SAS® Customer Intelligence 360 Mobile SDK Integration with a Flutter App: Cookbook

SAS® Documentation
March 29, 2024

Contents

Overview

SAS Customer Intelligence 360 mobile SDKs (also called *SASCollector*) enable you to add support for event collection and to publish content to native Android and iOS apps. You can use collected events to understand how your app is performing and target users for distribution of content.

- The Android mobile SDK for SAS Customer Intelligence 360 is a self-contained Java library in the form of a JAR file.
- The iOS mobile SDK for SAS Customer Intelligence 360 is an iOS framework that is a directory of files in a particular structure. The directory includes headers, binaries, and resource files.

You can use Flutter, an open-source software development kit, to design a native mobile application that uses only one codebase for both Android and iOS. The programming language that is used to develop a mobile app with Flutter is Dart.

The purpose of this document is to provide guidance on how you can integrate SAS Customer Intelligence 360 mobile SDKs for Android and iOS with a mobile app that is built using Flutter technology. This document shows how to create a plug-in that adds the capabilities of SAS Customer Intelligence mobile SDKs.

In addition, there is a Mobile SDK Flutter Package (.zip) that contains a sample mobile_sdk_flutter project.

What You Should Know in Order to Use This Cookbook

This cookbook assumes that the following statements are true:

- You are familiar with SAS Customer Intelligence 360 mobile SDKs.
- You have experience with the development of Android, iOS, and Flutter mobile apps and the programming languages that are used to design them.
- You understand the roles and responsibilities of the individuals who work with a mobile app, mobile in-app messages, and push notifications.

Roles and Responsibilities

Collaboration between marketers, business analysts, and mobile app developers is critical. To ensure success, it is important that each of the individuals in these key roles has direct access to the required resources. A successful integration of a mobile application with SAS Customer Intelligence 360 depends on proper configuration.

Note: In SAS Customer Intelligence 360, the individual who is working in the application is sometimes referred to as the SAS Customer Intelligence 360 user. In the context of delivering mobile content, this individual is typically a mobile marketer.

Here are examples of items that require collaboration:

- **Mobile messaging.** Firebase Cloud Messaging (FCM) for Android devices and Apple Push Notification service (APNs) for iOS devices are used to deliver mobile messages (push notifications and in-app messages). The mobile app developer registers the mobile app with those services and obtains certificates and keys that a SAS Customer Intelligence 360 user uses to register the mobile app with SAS Customer Intelligence 360. For more information, see Register a Mobile Application in *SAS Customer Intelligence 360: Administration Guide*.
- **Mobile spots.** The marketer and the mobile app developer work together to identify places (referred to as *spots*) in the mobile app where the marketer can use SAS Customer Intelligence 360 to deliver content. The mobile app developer must provide the SAS Customer Intelligence 360 user with spot IDs and details such as spot dimensions. In SAS Customer Intelligence 360, the spot ID is required to create a task that delivers content to a specific location in the mobile app. For more information, see Creating Mobile Spots in *SAS Customer Intelligence 360: User's Guide*.
- **Custom mobile events.** The mobile app developer provides a SAS Customer Intelligence 360 user with mobile event keys and custom attributes (if any). In SAS Customer Intelligence 360, the mobile event key is required to create custom events that represent specific behaviors in the mobile app. These behaviors can act as triggers for sending content to the app, or they can be used for personalization. For more information, see Create a Custom Event for a Mobile App in *SAS Customer Intelligence 360: User's Guide*. Also see Working with Events for Android and Working with Events for iOS in *SAS Customer Intelligence 360: Developer's Guide for Mobile Applications*.
- **Geofences and beacons.** The marketer or SAS Customer Intelligence 360 user can define (and upload to SAS Customer Intelligence 360) virtual geographic boundaries called *geofences* or points called *beacons* that can determine content that a mobile app user receives when they enter that space. The mobile app developer codes the mobile app (using the mobile SDKs) to include location services and monitor location

events. For more information, see Upload Location Data in *SAS Customer Intelligence 360: Administration Guide*. Also see Enable Location-Based Features for iOS and Enable Location-Based Features for Android in *SAS Customer Intelligence 360: Developer's Guide for Mobile Applications*.

- **Session settings.** The marketer defines settings for mobile app sessions so that SAS Customer Intelligence 360 mobile SDKs know when to continue a current session or start a new one. For more information, see Page and Session in *SAS Customer Intelligence 360: Administration Guide*.

Initial Setup

The following applications are used in this cookbook:

- Flutter SDK. See https://docs.flutter.dev/get-started/install/macos for Mac and https://docs.flutter.dev/get-started/install/windows for Windows.

  Note: Flutter cannot run iOS on Windows. It is therefore recommended to use a Mac for development.

- Android Studio and Xcode. Android Studio Chipmunk 2021.2.1 and Xcode 13.4.1 are used in this cookbook.
- For developing the Flutter plug-in, both Android Studio and Visual Studio Code (VSCode) can be used. In this cookbook, VSCode 1.70.2 is used. Go to this link to download and install VSCode: https://code.visualstudio.com/download.
- Flutter and Dart plug-ins from VSCode's Extensions. The plug-ins are needed for Flutter to work on VSCode.

Naming Convention

In this cookbook, `com.sas.SASIA.mobile_sdk_flutter` refers to the package name for the example Flutter project. For Android, the name is the package ID and for iOS the name is the bundle ID. That package name is displayed in sample code, directory paths, and figures throughout this cookbook.

Create a Flutter Plug-in Project

A Flutter app is built using Dart, a programming language. Flutter does not read native Android (Java or Kotlin) and iOS (Objective-C or Swift) languages. To enable you to use the Android and iOS SAS Customer Intelligence 360 mobile SDKs, the easiest approach is to build a wrapper that is a Flutter plug-in, around the SDKs to make them usable by Flutter apps.

The Flutter plug-in works by passing messages through channels between the Dart plug-ins and the native Android or iOS platforms. There are two types of channels in Flutter: the event channel and the method channel. The procedures in this guide use only the method channel.

Create a Plug-in Template

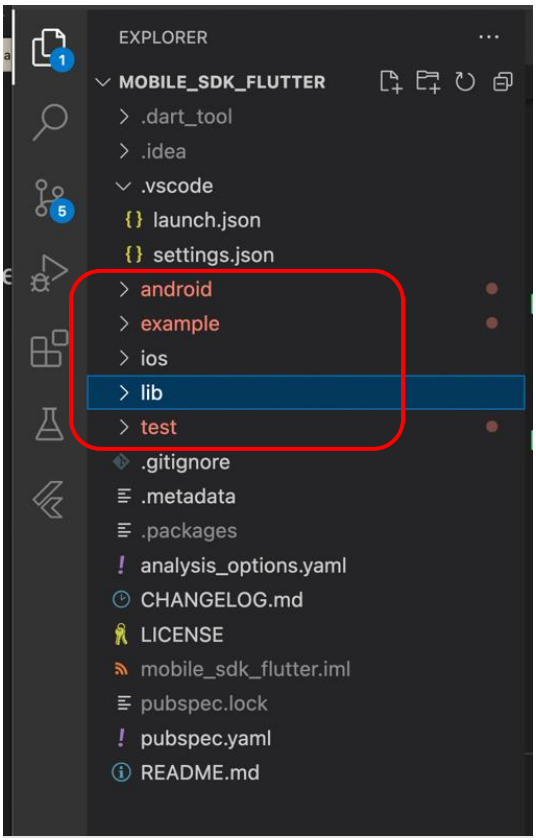
To generate the plug-in template that contains the folders that you need for your Flutter plug-in project:

1. Open a terminal session and navigate to the desired location for this project.
2. Use the command shown in the example below to create a Flutter plug-in project that specifies to use Java for Android and Objective-C for iOS:

   ```
   flutter create --org com.sas.SASIA --template=plugin  --
   platforms=android,ios  -a java  -i objc  mobile_sdk_flutter
   ```

   As mentioned in “Initial Setup”, com.sas.SASIA.mobile_sdk_flutter is the package name being used as an example in this cookbook. Replace that with the name of your project.

The resulting project includes these folders: `android`, `example`, `ios`, `lib`, and `test`.



Here is a description of the folders:

- The `android` and `ios` folders contain code that exposes native functionality to the rest of the Flutter app in Dart.
- The `lib` folder is where the Dart files that are used by the app are stored. It contains the definition of the functions that can be understood and used by Flutter apps.
- The `example` folder contains a starter Flutter app, sometimes referred to as the example project. It can be used for testing the Flutter plug-in.
- The `test` folder can be used to write unit test code.

Obtain the SAS Customer Intelligence 360 Mobile SDKs

These are the two ways to obtain SAS Customer Intelligence 360 mobile SDKs:

- A SAS Customer Intelligence 360 user can download the mobile SDKs through the user interface for SAS Customer Intelligence 360 and deliver the SDK ZIP file (SASCollector_<applicationID>.zip) to you to install.

  The Android SDK and the iOS SDK are distributed together as a single ZIP package.
- You can access the mobile SDKs from a public repository.

- For Android, see Configure a Dependency on the Maven Repository for the Mobile SDK in *SAS Customer Intelligence 360: Developer's Guide for Mobile Applications*.
- For iOS, see Use Swift Package Manager to Set Up the Mobile SDK in *SAS Customer Intelligence 360: Developer's Guide for Mobile Applications*.

Note: A SASCollector.properties file (for Android) and a SASCollector.plist file (for iOS) contain necessary information to successfully implement the mobile SDKs, including the customer's selected tenant and mobile app ID. The files are not included in the public repository. The files must be obtained from the mobile SDK ZIP package that is downloaded from SAS Customer Intelligence 360.

Add SAS Customer Intelligence 360 Mobile SDK Libraries

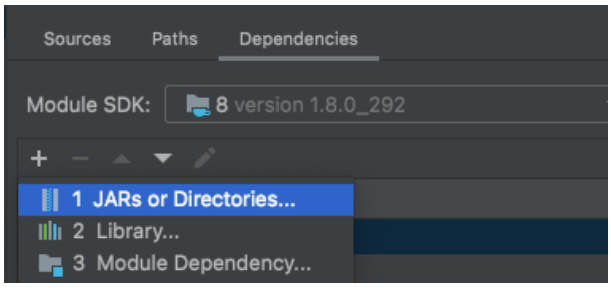
You need to add the SASCollector framework (library) to the Flutter plug-in project that you created.

Android

1. In Android Studio, open the Flutter plug-in project.
2. In the `android` folder, create a folder called `libs`.
3. Navigate to the folder that contains the SAS Customer Intelligence 360 mobile SDK ZIP file (SASCollector_<applicationID>.zip). Unzip the file, navigate to the `android` folder, and find SASCollector.jar. Copy SASCollector.jar from SASCollector_<applicationID>.zip into the `libs` folder.
4. Go to **File** => **Project Structure** => **Modules**.
5. Select the `android` folder.

   Note: In Android Studio, the folder name appears as mobile_sdk_flutter_android.

6. In the center pane, click the **Dependencies** tab, click **+**, and then select **JARs or Directories.**



7. Find SASCollector.jar and click **Open**.

   Note: Do not select **Export**.

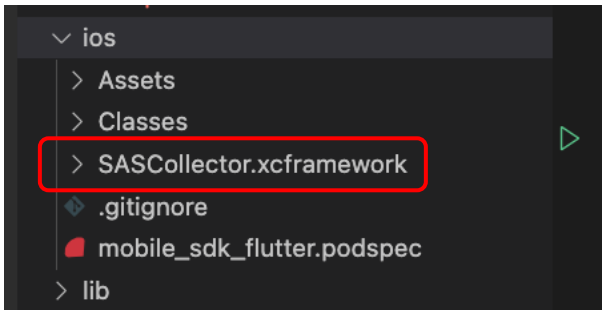
8. In Android Studio or VSCode, in your Flutter plug-in project, add the following JAR file dependency under **Dependencies** in build.gradle (inside the `android` folder.)

```
implementation files('./libs/SASCollector.jar')
```

9. If both the iPhone simulator and Android simulator are installed, you are prompted to choose which one to use. Choose the Android simulator. Verify that the build succeeds, and that the app starts.

iOS

1. Navigate to the folder that contains the SAS Customer Intelligence 360 mobile SDKs ZIP file (SASCollector_<applicationID>.zip).
2. Open the `iOS` folder and double-click the SASCollector.zip file to unzip it to xcframework named `SASCollector.xcframework`.
3. Open your Flutter plug-in project.
4. Add SASCollector.xcframework to the `iOS` folder.



5. Open mobile_sdk_flutter.podspec (shown in the figure above) and add these lines before "`end`":

```
s.preserve_path = 'SASCollector.xcframework/**/*'
s.xcconfig = {'OTHER_LDFLAGS' => '-framework SASCollector'}
s.vendored_framework = 'SASCollector.xcframework'
```

6. In VSCode's integrated terminal, navigate (by using the *cd* command, for example) to the `example` folder (which is where the example project resides) in your Flutter plug-in project, and then type "flutter run". An iPhone simulator needs to be open for the command to find it and run.
7. Verify that the build is successful, and that the app starts.

IMPORTANT: Before proceeding to implement the plug-in, please be aware that the plug-in will not work on your app until you add SASCollector.properties to Android and SASCollector.plist files to iOS. For instructions, see "Configure the Example Flutter App".

Basic Functionality

Some mobile app events, such as focus and defocus, do not need an explicit API call in the Flutter plug-in to make them work. The integration of SAS Customer Intelligence mobile SDKs and the Flutter app is sufficient.

Other basic functions, such as custom events, page loads, and identity, need to be converted to Flutter functions to be used by a Flutter app.

To define custom events, app developers work with the marketing team.

- Marketers define the custom events that are needed. Those custom events and their attributes are created in the SAS Customer Intelligence 360 user interface.
- Developers include the custom events and their associated attributes in the app. Then, the custom events can be leveraged by the Flutter app without any further code changes.

Note: The procedures in this section include more SASCollector public methods (functions) than just custom events, page loads, and identity. Some methods, such as getDeviceId, setDeviceId, can be used by developers for testing purposes. Others, such as startMonitoringLocation, disableLocationMonitoring, are used for location-based functionality.

Examples of how to use custom events, page loads, and identity in the code are included in "Configure the Example Flutter App" at the end of this section. The "Configure Flutter (Dart)", "Configure Android", and "Configure iOS" sections describe how to create the method channel to pass method calls from the native side to the Flutter (Dart) side.

Configure Flutter (Dart)

1. In the Flutter plug-in project, navigate to the `libs` folder.

   The folder contains three files: mobile_sdk_flutter_platform_interface.dart, mobile_sdk_flutter_method_channel.dart and mobile_sdk_flutter.dart. Each file contains boilerplate code.

2. In mobile_sdk_flutter_platform_interface.dart, add the methods from the SAS Customer Intelligence 360 mobile SDKs that you want to use in your Flutter app.

   For example, you might start by adding newPage, addAppEvent, identity, detachIdentity, startMonitoringLocation, and disableLocationMonitoring. Other public methods in SASCollector can be added later, such as getDeviceId and resetDeviceId, which are primarily used by developers for debugging purposes.

