



# Mobile Application Performance SDK

December 9, 2020  
Version 21.80

## iOS Integration Guide



[Introduction](#)

[Getting Started](#)

[Requirements](#)

[Integration with your iOS Application](#)

[Import SDK Frameworks](#)

[CocoaPods](#)

[CocoaPods Integration with mPulse SDK](#)

[Carthage](#)

[Carthage Integration with mPulse SDK](#)

[Adding Frameworks Manually](#)

[Initialize SDKs](#)

[Using MAP APIs](#)

[API Key](#)

[Integrating with Firebase Cloud Messaging \(Optional\)](#)

[API Reference](#)

[Pre-Positioned Content](#)

[Updating Pre-Positioning Subscriptions](#)

[MAP Cache Size](#)

[NSURLConnection](#)

[UIWebView](#)

[WKWebView](#)

[NSURLSession](#)

[Custom Event Tracking](#)

[Timed Events](#)

[Instantaneous Events](#)

[Network Aware Experience](#)

[Pinned Certificates / Custom TLS Certificate Handling](#)

[Receiving MAP Service Callbacks](#)

[Cache-Control Request Parameters](#)

[Debugging APIs](#)

[MAP State Properties](#)

[Enabled Flag](#)

[Subscribed Segments](#)

[Direct Access to Cached Files](#)

[Cache API and Record Format](#)

[Implementing a Cache Listener](#)

[QUIC Library Integration](#)

[Integrating QUIC](#)

[Brotli Library Integration](#)

[mPulse Integration](#)

[Appendix - Requirements and Dependencies](#)



[MAP Debug Logs](#)

[Bitcode](#)

[App Transport Security](#)

[Disabling Redirect Behavior](#)

[Framework Size](#)

## Introduction

MAP SDK includes various performance-enhancing features for your mobile app. It pre-positions web content onto the mobile device based on subscribed content groups (“segments”), and policies set up between the client and server. Acceleration of live traffic, as well as statistics collection, are handled internally by the SDK. MAP SDK’s API allows defining user subscriptions, control over which connections receive acceleration, and other controls defined in this guide.

The API also provides developers access to real-time network conditions. This information can be used to expand the user experience by taking necessary actions based on the network state.

API calls are available for logging user-initiated events to the server. These can be used to associate network traffic originating from the app with events such as tapping a button.

## Getting Started

The iOS platform provides several ways to request network resources via HTTP and HTTPS. MAP SDK accelerates the two direct download approaches, NSURLSession and NSURLConnection. It also enhances web pages loaded through UIWebView. WKWebView and SFSafariViewController are run by the OS outside of the app process, and are not enhanced by the SDK at this time.

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, UIWebView, and the default NSURLSession are automatically accelerated once the SDK is installed. NSURLSessions that use custom configurations require a MAP configuration call. Each approach is covered in the API Reference section.

The MAP SDK library also collects network-related statistics while serving content. These include HTTP time to first byte (TTFB), request size, response size, duration, and others. These stats are periodically sent to the MAP SDK server for access via the Web portal.

## Requirements

To integrate MAP SDK into your iOS, you need:

- iOS 8 or higher.
- An API key to enable features. The key is provisioned through the MAP SDK web portal.
- Your app's bundle ID and "iOS Application ID" to match. You can find your app bundle ID by going to Xcode at **Project** → choose target → **General** → **Identity** → **Bundle Identifier**. The "iOS Application ID" is provided in the MAP SDK web portal configuration.
- Firebase Messaging (FCM) project to use with [Pre-Positioned Content](#) (optional)

## Integration with your iOS Application

### Import SDK Frameworks

Import frameworks with CocoaPods, Carthage, or manually.

#### 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 MAP SDK to your Podfile. This will automatically pull in the AkaCommon framework.

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

MAP and mPulse SDKs share the same AkaCommon framework. CocoaPods takes care of the proper framework imports. Replace the MAP block in your Podfile with the following. Cronet may be added as well.

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

Refer to the mPulse 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 MAP SDK using Carthage:

1. Create or reuse an existing Cartfile. Open it in a text editor.
2. Add MAP 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/MAP/aka-map-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 **AkaMap.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 framework 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/AkaMap.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 mPulse SDK

The MAP SDK works seamlessly with Akamai's mPulse SDK. Both share the same AkaCommon framework. To install MAP SDK with mPulse 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/MAP/aka-map-ios.json"  
binary "https://downloads.pvoc-anaina.com/ios/mPulse/aka-mpulse-ios.json"
```

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

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

## Adding Frameworks Manually

Manual integration requires the AkaCommon and AkaMap frameworks. Frameworks are located in the package bundle, *Akamai\_iOS\_SDKs.zip*. The package can be obtained from Akamai's MAP 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 AkaCommon and AkaMap. 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 **AkaMap.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 .xcodproj file and may need modification depending on where you unzipped it (/frameworks/MAP-sdk-ios-21.11.1/ in this example). The strip\_frameworks.sh file is bundled alongside the AkaMap.framework file.

