THE POWER TO KNOW.

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

SAS® Documentation

June 30, 2023

Contents

Overview 4

What You Should Know in Order to Use This Cookbook 4

Roles and Responsibilities 5

Initial Setup 6

Create Xamarin Binding Libraries 6

- Create a Xamarin App Project 7
- Obtain the SAS Customer Intelligence 360 Mobile SDKs 8
- Add SAS Customer Intelligence 360 Mobile SDK Libraries 9
  - Android 9
  - iOS 13

Basic Functionality 18

- Configure the Cross-Platform Project 19
- Configure Android 21
- Configure iOS 25
- Configure the Example Xamarin App 28
  - Cross-Platform 28
  - Android 31
  - iOS 32

Mobile Spot Functionality 33

- Configure the Cross-Platform Project 34
- Configure Android 36
- Configure iOS 37
- Configure the Example Xamarin App 38

Location Functionality 40

- Configure the Cross-Platform Project 42
- Configure Android 44
- Configure iOS 46
- Test Geofencing and Beacon Functionality 47
  - Android 47
  - iOS 47

Mobile Message Functionality 49

- Configure the Cross-Platform Project 49

Configure Android 53
Configure iOS 59
Test Push Notifications and In-App Messages 66
Test Push Notifications 66
Test In-App Messages 67
Access API Reference Documentation 67

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 Xamarin, 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 Xamarin is C#.

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 Xamarin 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 Xamarin Sample Package (.zip) that contains a sample Xamarin project (mobile_sdk_xamarin).

IMPORTANT The sample files and code examples are provided by SAS Institute Inc. "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Recipients acknowledge and agree that SAS Institute shall not be liable for any damages whatsoever arising out of their use of this material. In addition, SAS Institute will provide no support for the materials contained herein.

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 Xamarin 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:

- Visual Studio 2022 for Mac, build version 17.4. Make sure that Xamarin Android and Xamarin iOS are included in Visual Studio. If not, rerun Visual Studio Installer to add them. In the future, VS2022 Mac will be the shortened name for Visual Studio 2022 for Mac.

Note: Visual Studio 2022 for Windows can also be used. However, when the project is built to run on an iOS simulator or iOS device, your Windows computer must be paired to a Mac computer. It is therefore recommended to use a Mac for development.

- Android Studio and Xcode. Android Studio Chipmunk 2021.2.1 and Xcode 14.1 are used in this cookbook.
- Microsoft's Objective Sharpie. See Creating Bindings with Objective Sharpie for information about Sharpie. To download Sharpie, click https://aka.ms/objective-sharpie.

Note: The Xcode version that is used to create SASCollector.framework must match the Xcode version on the Mac where Sharpie runs. For example, if SASCollector.framework is created with Xcode 13, and Sharpie is run on a Mac where the Xcode version is 14, Sharpie fails.

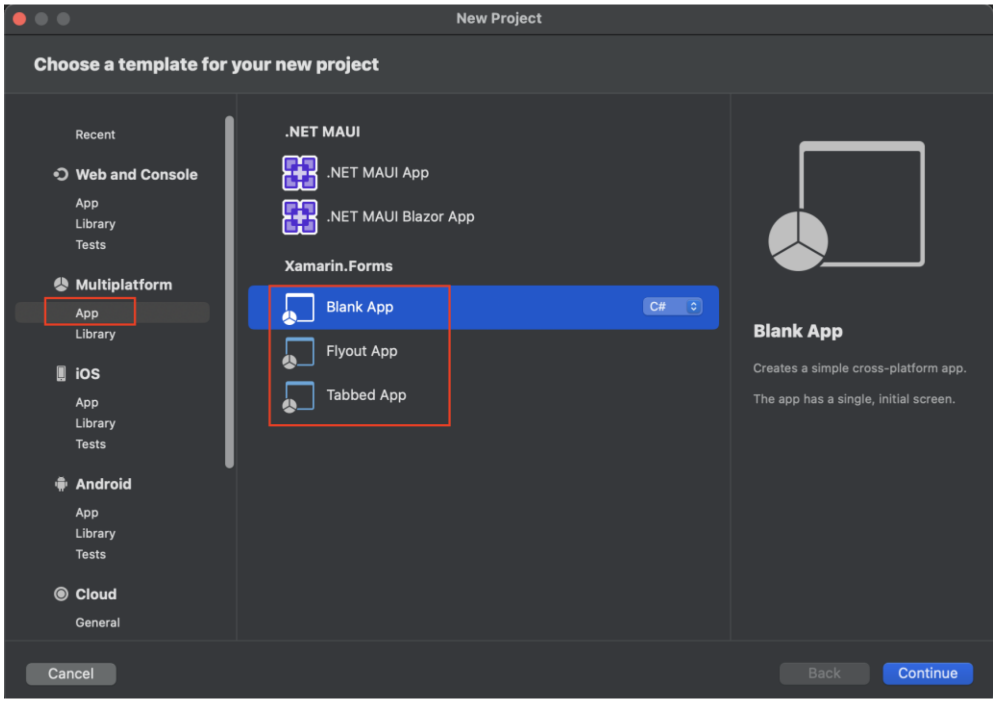
Create Xamarin Binding Libraries

In Xamarin, the native Android Java/Kotlin library and the native iOS Objective-C/Swift library are exposed to Xamarin apps through binding libraries. Unlike other cross-platform mobile frameworks (such as Flutter or React Native), Xamarin automates the process of creating binding libraries that wrap native libraries.

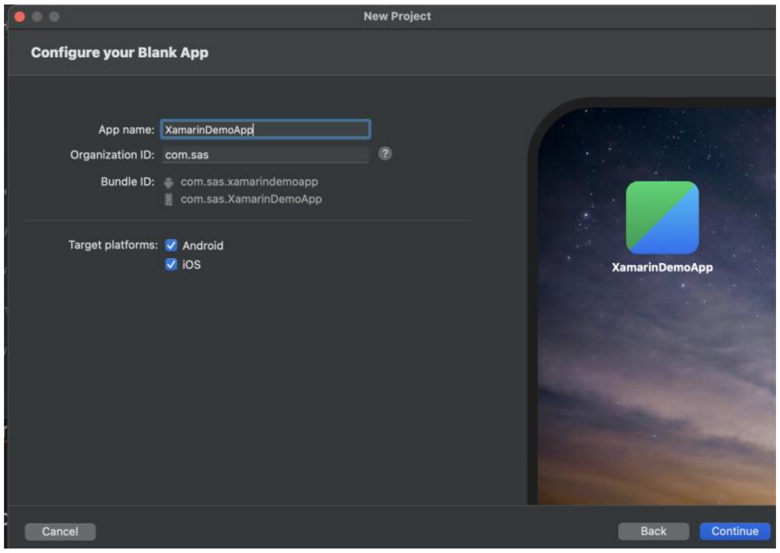
You can create a Xamarin project that hosts the binding libraries, and then import those libraries in Xamarin app projects. For simplicity, this cookbook creates the binding libraries in an existing Xamarin app project.

Create a Xamarin App Project

1. Open VS2022 Mac and select **New** to create a new project.
2. In the New Project window, under **Multiplatform**, select **App**, and then select any one of the three options under **Xamarin.Forms**. (For this cookbook, we chose **Blank App**.)



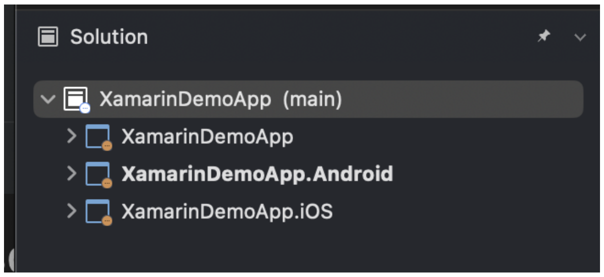
3. Click **Continue.**
4. Enter your app name and organization ID. Under **Target platforms**, select both **Android** and **iOS**.



App name: XamarinDemoApp
Organization ID: com.sas
Bundle ID: com.sas.xamarindemoapp
com.sas.XamarinDemoApp
Target platforms: ✓ Android
✓ iOS

Click **Continue** to finish creating the project.

The resulting project includes these folders: `XamarinDemoApp`, `XamarinDemoApp.Android`, and `XamarinDemoApp.iOS`.



Here is a description of the folders:

- The `XamarinDemoApp` folder contains code that is platform agnostic.
- The `XamarinDemoApp.Android` folder contains code that is specific to Android. The folder includesMainActivity.cs and AndroidManifest.xml. Those files are used to initialize SASCollector and to add other SASCollector functionality, such as setting up push notifications.
- The `XamarinDemoApp.iOS` folder contains code that is specific to iOS. The folder includes AppDelegate.cs and Info.plist. Those files are used to add SASCollector initialization and for push notification setup.

Note: Throughout this cookbook, the cross-platform project is referred to as *XamarinDemoApp*, the Android project is referred to as *XamarinDemoApp.Android*, and the iOS project is referred to as *XamarinDemoApp.iOS*.

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*.

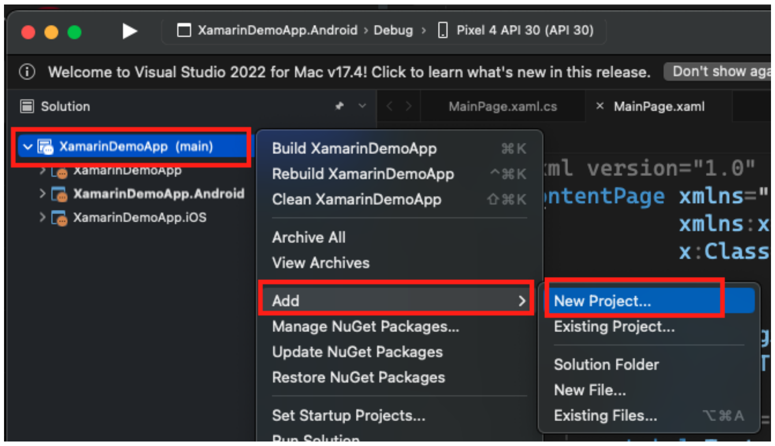
Note: A SASCollector.properties file (for Android) and a SASCollector.plist file (for iOS) contain necessary information to successfully implement the mobile SDKs, including the customer's selected tenant and mobile app ID. The files are not included in the public repository. The files must be obtained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

Add SAS Customer Intelligence 360 Mobile SDK Libraries

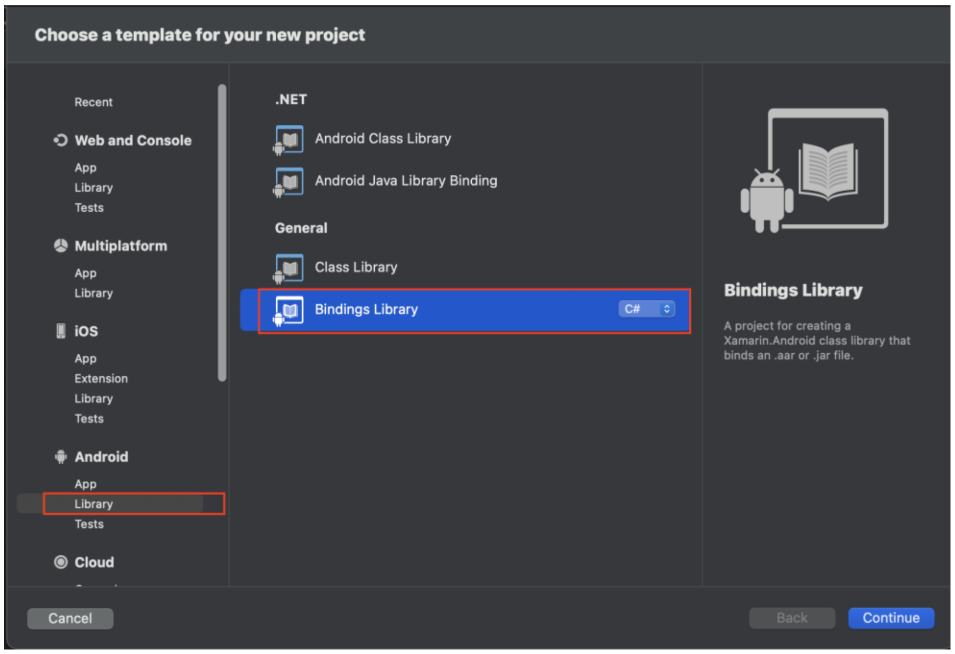
You need to create binding library projects to which you will add the SASCollector framework (library).

Android

1. In the app project that you created, right-click the solution and then select **Add** => **New Project…**, as shown in the example below.



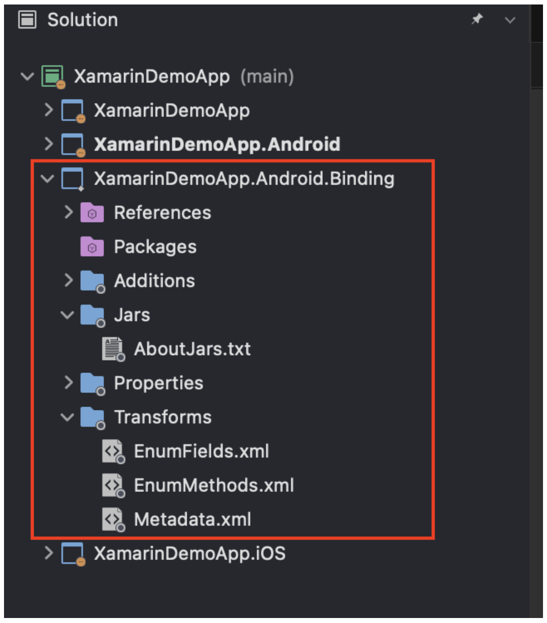
2. In the New Project window, in the left-hand pane, under **Android**, select **Library**. Then, in the center pane, under **General**, select **Binding Library**, as shown below.