   Here is an example:

```
Future<void> newPage(String uri) {
   throw UnimplementedError('newPage has not been
   implemented.');
}
```

Additional implementation examples of the methods are provided in mobile_sdk_flutter.zip.

Detailed information about the methods is provided in the API reference documentation that is included in SASCollector_<applicationID>.zip. To obtain the documentation, see "Access API Reference Documentation".

3. In mobile_sdk_flutter_method_channel.dart, add the implementation of the methods that you defined in mobile_sdk_flutter_platform_interface.dart in step 2.

   Here is an example:

```
@override
  Future<void> newPage(String uri) async {
    return await methodChannel.invokeMethod('newPage', {'uri':
       uri});
  }
```

4. Create a file called constants.dart in the `lib` folder. Add the content from the SASCollector library. The public constants in the library are exported and exposed to the Flutter plug-in's app users. The following constants are needed at this point in the constants file if you want to add the identity function to the plug-in and use it in your app:

```
const String identityTypeEmail = "email_id";
const String identityTypeLogin = "login_id";
const String identityTypeCustomerId = "customer_id";
```

   Additional constants can be added later.

5. Create a file called sas_collector_sdk.dart in the `lib` folder. Add these exports to the file:

```
export 'mobile_sdk_flutter.dart';
export 'constants.dart';
```

Note: You will import the sas_collector_sdk.dart file when you are ready to use the SASCollector's features.

6. In mobile_sdk_flutter.dart, add the implementation of the methods that are defined in the mobile_sdk_flutter.zip.

Configure Android

1. In the Flutter plug-in project, navigate to the `android` folder. In the `android` folder, navigate to `src/main/java/com/sas/SASIA/mobile_sdk_flutter`, and find MobileSdkFlutterPlugin.java.
2. VSCode cannot automatically add imports, so you must manually add the following imports to MobileSdkFlutterPlugin.java:

```
import android.annotation.NonNull;
import android.content.Context;
import android.content.pm.PackageManager;
import android.app.Activity;
import android.os.Handler;
import android.os.Looper;
import io.flutter.embedding.engine.plugins.activity.
  ActivityAware;
import io.flutter.embedding.engine.plugins.activity.
  ActivityPluginBinding;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugin.common.BinaryMessenger;
import java.util.*;
import com.sas.mkt.mobile.sdk.SASCollector;
```

   If the build fails when running this code from the `example` folder, review the finished project to find the missing imports.

3. In the MobileSdkFlutterPlugin class definition, implement ActivityAware using this code:

```
public class MobileSdkFlutterPlugin implements FlutterPlugin,
MethodCallHandler, ActivityAware {
```

4. In the MobileSdkFlutterPlugin class, at the start of the class definition, add these variables:

```
private MethodChannel channel;
private Context context;
```

5. Update onAttachedToEngine, as shown below:

```
@Override
  public void onAttachedToEngine(@NonNull FlutterPluginBinding
   flutterPluginBinding) {
```

```
    channel = new
      MethodChannel(flutterPluginBinding.getBinaryMessenger(),
        "mobile_sdk_flutter");

    channel.setMethodCallHandler(this);

    this.context =
flutterPluginBinding.getApplicationContext();

  }
```

6. Update onMethodCall by adding native implementations of the exposed methods discussed in "Configure Flutter (Dart)".

   An implementation example is provided in mobile_sdk_flutter.zip. In the mobile_sdk_flutter project example, navigate to `mobile_sdk_flutter/android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/MobileSdkFlutterPlugin.java`.

7. Add the onDetachedFromEngine override method:

```
@Override
   public void onDetachedFromEngine(@NonNull
     FlutterPluginBinding binding) {

       channel.setMethodCallHandler(null);

       }
```

8. Because the MobileSdkFlutterPlugin class implements ActivityAware, override methods such as onDetachedFromActivity, onAttachedToActivity, onReattachedToActivityForConfigChanges, and onDetachedFromActivityForConfigChanges are required. Only onAttachedToActivity needs to be overridden as shown below:

```
@Override

public void onAttachedToActivity(

  @NonNullActivityPluginBinding binding) {

    SASCollector.getInstance().initialize(context);

}
```

9. The SAS Customer Intelligence 360 mobile SDK's Android initialization requires google services and gson dependencies:

   a. Navigate to `example/android`. Add this line in the dependencies section of the project level build.gradle:

   ```
   classpath 'com.google.gms:google-services:4.3.13'
   ```

   b. Navigate to `example/android/app/`. Add this line in the dependencies section of the app level build.gradle:

   ```
   implementation 'com.google.code.gson:gson:2.8.9'
   ```

Configure iOS

1. In the Flutter plug-in project, navigate to the `ios/Classes` folder. Find MobileSdkFlutterPlugin.m.
2. At the top of the MobileSdkFlutterPlugin.m file, add this import:

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

3. Add the Method channel to the interface so that it can be referenced later:

```
@interface MobileSdkFlutterPlugin ()
  @property(nonatomic, retain) FlutterMethodChannel *channel;
@end
```

4. Update registerWithRegistrar:

```
+
(void)registerWithRegistrar:(NSObject<FlutterPluginRegistrar>*
)registrar {
  FlutterMethodChannel* channel = [FlutterMethodChannel
     methodChannelWithName:@"mobile_sdk_flutter"
       binaryMessenger:[registrar messenger]];
  MobileSdkFlutterPlugin* instance = [[MobileSdkFlutterPlugin
     alloc] init];
  instance.channel = channel;
  [registrar addMethodCallDelegate:instance channel:channel];
}
```

5. Update handleMethodCall to implement the methods that are defined in "Configure Flutter (Dart)".

   An implementation example is provided in mobile_sdk_flutter.zip. In the mobile_sdk_flutter project example, navigate to `mobile_sdk_flutter/ios/Classes/MobileSdkFlutterPlugin.m`.

Configure the Example Flutter App

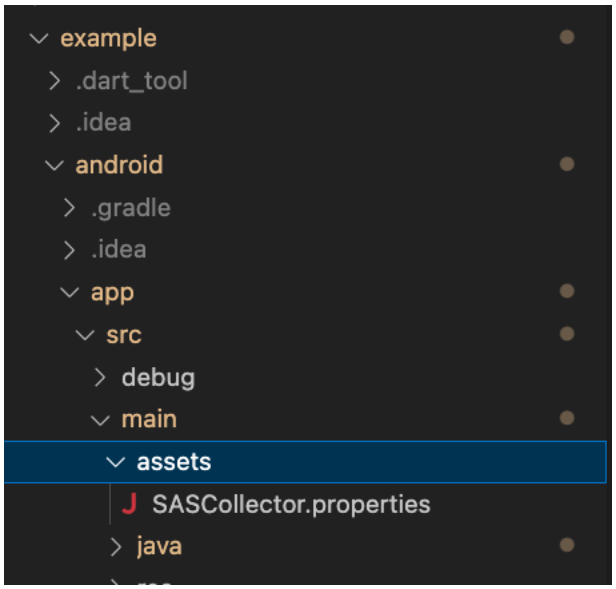
The `example` folder in your Flutter plug-in project includes these three folders: `android`, `ios`, and `lib`.

To configure and test identity, page load, and custom event functionality in the example Flutter app:

1. Add SASCollector.properties to Android:
   a. In VSCode, navigate to `android/app/src/main` and create an `assets` folder.

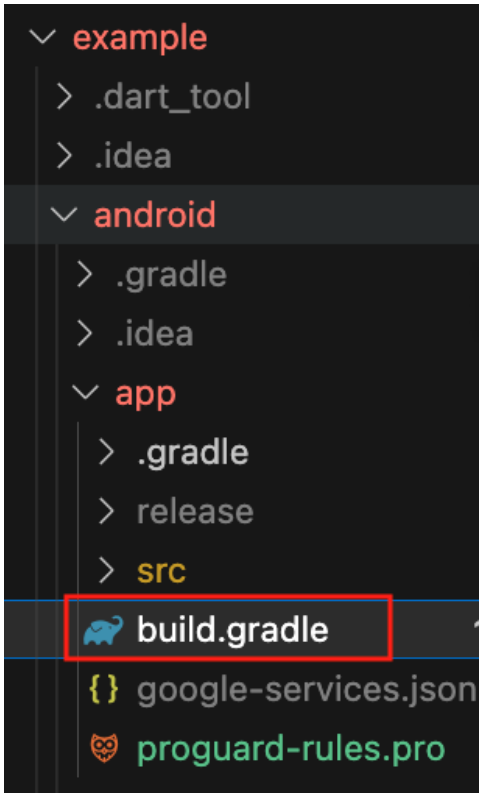
b. Find the SASCollector.properties file. The file is in the mobile SDK ZIP file for SAS Customer Intelligence 360 (SASCollector_<applicationID>.zip) in the `android` folder.

c. Copy SASCollector.properties into the `assets` folder.



2. If you will build the Android application's release APK and want to reduce the APK's size, then follow the following two steps:

a. Find build.gradle in example/android/app,



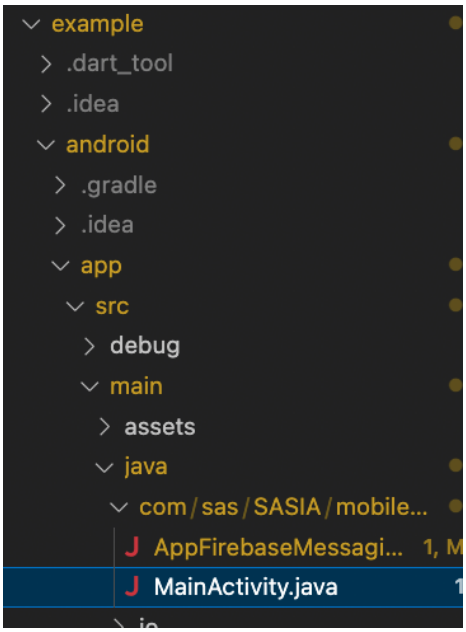
and add this code inside `release {}`:

```
minifyEnabled true
     proguardFiles getDefaultProguardFile(
     'proguard-android.txt'), 'proguard-rules.pro'
```

b. Create a file called proguard-rules.pro in example/android/app as shown in the above screenshot. Add this code inside proguard-rules.pro:

```
-keep class com.sas.mkt.mobile.sdk.** { *;}
```

3. Android only: The SAS Customer Intelligence 360 mobile SDK might not initialize in time to use functionality such as Identity. To avoid this issue:
   a. Navigate to `example/android/app/src/main/java/MainActivity.java` in the example project:



   b. Add this line in MainActivity.java:

```
SASCollector.getInstance().initialize(this);
```

4. Add SASCollector.plist to iOS:
   a. In Xcode, navigate to `ios` /`Runner`.
   b. Find the SASCollector.plist file. The file is in the mobile SDK ZIP file for SAS Customer Intelligence 360 (SASCollector_<applicationID>.zip) in the `ios` folder.
   c. Drag SASCollector.plist into the `Runner` folder.

5. Most of the code that a Flutter app developer writes is in the `lib` folder. Navigate to the `lib` folder. In main.dart, create a plug-in instance to make the plug-in available for all other pages, as shown in the example below.

```
final mobileSdkFlutterPlugin = MobileSdkFlutter();
```

The reference is passed to the pages that need to access the plug-in's functions.

6. To test the Identity API, in the `lib` folder create a login page Dart file (login_page.dart) like the one in the example project. Put the following code inside a login button's onPress function, as shown in the example below:

```
  ElevatedButton(
    style: ElevatedButton.styleFrom(
      fixedSize: const Size(300, 40),
    ),
    onPressed: () {
      widget.mobileSdkFlutter
        .identity(textFieldController.text, selectedType)
        .then((success) => {
          if (success){
            Navigator.of(context)
              .push(MaterialPageRoute(
                builder: (BuildContext context) {
                  return DetailsPage(
                    textFieldController.text,
                    widget.mobileSdkFlutter);
                }))
          } else {
            showDialog(context: context,
              builder: (_) =>
                const AlertDialog(
                  title: Text("Error"),
                  content: Text("Login failed."),
                ))
            }
          });
        },
        child: const Text("Log In"),
      ),
```

Note: widget.mobileSdkFlutter.identity is the Flutter plugin method that is created when you configured Dart. It communicates with SDK's native identity method.

7. To test page loads and custom events, in the `lib` folder create a home page dart file (home_page.dart) like the one in the example project. Events are created because of an activity such as tapping a button. Examples are shown below:

```
ElevatedButton(
  style: ElevatedButton.styleFrom(
    fixedSize: const Size(250, 40)),
    onPressed: () {
      if (pageUriController.text.isNotEmpty) {
        Widget.mobileSdkFlutter
          .newPage(pageUriController.text);
      }
    },
    child: const Text('Invoke New Page Event'),
),

ElevatedButton(
  style: ElevatedButton.styleFrom(
    fixedSize: const Size(250, 40)),
  onPressed: () {
    if(eventNameController.text.isEmpty ||
       attributeNameController.text.isEmpty ||
       attributeValueController.text.isEmpty) {
       return;
    }
     widget.mobileSdkFlutter.addAppEvent(
      eventNameController.text, {
      attributeNameController.text:
      attributeValueController.text
     });
  },
  child: const Text('Invoke App Event'),
)
```

Note: widget.mobileSdkFlutter.newPage and widget.mobileSdkFlutter.addAppEvent are the Flutter plugin methods that were created when you configured Dart. They communicate with SDK's native newPage and addAppEvent methods.

Send an Event

The system uses a unique mobile event key to identify the event type to send; you do not need to specify the event type in the code. All event types are sent the same way.

To send an event (such as tapping a button) to the mobile SDK, call this event:

```
mobileSdkFlutter.addAppEvent(eventId, attrs)
```

Use these parameters:

- a string identifier for the event. This string identifier should be the mobile event key that is specified in SAS Customer Intelligence 360.
- a map of name-to-value pairs of associated metadata to be sent with the hash map. mobileSdkFlutter.addAppEvent("*myEventId*", {*myAttributeName: myAttributeValue*})

  The map can be null if you do not want to send any metadata (attrs = null): `mobileSdkFlutter.addAppEvent("myEventId", null)`

To see more details about how the plug-in's exposed methods are used, you can find them in the completed project's `example` folder.

Mobile Spot Functionality

With SAS Customer Intelligence 360, you can include personalized content, such as advertising, in your mobile apps. In SAS Customer Intelligence 360, the location in the mobile app where the content is delivered is called a *spot*.

SAS Customer Intelligence 360 mobile SDKs provide two types of spots: inline spots and interstitial spots. Spots have delegate methods that are invoked at the different stages of the life cycle of the spots. For example, when the user closes an interstitial spot, the didClose method is called. Developers specify what action to take when a method is called.

As with custom events, app developers work with marketers to define where to include spots in the app and the content of those spots.