```
./frameworks/MAP-sdk-ios-21.11.1/strip_frameworks.sh
```

In this example, the folder structure is as follows:

```
/myproject/myproject.xcodeproj
/myproject/frameworks/MAP-sdk-ios-21.11.1/strip_frameworks.sh
/myproject/frameworks/MAP-sdk-ios-21.11.1/AkaMap.framework
```

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

```
Swift:
#ifdef Demo_Bridging_Header_h
#define Demo_Bridging_Header_h

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

#endif /* Demo_Bridging_Header_h */
```

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

## Initialize SDKs



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

**Objective-C:**

```
#import <AkaCommon/AkaCommon.h>
```

```
[AkaCommon configure];
```

**Swift:**

```
import AkaCommon
```

```
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 MAP APIs

Optional calls to the MAP framework are made on the AkaMap shared instance. Its API is defined through AkaMapProtocol.

**Objective-C:**

```
#import <AkaMap/AkaMap.h>
```

```
@property (strong, nonatomic) id<AkaMapProtocol> mapService;
```

```
self.mapService = [AkaMap shared];
```

**Swift:**

```
import AkaMap
```

```
var mapService: AkaMapProtocol?
```

```
mapService = AkaMap.shared()
```

## API Key

The MAP API key is specified in your Info.plist file. An API key is obtained from Akamai Control Center.

Optional: Initial pre-position content segments are also specified here. These segments are immediately joined the first time the API key is provided (i.e., the first time running the app with this API key).

Changing content segments after the first install requires the [‘subscribeSegments’ API](#).

It is strongly recommended that PII (personally identifiable information) not be directly used in naming your content segments.

Key	Type	Example
<code>com.akamai</code>	<code>Dictionary</code>	<code>{:}</code>
<code>&gt; map</code>	<code>Dictionary</code>	<code>{:}</code>
<code>&gt; api_key</code>	<code>String</code>	<code>123456ABCDEF</code>
<code>&gt; segments</code>	<code>Array</code>	<code>{}</code>
<code>&gt; Item 0</code>	<code>String</code>	<code>banner_images</code>
<code>&gt; Item ...</code>	<code>String</code>	<code>core_files</code>



## Integrating with Firebase Cloud Messaging (Optional)

*This step is optional and is intended for pre-positioning content in the background.*

MAP SDK uses Firebase Cloud Messaging (FCM) notifications to sync pre-positioning content and MAP configuration, including while the app is in the background.

In order for FCM background notifications to work, follow Google’s [Firebase Messaging integration guide](#). This involves enabling the remote-notification property for your app in Xcode. After FCM is integrated in the app, complete the following two steps:

1. Add or update the FCM server API key to the MAP portal under the app's configuration screen. The FCM server API key can be found in the FCM console's project settings, under Cloud Messaging.
2. Then add or update the key in the "Google FCM Key" field in the MAP SDK portal. The API Key is used by MAP's control server to trigger MAP-related notifications for pre-positioning the app's ingested content and notifying apps of configuration changes.

App Name	<input type="text" value="MapEngTest"/>	SDK License Key	<input type="text" value="087b2c362f803c16eaccd8808fcac492edececa"/> 
iOS Bundle ID	<input type="text" value="test"/>	Android Package Name	<input type="text" value="test"/>
Upload Push Notification Certificates			
Apple Prod APNS	<input type="text"/>	Google FCM Key	<input type="text" value="AAAA1b7auR4:APA91bER9Fe3jr3yhdPxWfBb6ZwbSr"/> 
	<input type="button" value="Upload File"/>		<input type="button" value="Upload File"/>
Apple Dev APNS	<input type="text"/>		
	<input type="button" value="Upload File"/>		
<input type="button" value="Save Changes"/>			

3. The Firebase portal provides step-by-step instructions for integrating and testing your app with Firebase. These instructions are available at <https://firebase.google.com/docs/cloud-messaging/ios/client>
4. Your app will need to pass remote notifications to MAP SDK for processing. Here is an example that implements the system callback for remote notifications and passes its message to AkaCommon. If AkaCommon returns a false response, MAP SDK will not process the message and your app must call the completion handler.

#### Objective-C:

```
- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler
{
    AkaCommon *akaCommon = [AkaCommon shared];
    BOOL mapHandledNotification = [akaCommon didReceiveRemoteNotification:userInfo fetchCompletionHandler:completionHandler];

    if (mapHandledNotification) {
        // MAP will call completionHandler after processing notification.
    } else {
        // otherwise your app should handle remote notification and call completion
        completionHandler(UIBackgroundFetchResultNoData);
    }
}
```

#### Swift:

```
func application(_ application: UIApplication, didReceiveRemoteNotification userInfo:
[AnyHashable : Any], fetchCompletionHandler completionHandler: @escaping
(UIBackgroundFetchResult) -> Void)
{
    let akaCommon = AkaCommon.shared()

    let mapHandledNotification = akaCommon.didReceiveRemoteNotification(userInfo,
fetchCompletionHandler: completionHandler)

    if (mapHandledNotification) {
        // MAP will call completionHandler after processing notification.
    } else {
        // otherwise your app should handle remote notification and call completion
completionHandler(UIBackgroundFetchResult.noData)
    }
}
```

MAP looks for “mapsdk” in the notification and returns *true* if so. Then it processes the notification and calls the fetch completion handler. If “mapsdk” is not present, *false* is returned, no processing is done by MAP SDK, and your app should handle the message. MAP will not call the completion handler in this case.