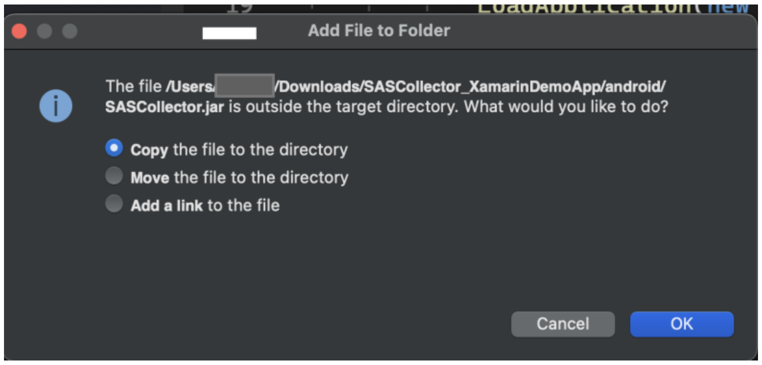
Click **Continue** to create the binding library.

Note: In this cookbook, the binding library is XamarinDemoApp.Android.Binding. You can name the library anything that makes sense to you, for example, SASCollector.Android.Binding.

The binding library that you create contains several folders and files. The files that you will work with most are in the `Jars` folder and in the `Transforms` folder.



3. Locate the SASCollector.jar file that you downloaded earlier and drag it into the `Jars` folder. Choose either the Copy option or the Move option, as shown below:


Add File to Folder

The file /Users/[redacted]/Downloads/SASCollector_XamarinDemoApp/android/SASCollector.jar is outside the target directory. What would you like to do?

[✓ Copy the file to the directory]
Move the file to the directory
Add a link to the file

Click **OK**.

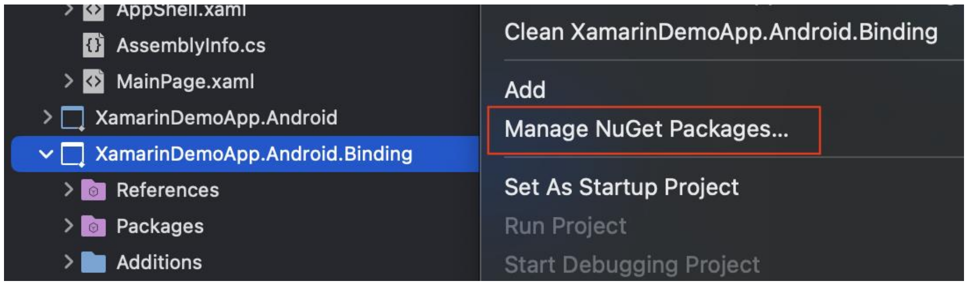
4. Navigate to the `Transforms` folder and open the Metadata.xml file. Add the following code between `<metadata>` and `</metadata>`:

```
<remove-node
path="/api/package[@name='com.sas.mkt.mobile.sdk.iam']/class[@
name='LargeMessageFragmentX']" />
  <remove-node
path="/api/package[@name='com.sas.mkt.mobile.sdk.iam']/class[@
name='SmallMessageFragmentX']" />
```

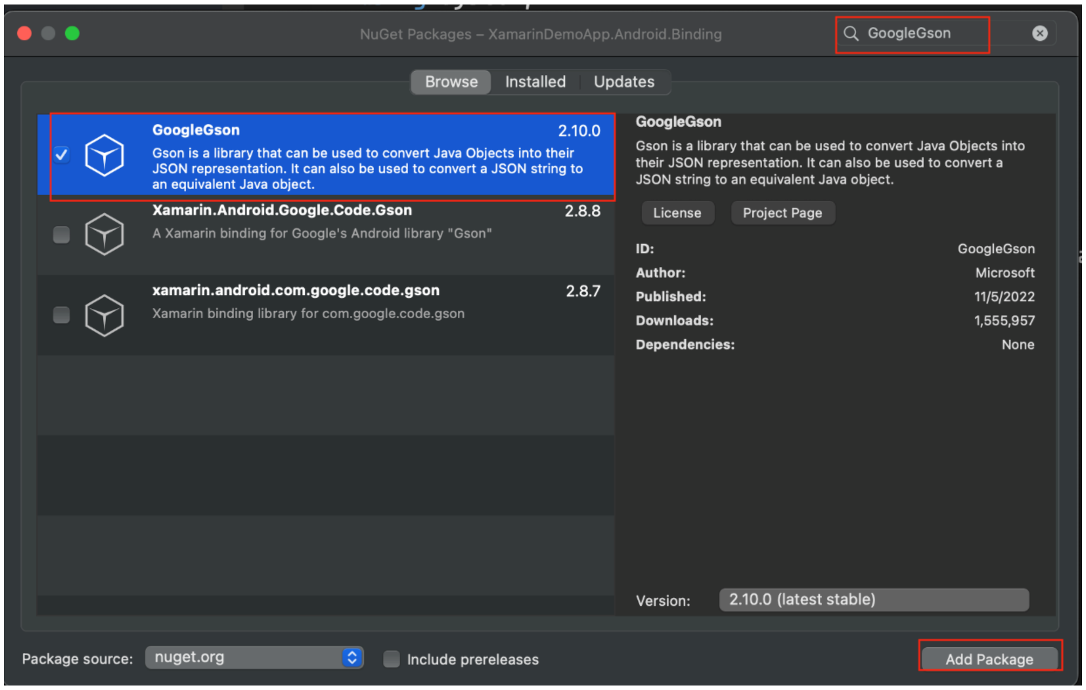
5. Create a new file called Extension.cs directly inside XamarinDemoApp.Android.Binding and replace its content with the following code:

```
namespace Com.Sas.Mkt.Mobile.Sdk.Ads

{

    public partial class DataTagParser :
     global::Java.Lang.Object

    {

     public partial class DataTag : global::Java.Lang.Object

     {

     }

    }

}
```

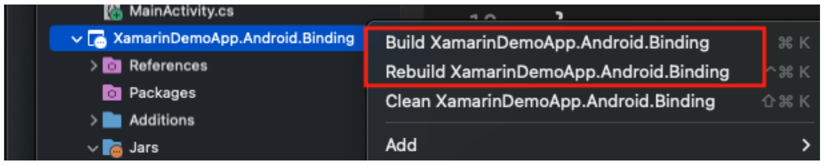
6. Right-click the binding library project and select **Manage NuGet Packages...**.



7. In the NuGet Packages window, search for `GoogleGson`, select it from the results list, and then click **Add Package**.

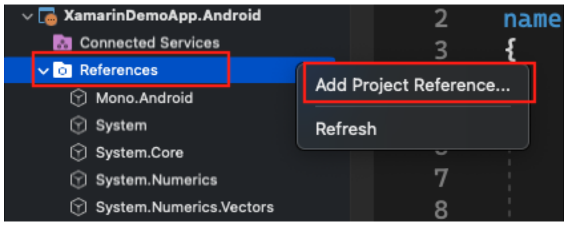

NuGet Packages – XamarinDemoApp.Android.Binding
GoogleGson
Browse
Installed
Updates
GoogleGson 2.10.0
Gson is a library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.
Xamarin.Android.Google.Code.Gson 2.8.8
A Xamarin binding for Google's Android library "Gson"
xamarin.android.com.google.code.gson 2.8.7
Xamarin binding library for com.google.code.gson
GoogleGson
Gson is a library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.
License
Project Page
ID: GoogleGson
Author: Microsoft
Published: 11/5/2022
Downloads: 1,555,957
Dependencies: None
Version: 2.10.0 (latest stable)
Package source: nuget.org
Include prereleases
Add Package

8. Right-click the binding library project and select either the Build option or the Rebuild option, as shown in the figure below:

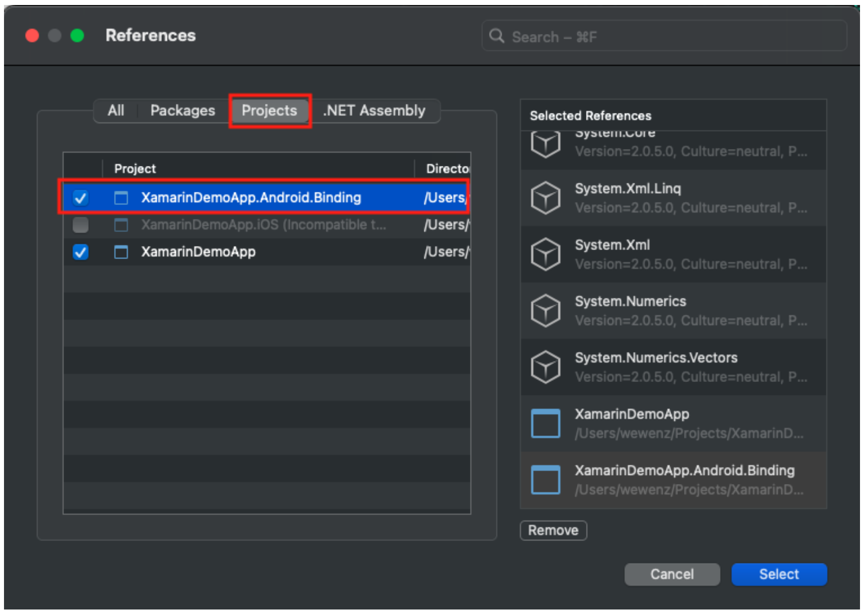

MainActivity.cs
XamarinDemoApp.Android.Binding
References
Packages
Additions
Jars
Build XamarinDemoApp.Android.Binding ⌘ K
Rebuild XamarinDemoApp.Android.Binding ^⌘ K
Clean XamarinDemoApp.Android.Binding ⇧⌘ K
Add

Note: The Rebuild option always builds the project, whereas the Build option might not build the project. This might occur if there is no change since the last time you built the project. Changes to SASCollector's public APIs affect the Metadata.xml and Extension.cs files.

9. After the binding library is successfully built, add it as a project reference in the XamarindDemoApp.Android project. To do this, right-click the **References** folder in the XamarinDemoApp.Android project and select **Add Project Reference...**



10. In the References window, select the **Projects** tab and the select the check box next to the XamarinDemoApp.Android.Binding project. Click **Select**.



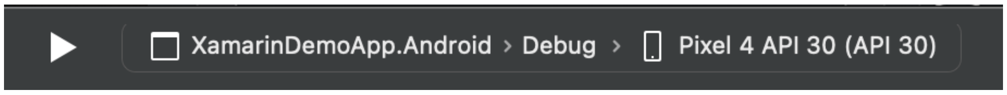
	Project	Directo
✓	XamarinDemoApp.Android.Binding	/Users/
	XamarinDemoApp.iOS (Incompatible t...	/Users/
✓	XamarinDemoApp	/Users/

System.Core
Version=2.0.5.0, Culture=neutral, P...
System.Xml.Linq
Version=2.0.5.0, Culture=neutral, P...
System.Xml
Version=2.0.5.0, Culture=neutral, P...
System.Numerics
Version=2.0.5.0, Culture=neutral, P...
System.Numerics.Vectors
Version=2.0.5.0, Culture=neutral, P...
XamarinDemoApp
/Users/wewenz/Projects/XamarinD...
XamarinDemoApp.Android.Binding
/Users/wewenz/Projects/XamarinD...

After the binding library project is referenced by the XamarinDemoApp.Android project, all the public classes and their public methods in SASCollector are available in the XamarinDemoApp.Android project.

Note: Xamarin wraps the native library in the binding library and makes it available in C# language.

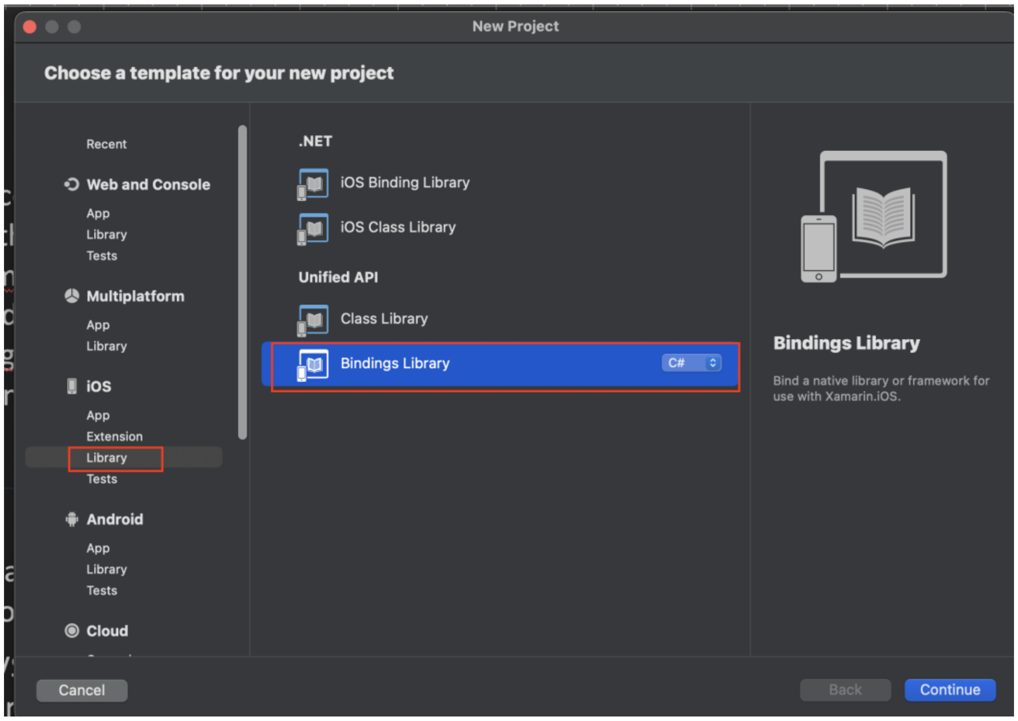
11. Verify that the inclusion of the binding library in the Android project works by running the app on an Android emulator. If there are any errors, the app does not run.



iOS

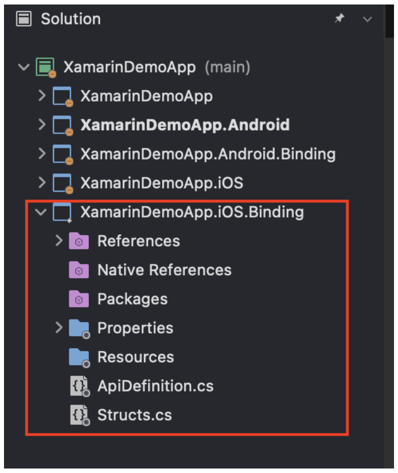
Note: The process of creating an iOS binding library that wraps the native library is more complex and error-prone than the process of creating the Android binding library.