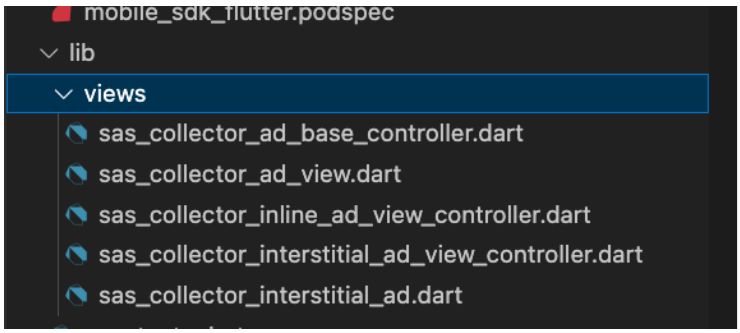
- The app developer includes the new mobile spots and the associated attributes in the app.
- Marketers register the mobile spots in the CI360 user interface so that they can be leveraged in campaigns without any further code changes.
- Marketing users design HTML creatives in SAS Customer Intelligence 360. Those creatives are delivered to the mobile spots via *tasks* that specify the mobile app, the spot, the target audience, and various other criteria.

Currently, the implementation of spots in the Flutter plug-in requires only the spotID parameter. If other parameters for spots are needed, developers can follow similar procedures to add them in the plug-in.

This section describes how to implement mobile spot features in the Flutter plug-in to be used in a Flutter app. The creation of the Flutter spots functions is described in three sections: "Configure Flutter (Dart)", "Configure Android", and "Configure iOS". The Dart functions are created as an interface that can be used by the Flutter widgets to get the spots. Most of the work that is involved in constructing and presenting spots is in Android and iOS.

Configure Flutter (Dart)

1. In the Flutter plug-in project, navigate to the `lib` folder and create a `views` folder.



2. In the `views` folder, create the following Dart files:
   - sas_collector_ad_base_controller.dart
   - sas_collector_ad_view.dart
   - sas_collector_inline_ad_view_controller.dart
   - sas_collector_interstitial_ad_view_controller.dart
   - sas_collector_interstitial_ad.dart

   Each of the views has delegate methods that correspond to the methods that are defined in AdDelegate (for Android) and SASIA_AdDelegate (for iOS) in the SAS Customer Intelligence 360 mobile SDKs. Therefore, they need controllers to perform actions (such as onLoaded and onClosed for Android and didLoad and didClose for iOS).

   sas_collector_ad_base_controller.dart is the base controller that the controllers of inline ad view and interstitial ad view inherit their values from. It defines all the delegate functions that an app can use. The app can also choose to use specific functionality. Please see the example project's view_page.dart file to see how these functions are used.

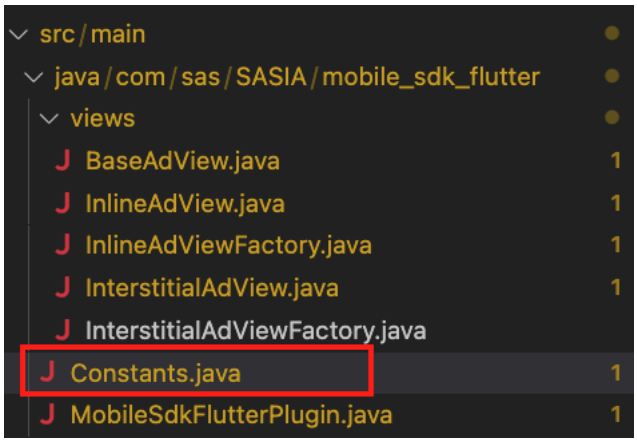
An implementation example of the ad views and their controllers is provided in mobile_sdk_flutter.zip. In the mobile_sdk_flutter project example, navigate to `mobile_sdk_flutter/lib/views`.

3. In the `lib` folder, update sas_collector_sdk.dart to include this code:

```
export 'views/sas_collector_ad_view.dart';
export 'views/sas_collector_interstitial_ad.dart';
export views/sas_collector_interstitial_ad_view_
  controller.dart';
export 'views/sas_collector_inline_ad_view_controller.dart';
```

Configure Android

1. In the Flutter plug-in project, navigate to the `Android` folder. In `/src/main/java/com/sas/SASIA/mobile_sdk_flutter`, create a Constants.java file.



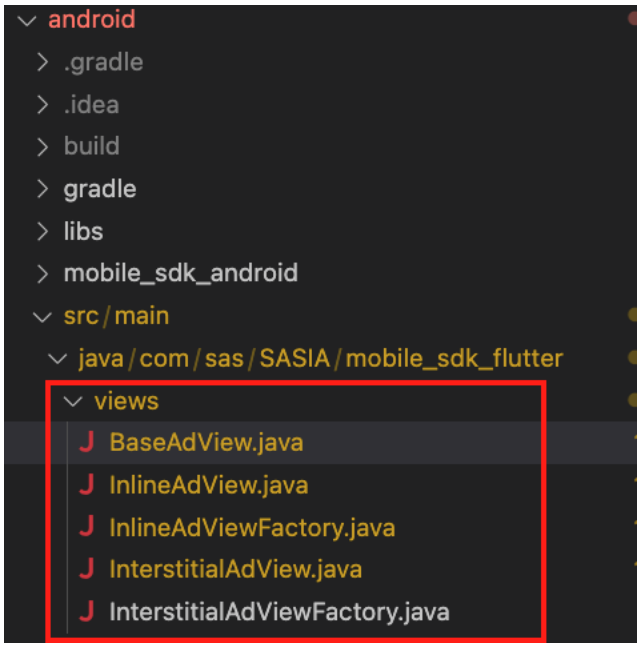
2. Add the following string constants.

TIP: The use of constants avoids typographical errors.

```
package com.sas.SASIA.mobile_sdk_flutter;

public class Constants {
    public static String Interstitial_Controller_Channel =
      "interstitial_controller_channel";
    public static String Inline_Ad_Controller_Channel =
      "inline_ad_controller_channel";
    public static String Spot_ID = "spotID";
    public static String Inline_Ad_View = "inlineAdView";
    public static String Interstitial_Ad_View =
      "interstitialAdView";
}
```

Additional string constants can be added later as needed.

3. In `src/main/java/com/sas/SASIA/mobile_sdk_flutter`, create a `views` folder.
4. In mobile_sdk_flutter.zip, navigate to `mobile_sdk_flutter/android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/views`.



5. Copy the following files and paste them in the `views` folder that you just created:
   - BaseAdView.java
   - InlineAdView.java
   - InlineAdViewFactory.java
   - InterstitialAdView.java
   - InterstitialAdViewFactory.java

   BaseAdView includes functionality that is common to both InterstitialAdView and InlineAdView. These two classes inherit features from BaseAdView and add their own features on top of it.

6. Navigate to `src/main/java/com/sas/SASIA/mobile_sdk_flutter`. In MobileSdkFlutterPlugin.java , update the onAttachedToEngine method with this code to register the views:

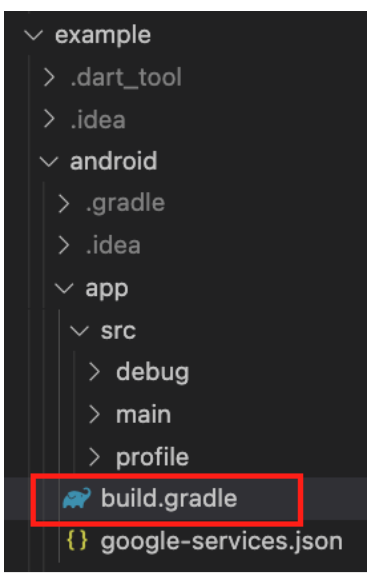
```
PlatformViewRegistry registry = flutterPluginBinding.
  getPlatformViewRegistry();
```

```
BinaryMessenger messenger = flutterPluginBinding.
  getBinaryMessenger();

registry.registerViewFactory(Constants.Inline_Ad_View, new
InlineAdViewFactory(messenger));
registry.registerViewFactory(Constants.Interstitial_Ad_View,
new InterstitialAdViewFactory(messenger));

channel = new
MethodChannel(flutterPluginBinding.getBinaryMessenger(),
"mobile_sdk_flutter");
channel.setMethodCallHandler(this);
```

This figure shows the update:

```
@Override
public void onAttachedToEngine(@NonNull FlutterPluginBinding flutterPluginBinding) {
  PlatformViewRegistry registry = flutterPluginBinding.getPlatformViewRegistry();
  BinaryMessenger messenger = flutterPluginBinding.getBinaryMessenger();
  registry.registerViewFactory(Constants.Inline_Ad_View, new InlineAdViewFactory(messenger));
  registry.registerViewFactory(Constants.Interstitial_Ad_View, new InterstitialAdViewFactory(messenger));

  channel = new MethodChannel(messenger, "mobile_sdk_flutter");
  channel.setMethodCallHandler(this);
  this.context = flutterPluginBinding.getApplicationContext();
```

7. In addition to the updates for the classes above, an update is needed in the example project. Navigate to `android/app/src/build.gradle`:



Add this dependency:

```
implementation files('../../../android/libs/SASCollector.jar')
```

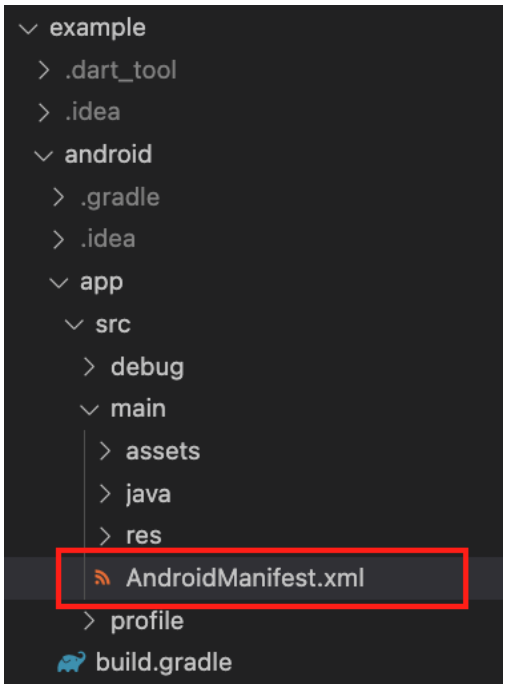
The figure below shows this change:

```
dependencies {
    implementation 'com.google.code.gson:gson:2.8.9'
    implementation 'com.google.android.gms:play-services-location:19.0.1'
    implementation platform('com.google.firebase:firebase-bom:30.3.1')
    implementation 'com.google.firebase:firebase-analytics'
    implementation 'com.google.firebase:firebase-core'
    implementation 'com.google.firebase:firebase-messaging'
    implementation files('../../../android/libs/SASCollector.jar')
}
```

A direct reference to the mobile SDK (as shown above) is needed for some native code-related operations, such as push notifications.

Note: This becomes clear when additional functionality is included.

8. Include the mobile SDK's implementation of the ad view activities in the example project's AndroidManifest.xml file so that the Android version of the Flutter app works. Navigate to `android/app/src/main` in the example project.

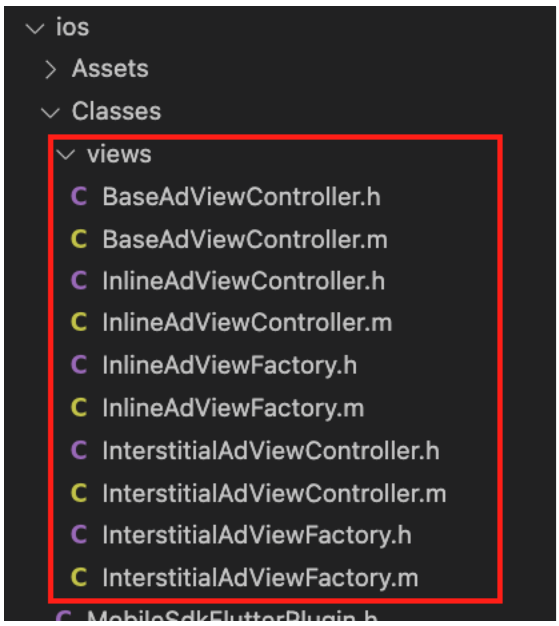


Add these lines to AndroidManifest.xml:

```
<activity
  android:name="com.sas.ia.android.sdk.
  InterstitialActivity" />
<activity
  android:name="com.sas.ia.android.sdk.
  InterstitialWebActivity"
/>
```

Configure iOS

1. Navigate to the `ios\Classes` folder and create a `views` folder with a few Objective-C files, such as the files shown in the figure below.



Note: BaseAdViewController is not used at this time. It was originally created for inheritance by InlineAdViewController and InterstitialAdView Controller.

An implementation example of the classes is provided in mobile_sdk_flutter.zip. In the mobile_sdk_flutter project example, navigate to `mobile_sdk_flutter/ios/classes/views`. Copy the content of the classes from the example files to the files that you just created.

2. Find MobileSdkFlutterPlugin.m in the `ios\Classes` folder, and add these imports after the other imports:

```
#import "./views/InlineAdViewFactory.h"
#import "./views/InterstitialAdViewFactory.h"
```

3. Also, in MobileSdkFlutterPlugin.m, register the inline and interstitial ad views factories in the registerWithRegistrar method:

```
InlineAdViewFactory* factory =
  [[InlineAdViewFactory alloc]
    initWithMessager:registrar.messenger];

[registrar registerViewFactory:factory
  withId:@"inlineAdView"];

InterstitialAdViewFactory *interstitialFactory =
  [[InterstitialAdViewFactory alloc]
    initWithMessager:registrar.messenger];

[registrar registerViewFactory:interstitialFactory
  withId:@"interstitialAdView"];
```

Configure the Example Flutter App

The size of inline and interstitial spot widgets on the Dart side depends on the size of the parent. Therefore, inline and interstitial spot widgets need to be wrapped in a widget (parent) such as SizedBox. The following is sample code for an inline spot widget:

```
SizedBox(
      height: 100,
      width: 300,
      child: SASCollectorInlineAdView(
      spotID: 'weather_spot_1',
      onCreated: onInlineAdCreated,
)
```

An interstitial spot widget does not render itself when it is placed on the screen. An interstitial spot widget needs a button to invoke its controller to display it. The following sample code provides that functionality:

```
Card(
    child: Padding(
       padding: const EdgeInsets.all(16.0),
       child: Column(
           children: [
              const Text('Interstitial Ad View'),
              ElevatedButton(
                 onPressed: () {
                      interstitialAdController.showAd();
                 },
                 child: const Text('Show Interstitial Ad')),
           ],
        ),
      ),
),
```

```
SizedBox(
    width: 3,
    height: 4,
    child: SASCollectorInterstitialAdView(
              spotID: 'interstitial_spot',
              onCreated: onInterstitialAdCreated),
     ),
```

The controllers for inline and interstitial ad views are defined at the start of the State class of the StatefulWidget. The controllers' handler methods are the equivalent of the handler methods for inline and interstitial ad views. You can modify them to suit your needs. See view_page.dart in the example app of the finished project for more details.

Location Functionality

Location features include precise location query (the ability to identify the local of a mobile device), geofence registration and detection, and beacon detection.