MAP uses notifications to alert the app of new content or configuration changes. Pre-positioned content will download in the background for a limited time, with the SDK resuming any unfinished downloads later.

## API Reference

This section discusses MAP features and provides example API usage. Features are enabled at the MAP Web portal.

### Pre-Positioned Content

Pre-positioned content begins loading onto the device at app start and continues downloading at regular intervals, including in background, until it is downloaded. Firebase Messaging (FCM) is used to notify the app that additional content is available.

At app start, the SDK refreshes content in cache using If-Modified-Since checks.

When changing subscriptions, content that is no longer subscribed to is purged from the cache.

Segments are joined in two ways.

- The first time the SDK starts, it joins the segments specified in Info.plist. These are used only on first app install or upon upgrade if it is the first time MAP SDK is installed.
- Segments are changed at runtime via the [subscribeSegments API](#).

Subsequent app starts continue using the latest subscribed segments. Segments in Info.plist are ignored after the first run.

Pre-positioning occurs automatically while your app runs. Content is used as follows:

- Your app's network requests are served from matching pre-positioned content, if available. Analytics record this as a CACHE\_FETCH event.
- If the content is not pre-positioned then it will be fetched from the network. Analytics record this as a CACHE\_MISS event.

## Updating Pre-Positioning Subscriptions

Segments are named collections of URLs defined on the MAP Web portal. With MAP's pre-positioning feature, the URLs in a segment are preemptively downloaded into the app's MAP SDK cache. Pre-positioned files are instantly available when the app's requests are served by MAP SDK.

The first app launch of MAP SDK will join segments listed in the Info.plist file. This prepares the first downloads as soon as possible. Relaunching the app ignores segments in the plist file, and instead continues subscriptions from when the app was last used.

Segment subscriptions may be changed any time through the MAP API.

### Objective-C:

```
// Example: join 2 segments, unsubscribing any previous segments.
NSSet *segments = [NSSet setWithObjects:@"basics", @"banner-images", nil];
[mapService subscribeSegments:segments];
```

### Swift:

```
// Example: join 2 segments, unsubscribing any previous segments.
let segments = Set(["basics", "banner-images"])
mapService?.subscribeSegments(segments)
```

To unsubscribe from a segment, pass the whole list of subscribed segments excluding the ones you want to remove. An empty set is used to unsubscribe from all segments.

For example, if you are subscribed to segments A, B, and C and you want to remove segment B, call subscribeSegments with A and C. Since B is no longer subscribed, its contents will be purged. Files are kept in cache if they remain in at least one subscribed segment.

### Objective-C:

```
// Join A, B, C.
NSMutableSet *segments = [NSMutableSet setWithObjects:@"A", @"B", @"C", nil];
[mapService subscribeSegments:segments];

// Unsubscribe B.
```

```
[segments removeObject:@"B"];
[mapService subscribeSegments:segments]; // segments = [A, C]
```

#### Swift:

```
// Join A, B, C.
var segments = Set(["A", "B", "C"])
mapService?.subscribeSegments(segments)

// Unsubscribe B.
segments.remove("B")
mapService?.subscribeSegments(segments) // segments = [A, C]
```

## MAP Cache Size

The MAP cache is shared by all features: pre-positioning, foreground pre-caching, and universal cache. Its total size is configurable as follows. This should be called early in your AppDelegate before the cache is filled.

#### Objective-C:

```
// Change cache size to 1 GB
mapService.cacheSize = 1 * 1024 * 1024 * 1024;
```

#### Swift:

```
// Change cache size to 1 GB
mapService?.cacheSize = 1 * 1024 * 1024 * 1024
```

## NSURLConnection

Requests using NSURLConnection automatically received MAP SDK features. For example, an asynchronous NSURLConnection can be created as before and will see MAP benefits such as pre-positioning.

#### Objective-C:

```
NSURL *requestURL = [NSURL URLWithString:@"https://www.akamai.com"];
NSURLRequest *request = [NSURLRequest requestWithURL:requestURL];
NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:request
delegate:self];
// ...followed by the asynchronous response handlers: connection:didReceiveResponse:,
connection:didReceiveData:, etc.
```

#### Swift:

```
let requestURL = URL(string: "https://www.akamai.com")!
let request = URLRequest(url: requestURL)
let connection = NSURLConnection(request: request, delegate: self)
// ...followed by asynchronous response handlers.
```

Synchronous connections are similarly straightforward. No changes to the connection are required to benefit from MAP SDK acceleration.

```
Objective-C (deprecated in iOS 9):
NSData *data = [NSURLConnection sendSynchronousRequest:request returningResponse:&response
error:&error];
```

## UIWebView

UIWebView will also use prepositioned content automatically and without modification.