1. In VS2022 Mac, right-click the app solution and select **Add** => **New Project**.
2. In the New Project window, in the left-hand pane, under **iOS**, select **Library**. Then, in the center pane, under **Unified API**, select **Binding Library**, as shown in the figure below.



Click **Continue** to create the binding library project. (In this cookbook, the project is XamarinDemoApp.iOS.Binding.)

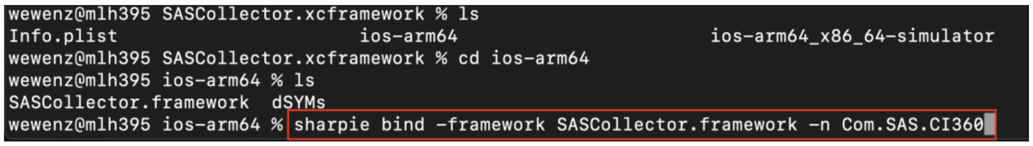
The binding library project contains several folders and files. The folder and files that you use to add SASCollector are the `Native References` folder, and the ApiDefinition.cs and Structs.cs files.



3. Open a terminal session, navigate to the SASCollector.xcframework package that you downloaded earlier, and then find the `ios-arms64` folder. Run this command:

```
sharpie bind -framework SASCollector.framework -n
Com.SAS.CI360
```

The figure below shows the path and the command:



```
wewenz@mlh395 SASCollector.xcframework % ls
Info.plist                      ios-arm64                      ios-arm64_x86_64-simulator
wewenz@mlh395 SASCollector.xcframework % cd ios-arm64
wewenz@mlh395 ios-arm64 % ls
SASCollector.framework  dSYMs
wewenz@mlh395 ios-arm64 % sharpie bind -framework SASCollector.framework -n Com.SAS.CI360
```

Note: SASCollector.framework is found in these two places: the `ios-arm64` folder and ios-arm64_x86_64-simulator. Use the SASCollector.framework that is in the `ios-arm64` folder because the framework is built for real devices. (Xamarin iOS must run on a real device for push notifications and mobile messages that are available only to real devices.)

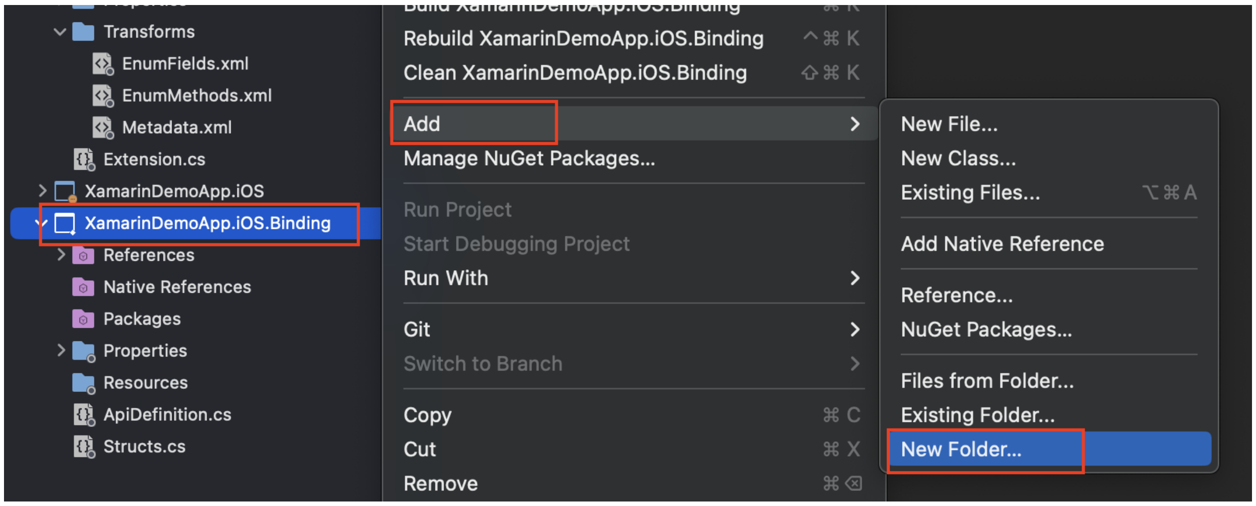
These two files are created by Sharpie: ApiDefinitions.cs and StructsAndEnums.cs.

TIP If you receive an error in Sharpie about a mismatch between SASCollector.framework and your version of Xcode, try copying the ApiDefinition.cs and Structs.cs files from the Mobile SDK Xamarin Sample Package ZIP file.

4. In VS2022 Mac, navigate to XamarinDemoApp.iOS.Binding. Replace the contents of ApiDefinition.cs with the contents of the ApiDefinitions.cs file that was created by Sharpie. Replace the contents of Structs.cs with the contents of the StructsAndEnums.cs file that was created by Sharpie.

Note: You will make additional changes to these two files after SASCollector.framework is added in the binding library.

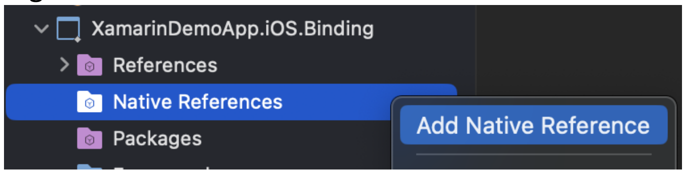
5. Right-click the XamarinDemoApp.iOS.Binding project and select **Add** => **New Folder…**.



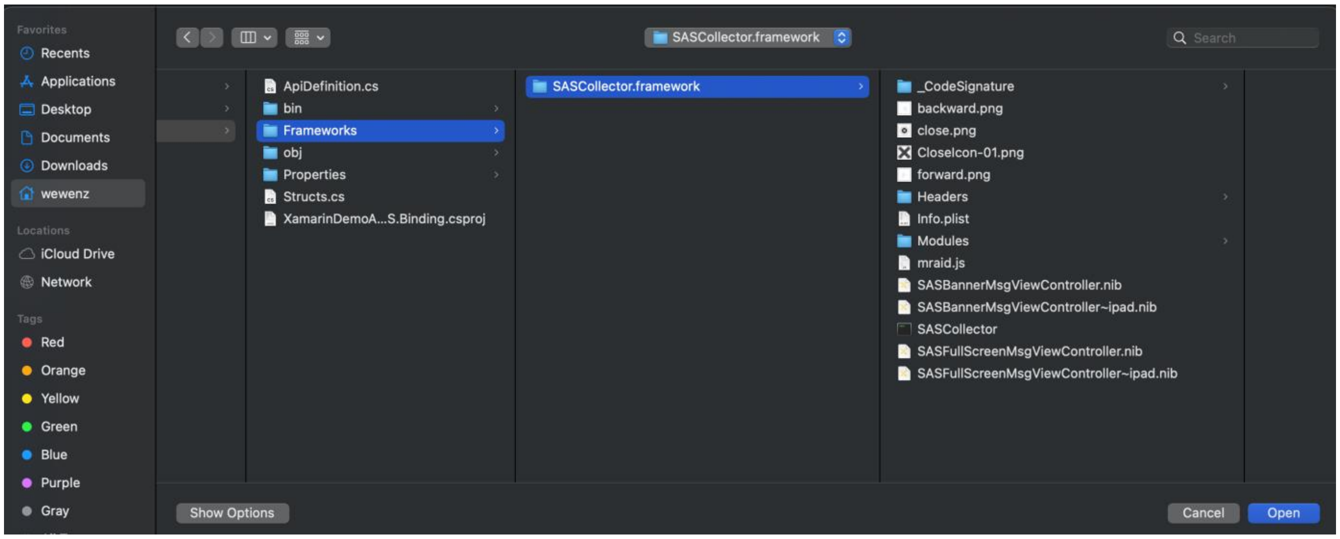
Name the folder **`Frameworks`**.

6. Find the SASCollector.framework that was used by Sharpie to generate ApiDefinitions.cs and StructsAndEnums.cs and copy it into the `Frameworks` folder.

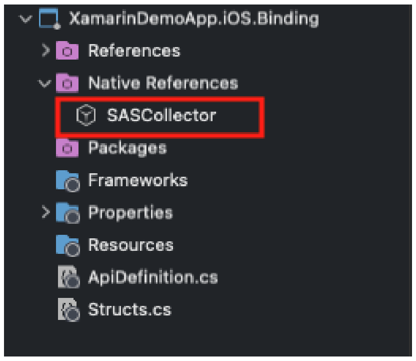
7. Right-click **Native References** and select **Add Native Reference**.



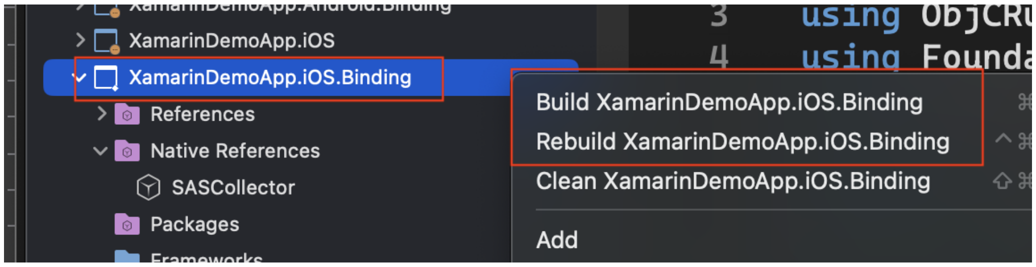
8. Navigate to **Frameworks**, select **SASCollector.framework**, and then click **Open**.



After SASCollector.framework is included in Native References, it is displayed under it as shown below:



9. Build the binding library project. To do this, right-click **XamarinDemoApp.iOS.Binding** and select either the Build XamarinDemoApp.iOS.Binding option or the Rebuild XamarinDemoApp.iOS.Binding option.


XamarinDemoApp.iOS
XamarinDemoApp.iOS.Binding
References
Native References
SASCollector
Packages
Build XamarinDemoApp.iOS.Binding
Rebuild XamarinDemoApp.iOS.Binding
Clean XamarinDemoApp.iOS.Binding
Add

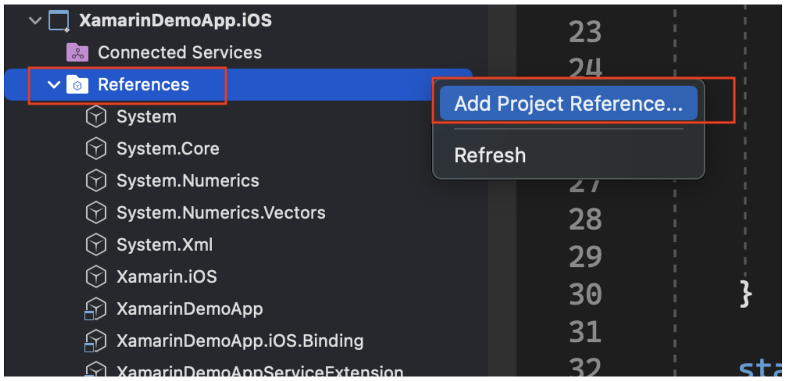
You can expect build errors. Here are several ways that you can fix them:

- Replace ApiDefinition.cs and Structs.cs with the original files from the Mobile SDK Xamarin Sample Package ZIP file and build the project again.

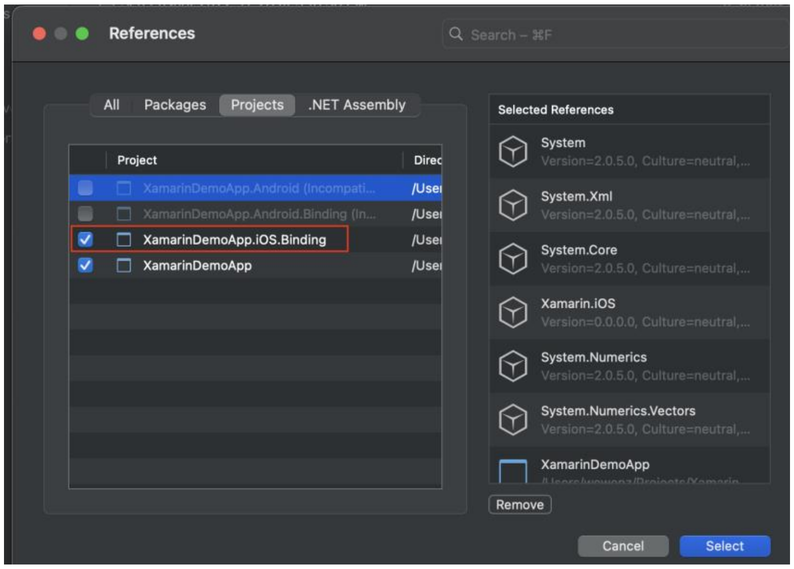
  Note: This option might not work if SASCollector.framework changed significantly. For example, some of its public classes and methods might be different from what was in the framework when you first downloaded it.

- Make changes to ApiDefinition.cs and Structs.cs. There are many corrections to make. Here are a few common ones:
  - Remove `using SASCollector` directives
  - Remove lines that start with `[Verify (…)]`; for example, `[Verify (MethodToProperty)]`
  - Change `NativeHandle` to `IntPtr`
- As stated in step 3, if there is a mismatch between SASCollector.framework and Xcode, copy ApiDefinition.cs and Structs.cs from the zipped project.

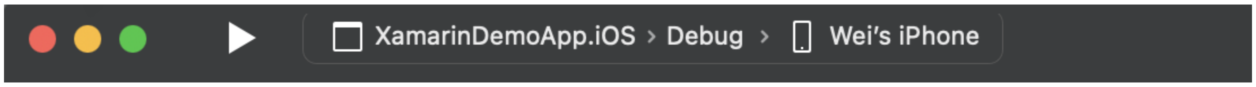
10. After the binding library project build succeeds, add it as a project reference in the iOS project. To do this, under **XamarinDemoApp.iOS**, right-click **References**, and then select **Add Project Reference...**.



