction"];

// ... do stuff ...

// stop the Action
[[MPulse sharedInstance] stopAction];

Swift:
// or, start an Action with a specific name
MPulse.sharedInstance()?.startAction(withName: "MyAction")

// ... do stuff ...

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

Timeout Mode

In Timeout mode, the mPulse iOS 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 new activity is triggered during the waiting period, the SDK will wait until all network activity has finished again.

Cancelling an Action

To cancel an Action, you call the cancelAction API:

```
Objective-C:
// start an Action
[[MPulse sharedInstance] startAction];

// ... do stuff ...
```

```
// cancel the Action
[[MPulse sharedInstance] cancelAction];

Swift:
// start an Action
MPulse.sharedInstance()?.startAction()

// ... do stuff ...

// cancel the Action
MPulse.sharedInstance()?.cancelAction()
```

Cancelling an Action will delete the active Action and all recorded Beacons for this Action.

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:

```
Objective-C:
// Timeout in milliseconds
MPulseSettings* settings;
[settings setActionTimeout:[NSNumber numberWithInt:2000]];

// Maximum number of resources on an Action's Waterfall
[settings setMaxActionResources:200];

// Timeout mode
[settings timeoutToStop];

// or, Wait mode
[settings waitForStop];

// Update the Global defaults
[[MPulse sharedInstance] updateSettings:settings];
```

```
Swift:
// Timeout in milliseconds
let settings = MPulseSettings()
settings.actionTimeout = 2000

// Maximum number of resources on an Action's Waterfall
settings.maxActionResources = 200

// Timeout mode
settings.timeoutToStop()

// or, Wait mode
settings.waitForStop()

// Update the Global defaults
MPulse.sharedInstance()?.update(settings)
```

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

```
Objective-C:
// Timeout in milliseconds
MPulseSettings* settings;
[settings setActionTimeout:[NSNumber numberWithInt:1000]];

// Maximum number of resources on an Action's Waterfall
[settings setMaxActionResources:300];

// Use these settings for the Action
[[MPulse sharedInstance] startActionWithSettings:settings];

Swift:
// Timeout in milliseconds
let settings = MPulseSettings()
settings.actionTimeout = 1000

// Maximum number of resources on an Action's Waterfall
settings.maxActionResources = 300

// Use these settings for the Action
MPulse.sharedInstance()?.startAction(with: settings)
```

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 what to do if an Action is ongoing. The MPulseMetricTimerOptions class configures this:

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

If the Custom Timer / Custom Metric is set to MPulseDataDuringActionIncludeOnActionBeacon, 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
  - MPulseDataOnDuplicateOverwrite: Overwrite the old value
  - MPulseDataOnDuplicateIgnore: Ignore the new value
  - MPulseDataOnDuplicateSum: Add the two values together
  - MPulseDataOnDuplicateSendDirectBeacon: (default) Convert the new value to an individual Custom Timer / Custom Metric beacon

Example usage:

```
Objective-C:
MPulseMetricTimerOptions* options;

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

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

[[MPulse sharedInstance] sendMetric:@"MetricName" value:[NSNumber numberWithInt:10]
withOptions:options];

Swift:
let options = MPulseMetricTimerOptions()
```

```
// include on the Action beacon (instead of sending a separate beacon)
options.duringAction = MPulseDataDuringAction.includeOnActionBeacon

// if the same Custom Metric was used twice on this Action, SUM the results
options.onActionDuplicate = MPulseDataOnDuplicate.sum

MPulse.sharedInstance()?.sendMetric("MetricName", value: 10, with: options)
```

Pinned Certificates / Custom TLS Certificate Handling

mPulse SDK uses the device's default certificate chain to decide which servers to trust. There are cases where an app needs to customize this behavior. These apps should implement the optional delegate callback below.

Swift example using the app delegate as the cert handler. The handler may be any class.

```
Objective-C:

// Implement AkaCommonDelegate to be notified with TLS challenges.
@interface AppDelegate () <AkaCommonDelegate>

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
        // Initialize the SDK.
        [AkaCommon configure];

        // Provide AkaCommon with an AkaCommonDelegate to handle the pinned cert.
        AkaCommon *akaCommon = [AkaCommon shared];
        akaCommon.delegate = self;

        return YES;
}

// Implement the AkaCommonDelegate handler. See function definition for full details.
- (void)didReceiveChallengeForRequest:(NSURLRequest *)originalRequest
  currentRequest:(NSURLRequest *)currentRequest
  challenge:(NSURLAuthenticationChallenge *)challenge
  modifiedTrust:(SecTrustRef)modifiedTrust
  completion:(void (^)(NSURLSessionAuthChallengeDisposition, NSURLCredential *
_Nullable))completion
{
        // Check the challenge against your pinned certificate.
        NSLog(@"App received challenge -- sending it for default handling.");

        // Your function must call the completion for the request to proceed.
        if (completion) {
                completion(NSURLSessionAuthChallengePerformDefaultHandling, nil);
        }
}
```

```swift
Swift:

// Implement AkaCommonDelegate to be notified with TLS challenges.
class AppDelegate: UIResponder, UIApplicationDelegate, AkaCommonDelegate {

  func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions:
        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    // Initialize the SDK.
    AkaCommon.configure()

    // Provide AkaCommon with an AkaCommonDelegate to handle the pinned cert.
    let akaCommon = AkaCommon.shared()
    akaCommon.delegate = self

    return true
  }

}

// Implement the AkaCommonDelegate handler. See function definition for full details.
func didReceiveChallenge(for originalRequest: URLRequest,
                         currentRequest: URLRequest,
                         challenge: URLAuthenticationChallenge,
                         modifiedTrust: SecTrust?,
                         completion: @escaping (URLSession.AuthChallengeDisposition,
URLCredential?) -> Void)
{
  // Check the challenge against your pinned certificate.
  print("App received challenge -- sending it for default handling.")

  // Your function must call the completion for the request to proceed.
  completion(.performDefaultHandling, nil)
}
```