## WKWebView

Loading cached resources in the **WKWebView** is challenging because it is executed in the separate process/thread. MAP SDK does not intercept traffic of WKWebView by default. To use MAP cache in the **WKWebView** you need to use the **WKURLSchemeHandler**. The **WKURLSchemeHandler** protocol is for loading resources with URL schemes that WebKit doesn't know how to handle.

Notice that the website which you're loading with the **WKWebView** must implement that protocol.

```
HTML:

```

In the **WKURLSchemeHandler** you can retrieve resources from the MAP cache and pass them to the **WKWebView** or perform the HTTP request if the cache does not contain a particular resource.

```
Objective-C:
// Setup your custom handler
#import <WebKit/WebKit.h>

@interface CustomURLSchemeHandler: NSObject<WKURLSchemeHandler>
@property (strong, nonatomic) NSString *scheme;
- (instancetype)initWithScheme:(NSString *)scheme;
@end

@implementation CustomURLSchemeHandler

- (instancetype)initWithScheme:(NSString *)scheme {
    self = [super init];
```

```

    if (self) {
        self.scheme = scheme;
    }
    return self;
}

- (void)webView:(nonnull WKWebView *)webView startURLSchemeTask:(nonnull
id<WKURLSchemeTask>)urlSchemeTask {
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        NSURL *requestUrl = urlSchemeTask.request.URL;
        NSString *rewrittenUrl = [requestUrl.absoluteString
stringByReplacingOccurrencesOfString:self.scheme withString:@"https"];
        NSURL *endpoint = [NSURL URLWithString:rewrittenUrl];

        AkaCachedFile *cachedFile = [[AkaMap shared] cachedFileForURL:rewrittenUrl];
        if (cachedFile != nil) {
            NSLog(@"Cache hit. %@", rewrittenUrl);

            NSHTTPURLResponse *httpResponse = [[NSHTTPURLResponse alloc] initWithURL:endpoint
statusCode:200 HTTPVersion:@"HTTP/1.1" headerFields:cachedFile.responseHeaders];
            [urlSchemeTask didReceiveResponse:httpResponse];

            NSData *httpBody = [NSData dataWithContentsOfFile:cachedFile.localPath];
            [urlSchemeTask didReceiveData:httpBody];

            [urlSchemeTask didFinish];
        } else {
            NSLog(@"Cache miss, performing HTTP request to %@", rewrittenUrl);

            [[[NSURLSession sharedSession] dataTaskWithURL:endpoint completionHandler:^(NSData
*data, NSURLResponse *response, NSError *error) {
                if (error != nil) {
                    [urlSchemeTask didFailWithError:error];
                    return;
                }
                [urlSchemeTask didReceiveResponse:response];
                [urlSchemeTask didReceiveData:data];
                [urlSchemeTask didFinish];
            }] resume];
        }
    });
}

- (void)webView:(nonnull WKWebView *)webView stopURLSchemeTask:(nonnull
id<WKURLSchemeTask>)urlSchemeTask {
}

@end

```



```
// and utilize it with your WKWebView
NSString *scheme = @"my-custom-scheme";
WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];
[configuration setURLSchemeHandler:[[CustomURLSchemeHandler alloc] initWithScheme:scheme]
forURLScheme:scheme];

webView = [[WKWebView alloc] initWithFrame:CGRectZero configuration:configuration];
NSURL *myURL = [NSURL URLWithString:@"my-custom-scheme://your-origin.com"]; // this should
be set to my-custom-scheme if you want to cache response to this request
NSURLRequest *myRequest = [NSURLRequest requestWithURL:myURL];
[webView loadRequest:myRequest];
```

#### Swift:

```
// Setup your custom handler
import WebKit

class CustomURLSchemeHandler: NSObject, WKURLSchemeHandler {
    let mapService = AkaMap.shared()
    let scheme: String

    init(scheme: String) {
        self.scheme = scheme
    }

    func webView(_ webView: WKWebView, start urlSchemeTask: WKURLSchemeTask) {
        DispatchQueue.global().async {
            let requestUrl = urlSchemeTask.request.url!
            let rewrittenUrl = requestUrl.absoluteString.replacingOccurrences(of: self.scheme,
with: "https")
            let endpoint = URL(string: rewrittenUrl)!

            if let cachedFile = self.mapService.cachedFile(forURL: rewrittenUrl) {
                print("Cache hit. \(rewrittenUrl)");

                let httpResponse = HTTPURLResponse(url: endpoint, statusCode: 200, httpVersion:
"HTTP/1.1", headerFields: cachedFile.responseHeaders)
                urlSchemeTask.didReceive(httpResponse!)

                let httpBodyFilePath = URL(fileURLWithPath: cachedFile.localPath)
                if let httpBody = try? Data(contentsOf: httpBodyFilePath) {
                    urlSchemeTask.didReceive(httpBody)
                }

                urlSchemeTask.didFinish()
            } else {
                print("Cache miss, performing HTTP request to \(rewrittenUrl).");
```

```

        URLSession.shared.dataTask(with: endpoint) {(data, response, error) in
            if let error = error {
                urlSchemeTask.didFailWithError(error)
                return
            }
            urlSchemeTask.didReceive(response!)
            urlSchemeTask.didReceive(data!)
            urlSchemeTask.didFinish()
        }.resume()
    }
}

func webView(_ webView: WKWebView, stop urlSchemeTask: WKURLSchemeTask) {
}

// and utilize it with your WKWebView
let scheme = "my-custom-scheme"
let configuration = WKWebViewConfiguration()
configuration.setURLSchemeHandler(CustomURLSchemeHandler(scheme: scheme), forURLScheme:
scheme)

let webView = WKWebView(frame: .zero, configuration: configuration)
let myURL = URL(string: "my-custom-scheme://your-origin.com") // this should be set to
my-custom-scheme if you want to cache response to this request
let myRequest = URLRequest(url: myURL!)
webView.load(myRequest)