11. Click the **Projects** tab, select **XamarinDemoApp.iOS.Binding**, and then click **Select**.



12. To verify that the inclusion of the binding library in the iOS project works, run the app on an iPhone device. If there are any errors, the app does not run.



IMPORTANT: Before you use binding libraries in a Xamarin app project, please be aware that the exposed functionality from libraries does not work on your app until you add SASCollector.properties to Android and SASCollector.plist files to iOS. For instructions, see "Configure the Example Xamarin App".

Basic Functionality

Some mobile app events, such as focus and defocus, do not need an explicit API call exposed through an interface to make them work. The integration of SAS Customer Intelligence mobile SDKs in the binding libraries and the Xamarin app is sufficient. The interface and its implementation classes are covered later in this section.

Other basic functions, such as custom events, page loads, and identity, need to be defined in the cross-platform project's interface and then implemented in iOS and Android projects to be used by the Xamarin 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 Xamarin app without any further code changes.

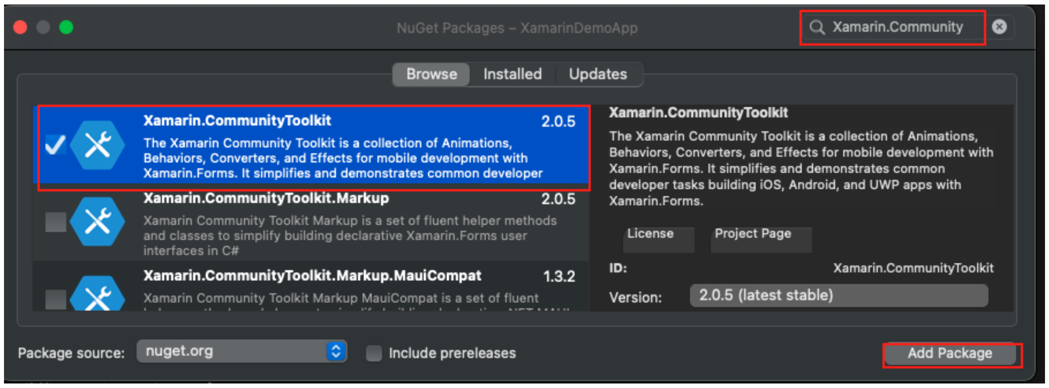
Note: The procedures in this section include more SASCollector public methods (functions) than just custom events, page loads, and identity. Some methods, such as getDeviceId, setDeviceId, can be used by developers for testing purposes. Others, such as startMonitoringLocation, disableLocationMonitoring, 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 Xamarin App” at the end of this section.

Configure the Cross-Platform Project

This cookbook uses the DisplayToastAsync method, which is included in this package, to show toast messages. For example, when an identity call fails, the mobile spots delegate methods and the mobile message delegate methods are called. If you need to show toast messages, complete step 1. Otherwise, skip to step 2.

1. To show toast messages:
   a. Right-click **XamarinDemoApp** and select **Manage NuGet Packages**.
   b. In the NuGet Packages window, search for **`Xamarin.CommunityToolKit`**, select it from the results list, and then click **Add Package**.



Xamarin.CommunityToolkit 2.0.5
The Xamarin Community Toolkit is a collection of Animations, Behaviors, Converters, and Effects for mobile development with Xamarin.Forms. It simplifies and demonstrates common developer

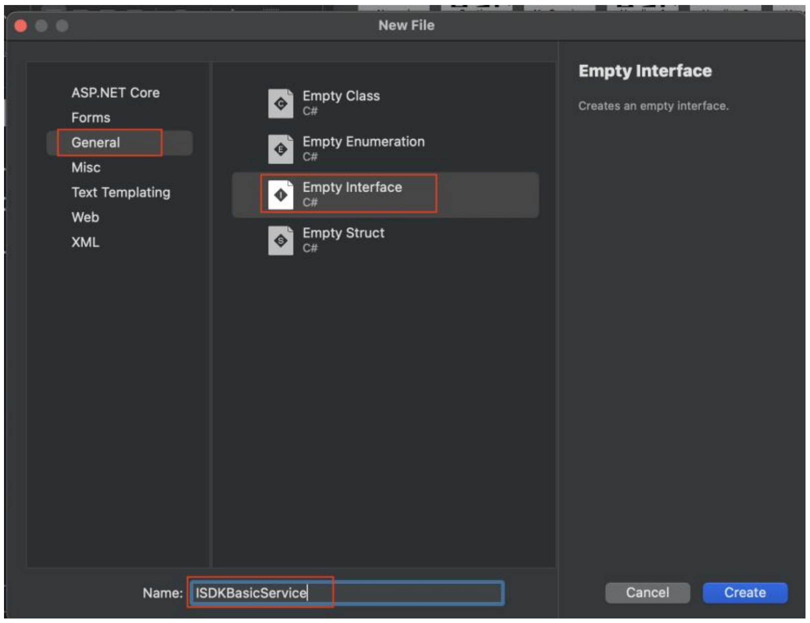
Xamarin.CommunityToolkit.Markup 2.0.5
Xamarin Community Toolkit Markup is a set of fluent helper methods and classes to simplify building declarative Xamarin.Forms user interfaces in C#

Xamarin.CommunityToolkit.Markup.MauiCompat 1.3.2
Xamarin Community Toolkit Markup MauiCompat is a set of fluent

Xamarin.CommunityToolkit
The Xamarin Community Toolkit is a collection of Animations, Behaviors, Converters, and Effects for mobile development with Xamarin.Forms. It simplifies and demonstrates common developer tasks building iOS, Android, and UWP apps with Xamarin.Forms.

ID: Xamarin.CommunityToolkit
Version: 2.0.5 (latest stable)

2. Right-click **XamarinDemoApp** and select **Add** => **New Folder**. Name the folder **`Services`**.
3. Right-click the `Services` folder and select **Add** => **New File**.
4. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Interface**, and enter **`ISDKBasicService`** for the name. Click **Create**.



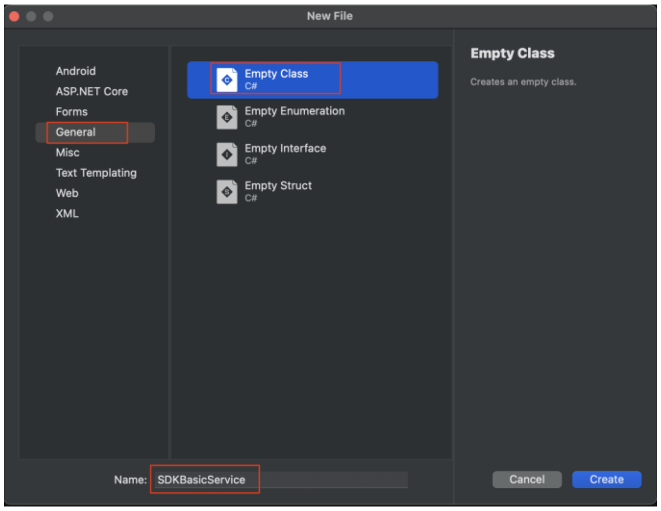
5. In ISDKBasicService.cs, add this code:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace XamarinDemoApp.Services
{
    public interface ISDKBasicService
    {
        void NewPage(string url);
        Task Identity(string type, string value);
        Task DetachIdentity();
        void AddAppEvent(string eventName,
            Dictionary<string, string> attrs);
        void StartMonitoringLocation();
        void DisableLocationMonitoring();


        string GetDeviceId();
        void ResetDeviceId();
        string GetTagServer();
        void SetTagServer(string newServer);
        string GetTenantId();
        void SetTenantId(string newTenant);
```

```
      }
}
```

Configure Android

1. Right-click **XamarinDemoApp.Android**, and then select **Add** => **New Folder**. Name the folder **`Services`**.
2. Right-click the `Services` folder, and then select **Add** => **New File**.
3. In the New File window, in the left-hand pane select **General,** in the center pane, select **Empty Class** in the center pane, and enter **`SDKBasicService`** for the name. Click **Create**.



4. Make the SDKBasicService class implement ISDKBasicService.

Note: The following code includes a few of the most often-used methods. For an example of other methods implementation and namespaces inclusion, please refer to `XamarinDemoApp.Android/Services/SDKBasicService.cs` in the mobile_sdk_xamarin project example.

```
[assembly: Xamarin.Forms.Dependency(typeof(SDKBasicService))]
namespace XamarinDemoApp.Droid.Services
{
   public class SDKBasicService : ISDKBasicService
     {
         public void AddAppEvent(string eventName,
           Dictionary<string, string> attrs)
         {
             SASCollector.Instance.AddAppEvent(eventName,
               attrs);
```

```
        }


        public void NewPage(string url)
        {
            SASCollector.Instance.NewPage(url);
        }
        public async Task Identity(string type,
            string value)
        {
            await Task.Run(() =>
             SASCollector.Instance.Identity(value, type,
                 new IdentityCallback(value)));
        }
        public async Task DetachIdentity()
        {
            await Task.Run(() =>
                SASCollector.Instance.DetachIdentity(
                    new DetachIdentityCallback()));
        }
        public void StartMonitoringLocation()
        {
            SASCollector.Instance.StartMonitoringLocation();
        }
        public void DisableLocationMonitoring()
        {
          SASCollector.Instance.DisableLocationMonitoring();
        }
    }
}
```

5. The Identity method takes as an input parameter a class instance that implements the SASCollector.IIdentityCallback interface. The DetachIdentity method takes as an input parameter a class instance that implements the SASCollector.IDetachIdentityCallback interface. Therefore, the two callback interfaces are used only in these two methods.

   Define two classes in SDKBasicService.cs file in the same namespace (for example, XamarinDemoApp.Droid.Services) as shown below:

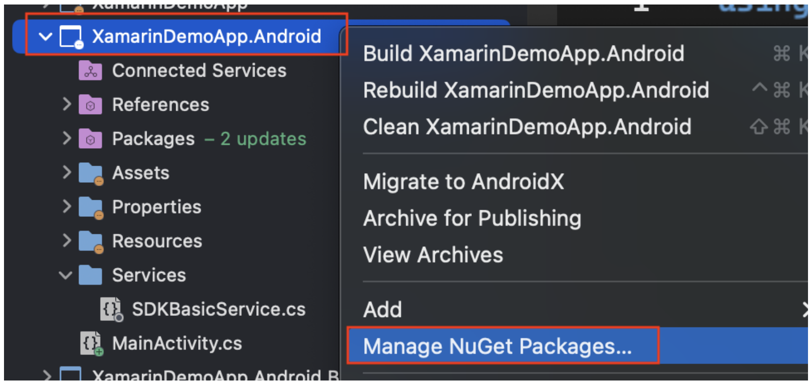
```
public class IdentityCallback : Java.Lang.Object,
SASCollector.IIdentityCallback
    {
        private string loginId;
        public IdentityCallback(string loginId)
        {
            this.loginId = loginId;
        }

        public void OnComplete(bool isSuccess)
        {
            MainThread.InvokeOnMainThreadAsync(() =>
            {
                if (isSuccess)
                {
                    (App.Current as
                      XamarinDemoApp.App)
                          .GoToDetails(loginId);
                }
                else
                {
                    (App.Current as
                      XamarinDemoApp.App).DisplayToastMsg(
                       "Login failed");
                }
            });
        }
    }

    public class DetachIdentityCallback : Java.Lang.Object,
     SASCollector.IDetachIdentityCallback
    {
        public void OnComplete(bool isSuccess)
        {
            MainThread.InvokeOnMainThreadAsync(() =>
            {
                if (isSuccess)
```