This passes the app several pieces of information, with full details in the header file:

- the **original request** made by the app for identifying the URL in question
- the **TLS server challenge**
- the modified trust object. This is an exact copy of challenge.protectionSpace.serverTrust. It is reserved for future use.
- a **completion block** to be called with the result of the evaluation. This must be called before the request will proceed.

The callback will be made for all requests and prepositioned downloads made through the SDK. Its usage and parameters are fully explained in the header file.

Debugging APIs

mPulse's error information is output to both SDK log files and the Xcode console. Developers may print extended debug output to the console with the following calls.

Note: **do not** use these in production/App Store builds as the extra output can lead to performance slowdowns.

debug property	frequency	purpose
[In AkaCommon framework] debugConsoleEnabled = <BOOL>	continuous until disabled	Running reports of URLs intercepted by mPulse SDK.

```
Objective-C:

// enable real-time extended debug info to Xcode console
AkaCommon.shared.debugConsoleEnabled = YES;

Swift:

// enable real-time extended debug info to Xcode console
AkaCommon.shared().debugConsoleEnabled = true
```

MAP Integration

See Integrating with MAP SDK for setting up an Xcode workspace with both mPulse and MAP SDKs.

Appendix

Bitcode

mPulse SDK is compiled with bitcode enabled. It works in both bitcode- and non-bitcode-enabled apps.

Troubleshooting

No Beacons Being Sent

If you are not seeing beacons in the mPulse dashboards, please ensure the app has been configured correctly:

- Ensure you are initializing the SDK with AkaCommon.configure().
- Ensure you are using the correct mPulse API Key in your plist file.

Ensure you see the following lines in the debug log after the application has started:

mPulse Mobile build: n.n.n

mPulse initialized.
mPulse session has started.

- If you are trying to send a Custom Timer or Custom Metric, ensure they have already been defined in the mPulse app configuration.
- Using a system proxy such as Fiddler or Charles, validate that:
    - There is a Config request to https://c.go-mpulse.net/api/config.json?...
    - There is a beacon being sent to https://*.mpstat.us or https://*.akstat.io

mPulse Debug Logs

mPulse SDK's debug logs are available on the device or simulator:

<application sandbox>/Library/Caches/Logs/<date>_com.akamai.akasdk.log

Limited runtime info is printed to the Xcode console. To enable debug logging:

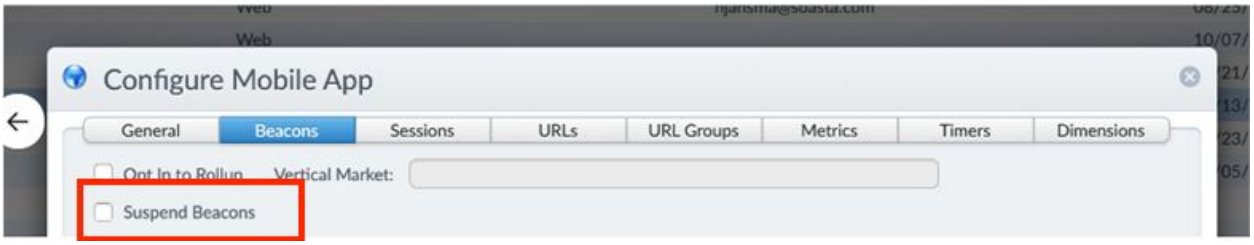
```
Objective-C:
[MPulse setDebug:true];

Swift:
MPulse.setDebug(true)
```

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 transfers.

This can be done via the Soasta Portal.



- Checking ***Suspend Beacons*** from the mPulse App Editor will send a 204 No Content for config.json so that 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.

FAQ

Why is this warning in my build logs?

Ignoring file libMPulse[Sim].a, missing required architecture [arch] in file libMPulse.a ([n] slices)

Answer: This is just a warning, and is expected. The mPulse libraries are split into two files (libMPulse.a and libMPulseSim.a) because of GitHub file size limits (100 MB), where the CocoaPod libraries are hosted. libMPulseSim.a is for simulators and libMPulse.a is for devices, and only one will be used at a time.

Opening Support Tickets

When opening new support tickets, please include the following information:

- Platform: iOS
    - iOS SDK version
- mPulse SDK version
- Complete debug logs (i.e. console) from a device or simulator
    - Enable debug mode via [MPulse setDebug:true] after initialization
- If possible, the project's Podfile or build configuration
- If possible, a compiled IPA with the mPulse SDK integrated
- If possible, code samples of:
    - mPulse initialization
    - Beacon filtering (if used)
- If possible, the full project source code