```

## NSURLSession

NSURLSession may use either the shared app session or a custom configuration. MAP SDK automatically handles the shared session.

### Objective-C:

```

// MAP is used implicitly with the shared session
NSURLSession *sharedSession = [NSURLSession sharedSession];
NSURL *requestURL = [NSURL URLWithString:@"http://www.akamai.com/"];
[[sharedSession dataTaskWithURL:requestURL] resume];

```

### Swift:

```

// MAP is used implicitly with the shared session
let sharedSession = URLSession.shared
if let requestURL = URL(string: "https://www.akamai.com/") {
    sharedSession.dataTask(with: requestURL).resume()
}

```

More common for NSURLSessions to have a custom configuration. Custom configurations must be passed into MAP SDK for setup.

#### Objective-C:

```
// MAP must be added to custom URLSession configurations

NSURLSessionConfiguration *sessionConfig = [NSURLSessionConfiguration
defaultSessionConfiguration];

// ... modify sessionConfig as required by the app ...

[[AkaCommon shared] interceptSessionsWithConfiguration:sessionConfig]; // sessionConfig now
uses MAP

NSURLSession *customSession = [NSURLSession sessionWithConfiguration:sessionConfig
delegate:self delegateQueue:nil];

NSURL *requestURL = [NSURL URLWithString:@"http://www.akamai.com/"];

[[customSession dataTaskWithURL:requestURL] resume];
```

#### Swift:

```
// MAP must be added to custom URLSession configurations
let sessionConfig = URLSessionConfiguration.default

// Tell sessionConfig to use MAP SDK
AkaCommon.shared().interceptSessions(with: sessionConfig)

let customSession = URLSession.init(configuration: sessionConfig, delegate: self,
delegateQueue: nil)

if let requestURL = URL(string:"https://www.akamai.com/") {
    customSession.dataTask(with: requestURL).resume()
}
```

## Custom Event Tracking

Custom events are actions triggered by the app to produce metrics on the Web portal. This could be a user activity such as a button click, a timer for duration spent on a particular app screen, or an internal timer for resources to load. Multiple events may be tracked concurrently. Custom events are classified as timed or instantaneous.

### Timed Events

A *timed* event has two endpoints, start and end. The two endpoints are paired by calling `startEvent` and `stopEvent` with matching event names and recording the time between these endpoints. In addition to logging durations, timed events are useful for monitoring the network activity between endpoints. For

example, custom event start and stop points can be recorded in-line with network activity and reviewed collectively from the Web portal. Note that unrelated, asynchronous requests may be recorded during user events depending on your app design. Do not add any data to the event name that has privacy implications.

**Objective-C:**

```
// timed user event
[mapService startEvent:@"Event Name"];
// ...perform a measurable activity...
[mapService stopEvent:@"Event Name"];
```

**Swift:**

```
// timed user event
mapService?.startEvent("Event Name")
// ...perform a measurable activity...
mapService?.stopEvent("Event Name")
```

## Instantaneous Events

*Instantaneous* events are recorded in the server logs as a name and a series of timestamps when the event occurred. They are useful for recording a sequence of activities or to form a timeline of events. Note that instantaneous events are in Tech Preview and are not yet displayed in the portal.

**Objective-C:**

```
// instantaneous event
[mapService logEvent:@"tapped home button"];
```

**Swift:**

```
// instantaneous event
mapService?.logEvent("tapped home button")
```

## Network Aware Experience

The SDK provides API access to the client-side network quality state in order to help developers augment client requests. The return value is either excellent, good, or poor. The meaning of these values is defined in the configuration portal.

The next example suggests how loading content may be tweaked based on network quality status.