```
                    {
                         (App.Current as
                           XamarinDemoApp.App).GoToLogin();
                    }
                    else
                    {
                         (App.Current as
                           XamarinDemoApp.App).DisplayToastMsg
                             ("Logout failed");
                    }
               });
          }
     }
```

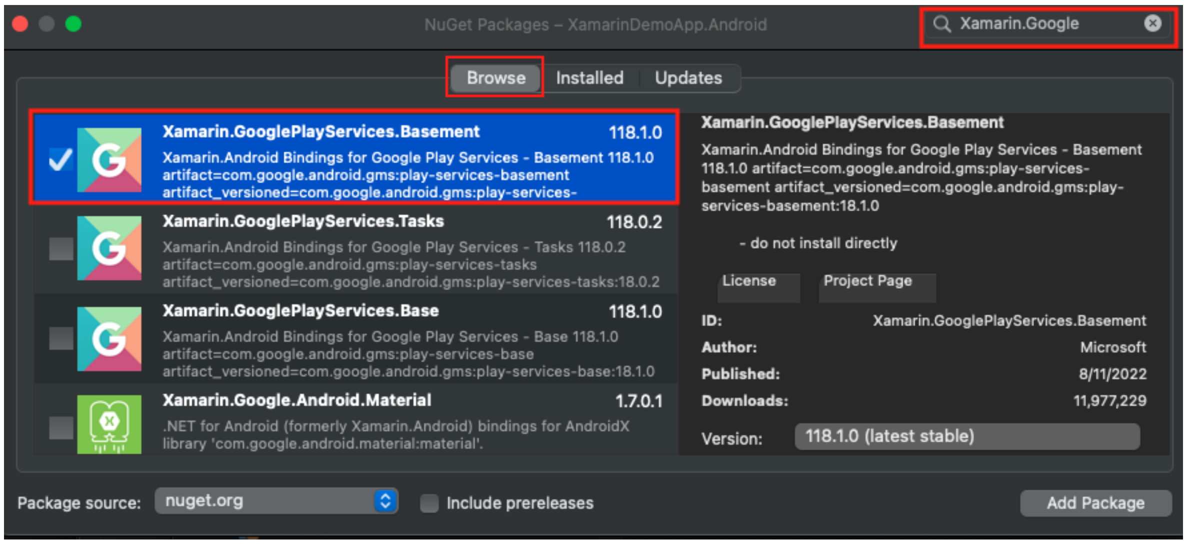
Note: The GoToDetails(), GoToLogin(), and DisplayToastMsg() methods are implemented in the App.xaml.cs file in the cross-platform project. You can create similar methods to achieve the same results.

6. Install Xamarin.GooglePlayServices.Basement in XamarinDemoApp.Android:

   a. Right-click **XamarinDemoApp.Android** and select **Manage NuGet Packages…**.



   b. In the NuGet Packages window, search for `Xamarin.Google`. Xamarin.GooglePlayServices.Basement is the first item in the list. Select it and then click **Add Package**.


NuGet Packages – XamarinDemoApp.Android

Xamarin.Google

Browse | Installed | Updates

✓	**Xamarin.GooglePlayServices.Basement** Xamarin.Android Bindings for Google Play Services - Basement 118.1.0 artifact=com.google.android.gms:play-services-basement artifact_versioned=com.google.android.gms:play-services-	118.1.0
	Xamarin.GooglePlayServices.Tasks Xamarin.Android Bindings for Google Play Services - Tasks 118.0.2 artifact=com.google.android.gms:play-services-tasks artifact_versioned=com.google.android.gms:play-services-tasks:18.0.2	118.0.2
	Xamarin.GooglePlayServices.Base Xamarin.Android Bindings for Google Play Services - Base 118.1.0 artifact=com.google.android.gms:play-services-base artifact_versioned=com.google.android.gms:play-services-base:18.1.0	118.1.0
	Xamarin.Google.Android.Material .NET for Android (formerly Xamarin.Android) bindings for AndroidX library 'com.google.android.material:material'.	1.7.0.1

Xamarin.GooglePlayServices.Basement

Xamarin.Android Bindings for Google Play Services - Basement 118.1.0 artifact=com.google.android.gms:play-services-basement artifact_versioned=com.google.android.gms:play-services-basement:18.1.0

- do not install directly

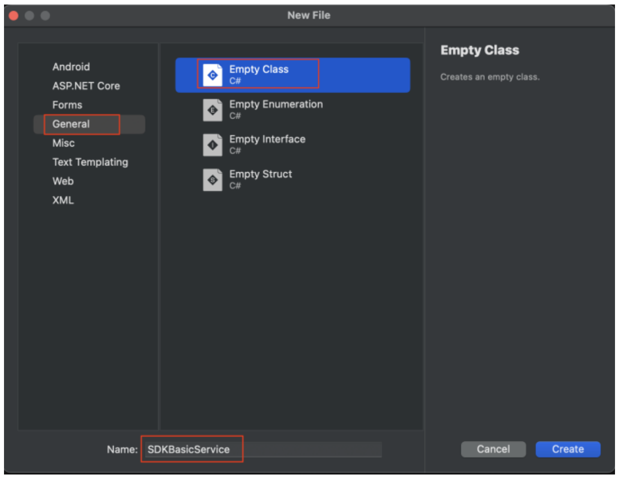
License | Project Page

ID:	Xamarin.GooglePlayServices.Basement
Author:	Microsoft
Published:	8/11/2022
Downloads:	11,977,229
Version:	118.1.0 (latest stable)

Package source: nuget.org | Include prereleases | Add Package

Configure iOS

1. Right-click **XamarinDemoApp.iOS** and select **Add** => **New Folder**. Name the folder **`Services`**.
2. Right-click the `Services` folder and select **Add** => **New File**.
3. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Class**, and enter **`SDKBasicService`** for the name. Click **Create**.



Name: SDKBasicService

4. Make the SDKBasicService class implement ISDKBasicService.

Note: The following code includes a few of the most often-used methods. For complete details about methods implementation and namespaces inclusion, please refer to `XamarinDemoApp.iOS/Services/SDKBasicService.cs` in the mobile_sdk_xamarin project example.

```
[assembly: Xamarin.Forms.Dependency(typeof(SDKBasicService))]
namespace XamarinDemoApp.iOS.Services
```

```
{
    public class SDKBasicService : ISDKBasicService
    {
        public async Task Identity(string type,
            string value)
        {
            Action<bool> completionHandler = (
                bool isSuccess) =>
            {
                MainThread.InvokeOnMainThreadAsync(() =>
                {
                    if (isSuccess)
                    {
                        (App.Current as XamarinDemoApp.App)
                            .GoToDetails(value);
                    }
                    else
                    {
                        (App.Current as XamarinDemoApp.App)
                            .DisplayToastMsg(
                                "Login Failed");
                    }
                });
            };
            await Task.Run(
               () => SASCollector.Identity(value,
                      type, completionHandler));
        }
        public async Task DetachIdentity()
        {
            Action<bool> completionHandler =
              (bool isSuccess) =>
            {
                MainThread.InvokeOnMainThreadAsync(() =>
                {
                    if (isSuccess)
                    {
            ((XamarinDemoApp.App)App.Current).GoToLogin();
                    }
                    else
                    {
                        (App.Current as XamarinDemoApp.App)
                            .DisplayToastMsg(
                                "Logout failed");
                    }
                });
            };
            await Task.Run(() =>
             SASCollector.DetachIdentity(
                 completionHandler));
        }
        public void NewPage(string url)
        {
            SASCollector.NewPage(url);
        }
        public void AddAppEvent(string eventName,
            Dictionary<string, string> attrs)
        {
            NSMutableDictionary<NSString, NSString>
            eventAttrs
            = new NSMutableDictionary<NSString, NSString>();
            if (attrs == null)
            {
                SASCollector.AddAppEvent(eventName, null);
                return;
            }
            foreach(var attr in attrs)
            {
                NSString key = NSString.FromData(attr.Key,
                 NSStringEncoding.UTF8);
                NSString value =
                   NSString.FromData(attr.Value,
                       NSStringEncoding.UTF8);
                eventAttrs.Add(key, value);
            }
            SASCollector.AddAppEvent(eventName, eventAttrs);
        }
        public void StartMonitoringLocation()
        {
            SASCollector.StartMonitoringLocation();
        }
        public void DisableLocationMonitoring()
        {
            SASCollector.DisableLocationMonitoring();
        }
}
```

Note: The GoToDetails(),GoToLogin(), and DisplayToastMsg() methods are implemented in the App.xaml.cs file in the cross-platform project. You can create similar methods to achieve the same results.

Configure the Example Xamarin App

The example Xamarin app in this cookbook refers to the cross-platform project, the Android project, and the iOS project. These are the configuration differences:

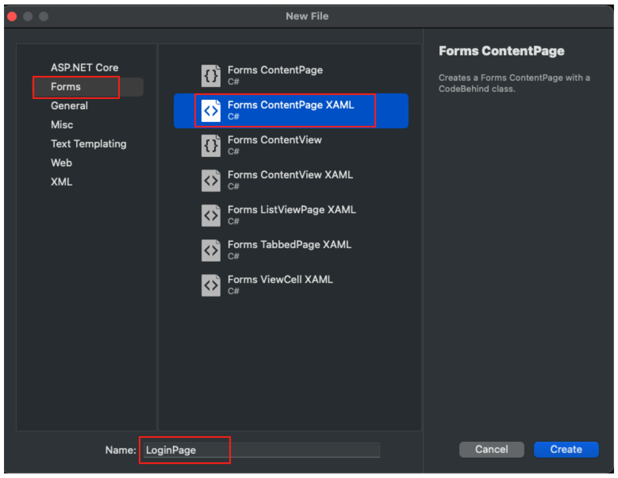
- In the cross-platform project, the presentations of the app that involve views, view models, and the App file (which is the entry point of the Xamarin app) are configured.
- In the Android project, MainActivity.cs and AndroidManifest.xml are configured, and SASCollector.properties is added.
- In the iOS project, AppDelegate.cs is configured and SASCollector.plist is added.

Cross-Platform

This cookbook describes more than basic SASCollector functionality in some of its examples. In the mobile_sdk_xamarin project example, each tabbed page shows different areas of functionality. The following procedure assumes that the tabbed structure is already created.

1. Right-click **XamarinDemoApp**, select **Add** => **New Folder**, and enter `Views` for the name.
2. Right-click the `Views` folder and select **Add** => **New File**.

3. In the New File window, in the left-hand pane, select **Forms**, in the center pane, select **Forms ContentPage XAML**, and then enter **`LoginPage`** for the name. Click **Create**.



4. In the New File window, in the left-hand pane, select **Forms**, in the center pane, select **Forms ContentPage XAML**, and then enter **`HomePage`** for the name. Click **Create**.

5. In the mobile_sdk_xamarin project example, copy the contents of LoginPage and HomePage from `XamarinDemoApp/Views/LoginPage.xaml` and `XamarinDemoApp/Views/HomePage.xaml.` Replace the content in the LoginPage and HomePage files that you created with the content that you copied from the mobile_sdk_ xamarin project example.

6. Right-click **XamarinDemoApp**, select **Add** => **New Folder**, and enter **`ViewModels`** for the name. Click **Create**.

7. Right-click the `ViewModels` folder and select **Add** => **New File**.

8. In the New File window, under **General**, select **Empty Class**, and name the class **`BaseViewModel`**. Click **Create**.

9. Repeat steps 6 through 8 to create folders and files for LoginViewModel, DetailsViewModel, HomeViewModel.

Note: The file names are the same as the folder names but with "Base" in front.

10. Replace the content of BaseViewModel.cs with this code:

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace XamarinDemoApp.ViewModels
{
    public class BaseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler
          PropertyChanged;

        protected virtual void OnPropertyChanged(
          [CallerMemberName] string propertyName = null)
        {
            PropertyChangedEventHandler handler =
              PropertyChanged;
            if (handler != null)
            {
                handler(this, new
              PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

11. Copy the content of LoginViewModel.cs from `XamarinDemoApp/ViewModels/LoginViewModel.cs` in the mobile_sdk_xamarin project example into the LoginViewModel.cs file that you created. The following code shows how the Identity method is called when a logon button is clicked:

```
private async void OnLoginClicked(object obj)
{
   await DependencyService.Get<ISDKBasicService>()
       .Identity(identityType, identityValue);
}
```

12. Copy the content of DetailsViewModel.cs from `XamarinDemoApp/ViewModels/DetailsViewModel.cs` in the mobile_sdk_xamarin project example into the DetailsViewModel.cs that you created. The following code shows how the DetachIdentity method is called when a logoff button is clicked:

```
private async void OnDetachIdentityClicked()
{
    await DependencyService.Get<ISDKBasicService>()
        .DetachIdentity();
}
```

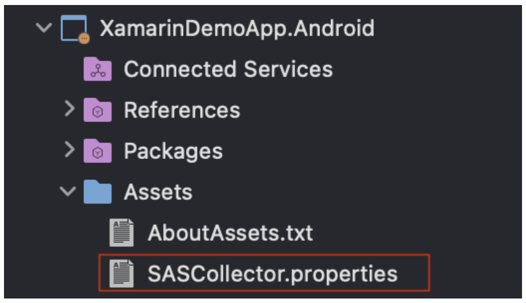
13. Copy the content of HomeViewModel.cs from `XamarinDemoApp/ViewModels/HomeViewModel.cs` in the mobile_sdk_xamarin project example into the HomeViewModel.cs that you created. The following code shows how NewPage and AddAppEvent are used:

```
private async void OnNewPageClicked()
```

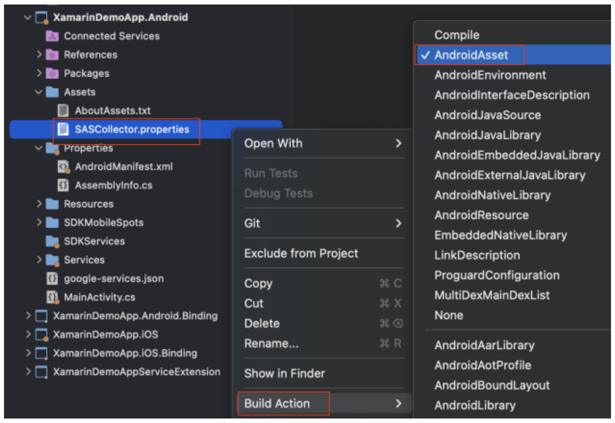
```
{
    DependencyService.Get<ISDKBasicService>().NewPage(
         NewPageUrl);
    await Application.Current.MainPage.DisplayToastAsync(
        "New Page event is sent");
}