Developers collaborate with marketers on when to send push notifications. If the location of a mobile app is known, a triggered push notification can be sent when users enter or leave geolocations, or when a beacon is discovered. For example, when a user enters the geofence of a drugstore, the mobile app can send a push notification that entitles the user to a discount.

A SAS Customer Intelligence 360 user creates a triggered push notification task with the trigger set (on the **Orchestration** tab) to one of these mobile location options:

- Beacon Discovered
- Geofence Entered
- Geofence Exit

The SAS Customer Intelligence 360 user selects the trigger event's attribute condition, which is the action that triggers the event. For example, if the Geofence Entered trigger is an airport, the event's name might be Airport. Note that the CSV file that the developer delivered to the SAS Customer Intelligence 360 user to upload contains the event attributes to choose from.

To enable location features, these actions are required:

- **Add startMonitoringLocation and disableLocationMonitoring**. For geofences and beacons to work, these two functions are needed from the SDK.

Note: The startMonitoringLocation and disableLocationMonitoring functions were already added in MobileSdkFlutterPlugin.m and MobileSdkFlutterPlugin.java on the

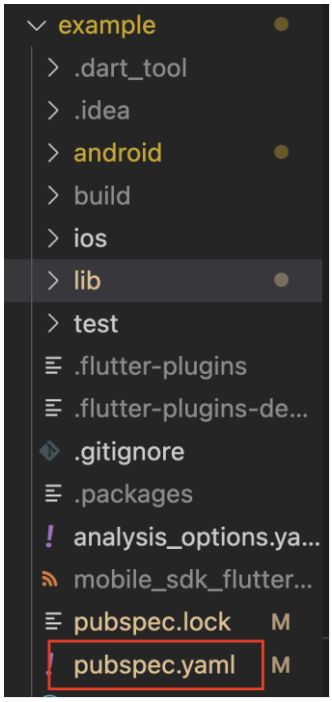
native side, and in mobile_sdk_flutter_platform_interface.dart, mobile_sdk_flutter_method_channel.dart and mobile_sdk_flutter.dart when you configured Dart in the "Basic Functionality" section of this guide.

- **Request location tracking permission.** A developer requests location tracking permission from the user through the mobile app. For information, for iOS, see Enable Location-Based Features and for Android, see Enable Location-Based Features in *SAS Customer Intelligence 360: Developer's Guide for Mobile Applications*.
- **Upload geofence and beacon data.** A developer provides geofence and beacon information in a CSV file to the SAS Customer Intelligence 360 user who uploads the file to the mobile application that was created in SAS Customer Intelligence 360. For information, see Upload Geofence and Beacon Data in *SAS Customer Intelligence 360: Administration Guide*.

The topics in this section cover how to configure startMonitoringLocation and disableLocationMonitoring in a Flutter app.

Configure Flutter (Dart)

1. In the example project, find pubspec.yaml:



2. Add the following dependencies:

```
permission_handler: ^9.2.0
```

```
location: ^4.4.0
fluttertoast: ^8.0.9
```

Note: In a yaml file, alignment is very important. Make sure these dependencies are indented properly.

Here are the definitions of those dependencies:

- `permission_handler`: Provides cross-platform APIs to request the mobile app user's permission to track their location. `Permission_handler` checks whether permission was granted or denied. Although `permission_handler` contains additional categories of permissions, only the location-related permission is used.
- `location`: Provides location-related functionality including location permission request and status check.

  Note: The reason for using the `location` dependency rather than `permission_handler` alone is due to a bug in `permission_handler`. `permission_handler` cannot correctly check the permission status on iOS. Although `location` dependency is not a perfect solution, `location` makes the user aware that they need to grant permission for the mobile app to track their location. When the bug is resolved, the `location` dependency is no longer needed.

- `fluttertoast`: Provides a toast message like the native Android's toast message. It behaves the same on both Android and iOS.

3. In `example/lib/main.dart`, add imports at the start of the file, as shown in the example code below:

```
import 'package:fluttertoast/fluttertoast.dart';
import 'package:permission_handler/permission_
       handler.dart';
import 'package:location/location.dart' as loc;
```

Then create this method:

```
void getLocationPermissionsAndStartGeofence() async {
  if (await Permission.locationAlways.isGranted) {
    mobileSdkFlutterPlugin.startMonitoringLocation();
    _geofenceStarted = true;
    return;
  }
  if (await Permission.locationAlways.isDenied ||
    await Permission.locationAlways.isPermanentlyDenied) {
      Fluttertoast.showToast(
        msg: 'For location-related features to work, '
```

```
              'please always allow "appname" to '
              ' access your location',
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.CENTER);
    await Future.delayed(const Duration(seconds: 2), () {});
    openAppSettings();
  }
}
```

4. After the user grants permission in app settings and returns to the mobile app, the permission change is not apparent in the app. For the app to detect the permission change, the state class needs to implement WidgetsBindingObserver mixin, set itself as the observer, and override didChangeAppLifecycleState as follows:

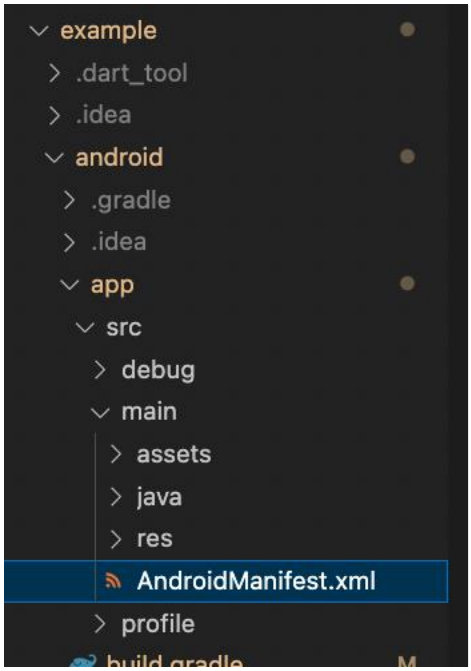
```
@override
  void didChangeAppLifecycleState(AppLifecycleState state)
async {
    await Future.delayed(const Duration(seconds: 1), () {});
    if (!_geofenceStarted) {
      getLocationPermissionsAndStartGeofence();
    }

    super.didChangeAppLifecycleState(state);
  }
```

5. iOS only: Due to the bug in permission_handler (described in step 2), create this function for iOS: getLocationPermissionsIOSAndStartGeofence. Refer to the example project for details.
6. In the initState method, add this code:

```
if (Platform.isAndroid) {
   getLocationPermissionsAndStartGeofence();
} else if (Platform.isIOS) {
   getLocationPermissionsIOSAndStartGeofence();
}
```

Configure Android

1. In the example project's `android` folder, navigate to `app/src/main` and find the AndroidManifest.xml file.



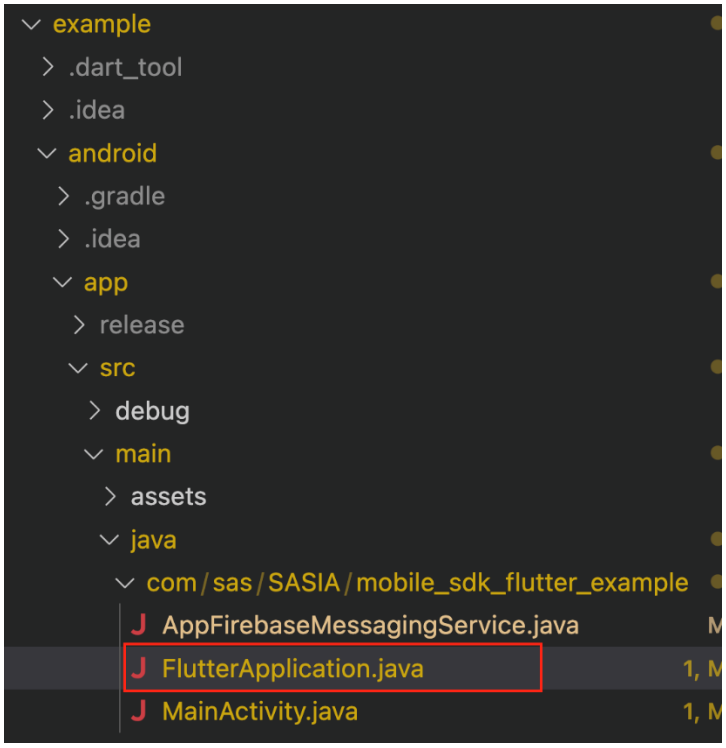
2. Add permissions for locations:

```
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission
android:name="android.permission.ACCESS_BACKGROUND_LOCATION"
/>

<uses-permission
android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH"
/>
<uses-permission
android:name="android.permission.BLUETOOTH_ADMIN" />
```

3. In `<application></application>`, add this code:

```
<service android:name=
 "com.sas.mkt.mobile.sdk.SASCollectorIntentService">
</service>
<receiver android:name=
 "com.sas.mkt.mobile.sdk.SASCollectorBroadcastReceiver"
 android:exported = "true">
  <intent-filter>
    <action
      android:name="android.intent.action.BOOT COMPLETED" />
  </intent-filter>
</receiver>
```

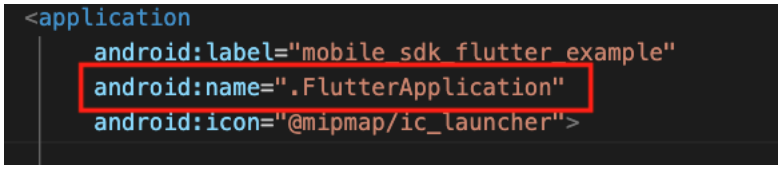
4. To enable detailed logging in the mobile SDK, create FlutterApplication.java in `app/src/main/java/com/sas/SASIA/mobile_sdk_flutter_example`:



5. Add this code to the FlutterApplication.java file that you created:

```
package com.sas.SASIA.mobile_sdk_flutter_example;
import android.app.Application;
import com.sas.mkt.mobile.sdk.util.SLog;

public class FlutterApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        SLog.setLevel(SLog.ALL);
    }
}
```

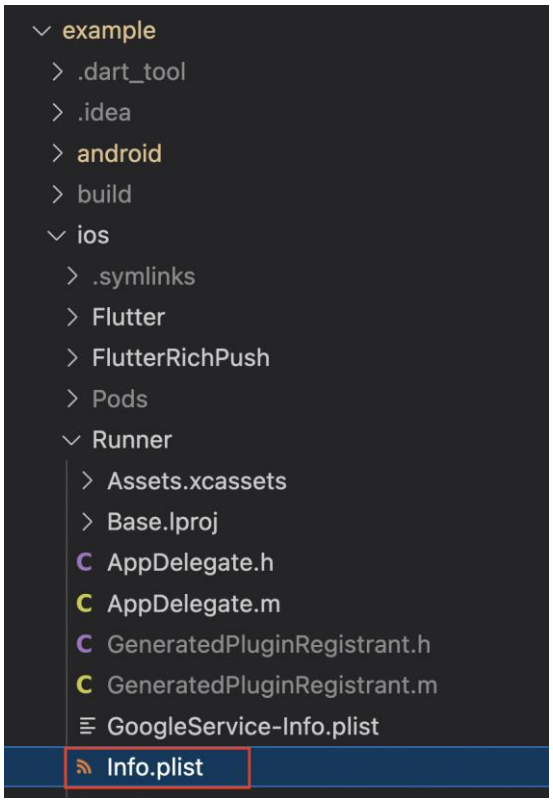
6. In the AndroidManifest.xml file find `android:name=" <name of application>"` and change it to `android:name=".FlutterApplication"`:



Note: Flutter creates two AndroidManifest.xml files, one inside `android/app/src/main`, and another inside `android/app/src/debug`. The content in both files must match to avoid merge error.

Configure iOS

1. In the example project's `ios/Runner` folder, find Info.plist:



2. In Info.plist, add these location request permissions:

```
<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>We need to access your location for geofence</string>
<key>NSLocationAlwaysUsageDescription</key>
<string>We need to access your location for geofence</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>We need to access your location for geofence</string>
```

3. To enable detailed logging in the mobile SDK, find AppDelegate.m in `example/ios/Runner`, and add the import at the beginning of the file, after all other imports:

```
#import <SASCollector/SASLogger.h>
```

4. Still in AppDelegate.m, in application:didFinishLaunchingWithOptions:, add this line after the registerWithRegistry method call:

```
[SASLogger setLevel:SASLoggerLevelAll];
```

TIP SDK logging in the integrated terminal is not visible when you test on an iOS simulator. Use a real device to see the event traffic.

Test Geofencing and Beacon Functionality

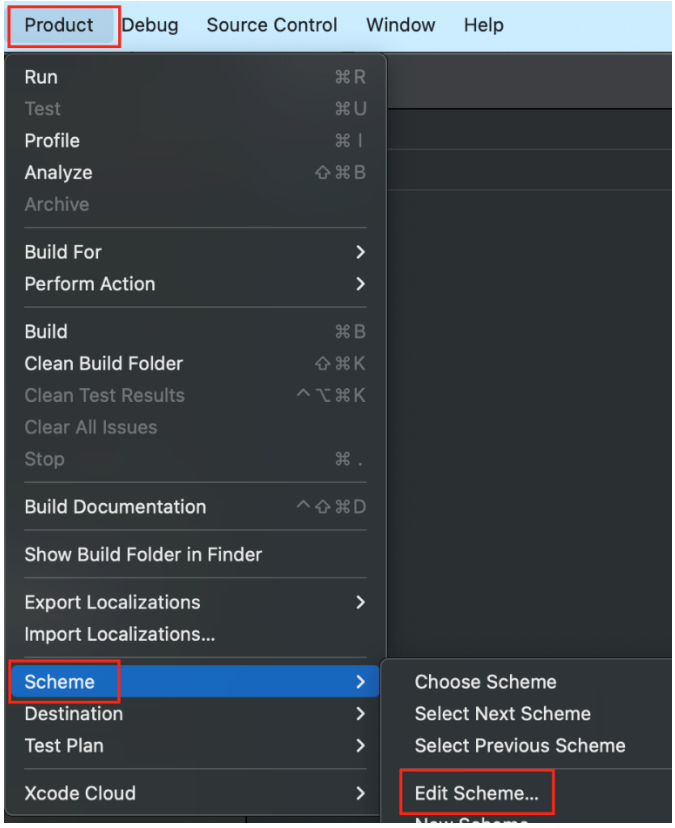
Create a geofence CSV file with the mobile app ID, longitude, latitude, radius, and so on. Give the file to the SAS Customer Intelligence 360 user to upload in SAS Customer Intelligence 360 where the mobile application is created. For information, see Upload Geofence and Beacon Data in *SAS Customer Intelligence 360: Administration Guide*.