**Objective-C:**

```
id<VocNetworkQuality> networkQuality = mapService.networkQuality;
switch (networkQuality.qualityStatus) {
case VocNetworkQualityPoor:
    [self flashMessage:nil withTitle:@"Network Quality: Poor"];
    // Exit download
    break;
```

```

case VocNetworkQualityGood:
    [self flashMessage:nil withTitle:@"Network Quality: Good"];
    // Throttle download
    break;
case VocNetworkQualityExcellent:
    [self flashMessage:nil withTitle:@"Network Quality: Excellent"];
    // Download content
    break;
case VocNetworkQualityUnknown:
    [self flashMessage:nil withTitle:@"Network Quality: Unknown"];
    break;
}

```

```

Swift:
if let netQuality = mapService?.networkQuality()?.qualityStatus {
    switch (netQuality) {
    case .excellent:
        print("Network Quality: Excellent")
    case .good:
        print("Network Quality: Good")
    case .poor:
        print("Network Quality: Poor")
    case .notReady:
        print("Network Quality: Not ready")
    case .unknown:
        print("Network Quality: Unknown")
    }
}
}

```

To receive callbacks as the network is measured, see [Receiving MAP Service Callbacks](#).

## Pinned Certificates / Custom TLS Certificate Handling

MAP 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.

```

Objective-C:
- (void)vocService:(nonnull VocService *)vocService
  didReceiveChallengeForRequest:(nonnull NSURLRequest *)originalRequest
  currentRequest:(nonnull NSURLRequest *)currentRequest
  challenge:(nonnull NSURLAuthenticationChallenge *)challenge
  modifiedTrust:(nullable SecTrustRef) modifiedTrust
  completion:(nonnull void (^)(NSURLSessionAuthChallengeDisposition disposition,
                               NSURLCredential * _Nullable credential))completion
{
    // custom pinned certificate handling
}

```

**Swift:**

```
func vocService(_ vocService: VocService, didReceiveChallengeFor originalRequest:
URLRequest, currentRequest: URLRequest, challenge: URLAuthenticationChallenge,
modifiedTrust: SecTrust?, completion: @escaping (URLSession.AuthChallengeDisposition,
URLCredential?) -> Void) {
    // custom pinned certificate handling
}
```

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

- the original request made by the app for identifying the URL in question,
- the TLS server challenge,
- the modified trust object that can be used for verification, and
- a completion block to be called with the result of the evaluation.

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

To define a delegate handler that receives this callback, see the next section, [Receiving MAP Service Callbacks](#).

## Receiving MAP Service Callbacks

Setting a custom MAP SDK delegate allows your app to receive optional callbacks. These are

- custom pinned certificate handling, as described in the previous section, or
- network quality state measurements, if that feature is enabled.

The following example makes AppDelegate the MAP SDK delegate. The delegate can be any object that implements *VocServiceDelegate*.

**Objective-C:**

```
#import <AkaCommon/AkaCommon.h>
#import <AkaMap/AkaMap.h>

@interface AppDelegate () <VocServiceDelegate>

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [AkaCommon configure];

    // Optional: set a custom MAP delegate.
    id<AkaMapProtocol> mapService = [AkaMap shared];
    [mapService setDelegate:self withOperationQueue:NSOperationQueue.mainQueue];
}
```

```

// Optional handler: Demonstrate receiving a callback.
// Called when Network Quality is enabled and measures a new value.
- (void)vocService:(id<VocService>)vocService
networkQualityUpdate:(id<VocNetworkQuality>)networkQuality
{
    NSLog(@"App received network quality update: %@", @(networkQuality.qualityStatus));
}

// Optional pinned certificate handler, -didReceiveChallengeForRequest:, described above

Swift:
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, VocServiceDelegate {

    var mapService: AkaMapProtocol?

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

        // Optional: set a custom MAP delegate.
        mapService = AkaMap.shared()
        mapService?.setDelegate(self, with: .main)
    }

    // Optional MAP SDK callback
    func vocService(_ vocService: VocService, networkQualityUpdate networkQuality:
VocNetworkQuality) {
        print("App received network quality update: \(networkQuality.qualityStatus)")
    }

    // Optional pinned certificate handler, didReceiveChallengeForRequest:, described above
}

```

Setting `[mapService -setDelegate:nil withOperationQueue:nil]` will stop MAP SDK from sending delegate notifications.

## Cache-Control Request Parameters

Requests passing through MAP SDK are served transparently from either the MAP cache, the OS cache, or the server. The SDK ensures delivery of fresh content by following content expiration headers, performing refreshes as necessary. In certain cases, it may be desirable to override this behavior. For example, in case of poor connectivity, the app may accept stale responses for a particular request. In another case, the app may decide to force cached content to be revalidated by the origin server by controlling its expiry time and date.