private void OnAddAppEventClicked()
{
    if (string.IsNullOrEmpty(eventName))
    {
        return;
    }
    Dictionary<string, string> eventAttributes = null;
    if (!string.IsNullOrEmpty(eventAttrKey) &&
         !string.IsNullOrEmpty(eventAttrValue))
    {
        eventAttributes = new Dictionary<string, string>()
        {
            {eventAttrKey, eventAttrValue }
        };
    }
    DependencyService.Get<ISDKBasicService>()
        .AddAppEvent(eventName, eventAttributes);
}
```

Android

1. Drag the SASCollector.properties file (you downloaded it earlier) into the **XamarinDemoApp.Android** => **Assets** folder.



2. Right-click **SASCollector.properties**, select **Build Action**, and make sure that **AndroidAsset** is selected.



3. Under **XamarinDemoApp.Android** => **Properties,** in the AndroidManifest.xml file, add this line if it does not exist:

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

4. Under **XamarinDemoApp.Android**, in the MainActivity.cs file:

   a. Add these imports after the other `using` directives.

   ```
   using Com.Sas.Mkt.Mobile.Sdk;
   using Com.Sas.Mkt.Mobile.Sdk.Util;
   ```

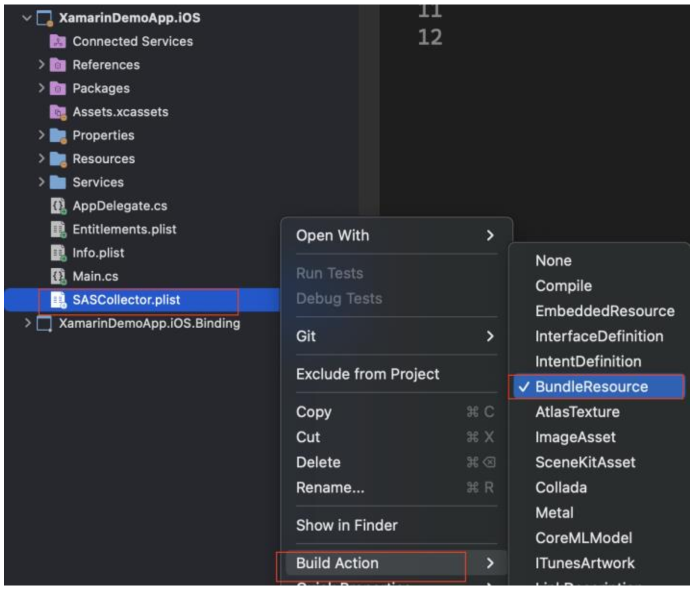
   b. In the OnCreate method, add this code after `LoadApplication()`:

   ```
   SLog.Level = SLog.All;
   SASCollector.Instance.Initialize(ApplicationContext);
   ```

   Note: Setting Slog.Level = Slog.All lets you view all events logs from SASCollector.

iOS

1. Drag SASCollector.plist (you downloaded it earlier) into **XamarinDemoApp.iOS**.
2. Right-click **SASCollector.plist**, select **Build Action** and make sure **BundleResource** is selected.



3. Under **XamarinDemoApp.iOS**, find AppDelegate.cs:
   a. Add this line after the other `using` directives:

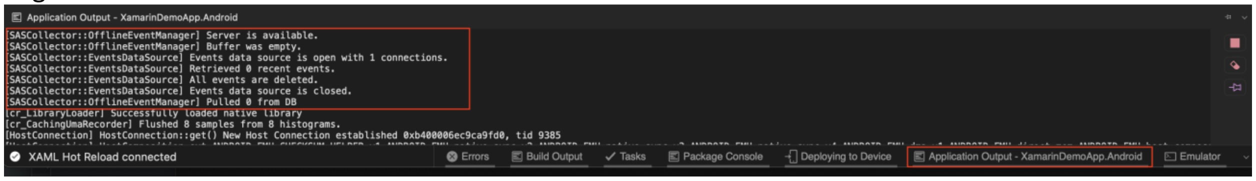
   ```
   using Com.SAS.CI360;
   ```

   b. Add this code to the FinishedLaunching() method , after `LoadApplication()`, to see all events logs from SASCollector:

   ```
   SASLogger.SetLevel(SASLoggerLevel.All);
   ```

To see SASCollector logs, at the bottom of VS 2022 Mac, click the **Application Output** tab, as shown in the following figures.

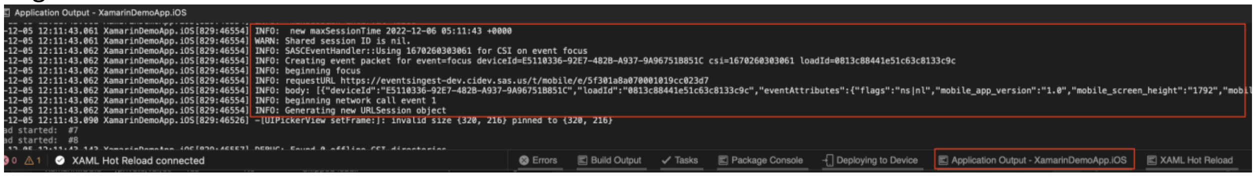
Logs in Android:



```
Application Output - XamarinDemoApp.Android
[SASCollector::OfflineEventManager] Server is available.
[SASCollector::OfflineEventManager] Buffer was empty.
[SASCollector::EventsDataSource] Events data source is open with 1 connections.
[SASCollector::EventsDataSource] Retrieved 0 recent events.
[SASCollector::EventsDataSource] All events are deleted.
[SASCollector::EventsDataSource] Events data source is closed.
[SASCollector::OfflineEventManager] Pulled 0 from DB
[cr_LibraryLoader] Successfully loaded native library
[cr_CachingUmaRecorder] Flushed 8 samples from 8 histograms.
[HostConnection] HostConnection::get() New Host Connection established 0xb400006ec9ca9fd0, tid 9385
```

XAML Hot Reload connected
Errors
Build Output
Tasks
Package Console
Deploying to Device
Application Output - XamarinDemoApp.Android
Emulator

Logs in iOS:



```
Application Output - XamarinDemoApp.iOS
-12-05 12:11:43.061 XamarinDemoApp.iOS[829:46554] INFO:  new maxSessionTime 2022-12-06 05:11:43 +0000
-12-05 12:11:43.061 XamarinDemoApp.iOS[829:46554] WARN: Shared session ID is nil.
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: SASCEventHandler::Using 1670260303061 for CSI on event focus
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: Creating event packet for event=focus deviceId=E5110336-92E7-482B-A937-9A96751B851C csi=1670260303061 loadId=0813c88441e51c63c8133c9c
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: beginning focus
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: requestURL https://eventsingest-dev.cidev.sas.us/t/mobile/e/5f301a8a070001019cc023d7
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: body: [{"deviceId":"E5110336-92E7-482B-A937-9A96751B851C","loadId":"0813c88441e51c63c8133c9c","eventAttributes":{"flags":"ns|nl","mobile_app_version":"1.0","mobile_screen_height":"1792","mobil
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: beginning network call event 1
-12-05 12:11:43.062 XamarinDemoApp.iOS[829:46554] INFO: Generating new URLSession object
-12-05 12:11:43.090 XamarinDemoApp.iOS[829:46526] -[UIPickerView setFrame:]: invalid size {320, 216} pinned to {320, 216}
ad started:  #7
ad started:  #8
```

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 Xamarin 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 Xamarin to be used in the Xamarin demo app. The creation of the Xamarin spots functions is described in three sections. The views that are created in “Configure the Cross-Platform Project” can be used either in the XAML file of the Xamarin app, or in XAML’s code-behind cs file, as described in “Configure the Example Xamarin App”. The real implementation of the views is in “Configure Android” and “Configure iOS”.

Configure the Cross-Platform Project

1. Right-click **XamarinDemoApp** and select **Add** => **New Folder**. Name the folder **`SDKMobileSpots`**.
2. Right-click the `SDKMobileSpots` folder, and then select **Add** => **New File**.
3. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Class**, and enter **`InlineAdView`** for the name. Click **Create**.
4. Replace the boilerplate code in InlineAdView.cs with this code:

```
using System;
using Xamarin.Forms;
namespace XamarinDemoApp.SDKMobileSpots
```

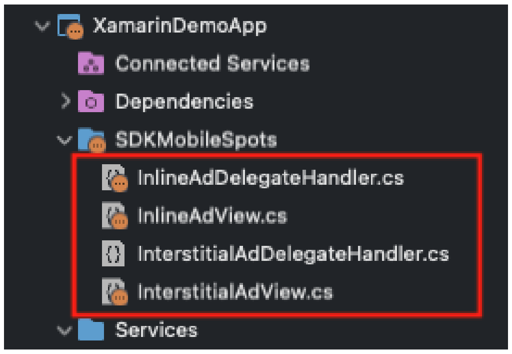
```
{
   public class InlineAdView : View
   {
           public static readonly BindableProperty
           SpotIdProperty =
               BindableProperty.Create(
                    propertyName: nameof(SpotId),
                    returnType: typeof(string),
                    declaringType: typeof(string),
                    defaultValue: string.Empty
               );

         public string SpotId
         {
                get { return (string)GetValue(SpotIdProperty);
}
                set { SetValue(SpotIdProperty, value); }
         }
   }
}
```

5. Repeat steps 2 and 3 to create InterstitialAdView.cs.
6. Replace the boilerplate code in InterstitialAdView.cs with this code:

```
using System;
using Xamarin.Forms;

namespace XamarinDemoApp.SDKMobileSpots
{
    public class InterstitialAdView : View
    {
        public static readonly BindableProperty
        SpotIdProperty =
            BindableProperty.Create(
                propertyName: nameof(SpotId),
                returnType: typeof(string),
                declaringType: typeof(string),
                defaultValue: string.Empty
            );

        public string SpotId
        {
            get { return (string)GetValue(SpotIdProperty); }
            set { SetValue(SpotIdProperty, value); }
        }
    }
}
```

7. Repeats step 2 and 3 to create InlineAdDelegateHandler.cs and InterstitialAdDelegateHandler.cs. Replace the boilerplate code with the code in the mobile_sdk_xamarin project example.

The following figure shows the files that you created in SDKMobileSpots.

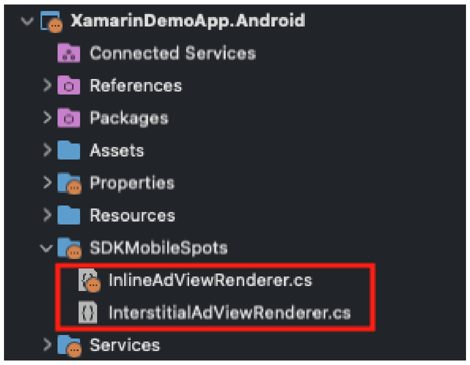


```
XamarinDemoApp
    Connected Services
  > Dependencies
  v SDKMobileSpots
      InlineAdDelegateHandler.cs
      InlineAdView.cs
      InterstitialAdDelegateHandler.cs
      InterstitialAdView.cs
  v Services
```

Configure Android

1. Right-click **XamarinDemoApp.Android** and select **Add** => **New Folder**. Name the folder **`SDKMobileSpots`**.
2. Right-click the `SDKMobileSpots` folder, and then select **Add** => **New File**.
3. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Class**. Name the class **`InlineAdViewRenderer`**, and then click **Create**.
4. Repeat steps 2 and 3 to create the InterstitialAdViewRenderer class.
5. Replace the content of InlineAdViewRender.cs with the content from `XamarinDemoApp.Android/SDKMobileSpots/InlineAdViewRenderer.cs` in the mobile_sdk_xamarin project example.
6. Replace the content of InterstitialAdViewRender.cs with the content from `XamarinDemoApp.Android/SDKMobileSpots/InterstitialAdViewRender.cs` in the mobile_sdk_xamarin project example.

The following figure shows the files that you created in `XamarinDemoApp.Android/SDKMobileSpots`.



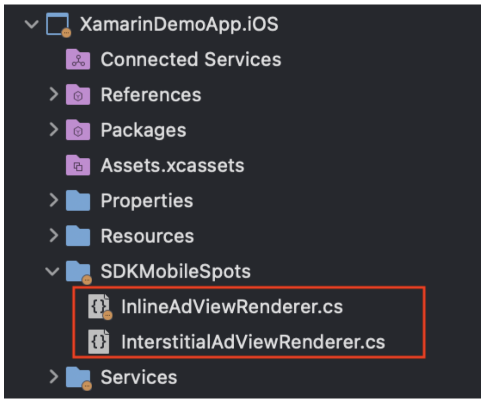
7. Find AndroidManifest.xml in `XamarinDemo.Android`/`Properties` and add this code in *`<application></application>`*:

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

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

Configure iOS

1. Right-click **XamarinDemoApp.iOS**, and then select **Add** => **New Folder**. Name the folder **`SDKMobileSpots`**.
2. Right-click the **`SDKMobileSpots`** folder, and then select **Add** => **New File**.
3. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Class,** and enter **`InlineAdViewRenderer`** for the name. Click **Create**.
4. Follow steps 2 and 3 to create InterstitialAdViewRenderer class.
5. Replace the content of InlineAdViewRender.cs with the content from `XamarinDemoApp.iOS/SDKMobileSpots/InlineAdViewRenderer.cs` in the mobile_sdk_xamarin project example.
6. Replace the content of InterstitialAdViewRender.cs with the content from `XamarinDemoApp.iOS/SDKMobileSpots/InterstitialAdViewRender.cs` in the mobile_sdk_xamarin project example.

   The following figure shows the files that you created in `XamarinDemoApp.iOS/SDKMobileSpots`:



The following notes apply to both Android and iOS.

Notes:

- Because they are related, InlineAdViewRenderer.cs and InterstitialAdViewRender.cs both include the view renderer class and the class that implements SASIA_AdDelegate class.
- The delegate classes InlineAdViewDelegate and InterstitialAdViewDelegate each contain a class instance called InlineAdDelegateHandler and InterstitialAdDelegateHandler, respectively. These delegate handlers are defined in the cross-platform project. You can change the content of the handlers to achieve your goal for events such as OnAdLoaded, OnAdClosed(). For demonstration purposes, the handlers in the cookbook display toast messages only when these events occur.

Configure the Example Xamarin App

1. Under **XamarinDemoApp**, right-click **Views** and select **Add** => **New File**.
2. In the New File window, in the left-hand pane, select **Forms**, in the center pane, select **Forms.ContentPage XAML**, and then enter **`MobileSpotsPage`** for the name. Click **Create**.
3. To set up an inline ad view and a button that opens an interstitial ad view, replace the content of MobileSpotsPage.xaml with this code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:mobilespot="clr-
namespace:XamarinDemoApp.SDKMobileSpots"
    x:Class="XamarinDemoApp.Views.MobileSpotsPage">
    <StackLayout Orientation="Vertical" Margin="0, 30,0,0"
Spacing="30" x:Name="ContainerLayout">
```

```
    <Label Text="Welcome to Mobile Spots Page"
        FontSize="26" FontAttributes="Bold"
        HorizontalOptions="CenterAndExpand"
        HorizontalTextAlignment="Center"
        VerticalOptions="Start" />

    <mobilespot:InlineAdView SpotId="your_inline_spot_id"
        HeightRequest="250"WidthRequest="250"
        HorizontalOptions="CenterAndExpand"
        BackgroundColor="LightGray"/>

    <Button HeightRequest="40" Margin="40,0"
        Clicked="InterstitialAd_Clicked"
        Text="View Interstitial Ad" />
    </StackLayout>
</ContentPage>
```

Note: In the code above, replace `your_inline_spot_id` with the inline spot ID for your app. For this example, the name of the button that opens the interstitial ad is called **`View Interstitial Ad`**, but you can name the button anything that makes sense to you.