Android

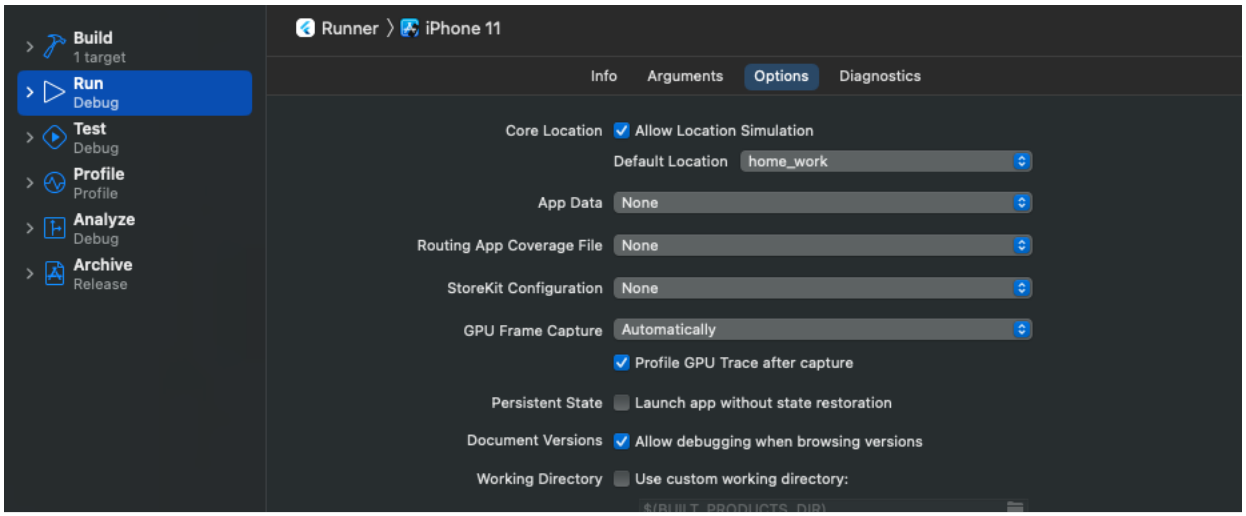
1. In the Android simulator, create a few location points. Make sure some, but not all, locations are also in the CSV file.
2. Start the example app and find a location in the simulator that is in the CSV file and set the location. The logs from Slog should have an enter_geofence event.
3. To test leaving a geofence, choose a location that is not in the CSV file, and set the location. The result is that an exit_geofence event is logged. Beacon events are also included in the logs.

iOS

1. Create a GPX file in the example (Runner) project. In the file, make sure some of the wpts (waypoints) have the same lat (latitude) and lon (longitude) values that are defined in the CSV file, and others do not.
2. In Xcode, go to **Product** => **Scheme** => **Edit Scheme**.



The following window is displayed:



Make sure the GPX file is in the **Default Location** field. In the figure, the file name is home_work. Also, select **Allow Location Simulation**.

Note:

- Currently, the example project cannot run on a real device from Xcode. However, you can run the project on a simulator.

- Once the app is run, if a map is open, it moves from one location to another based on the setup in the GPX file.
- Logs of geofence information can be found in the output pane at the bottom of the Xcode window.
- Beacons might not work on simulator.

Mobile Message Functionality

Mobile message features include token registration, in-app messages, push notifications, rich push notifications for iOS, and the delegate methods.

SAS Customer Intelligence 360 enables you to capture real-time impression data and connect other SAS Customer Intelligence 360 features with mobile messages.

Push notifications can display timely offers that invite a mobile app user back into the mobile app or into a store. For example, a mobile app user might drive to a store for which a geofence is defined in the mobile app. When the user (more specifically, the user's mobile device) enters that geofence, that action can trigger the mobile app to send a push notification that informs the user of a sale in the store.

In-app messages can display pop-up ads in the app. For example, the user might tap a button that triggers the in-app message event. The in-app message displays ads that might contain a link for the user to go to the website to learn more, or a button that takes the user to another page of the app to get more information. As the message is triggered by a SAS Customer Intelligence 360 custom event, this cannot be achieved using third-party plug-ins.

When the user clicks one of the buttons in an in-app message or opens a push notification, often the next action is to navigate to a particular section of your app. Design your delegate to be as flexible as possible so that it can perform navigation based on the link provided by the creative. This flexibility enables the SAS Customer Intelligence 360 user to achieve the desired calls to action more easily.

Like the configuration of location functionality, mobile messages require more native setup than Dart setup.

Note: Third-party push notification plug-ins (such as FlutterFire) are available for Flutter apps, but they do not provide the full functionality that SAS Customer Intelligence 360 mobile messaging delivers.

Configure Flutter (Dart)

For the methods of the SASMobileMessagingDelegate2 delegate to work, the Flutter method channel is used. When the delegate methods are called on the native side, messages are sent through the channel, and the Dart side responds to the messages passed from the channel.

In the example project's main.dart file, add this method:

```
void setupConnectionWithNative() {
  channel.setMethodCallHandler((call) async {
    switch (call.method) {
      case 'msgDismissed':
          Fluttertoast.showToast(
            msg: "User dismissed the message"
          );
          break;
      case 'actionLinkClicked':
          print('actionLinkClicked called');
          Map args = call.arguments;
          String link = args['link'];
          String type = args['type'];
          Fluttertoast.showToast(msg:
            "User clicked the push notification link: $link");
          if (link.contains('diagnostics')) {
            if (type == 'InAppMsg') {
              // diagnostics page has index of 3
              _tabController.animateTo(3);
            } else if (type == 'PushNotification') {
              if (Platform.isIOS) {
                _tabController.animateTo(3);
              } else if (Platform.isAndroid) {
                _pushNotificationReceived = true;
              }
            }
          }
          break;
      default:
          break;
      }
    });
  }
```

The example above shows how to handle delegate methods calls sent through the channel on the Dart side. You can use a similar method to achieve your goals.

Note: If you use CI360 Android SDK 1.80.2 or 1.80.3, replace this in the above code:

```
      if (Platform.isIOS) {
         _tabController.animateTo(3);
```

```
    } else if (Platform.isAndroid) {
      _pushNotificationReceived = true;
    }
```

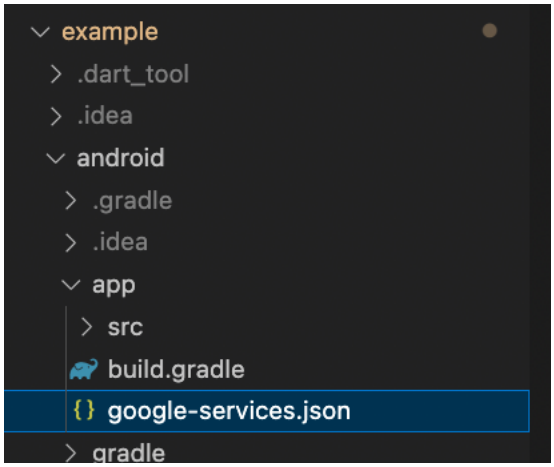
with this:

```
    _pushNotificationReceived = true;
    _tabController.animateTo(3);
```

CI360 Android SDK version 1.80.2 fixed an application relaunch issue, and thus makes Android and iOS behave the same when a push notification is clicked to open the application. There will be a corresponding update in the next section (Configure Android).

Configure Android

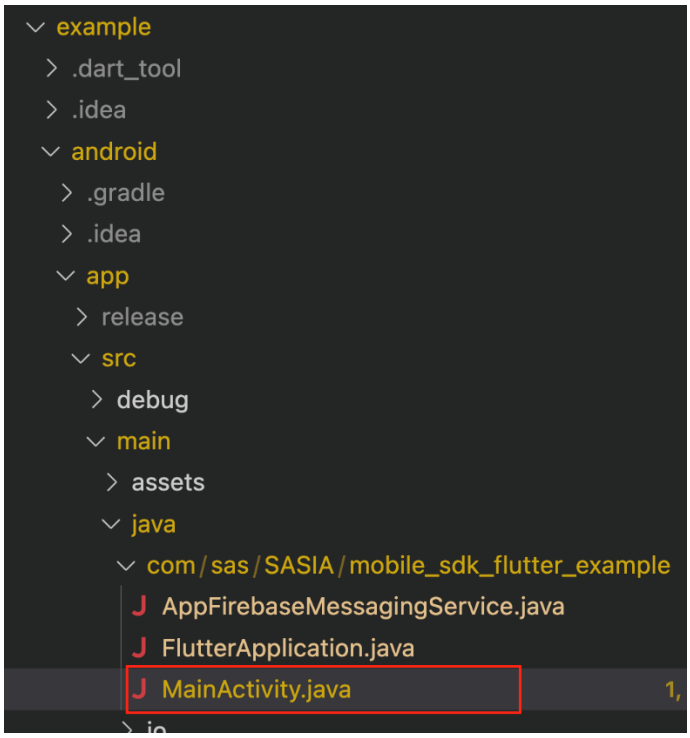
1. In the Firebase console, create a project and add the Flutter app's Android package ID to the project.
2. Get the google-services.json file and put it in the example project's `android/app` folder:



3. From the project in the Firebase console, get the server key and give it to the SAS Customer Intelligence 360 user. The user will add it to the SAS Customer Intelligence 360 mobile application that is created for the example project.

   For information, see Mobile Application Configuration in *SAS Customer Intelligence 360: Administration Guide*.

4. Under `android/app/src/main/java/com/sas/SASIA/mobile_sdk_flutter_example`, find MainActivity.java:



5. To MainActivity.java, do the following:

   a. Add the setPushChannel method:

   Note: Android version Oreo and above requires a push notification channel. By creating it in the application class, you can avoid having to re-create the channel. Slog can also be set in the application, so it is not needed in MainActivity.java.

```
@RequiresApi(api = Build.VERSION_CODES.O)
private void setPushChannel() {
  NotificationManager notificationManager =
    (NotificationManager)
    this.getSystemService(NOTIFICATION_SERVICE);
  String customAndroidChannel = "FlutterPushChannel";
  CharSequence channelName = "Flutter Channel";
  int importance = NotificationManager.IMPORTANCE_HIGH;
  NotificationChannel notificationChannel =
     new NotificationChannel(
       customAndroidChannel, channelName, importance);
  notificationChannel.enableLights(true);
  notificationChannel.setLightColor(Color.RED);
  notificationChannel.enableVibration(true);
```

```
    notificationChannel.setShowBadge(true);
    notificationChannel.setVibrationPattern(
      new long[]{100, 200, 300, 400, 500,
        400, 300, 200, 400}
    );
    notificationManager.createNotificationChannel(
      notificationChannel);
    SASCollector.getInstance()
      .setPushNotificationChannelId(customAndroidChannel);
  }
```

b. Also, add the setPushChannel method call to the onCreate method as follows:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
  setPushChannel();
}
```

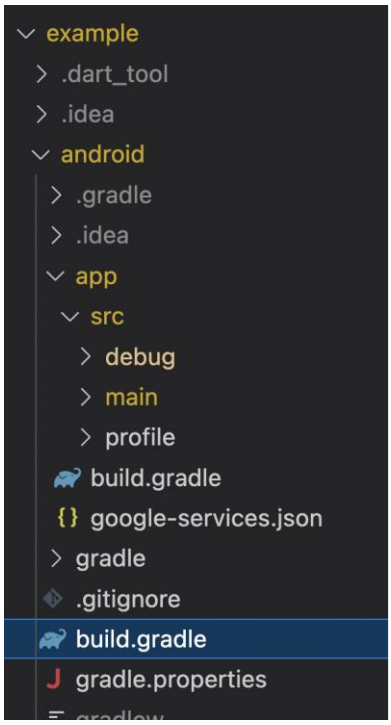
6. In the same folder where MainActivity resides, add AppFirebaseMessagingService.java with two override methods.

```
public class AppFirebaseMessagingService extends
FirebaseMessagingService {

  @Override
  public void onMessageReceived(RemoteMessage remoteMessage)
  {
    if (!SASCollector.getInstance().handleMobileMessage(
      remoteMessage.getData())) {
       //Handle non-SASCollector message
    }

  }

  @Override
  public void onNewToken(String token) {
    super.onNewToken(token);
    SLog.e("NEW_TOKEN",token);

    if(token != null) {
      SASCollector.getInstance()
        .registerForMobileMessages(token);
    }
  }
}
```

An implementation example of imports of this class is provided in mobile_sdk_flutter.zip. In the mobile_sdk_flutter project example, navigate to `mobile_sdk_flutter/`

`example/android/app/src/main/java/com/sas/SASIA/mobile_sdk_flutter_example`.

7. Find the build.gradle file in the example project's `android` folder:



8. In the project level build.gradle file, add this line in dependencies section:

```
classpath 'com.google.gms:google-services:4.3.13'
```

9. Find the app level build.gradle file in the `example/android/app` folder, and add these lines to the dependencies section:

```
implementation 'com.google.firebase:firebase-core'
implementation 'com.google.firebase:firebase-messaging'
```

In the plugin section, add:

```
apply plugin: 'com.google.gms:google-services'
```

10. Find MainActivity.java in `example/android/app/src/main/java/com/sas/SASIA/mobile_sdk__flutter_example`, and replace its content with this code:

```
public class MainActivity extends FlutterActivity {
  MethodChannel channel;
  String notificationLink;

  @Override
```

```
public void configureFlutterEngine(
  @NonNull FlutterEngine flutterEngine) {
  super.configureFlutterEngine(flutterEngine);
  channel = new MethodChannel(
    flutterEngine.getDartExecutor()
      .getBinaryMessenger(),
    "app_channel");
}

@Override
protected void onCreate(
  @Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  SASCollector.getInstance()
    .initialize(getApplicationContext());
  FirebaseMessaging.getInstance()
    .getToken().addOnSuccessListener(token -> {
      if(!TextUtils.isEmpty(token)) {
        SASCollector.getInstance()
          .registerForMobileMessages(token);
      }
    });
  Intent intent = getIntent();
  notificationLink =
    intent.getStringExtra("notificationWithLink");

  SASCollector.getInstance()
    .setMobileMessagingDelegate2(
      new SASMobileMessagingDelegate2() {
        @Override
        public void dismissed() {
          channel.invokeMethod("msgDismissed", null);
        }

        @Override
        public void action(String s,
          SASMobileMessageType sasMobileMessageType) {
          if(sasMobileMessageType.equals(
            SASMobileMessageType.IN_APP_MESSAGE)) {
            Map<String, String> args = new HashMap<>();
            args.put("link", s);
            args.put("type", "InAppMsg");
            channel.invokeMethod(
              "actionLinkClicked", args);
          }
        }

        @Override
        public Intent getNotificationIntent(String s) {
          SLog.i("getNotificationIntent", s);
          Intent intent = new Intent(
            MainActivity.this, MainActivity.class);
```

```
            intent.putExtra("notificationWithLink", s);
            intent.addFlags(
              Intent.FLAG_ACTIVITY_SINGLE_TOP);
            return intent;
          }
        });
    }

    @Override
    public void onPostResume() {
      super.onPostResume();
      if(notificationLink != null) {
        Map<String, String> args = new HashMap<>();
        args.put("link", notificationLink);
        args.put("type", "PushNotification");
        channel.invokeMethod("actionLinkClicked", args);
        notificationLink = null;
      }
    }
  }
```

An implementation example of the import is provided in mobile_sdk_flutter.zip. In the mobile_sdk_flutter project example, navigate to `mobile_sdk_flutter/example/android/app/src/main/java/com/sas/SASIA/mobile_sdk_flutter_example`.

11. If you use CI360 Android SDK 1.80.2, add this code in MainActivity.java:

```
@Override
protected void onNewIntent(@NonNull Intent intent) {
  super.onNewIntent(intent);
  notificationLink =
     intent.getStringExtra("notificationWithLink");
  if (notificationLink != null) {
     Map<String, String> args = new HashMap<>();
     args.put("link", notificationLink);
     args.put("type", "PushNotification");
     channel.invokeMethod("actionLinkClicked", args);
     notificationLink = null;
  }
}
```

12. If you use CI360 Android SDK 1.80.3, skip step 11, and add this code in MainActivity.java:

```
@Override
protected void onNewIntent(@NonNull Intent intent) {
  super.onNewIntent(intent);

  Bundle bundle = intent.getExtras();
```

```
    this.getIntent().putExtras(bundle);

    notificationLink =
       intent.getStringExtra("notificationWithLink");
    if (notificationLink != null) {
       Map<String, String> args = new HashMap<>();
       args.put("link", notificationLink);
       args.put("type", "PushNotification");
       channel.invokeMethod("actionLinkClicked", args);
       notificationLink = null;
    }
  }
```

13. If you use CI360 Android SDK 1.80.3, you also need to add this entry in your SASCollector.properties:

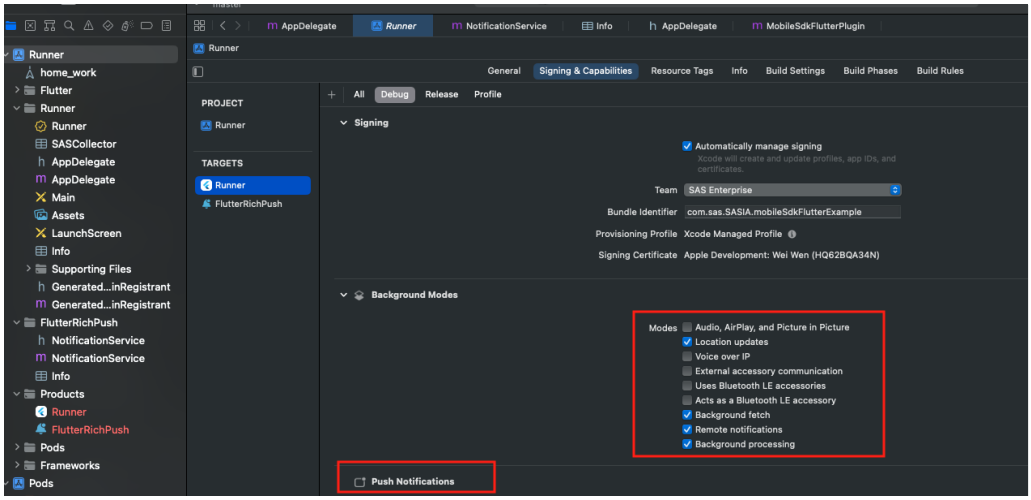
    ```
    apprelaunch.disabled.on.notification.open=true
    ```

14. In AndroidManifest.xml, add the Firebase Messaging service:

```
<service
  android:name=".AppFirebaseMessagingService"
  android:exported="false">
    <intent-filter>
      <action android:name=
        "com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>
```

Configure iOS

1. Go to developer.apple.com, enable push notifications for the app, and create a PEM file.
2. Copy the key and certificate and put them in SAS Customer Intelligence 360, where the mobile application is created. For information, see Mobile Application Configuration in *SAS Customer Intelligence 360: Administration Guide*.
3. Open Runner.xcworkspace from Xcode. Add push notifications and the checked capabilities in background modes:



4. In AppDelegate.h, replace the content with this code:

```
#import <Flutter/Flutter.h>
#import <UIKit/UIKit.h>
#import <UserNotifications/UserNotifications.h>
#import <SASCollector/SASCollector.h>

@interface AppDelegate :
FlutterAppDelegate<UIApplicationDelegate,
UNUserNotificationCenterDelegate, SASMobileMessagingDelegate2>
@end
```

5. In AppDelegate.m:

   a. Override two UIApplication methods didRegisterForRemoteNotificationsWithDeviceToken and didReceiveRemoteNotification. You can customize the completionHandler with your code:

```
-(void)application:(UIApplication *)application
  didRegisterForRemoteNotificationsWithDeviceToken:
  (NSData *)deviceToken {
    [SASCollector registerForMobileMessages:
      deviceToken completionHandler:^{
        [SASLogger info:@"Registering for remote\
          notifications is successful"];
   } failureHandler:^{
      [SASLogger info:@"Registering for remote\
        notifications failed"];
   }];
  }

-(void)application:(UIApplication *)application
  didReceiveRemoteNotification:(NSDictionary *)userInfo
  fetchCompletionHandler:
```

```
    (void (^)(UIBackgroundFetchResult))completionHandler{
       NSString* userInfoStr =
         [NSString stringWithFormat:
           @"didReceiveRemoteNotification,
           userInfo: %@", userInfo.description];
       [SASLogger info:userInfoStr];

       if (![SASCollector handleMobileMessage:userInfo
         WithApplication:application]) {
         //Handle non-SASCollector message
         NSLog(@"Remote Notification was not handled by
SASCollector");
  }

       completionHandler(UIBackgroundFetchResultNoData);
  }

-(void)userNotificationCenter:(UNUserNotificationCenter
*)center
didReceiveNotificationResponse:(UNNotificationResponse
*)response withCompletionHandler:(void
(^)(void))completionHandler {

NSLog(@"Remote Notification was handled by SASCollector");

  if (![SASCollector handleMobileMessage:response.
notification.request.content.userInfo
WithApplication:UIApplication.sharedApplication]) {
    //Handle non-SASCollector message
    NSLog(@"Remote Notification was not handled by
SASCollector");
}

completionHandler();
}
```

b. Create the method channel and set SASMobileMessagingDelegate2's delegate in the didFinishLaunchingWithOptions method:

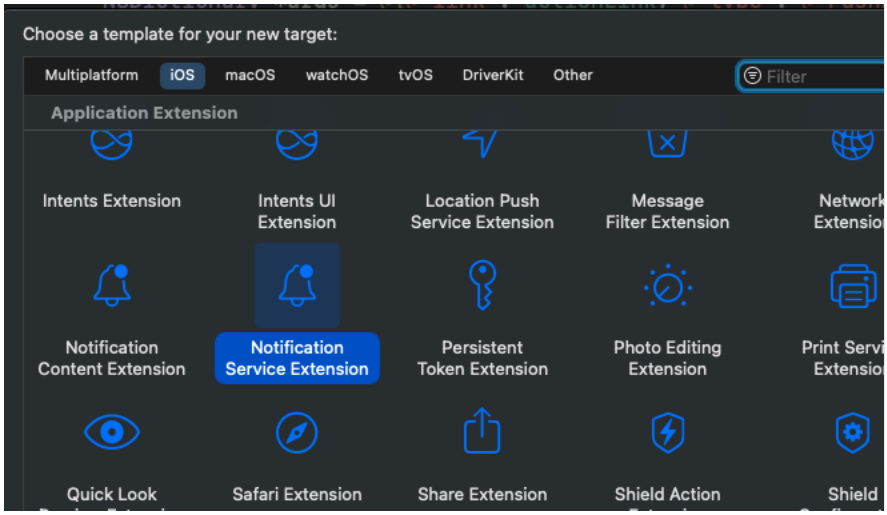
```
FlutterViewController* controller =
(FlutterViewController*)self.window.rootViewController;
    methodChannel = [FlutterMethodChannel
methodChannelWithName:@"app_channel"
binaryMessenger:[controller binaryMessenger]];

[SASCollector setMobileMessagingDelegate2:self];
```

c. Implement the delegate methods and methods that call methods on the Dart side:

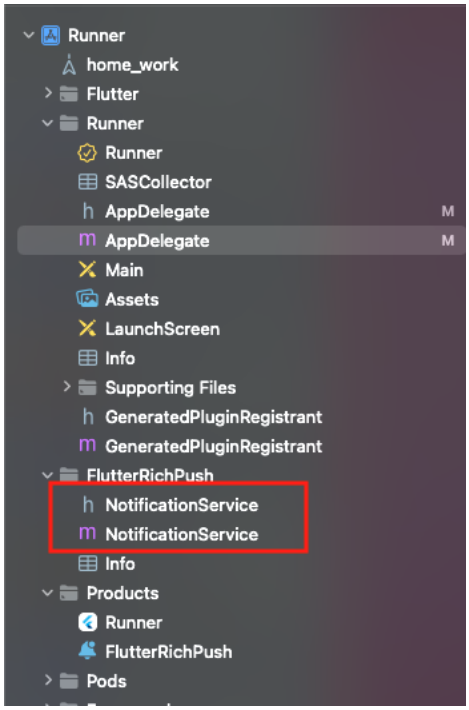
```
//SASMobileMessagingDelegate2 delegate methods
-(void)messageDismissed {
    [self handleUserDismissMsg];
}

-(void)actionWithLink:(NSString *)link
type:(SASMobileMessageType)type {
  [SASLogger info: @"actionWithLink called"];
  if(type == SASMobileMessageTypeInAppMessage) {
    NSDictionary *args =
      @{@"link": link, @"type": @"InAppMsg"};
    [methodChannel invokeMethod:
      @"actionLinkClicked" arguments:args];
  } else if (
    type == SASMobileMessageTypePushNotification){
    actionLink = link;
  }
}

// methods that calls methods on the dart side
-(void)handleUserDismissMsg {
  [methodChannel invokeMethod:@"msgDismissed"
                 arguments:nil];
}

-(void)applicationDidBecomeActive:(UIApplication
*)application {
  if(actionLink != nil) {
    NSString *msg =
      [NSString stringWithFormat:
        @"applicationDidBecomeActive,
        link: %@", actionLink];

    [SASLogger info:msg];
    NSDictionary *args = @{@"link": actionLink,
      @"type": @"PushNotification"};
    [methodChannel invokeMethod:@"actionLinkClicked"
                   arguments:args];

     actionLink = nil;
  }
}
```

For an example, in mobile_sdk_flutter.zip, navigate to `mobile_sdk_flutter/example/ios/Runner`.

6. To enable rich push notifications, create a new Notification Service Extension target from Xcode:



When the target is created, two new files are added:



7. Replace the didReceiveNotificationRequest method in NotificationService.m with this code:

```
-(void)didReceiveNotificationRequest:
  (UNNotificationRequest *)request
  withContentHandler:
  (void (^)(UNNotificationContent * _Nonnull))contentHandler
```

```
{
  self.contentHandler = contentHandler;
  self.bestAttemptContent = [request.content mutableCopy];

  NSDictionary *notificationData =
    (NSDictionary*)request.content.userInfo[@"data"];
  if(notificationData == nil) {
    return;
  }

  NSString *urlStr =
    (NSString*)[notificationData objectForKey:
      @"attachment-url"];

  if(urlStr == nil) {
    return;
  }

  NSURL *fileUrl = [NSURL URLWithString:urlStr];
  if(fileUrl == nil) {
      return;
  }

  NSURLSessionDownloadTask *downloadTask =
    [NSURLSession.sharedSession
      downloadTaskWithURL:fileUrl
      completionHandler:^(NSURL * _Nullable location,
      NSURLResponse * _Nullable response,
      NSError * _Nullable error) {

        if(location != nil && error == nil) {
          NSString *tempDir = NSTemporaryDirectory();
          NSString *suggestedName = [response
            suggestedFilename];
          if(suggestedName != nil) {
            NSString *fileName = [NSString
              stringWithFormat:@"file://%@%@",
                tempDir, suggestedName];

            NSString *tempFileName = [fileName
              stringByReplacingOccurrencesOfString:@" "
              withString:@"_"];

            NSURL *tempUrl = [NSURL
              URLWithString:tempFileName];

            NSError *removeFileError;

            if([NSFileManager.defaultManager
              fileExistsAtPath:tempUrl.path] &&
              [NSFileManager.defaultManager
                isDeletableFileAtPath:tempUrl.path]) {
```

```
                [NSFileManager.defaultManager
                  removeItemAtPath:tempUrl.path
                  error:&removeFileError];
              }

              if(removeFileError != nil) return;

              NSError *moveFileError;
              [NSFileManager.defaultManager
                moveItemAtURL:location toURL:tempUrl
                error:&moveFileError];

              if (moveFileError != nil) return;


              NSError *attachmentError;
              UNNotificationAttachment *attachment =
                [UNNotificationAttachment
                  attachmentWithIdentifier:@"ci360content"
                  URL:tempUrl options:nil
                  error:&attachmentError];

              self.bestAttemptContent.attachments =
                @[attachment];

              if (attachmentError != nil) return;
            }
        }
        self.contentHandler(self.bestAttemptContent);
    }];

    [downloadTask resume];
}
```

Test Push Notifications and In-App Messages

A SAS Customer Intelligence 360 user creates events, creatives, and tasks for push notifications and in-app messages.

Test Push Notifications

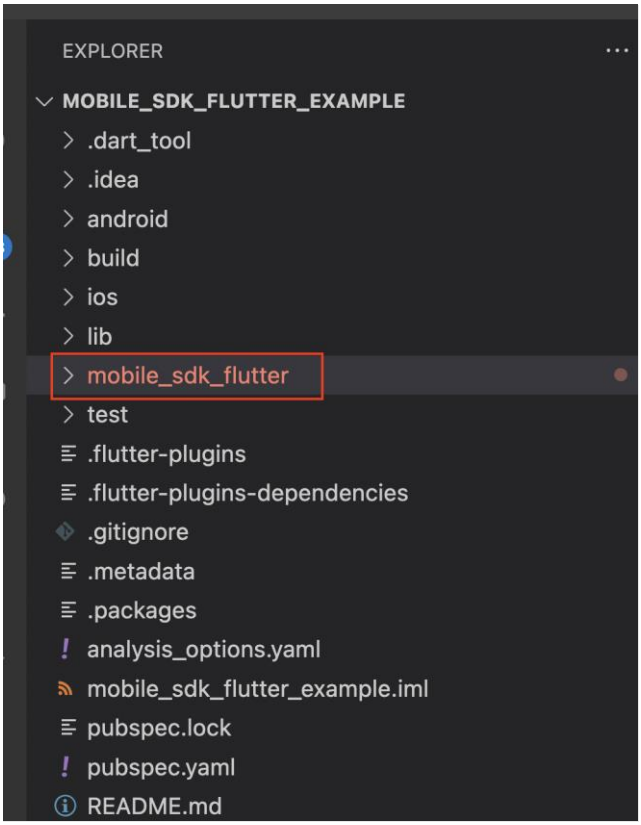
1. Start the app, log in, and put the app in the background.
2. In SAS Customer Intelligence 360, navigate to **General Settings**. Under **Content Delivery**, select **Diagnostics**. For **ID type**, select your device ID and click **Submit Test**. You should receive a test push notification on your device.

Test In-App Messages

To test an in-app message, a mobile in-app message task must be created in SAS Customer Intelligence 360. The task requires a trigger event. Call addAppEvent using the event ID for the task's trigger event. The in-app message should be displayed in your mobile app.

Integrate the Flutter Plug-in with the Existing Flutter App

To enable an existing Flutter app to use the plug-in, copy the plug-in folder (the zipped example project minus the example folder) in the existing project. The figure below shows an example where MOBLE_SDK_FLUTTER_EXAMPLE is the Flutter app.



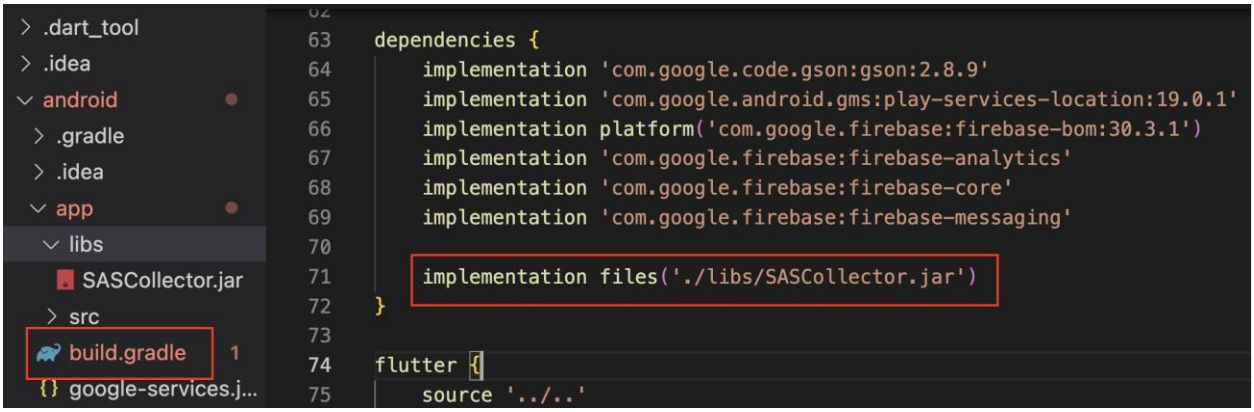
In the Flutter app's pubspec.yaml file, add the plug-in:

```
! pubspec.yaml
17  # versions available, run `flutter pub outdate
18  dependencies:
19    flutter:
20      sdk: flutter
21
22    mobile_sdk_flutter:
23      path: ./mobile_sdk_flutter
24
```

Configure Android

As discussed in the prior sections, in the Flutter plug-in, both Android and iOS need direct access to the SDKs for some features (for example, mobile messages). In the example project, it can be directly accessed by specifying the location of the library in the plug-in. However, this is not possible when the Flutter app hosts the plug-in as a library. In this case, create a `libs` folder in `android/app`, and add SASCollector.jar in it, as shown in the following figure:

In order for the native code (for example, MainActivity.java) to know where the library is, its location is added in the app level build.gradle as shown below:



Configure iOS

In iOS, there is nothing specific to configure. As long as mobile_sdk_flutter plug-in is specified in pubspec.yaml and `flutter pub get` is run, the iOS version works properly. But remember to enter `pod install` in the `ios` folder after `flutter pub get` to make sure the plug-in is included.

Access API Reference Documentation

API reference documentation is included in SASCollector_<applicationID>.zip.

1. Navigate to the `Android` folder or the `iOS` folder in the SDK ZIP file (SASCollector_<applicationID>.zip).

2. To view the API documentation in a browser:

    a. Extract the contents of SASCollector-javadoc.jar (for Android) or iOSDocumentation.zip (for iOS) to a local directory.

    b. To open API reference documentation, open index.html.

    TIP For ease of use, bookmark the API reference URL in your browser.

3. Android only: To view the API documentation in Android Studio, add the SASCollector-javadoc.jar to the `app/libs` folder in your Android Studio project.

Each time you upgrade to the latest SDK, remember to refer to the latest API reference. Information about changes to the SAS Customer Intelligence mobile SDKs is available in the SDK Change Log.

Updates

October 2023 Updates

Starting from Flutter 3.3, context passed to Android platform view has changed from MainActivity context to MutableContextWrapper context. However, SASCollector SDK's mobile spot views only accept activity context. Due to this change, an update is needed in Mobile Spot Functionality's Configure Android section. In addition, the original spot implementations have a repetitively loading bug, and it appears on both Android and iOS. This bug will also be fixed in this update.

Android

1. In the sample project, find InlineAdViewFactory.java in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/views, and update its create method:

```
@NonNull
@Override
public PlatformView create(@Nullable Context context, int viewId,
  @Nullable Object args) {
  Map<String, Object> creationParams = (Map<String, Object>)args;

  Context ctx = context;
  if (context instanceof MutableContextWrapper) {
    ctx = ((MutableContextWrapper) context).getBaseContext();
  }
  return new InlineAdView(messenger, ctx, viewId, creationParams);
}
```

2. Find InlineAdView.java in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/views and do the following updates:

a. Add a boolean field:

```
Boolean isLoaded;
```

b. Update the constructor:

```
public InlineAdView(BinaryMessenger messenger, @NonNull Context
context, int id, @Nullable Map<String, Object> creationParams){

  super(messenger, Constants.Inline_Ad_Controller_Channel);

  spotID = (String)creationParams.get(Constants.Spot_ID);
  ad = new SASCollectorAd(context);
  ad.setBackgroundColor(Color.LTGRAY);
  setContents(ad, spotID);
  isLoaded = false;
}
```

c. Update getView method:

```
@Nullable
@Override
public View getView() {
  if (spotID != null && !isLoaded) {
     ad.load(spotID, null);
     isLoaded = true;
  }
  return ad;
}
```

Note: Step 1 fixed inline spot not displayed issue, and steps 2 a-c fixed inline spot repetitively loading issue.

3. Find IntertstitialAdViewFactory.java in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/views, and update its create method:

```
@NonNull
@Override
public PlatformView create(@Nullable Context context, int viewId,
  @Nullable Object args) {
  Map<String, Object> creationParams = (Map<String, Object>)args;

  Context ctx = context;
  if (context instanceof MutableContextWrapper) {
      ctx = ((MutableContextWrapper) context).getBaseContext();
  }
  return new InterstitialAdView(messenger, ctx, viewId,
             creationParams);
}
```

Note: Step 3 fixed interstitial spot not displayed issue.

4. Find BaseAdView.java in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/views and update its onMethodCall as follows:

```
@Override
public void onMethodCall(@NonNull MethodCall call,
  @NonNull MethodChannel.Result result) {
  switch (call.method) {
    default:
      result.success(null);
  }
}
```

Note: Step 4 fixed “MissingPluginException” bug. This bug does not cause mobile spots to not appear, but only displays “MissingPluginException” message in log.

iOS

1. In the sample project, find InlineAdViewController.m file in ios/Classes/views folder.
   a. Create a bool variable in @implementation:

```
@implementation InlineAdViewController {
   //…
   BOOL _isLoaded;
}
```

b. Update initWithFrame method by initializing _isLoaded variable:

```
-(instancetype)initWithFrame:(CGRect)frame viewIdentifier:
  (int64_t)viewId arguments:(id _Nullable)args
   binaryMessenger:(NSObject<FlutterBinaryMessenger>*)messenger {
     if (self = [super init]) {
       //...
       _isLoaded = NO;
     }
     return self;
  }
```

c. Update view method:

```
-(UIView* _Nonnull)view {
  if (!_isLoaded) {
    [_view load];
    _isLoaded = YES;
  }
   return _view;
}
```

Note: Steps 1 a-c fixed inline spot repetitively loading issue.

Dart

Note: Flutter supports two modes of hosting Android native views, hybrid composition and virtual displays. The sample project originally used virtual displays. However, some people may want to use hybrid composition. For this reason, the sample project is updated to hybrid composition for the mobile spots. Unless you also want hybrid composition, steps 1 and 2 are optional.

1. Find sas_collector_inline_ad_view.dart in libs/views folder, and do the following update:
   a. Add these imports:

```
import 'package:flutter/gestures.dart';
import 'package:flutter/rendering.dart';
```

   b. Replace the Android part in build method with this code:

```
case TargetPlatform.android:
  return PlatformViewLink(
```

```
      surfaceFactory: (context, controller) {
        return AndroidViewSurface(
          controller: controller as AndroidViewController,
          hitTestBehavior: PlatformViewHitTestBehavior.opaque,
          gestureRecognizers:
           const <Factory<OneSequenceGestureRecognizer>>{},
        );
      },
      onCreatePlatformView: (params) {
        return PlatformViewsService.initSurfaceAndroidView(
          id: params.id,
          viewType: viewType,
          layoutDirection: TextDirection.ltr,
          creationParams: creationParams,
          creationParamsCodec: const StandardMessageCodec(),
          onFocus: () {
            params.onFocusChanged(true);
        })
  ..addOnPlatformViewCreatedListener(params.onPlatformViewCreated)
  ..create();
    },
    viewType: viewType);
```

2. Find sas_collector_interstitial_ad_view.dart in libs/views folder, and do the following update:
   a. Add these imports:

```
import 'package:flutter/gestures.dart';
import 'package:flutter/rendering.dart';
```

   b. Replace the Android part in build method with this code:

```
case TargetPlatform.android:
  return PlatformViewLink(
    surfaceFactory: (context, controller) {
      return AndroidViewSurface(
        controller: controller as AndroidViewController,
        hitTestBehavior: PlatformViewHitTestBehavior.opaque,
        gestureRecognizers:
         const <Factory<OneSequenceGestureRecognizer>>{},
      );
    },
    onCreatePlatformView: (params) {
      onPlatformViewCreated(params.id);

      return PlatformViewsService.initSurfaceAndroidView(
        id: params.id,
        viewType: viewType,
        layoutDirection: TextDirection.ltr,
```

```
          creationParams: creationParams,
          creationParamsCodec: const StandardMessageCodec(),
          onFocus: () {
            params.onFocusChanged(true);
        })
    ..addOnPlatformViewCreatedListener(params.onPlatformViewCreated)
    ..create();
      },
      viewType: viewType);
```

Note: The original use of Virtual Display (AndroidView) is commented out in the sample project. You can compare it with the Hybrid Composition in steps 1 and 2.

3. Find pubspec.yaml in the sample project's root folder, and pubspec.yaml in example folder. Copy the contents of the the pubspec.yaml files from the sample project into your project. You may need to delete the two pubspec.lock files in your project. They will be recreated once the packages included in pubspec.yaml are fetched.

Note: The sample project was updated to run on Flutter 3.7. The package versions in pubspec.yaml may not work for older Flutter versions.

4. Your cocoapods may need to be updated too. Update it if running iOS app failed and indicated that cocoapods needs update.

March 2024 Updates

Mobile Spot

The native SASCollector iOS SDK (1.74.0) and Android SDK (1.82.0) added a new feature that allows developers to use resources such as fonts and icons from inside their apps to style their mobile spots. To leverage the new functionality and to also keep the original functionality (i.e., the functionality before the update), follow the steps. Please note that the cookbook only shows you how to implement the feature on inline mobile spot. If you need to have the feature in interstitial mobile spot, the instructions can be applied similarly.

Configure Android

1. Add these two constants in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/Constants.java:

```
public static String Use_Local_Resources = "useLocalResources";

public static String Resource_Path = "resourcePath";
```

2. Update InlineAdView.java in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter/views.

a. Add these two fields:

```
boolean useLocalResources;
String resourcePath;
```

b. In InlineAdView constructor, add this code:

```
useLocalResources =

 (boolean)creationParams.get(Constants.Use_Local_Resources);
resourcePath =

 (String)creationParams.get(Constants.Resource_Path);
```

c. In getView, add this code:

```
ad.useLocalResources(useLocalResources, resourcePath);
```

d. Add this method:

```
@Override
public void onMethodCall(@NonNull MethodCall call,

  @NonNull   MethodChannel.Result result) {
  super.onMethodCall(call, result);
  switch (call.method) {
    case "setUseLocalResources":
      isLoaded = false;
      boolean isUseLocalRsc =

        Boolean.TRUE.equals(call.argument("useResources"));
      useLocalResources = isUseLocalRsc;
      ad.useLocalResources(useLocalResources);
      ad.load(spotID, null);
      isLoaded = true;
      break;

    case "setUseLocalResourcesWithPath":
      isLoaded = false;
      boolean isUseLocal =
```

```
            Boolean.TRUE.equals(call.argument("useResources"));
          String path = call.argument("path");
          ad.useLocalResources(isUseLocal, path);
          ad.load(spotID, null);
          isLoaded = true;
          break;
      }
    }
```

Configure iOS

1. Update InlineAdViewController.m in ios/Classes/views.

a. Add this local variable inside @implementation InlineAdViewController { }. The location where the resources are in does not matter, and so there is no resourcePath variable unlike in Android.

```
BOOL _useLocalResources;
```

b. In initWithFrame method, add this code in if block:

```
_useLocalResources = (BOOL)args[@"useLocalResources"];
```

c. in onMethodCall method, add another if branch:

```
else if ([call.method isEqualToString:@"setUseLocalResources"] ||

  [call.method isEqualToString:@"setUseLocalResourcesWithPath"]) {

 _useLocalResources = [call.arguments[@"useResources"] boolValue];

  [_view useLocalResources:_useLocalResources];

  [_view load];

}
```

d. In view method, add this code inside if block:

```
[_view useLocalResources:_useLocalResources];
```

Configure Flutter (Dart)

Update the mobile_sdk_flutter plugin

1. In lib/views/sas_collector_inline_ad_view.dart, perform the following steps.

   a. Add these properties:

```
bool? useLocalResources;

String? resourcePath;
```

   b. Update the constructors:

```
SASCollectorInlineAdView(
  {Key? key,
   this.spotID,
   this.useLocalResources,
   this.resourcePath,
   this.onCreated})
: super(key: key);


SASCollectorInlineAdView.withoutLocalResources(
  {Key? key, spotID, onCreated})
: this(
  key: key,
  spotID: spotID,
  useLocalResources: false,
  resourcePath: null,
  onCreated: onCreated);
```

   c. In build method, update creationParams to add two more parameters:

```
final Map<String,dynamic> creationParams = <String, dynamic>
{ "spotID": widget.spotID,
  "useLocalResources": widget.useLocalResources,
  "resourcePath": widget.resourcePath
};
```

d. Still in build method, update the Android branch of creating the view (initSurfaceAndroidView) as below:

```
..addOnPlatformViewCreatedListener((id) {
  params.onPlatformViewCreated(id);
  onPlatformViewCreated(id);
})
```

2. In lib/views/sas_collector_inline_ad_view_controller.dart, add the following two methods:

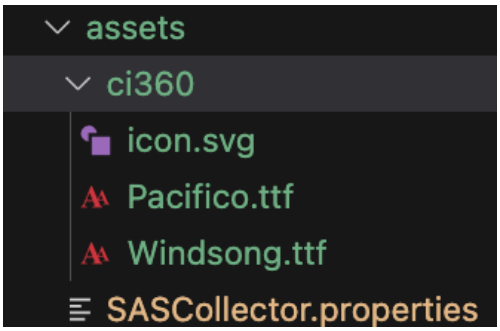
```
Future<void> setUseLocalResources(bool useLocalResources) async {
  return await channel.invokeMethod(
  'setUseLocalResources', {'useResources': useLocalResources});
}
Future<void> setUseLocalResourcesWithPath(
  bool useLocalResources, String path) async {
  return await channel.invokeMethod(
    'setUseLocalResourcesWithPath',
    {'useResources': useLocalResources, 'path': path});
}
```

Update the example app

1. Set up the html creative that uses local resources. Here is an example of the styles that is used in the example project.

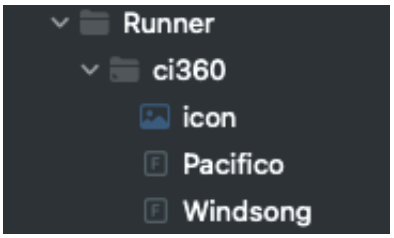
```
<style>
 @font-face {
   font-family: Pacifico;
   src: url('Pacifico.ttf')
 }
 @font-face {
   font-family: Windsong;
   src: url('Windsong.ttf')
 }
...
</style>
```

2. In example/android/app/src/main/assets, create a folder called ci360 and add the font and the icon files:

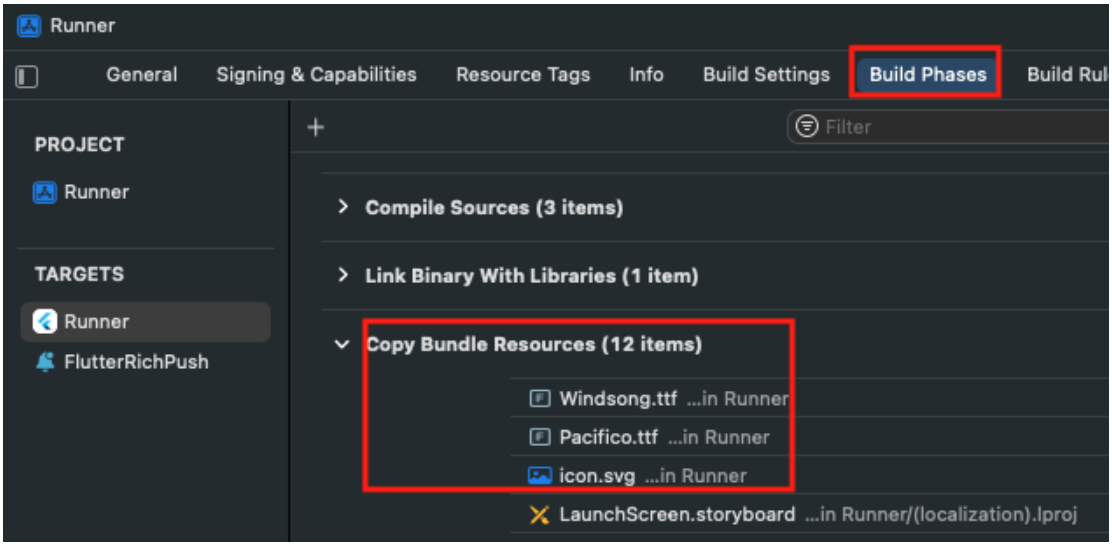


3. Open the example app project in Xcode and makes these updates:

a. Under Runner target, create a group called ci360 and add the font and icon files:



b. Select Runner target in edit window, then choose build phases. Make sure the fonts and icon files are in “Copy Bundle Resources”. If not, then click “+” button to add them.



4. In example/lib/view_page.dart, make these updates:

a. Add another inline view controller and a boolean variable:

```
late SASCollectorInlineAdViewController inlineAdController2;

bool toggleUseLocalResources = true;
```

b. Add this method:

```
void onInlineAd2Created(SASCollectorInlineAdViewController

  controller) {

   inlineAdController2 = controller;

   inlineAdController2.onLoadedHandler = () {

     Fluttertoast.showToast(

        msg: 'Inline Ad 2 is loaded',
```

```
        toastLength: Toast.LENGTH_LONG,
        gravity: ToastGravity.CENTER);
    };
  }
```

c. Add widgets for the new mobile spot. Replace the spotID with the one you created.

```
  Card(
    child: Padding(
      padding: const EdgeInsets.all(16),
      child: Column(
        children: [
          const Text(
            'Inline Spot 2',
            style: TextStyle(
              color: Colors.blue,
              fontSize: 20,
              fontWeight: FontWeight.bold),
          ),
          Container(
            color: Colors.grey,
            height: 2,
            width: double.infinity,),
          SizedBox(
            height: 180,
            width: 330,
```

```
            child: SASCollectorInlineAdView(
              spotID: 'Spot_UseLocalResources_WW',
              useLocalResources: true,
              resourcePath: 'ci360',
              onCreated: onInlineAd2Created,
            ),
          ),
          ElevatedButton(
            onPressed: () {
              toggleUseLocalResources =
                !toggleUseLocalResources;
              if (toggleUseLocalResources) {
              inlineAdController2.setUseLocalResourcesWithPath(
                toggleUseLocalResources, 'ci360');
              } else {
                inlineAdController2
                .setUseLocalResources(toggleUseLocalResources);
              }
            },
            child: const Text('Toggle using local resources'))
        ],
    ),),),
```

d. Update the existing mobile spot to be as below, with the changes in bold:

```
SASCollectorInlineAdView.withoutLocalResources(
```

```
            spotID: 'spot_id_1',

            onCreated: onInlineAd1Created,

  ),
```

Setting Application Version Programmatically

In addition to the mobile spot update, the iOS SDK also fixed a bug which made it impossible to set application version programmatically (i.e. calling SDK's setApplicationVersion method). Currently the cookbook also does not include instructions on how to set application version programmatically either. This is addressed in this section. Please note that the goal of the following steps is not only to set application versions, but also to have the application versions updated in UDM session_details table. Updating session_details table is not an explicit step in the following steps, but an implicit one performed in the backend. Please note that even though you can set application version programmatically as discussed in this section, the recommendation is still to set the version either in SASCollector.plist/SASCollector.properties or in the native apps (the version field in target's general setting in iOS, and the versionName in app build.gradle file in Android).

Configure the mobile_sdk_plugin library

This will include changes on Android, iOS and Flutter (dart).

Configure Android

1. In android/build.gradle, add these dependencies and sync the changes:

```
  implementation platform('com.google.firebase:firebase-bom:30.3.1')

  implementation 'com.google.firebase:firebase-core'

  implementation 'com.google.firebase:firebase-messaging'
```

2. Update MobileSdkFlutterPlugin.java in android/src/main/java/com/sas/SASIA/mobile_sdk_flutter:

   a. Add this variable. The import will be added if you do this in Android Studio. Otherwise, you can check the zip project for the imports.

```
private Activity activity;
```

b. Implement onAttachedToActivity:

```
@Override

public void onAttachedToActivity(

  @NonNull ActivityPluginBinding binding) {

  this.activity = binding.getActivity();

}
```

c. Remove setPushChannel in example/android/app/src/main/java/com/sas/SASIA/mobile_sdk_flutter_example, and add it in MobileSdkFlutterPlugin.java:

```
@RequiresApi(api = Build.VERSION_CODES.O)

private void setPushChannel() {

  NotificationManager notificationManager =

   (NotificationManager)

    activity.getSystemService(NOTIFICATION_SERVICE);



  String customAndroidChannel = "FlutterPushChannel";

  CharSequence channelName = "Flutter Channel";

  int importance = NotificationManager.IMPORTANCE_HIGH;

  NotificationChannel notificationChannel =

    new NotificationChannel(customAndroidChannel,

    channelName, importance);

  notificationChannel.enableLights(true);

  notificationChannel.setLightColor(Color.RED);
```

```
    notificationChannel.enableVibration(true);

    notificationChannel.setShowBadge(true);

    notificationChannel.setVibrationPattern(

      new long[]{100, 200, 300, 400, 500, 400, 300, 200, 400});

    notificationManager.createNotificationChannel(

      notificationChannel);

    SASCollector.getInstance()

      .setPushNotificationChannelId(customAndroidChannel);

  }
```

d. Remove the FirebaseMessaging code in example/android/app/src/main/java/com/sas/SASIA/mobile_sdk_flutter_example/MainActivity, and add a switch/case in MobileSdkFlutterPlugin's onMethodCall:

```
  case "setAppVersionAndInitSDK":

    String newAppVersion = call.argument("appVersion");

  SASCollector.getInstance().setApplicationVersion(newAppVersion);

  SASCollector.getInstance().initialize(activity);

  FirebaseMessaging.getInstance().getToken().addOnSuccessListener(

    tk -> {

      if(!TextUtils.isEmpty(tk)) {

        SASCollector.getInstance().registerForMobileMessages(tk);

    }

  });

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

    setPushChannel();
```

```
    }

    return;
```

Configure iOS

1. Update ios/Classes/MobileSdkFlutterPlugin.m.

   a. add two if/else branches:

```
  else if([call.method isEqualToString:@"setAppVersionAndInitSDK"]){

    NSString* newVersion = call.arguments[@"appVersion"];

    [SASCollector setApplicationVersion:newVersion];

    [SASCollector initializeCollection];

    result(nil);

  } else if([call.method isEqualToString:@"registerDeviceToken"]) {

    NSString* deviceTokenStr = call.arguments[@"deviceToken"];

    NSData *deviceToken = [self hexStrToData:deviceTokenStr];

    [SASCollector registerForMobileMessages:deviceToken

      completionHandler:^{

        NSLog(@"Registering token with SASCollector succeeded.");

      } failureHandler:^{

        NSLog(@"Registering token with SASCollector failed.");

     }];

     result(nil);

  }
```

   b. Create hexStrToData:

```
  -(NSData*)hexStrToData:(NSString*)hexStr {

     NSString *cleanedHex = [hexStr

       stringByReplacingOccurrencesOfString:@" " withString:@""];
```

```
    NSMutableData *data = [[NSMutableData alloc] init];
    unsigned char wholeByte;
    char byteChars[3] = {'\0', '\0', '\0'};
    int i;
    for(i=0; i<[cleanedHex length] / 2; i++) {
      byteChars[0] = [cleanedHex characterAtIndex:i*2];
      byteChars[1] = [cleanedHex characterAtIndex:i*2+1];
      wholeByte = strtol(byteChars, NULL, 16);
      [data appendBytes:&wholeByte length:1];
    }
    return data;
  }
```

Configure Flutter (dart):

1. In lib/mobile_sdk_flutter_platform_interface.dart, add this method:

```
Future<void> setAppVersionAndInitSDK(String appVersion){
  throw UnimplementedError(
    'setAppVersionAndInitSDK is not implemented');
}
```

2. In lib/mobile_sdk_flutter_platform_channel.dart, add this method:

```
@override
Future<void> setAppVersionAndInitSDK(String appVersion) async {
  return methodChannel
    .invokeMethod('setAppVersionAndInitSDK',
```

```
    {'appVersion': appVersion});
}
```

3. In lib/mobile_sdk_flutter.dart, add this method:

```
Future<void> setAppVersionAndInitSDK(String appVersion) {
  return MobileSdkFlutterPlatform.instance
    .setAppVersionAndInitSDK(appVersion);
}
```

Configure the example application

This section includes configuration of iOS and Flutter(dart). Android configuration in the example project is already updated in the section "*Configure the mobile_sdk_plugin library*".

Configure iOS

Because SASCollector initialization is done in the library (m*obile_sdk_plugin*), the device token registration in SASCollector that used to be performed in the native application side will not work. This is because the native side's AppDelegate lifecycle method calls are complete before the Flutter/dart methods are called, and so there is communication between the native and Flutter at this stage. This is also an issue in Android, and the solution was to move FirebaseMessaging code into the library. However, we cannot do so in iOS since the lifecycle method *didRegisterForRemoteNotificationsWithDeviceToken* can only be triggered in the application's AppDelegate. For this reason, we have to persist the device token until the native and the flutter sides can communicate and pass the token to SASCollector.

1. Make these updates in example/ios/Runner/AppDelegate.m.

   a. Add these two variables between *@implementation AppDelegate {}*:

```
NSData *deviceTokenToRegister;

BOOL hasRegisteredDeviceToken;
```

   b. Update didRegisterForRemoteNotificationsWithDeviceToken method:

```
-(void)application:(UIApplication *)application
```

```
    didRegisterForRemoteNotificationsWithDeviceToken:
    (nonnull NSData *)deviceToken {
    deviceTokenToRegister = deviceToken;
    hasRegisteredDeviceToken = NO;
  }
```

c. Add this method:

```
  -(NSString*)dataToHexStr:(NSData*)data {
    NSMutableString *str = [NSMutableString stringWithCapacity:64];
    NSUInteger length = [data length];
    char *bytes = malloc(sizeof(char)*length);
    [data getBytes:bytes length:length];
    for(int i=0; i<length; i++) {
      [str appendFormat:@"%02.2hhX", bytes[i]];
    }
    free(bytes);
    return str;
  }
```

d. Implement this method:

```
 - (void)applicationWillResignActive:(UIApplication *)application {
   if (!hasRegisteredDeviceToken) {
     NSString *deviceTokenStr =
       [self dataToHexStr:deviceTokenToRegister];
     NSDictionary *args = @{@"deviceToken": deviceTokenStr};
```

```
    [methodChannel invokeMethod:@"registerDeviceToken"
      arguments:args];
    hasRegisteredDeviceToken = YES;
  }
}
```

Configure Flutter (dart)

1. In example/lib/main.dart, make these changes.

a. In initState method, set application versions. In the below code, I set Android and iOS application versions differently. Depending on your use case, you can set the same application version.

```
if (Platform.isAndroid) {
  mobileSdkFlutterPlugin.setAppVersionAndInitSDK("2.0.1");
} else if (Platform.isIOS) {
  //other code skipped ...
  mobileSdkFlutterPlugin.setAppVersionAndInitSDK("1.1.1");
}
```

b. In setupConnectionWithNative method, add this switch case:

```
case 'registerDeviceToken':
  Map args = call.arguments;
  String deviceToken = args['deviceToken'];
  mobileSdkFlutterPlugin.registerForMobileMessages(deviceToken);
  break;
```

Optional SASMobileMessagingDelegate2

SASCollector iOS SDK release makes SASMobileMessagingDelegate2 optional when displaying mobile in-app messages. However, if you exclude SASMobileMessagingDelegate2, you will not get user interaction information of your app, such as when the user dismissed the in-app message. The following instructions show how to remove SASMobileMessagingDelegate2; however, the final example project will still include it.

Configure iOS

1. In example/ios/Runner/AppDelegate.h, remove SASMobileMessagingDelegate2, so it is changed to this:

```
@interface AppDelegate : FlutterAppDelegate<UIApplicationDelegate,

   UNUserNotificationCenterDelegate>

@end
```

2. In example/ios/ Runner/AppDelegate.m, remove SASMobileMessagingDelegate2's delegate methods:

```
- (void)messageDismissed

- (void)actionWithLink:(NSString * _Nonnull)link
type:(SASMobileMessageType)type
```