This can be controlled with standard HTTP parameters added directly to the `NSURLRequest`:

#### Objective-C:

```
NSMutableDictionary *cacheControlHeaders = [NSMutableDictionary new];
cacheControlHeaders[@"Cache-Control"] = @"no-cache";
NSURL *requestURL = [NSURL URLWithString:@"https://www.akamai.com/some_image.jpg"];
NSMutableURLRequest *mRequest = [NSMutableURLRequest requestWithURL:requestURL];
[mRequest setAllHTTPHeaderFields:cacheControlHeaders];

NSURLSessionDataTask *dataTask = [self.mySession dataTaskWithRequest: mRequest];
[dataTask resume];
```

#### Swift:

```
let cacheControlHeaders = ["Cache-Control": "no-cache"]
if let requestURL = URL(string:"https://www.akamai.com/some_image.jpg") {
    var request = URLRequest(url: requestURL)
    request.allHTTPHeaderFields = cacheControlHeaders
    let dataTask = session.dataTask(with: request)
    dataTask.resume()
}
```

MAP's cache behavior can be controlled with the following key-value pairs.

- Pragma:no-cache: Forces SDK to revalidate cached response.
- Cache-Control:no-cache: Same as Pragma:no-cache.
- Cache-Control:max-age='x': Forces SDK to select expiry for the content as Min('x', expiry calculated from response headers).
- Cache-Control:max-stale='x': If assigned a value, the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If present and no value is assigned to max-stale, then the client is willing to accept a stale response of any age. Developers can use this under poor network conditions to serve stale responses.

Note: max-age/max-stale is ignored if no-cache is present.

## Debugging APIs

MAP's error 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 function	frequency	purpose
setDebugConsoleLog:<BOOL>	continuous until disabled	Running reports of URLs intercepted by MAP SDK.
printCurrentConfiguration	one time	Display enabled features, timestamp of last configuration received.



printCache	one time	Lists subscribed segments. Also lists all URLs known to the cache with their statuses (cached, failed to download, in queue, ...).
------------	----------	--

#### Objective-C:

```
// enable real-time extended debug info to Xcode console
AkaCommon.shared.debugConsoleEnabled = YES;
[mapService setDebugConsoleLog:YES];

// print subscribed segments, followed by each URL with its download status
[mapService printCache];

// print last received SDK features as configured through the portal
[mapService printCurrentConfiguration];
```

#### Swift:

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

// print subscribed segments, followed by each URL with its download status
mapService?.printCache()

// print last received SDK features as configured through the portal
mapService?.printCurrentConfiguration()
```

The debugSendAnalytics call may be used to test that records are sent correctly from your app. It immediately sends the latest batch of analytics from the device and reports to the developer console. This results in additional uploads and should not be used in production code.

#### Objective-C:

```
// Debug only - test analytics upload by sending outside of regular cycle
[self.mapService debugSendAnalytics];
```

#### Swift:

```
// Debug only - test analytics upload by sending outside of regular cycle
mapService?.debugSendAnalytics()
```

Note: there will be a delay before analytics are aggregated for display on the portal.

## MAP State Properties

The current state of MAP may be queried at runtime.

## Enabled Flag

This returns whether MAP SDK has received a valid configuration from the Web portal and that MAP features are turned on.

### Objective-C:

```
BOOL mapEnabled = [mapService enabled];
```

### Swift:

```
let mapEnabled = mapService?.enabled()
```

## Subscribed Segments

This returns a set of content segments to which the app currently subscribes.

### Objective-C:

```
NSSet *subscribedSegments = [mapService subscribedSegments];
```

### Swift:

```
let subscribedSegments = mapService?.subscribedSegments()
```

These segments are subscribed but not necessarily downloaded yet. To print the current cache, see [Debugging APIs](#).

## Direct Access to Cached Files

There are occasions when the app wants access to the cached items, such as for moving them into a web cache that MAP does not intercept.

Cached files may be directly accessed from the app file system in one of two ways. There are API calls for getting all items at once or a specific cached URL. In addition, the app may subscribe to cache change callbacks.

Cached file metadata is returned in the *AkaCachedFile* class. This encapsulates a single cached file using these properties:

<i>property</i>	<i>type</i>	<i>description</i>
urlString	String	URL of cached resource.
responseHeaders	Dictionary	Response headers returned when the item was cached.
inferredMIMETYPE	String	Content MIME type, usually drawn from server response headers.
localPath	String	Path to body data stored in the app sandbox. This can be loaded directly into an NSData object.

## Cache API and Record Format

### Direct cache API examples

#### Objective-C:

```
// returns a set of metadata for all cached files.
NSSet <AkaCachedFile*> *allCached = [mapService cachedFiles];

// returns metadata for a single URL, or nil if not cached.
NSString *urlString = @"https://www.akamai.com/index.html";
AkaCachedFile *cachedFile = [mapService cachedFileForURL:urlString];

// example of creating HTTP response
NSURL *url = [NSURL URLWithString:urlString];
NSHTTPURLResponse *httpResponse = [[NSHTTPURLResponse alloc] initWithURL:url statusCode:200
HTTPVersion:@"HTTP/1.1" headerFields:cachedFile.responseHeaders];

// example of reading body data
NSData *bodyData = [NSData dataWithContentsOfFile:cachedFile.urlString];

// now there is both a response and body for caching elsewhere
```

#### Swift:

```
// returns a set of metadata for all cached files.
let cachedFiles = mapService?.cachedFiles()