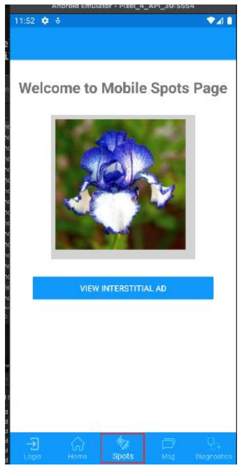
4. To render the interstitial ad view, replace the content of MobileSpotsPage.xaml.cs with this code:

```
using System;
using System.Collections.Generic;
using Xamarin.Forms;
using XamarinDemoApp.SDKMobileSpots;

namespace XamarinDemoApp.Views
{
    public partial class MobileSpotsPage : ContentPage
    {
        InterstitialAdView interstitialAdView;

        public MobileSpotsPage()
        {
            InitializeComponent();
         }

        private void InterstitialAd_Clicked(
            Object sender, EventArgs args)
        {
            if (interstitialAdView != null &&
      ContainerLayout.Children.Contains(interstitialAdView))
            {
       ContainerLayout.Children.Remove(interstitialAdView);
            }
            interstitialAdView = new InterstitialAdView();
```

```
             interstitialAdView.SpotId = "your_spot_id";
        ContainerLayout.Children.Add(interstitialAdView);
          }
     }
}
```

Note: In the code above, replace *`your_spot_id`* with the interstitial spot ID for your app.

In the example, the interstitial spot appears only when the user clicks the **View Interstitial Ad** button, whereas the inline ad view is the ad (represented by an iris in the figure below) that is shown when the user is on the mobile **Spots** tab, as illustrated by the following figure:



---

5. Add this code between <`TabBar`> and `</TabBar>` in the AppShell.xaml file:

```
<ShellContent Title="Spots" Icon="spotlight.png"
    Route="MobileSpotsPage"
    ContentTemplate="{DataTemplate local:MobileSpotsPage}" />
```

Note: In the code sample above, `spotlight.png` is the image used for the **Spots** tab that is shown in the figure in step 4. For this example, the image file was downloaded and added in `XamarinDemoApp.Android/Resources/drawable` folder and in `XamarinDemoApp.iOS/Resources` folder.

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.

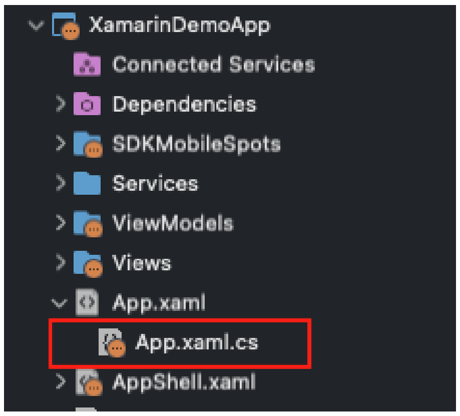
  Note: The startMonitoringLocation and disableLocationMonitoring functions were already added in `XamarinDemoApp/Services/ISDKBasicService.cs` as the interface methods, and in `XamarinDemoApp.Android/Services/SDKBasicService.cs` and `XamarinDemoApp.iOS/Services/SDKBasicService.cs` as concrete implementation 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 the Xamarin demo app.

Configure the Cross-Platform Project

1. Under **XamarinDemoApp** =>**App.xaml**, find App.xaml.cs.



2. Add these `using` directives at the start of the file:

```
using Xamarin.Essentials;

using System.Threading.Tasks;
```

3. Add the HasStartedLocationMonitoring property and the CheckLocationPermissionAndStartLocationService method:

```
public Boolean HasStartedLocationMonitoring { get; set; }

public async Task
CheckLocationPermissionAndStartLocationService()
    {
        PermissionStatus status = await
Permissions.CheckStatusAsync<Permissions.LocationAlways>();
        if (status == PermissionStatus.Denied &&
           DeviceInfo.Platform == DevicePlatform.iOS)
        {
            return;
        }
        if
(Permissions.ShouldShowRationale<Permissions.LocationAlways>()
)
        {
            await Shell.Current.DisplayAlert("Needs location
permission",
  "Location permission always is needed for enabling
geofencing and bluetooth functionality. \nPlease go to app
settings to set the permission."
                    , "OK");
            return;
         }

         status = await
Permissions.RequestAsync<Permissions.LocationAlways>();

         if (status == PermissionStatus.Granted)
         {
             DependencyService.Get<ISDKBasicService>()
                 .StartMonitoringLocation();
             HasStartedLocationMonitoring = true;
         }
     }
```

Note: Adding HasStartedLocationMonitoring ensures that CheckLocationPermissionAndStartLocationService is not repeatedly called. Making HasStartedLocationMonitoring a public property enables you to use it in other parts of the application. CheckLocationPermissionAndStartLocationService() can also be used in other locations outside of App.xaml.cs.

---

4. Add the OnStart override method after the class constructor:

```
protected async override void OnStart ()
{
    if (!HasStartedLocationMonitoring)
    {
       await
CheckLocationPermissionAndStartLocationService();
    }
}
```

Notes:

1. App.xaml.cs includes the OnResume override method. But OnResume is not called after OnStart is called, or when the app wakes up from sleep. Because of this, you cannot put CheckLocationPermissionsAndStartLocationService in OnResume.
2. When the user changes location permissions for the app, CheckLocationPermissionAndStartLocationService does not run again because OnStart is called only once. In order for CheckLocationPermissionAndStartLocationService to be called, the user must close the app and restart it.

---

5. Create the StopLocationMonitoring public method in App.xaml.cs:

```
public void StopLocationMonitoring()
{
    if (!HasStartedLocationMonitoring)
    {
       return;
    }
    DependencyService.Get<ISDKBasicService>()
        .DisableLocationMonitoring();
    HasStartedLocationMonitoring = false;
}
```

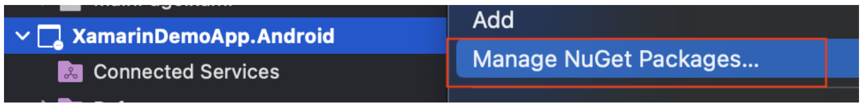
6. In AppShell.xaml.cs, within the OnAppearing() method, add the following code after `base.OnAppearing()`:

```
     await (App.Current as
XamarinDemoApp.App).CheckLocationPermissionAndStartLocationSer
vice();

             }
```

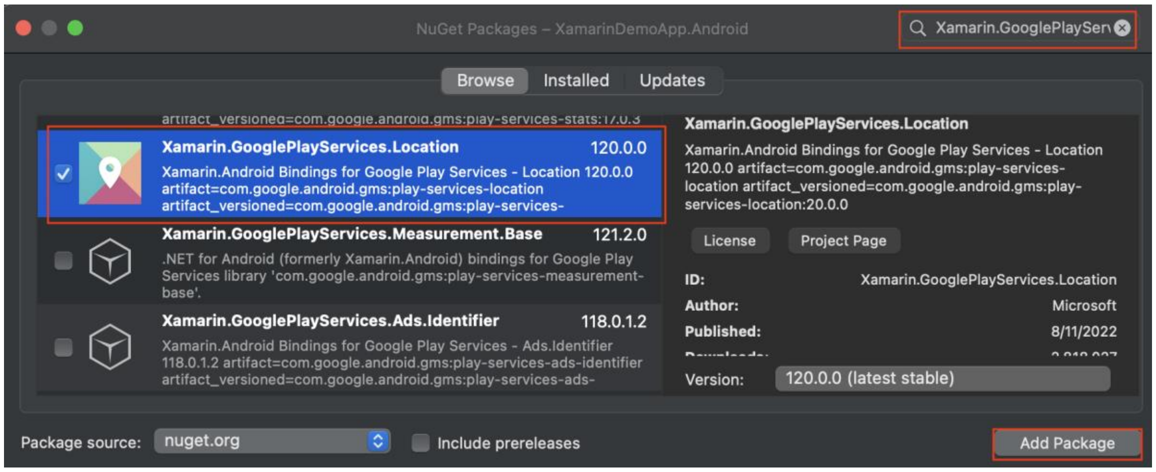
Note: This method is not used in the Xamarin demo app. You can call it anywhere you want in your app to stop monitoring location. For example, you might include a button which, when it is clicked, stops location monitoring.

Configure Android

1. Right-click **XamarinDemoApp.Android**, and then select **Manage NuGet Packages**.



2. In NuGet Packages window, search for **`Xamarin.GooglePlayServices.Location`**, select it from the results list, and then click **Add Package**.



artifact_versioned=com.google.android.gms:play-services-stats:17.0.3

Xamarin.GooglePlayServices.Location 120.0.0
Xamarin.Android Bindings for Google Play Services - Location 120.0.0 artifact=com.google.android.gms:play-services-location artifact_versioned=com.google.android.gms:play-services-

Xamarin.GooglePlayServices.Measurement.Base 121.2.0
.NET for Android (formerly Xamarin.Android) bindings for Google Play Services library 'com.google.android.gms:play-services-measurement-base'.

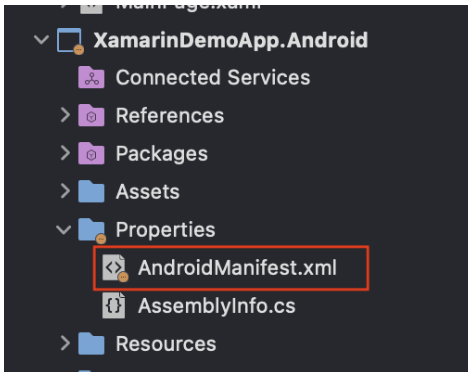
Xamarin.GooglePlayServices.Ads.Identifier 118.0.1.2
Xamarin.Android Bindings for Google Play Services - Ads.Identifier 118.0.1.2 artifact=com.google.android.gms:play-services-ads-identifier artifact_versioned=com.google.android.gms:play-services-ads-

Xamarin.GooglePlayServices.Location
Xamarin.Android Bindings for Google Play Services - Location 120.0.0 artifact=com.google.android.gms:play-services-location artifact_versioned=com.google.android.gms:play-services-location:20.0.0

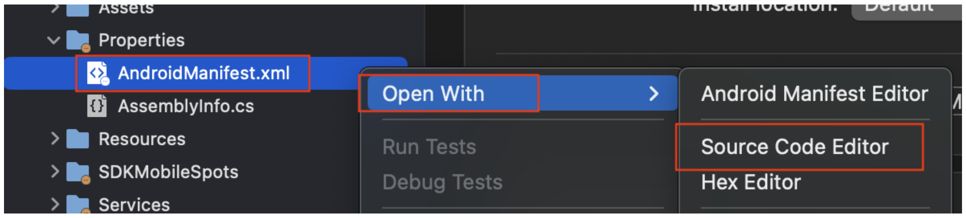
ID:	Xamarin.GooglePlayServices.Location
Author:	Microsoft
Published:	8/11/2022
Version:	120.0.0 (latest stable)

Package source: nuget.org
Include prereleases

3. Under **XamarinDemoApp.Android** => **Properties**, find AndroidManifest.xml.



4. Right-click **AndroidManifest.xml**, select **Open With**, and then select **Source Code Editor**.



5. In AndroidManifest.xml:

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

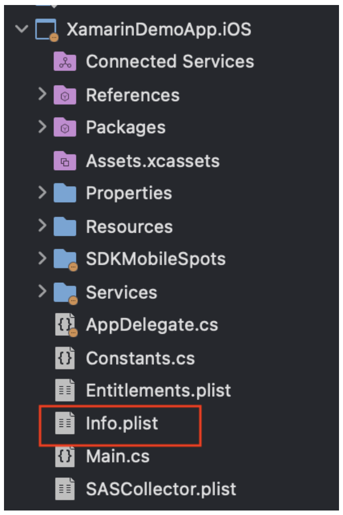
   b. In `<application></application>`, add this code:

```
<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
```

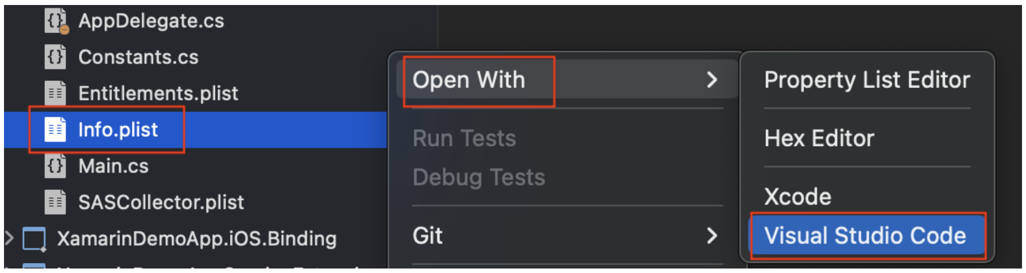
```
      android:name="android.intent.action.BOOT_COMPLETED"
/>
  </intent-filter>
</receiver>
```

Configure iOS

1. Under **XamarinDemoApp.iOS**, find Info.plist.



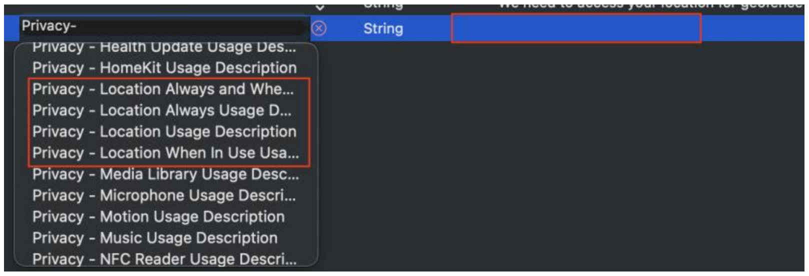
2. Right-click **Info.plist**, select **Open With**, and then select **Visual Studio Code** if Visual Studio Code is installed. Otherwise, select **Xcode** or **Property List Editor**.



   a. If you opened Info.plist in Visual Studio Code, copy the following location request permissions into your Info.plist file:

```
<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>
```

   b. If you opened Info.plist in Xcode or Property List Editor, click **Add new entry**, then select a key from the list of properties. Enter a value for the key. The figure below shows a list of keys to choose from after clicking **Add new entry**. (The String value is not yet entered.)