// returns metadata for a single URL, or nil if not cached.
let singleCachedFile = mapService?.cachedFile(forURL: "https://www.akamai.com/index.html")

// example of reading header and body data
let urlString = "https://www.akamai.com/index.html"
if let url = URL(string: urlString),
    let cachedFile = mapService?.cachedFile(forURL: urlString) {

    let httpResponse = HTTPURLResponse(url: url, statusCode: 200, httpVersion: "HTTP/1.1",
headerFields: cachedFile.responseHeaders)

    let httpBodyFilePath = URL(fileURLWithPath: cachedFile.localPath)
    if let httpBody = try? Data(contentsOf: httpBodyFilePath) {
        // now there is both a response and body for caching elsewhere
    }
}
```

## Implementing a Cache Listener

Implement `VocServiceDelegate` in order to receive cache changes in real-time. See [Receiving MAP Service Callbacks](#). The function to implement is:

#### Objective-C:

```
- (void)cacheAdded:(nullable NSSet<AkaCachedFile*> *)added  
    updated:(nullable NSSet<AkaCachedFile*> *)updated  
    removed:(nullable NSSet<AkaCachedFile*> *)removed;
```

#### Swift:

```
func cacheAdded(_ added: Set<AkaCachedFile>?,  
    updated: Set<AkaCachedFile>?,  
    removed: Set<AkaCachedFile>?) {  
}
```

*Added* items have already been added to cache and are ready for access.

*Updated* items indicate a change in the header or body data. The updated records include the complete updates.

*Removed* items have already been taken out of cache. These records contain only the URL indicating that that URL is no longer available in the MAP cache.

## QUIC Library Integration

MAP SDK has the option to download using the [QUIC protocol](#) as implemented in the Chromium/Cronet library. The following conditions must be present to use QUIC:

- QUIC must be enabled at the MAP portal.
- An additional framework needs to be included in the client app. See the Cronet option discussed in [CocoaPods](#).
- Your content server(s) must support QUIC.

With QUIC enabled and Cronet imported, all app requests through MAP attempt using QUIC by default. If the server supports QUIC the response is served over QUIC; otherwise it falls back to HTTP/S.

## Integrating QUIC

Cronet is an optional dependency for using the QUIC protocol. To use it, replace the MAP block in your Podfile with this:

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

## Brotli Library Integration

Brotli compression is added automatically by iOS 11 and up. Brotli compression must be enabled by your content server.

## mPulse Integration

See [Integrating with mPulse SDK](#) for setting up an Xcode workspace with both MAP and mPulse SDKs.

## Appendix - Requirements and Dependencies

### MAP Debug Logs

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

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

Additional debug info is available through the Xcode console. See [Debugging APIs](#).

### Bitcode

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

### App Transport Security

iOS 9.0 introduced a new app security feature called App Transport Security (ATS) and it is enabled by default. With ATS, connections must use secure HTTPS instead of HTTP. Additionally, if the app contents that MAP SDK needs to download contain HTTP URLs, those downloads will fail. Application developers must ensure that either:

1. [preferred] HTTPS is used for all content URLs. This is done during ingest at the MAP portal.
2. [insecure] Or ATS exceptions can be added for certain domains by adding the following keys to your app's Info.plist file

Optional ATS key to allow insecure (HTTP) content [not recommended]:

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSExceptionDomains</key>
```

```

<dict>
  <key>REPLACE-WITH-YOUR-CONTENT-DOMAIN.com</key>
  <dict>
    <key>NSIncludesSubdomains</key>
    <true/>
    <key>NSExceptionAllowsInsecureHTTPLoads</key>
    <true/>
  </dict>
  <key>EXAMPLE-TWO-DOMAIN.com</key>
  <dict>
    <key>NSIncludesSubdomains</key>
    <true/>
    <key>NSExceptionAllowsInsecureHTTPLoads</key>
    <true/>
  </dict>
</dict>
</dict>

```

## Disabling Redirect Behavior

MAP SDK automatically follows redirects. To return the redirect response directly to your app, set the following MAP config property.

```

Objective-C:
mapService.config.autoFollowRedirects = NO;

Swift:
mapService?.config.autoFollowRedirects = false;


```

Your app will then receive a response with the status code 301, 302, etc. and a location header, to which you can create a follow-up request.

This property supersedes NSURLSession's `-willPerformHTTPRedirection:` call, which should be omitted or should simply return the recommended request. Do not implement `-willPerformHTTPRedirection` with a nil return value even if you choose to not auto-redirect. MAP SDK will follow or not follow based on the *autoFollowRedirects* property.

## Framework Size

MAP SDK consists of the AkaMap framework, a shared AkaCommon framework, and an optional Cronet framework. These are all 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
AkaMap	18.0 MB	2.5 MB	2.3 MB
AkaCommon (shared)	3.0 MB	452 KB	416 KB
Cronet (optional)	17.0 MB (no bitcode)	4.3 MB	4.0 MB
Total download size in app store		2.9 or 7.2 MB depending on Cronet	2.7 or 6.7 MB depending on Cronet

Table 2 - Framework sizes