Privacy-
String
Privacy - Health Update Usage Des...
Privacy - HomeKit Usage Description
Privacy - Location Always and Whe...
Privacy - Location Always Usage D...
Privacy - Location Usage Description
Privacy - Location When In Use Usa...
Privacy - Media Library Usage Desc...
Privacy - Microphone Usage Descri...
Privacy - Motion Usage Description
Privacy - Music Usage Description
Privacy - NFC Reader Usage Descri...

Test Geofencing and Beacon Functionality

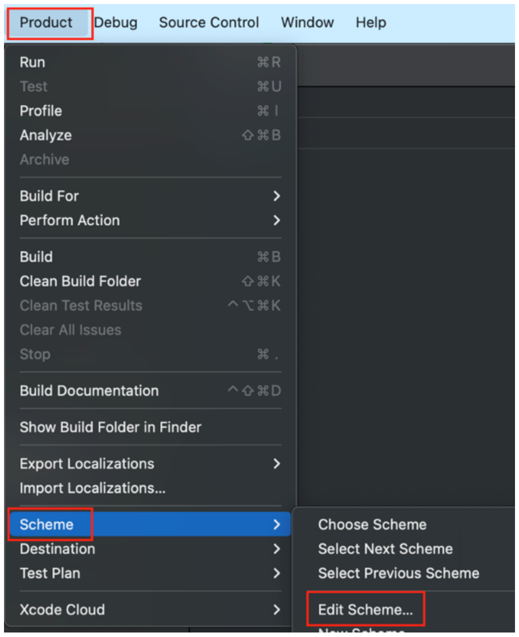
Create a geofence CSV file with the mobile app ID, longitude, latitude, radius, and so on. Give the file to the SAS Customer Intelligence 360 user to upload in SAS Customer Intelligence 360 where the mobile application is created. For information, see Upload Geofence and Beacon Data in *SAS Customer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

Android

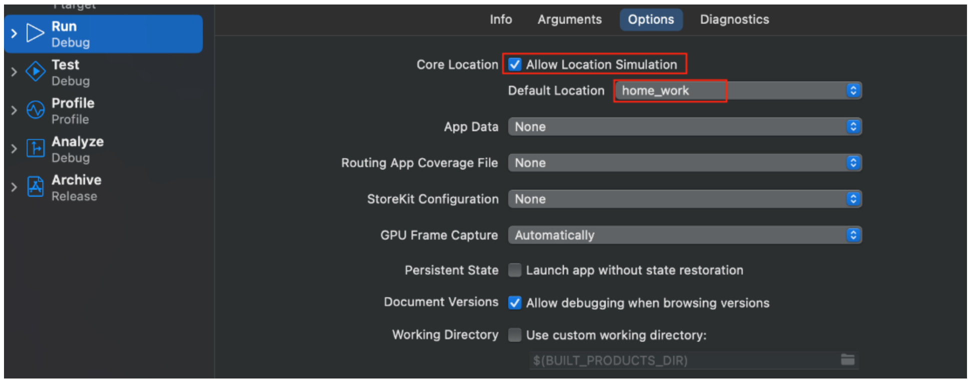
1. In the Android simulator, create a few location points. Make sure some, but not all, locations are also in the CSV file.
2. Start the example app and find a location in the simulator that is in the CSV file and set the location. The logs from Slog should have an enter_geofence event.
3. To test leaving 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 starter native iOS project in Xcode. You do not need to add any code in this project. It is used only to simulate location changes.
2. Create a GPX file in the native iOS 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.
3. In Xcode, navigate to **Product** => **Scheme** => **Edit Scheme**.



Select **Allow Location Simulation**. Make sure the GPX file is in the **Default Location** field. In the example below, the file name is home_work.



Core Location: ✓ Allow Location Simulation
Default Location: home_work
App Data: None
Routing App Coverage File: None
StoreKit Configuration: None
GPU Frame Capture: Automatically
Persistent State: Launch app without state restoration
Document Versions: ✓ Allow debugging when browsing versions
Working Directory: Use custom working directory:
$(BUILT_PRODUCTS_DIR)

4. In Xcode, run the starter native project on the real device that you use to run the Xamarin app.
5. In VS 2022 Mac, run the Xamarin demo app on the same real device.

Notes:

- After VS 2022 Mac deployed the Xamarin demo app on the device, it will terminate immediately. However, both the native and the Xamarin apps are still running. Since VS 2022 Mac stopped running, you cannot view SAS logs to verify geofence related events. You need to get events logs elsewhere to confirm that geofence works. If you can manage to have both the native and the Xamarin apps running from Xcode and VS 2022 Mac without getting VS 2022 Mac stopped, then you can use VS 2022 Mac to view geofence logs.

- When the native app is running, if a map is open, it moves from one location to another based on the setup in the GPX file.

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.

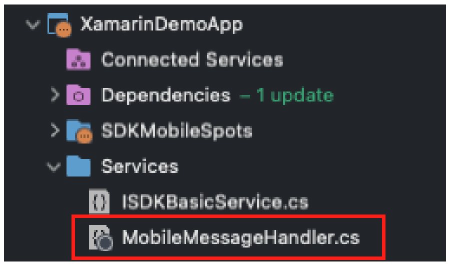
When the user clicks one of the buttons in an in-app message or opens a push notification, often the next action is to navigate to a particular section of your app. Design your delegate to be as flexible as possible so that it can perform navigation based on the link provided by the creative. This flexibility 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: In Xamarin, you can use the Firebase and Azure notification hub for push notifications, but they do not provide the full functionality that SAS Customer Intelligence 360 mobile messaging delivers.

Configure the Cross-Platform Project

1. Under **XamarinDemoApp**, select **Services**, right-click and then select **Add** => **New File**.
2. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Class**, and enter `MobileMessageHandler` for the name. Click **Create**.


XamarinDemoApp
Connected Services
Dependencies – 1 update
SDKMobileSpots
Services
ISDKBasicService.cs
MobileMessageHandler.cs

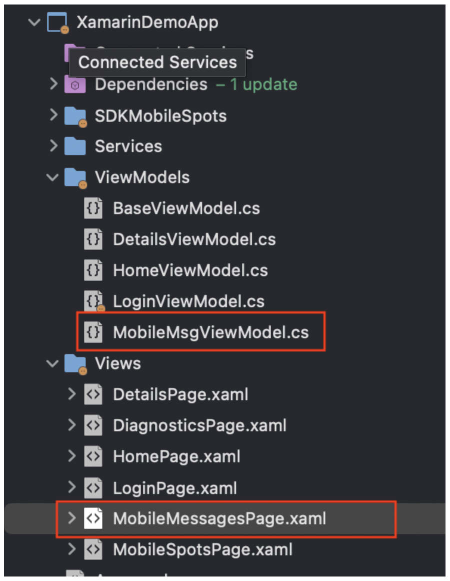
3. Add this code to the MobileMessageHandler.cs file:

```
using System;
using Xamarin.CommunityToolkit.Extensions;
using Xamarin.Forms;

namespace XamarinDemoApp.Services
{
    public class MobileMessageHandler
    {
        public static async void OnMsgDismissed()
        {
            await
Application.Current.MainPage.DisplayToastAsync(
                "Mobile Message is dismissed.");
        }
       public static async void OnMsgLinkClicked(
          string url, Boolean isPushNotification)
       {
 if (String.IsNullOrEmpty(url)) {
                return;
         }
         string msg = isPushNotification ? "Push notification
link: " : "In-App Message link: ";
          await
Application.Current.MainPage.DisplayToastAsync(
          msg + url + " is clicked");
 if (url.Equals("app://diagnostics")) {
                Shell.Current.CurrentItem =
                  Shell.Current.Items[0].Items[4];
                }
            }
        }
    }
```

Note: The code above demonstrates the handling of user actions for mobile messages with both a toast message and redirection to diagnostics tab. In the example project's AppShell.xaml file, the tabs are defined in the following code where TabBar is Items[0], and Diagnostics page is Items[4] in TabBar.

```
<TabBar>
        <ShellContent Title="Login" Icon="login.png"
          Route="LoginPage" ContentTemplate=
          "{DataTemplate local:LoginPage}" />
        <ShellContent Title="Home" Icon="home.png"
          Route="HomePage" ContentTemplate=
          "{DataTemplate local:HomePage}" />
        <ShellContent Title="Spots" Icon="spotlight.png"
          Route="MobileSpotsPage" ContentTemplate=
          "{DataTemplate local:MobileSpotsPage}" />
        <ShellContent Title="Msg" Icon="message.png"
          Route="MobileMessagesPage" ContentTemplate=
          "{DataTemplate local:MobileMessagesPage}" />
        <ShellContent Title="Diagnostics" Icon="stethoscope.png"
          Route="DiagnosticsPage" ContentTemplate=
          "{DataTemplate local:DiagnosticsPage}" />

    </TabBar>
```

4. Under **XamarinDemoApp**, right-click **Views,** and then select **Add** => **New File**.
5. In the New File window, in the left-hand pane, select **Forms**, in the center pane, select **Forms ContentPage XAML**, and enter **`MobileMessagesPage`** for the name. Click **Create**.
6. Under **XamarinDemoApp**, right-click **ViewModels**, and then select **Add** => **New File**.
7. In the New File window, in the left-hand pane, select **General**, in the center pane, select **Empty Class**, and enter **`MobileMsgViewModel`** for the name. Click **Create**.



8. Replace the content of the MobileMsgViewModel.cs file with the content from `XamarinDemoApp/ViewModels/MobileMsgViewModel.cs` in the mobile_sdk_xamarin project example.

   The following code shows that the SASCollector's AddAppEvent method is called when the small in-app and large in-app message buttons are clicked:

```
private void OnSmInAppEventClicked()
{
    if (string.IsNullOrEmpty(smInAppMsgEventName))
    {
       Application.Current.MainPage.DisplayToastAsync(
           "Please enter event name");
       return;
     }
     DependencyService.Get<ISDKBasicService>()
        .AddAppEvent(smInAppMsgEventName, null);
}

private void OnLgInAppEventClicked()
{
    if (string.IsNullOrEmpty(lgInAppMsgEventName))
    {
        Application.Current.MainPage.DisplayToastAsync(
            "Please enter event name");
        return;
    }
    DependencyService.Get<ISDKBasicService>()
        .AddAppEvent(lgInAppMsgEventName, null);
}
```

9. Replace the content of MobileMessagesPage.xaml with the content from `XamarinDemoApp/Views/MobileMessagesPage.xaml` in the mobile_sdk_xamarin project example.
10. In the code-behind file, MobileMessagesPage.xaml.cs, replace the boilerplate code with this code:

```
using Xamarin.Forms;
using XamarinDemoApp.ViewModels;

namespace XamarinDemoApp.Views
{
    public partial class MobileMessagesPage : ContentPage
    {
        public MobileMessagesPage()
        {
            InitializeComponent();
            BindingContext = new MobileMsgViewModel();
        }
    }
}
```

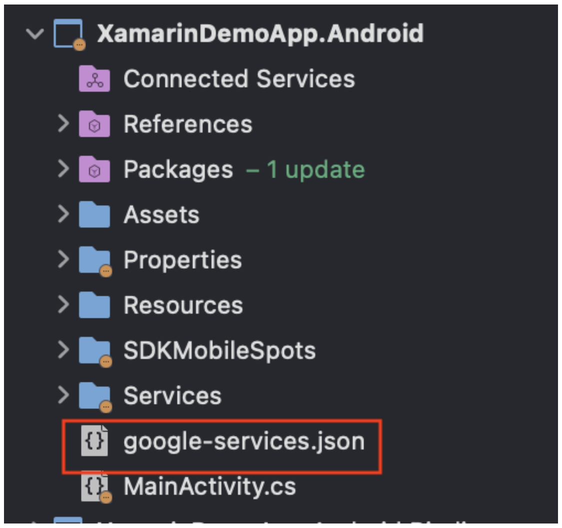
11. Add this code between `<TabBar></TabBar>` in AppShell.xaml, which is in XamarinDemoApp:

```
<ShellContent Title="Msg" Icon="message.png"
  Route="MobileMessagesPage"
  ContentTemplate="{DataTemplate local:MobileMessagesPage}" />
```

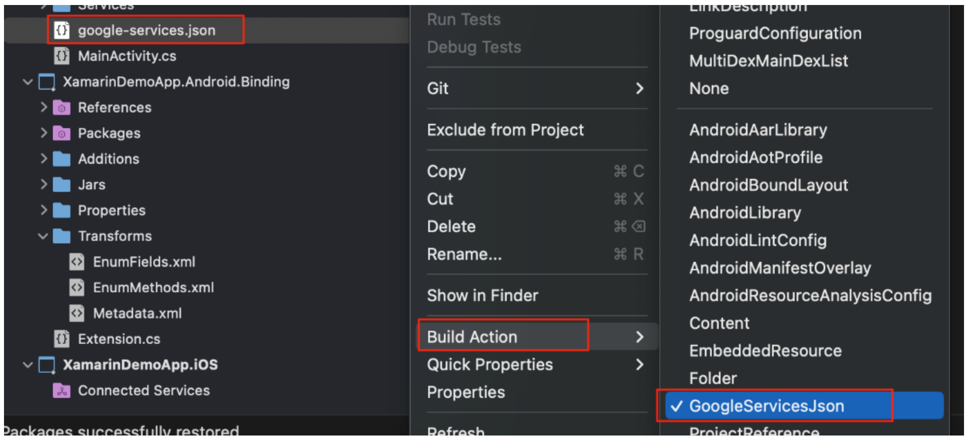
Note: You need to have message.png file in `XamarinDemoApp.Android/Resources/drawable` and in `XamarinDemoApp.iOS/Resources`.

Configure Android

1. In the Firebase console, create a project and add the Xamarin app's Android package name to the project. You can find the package name in the AndroidManifest.xml file.
2. Obtain the google-services.json file from the Firebase console, and then put it in the Xamarin project under **XamarinDemoApp.Android**.



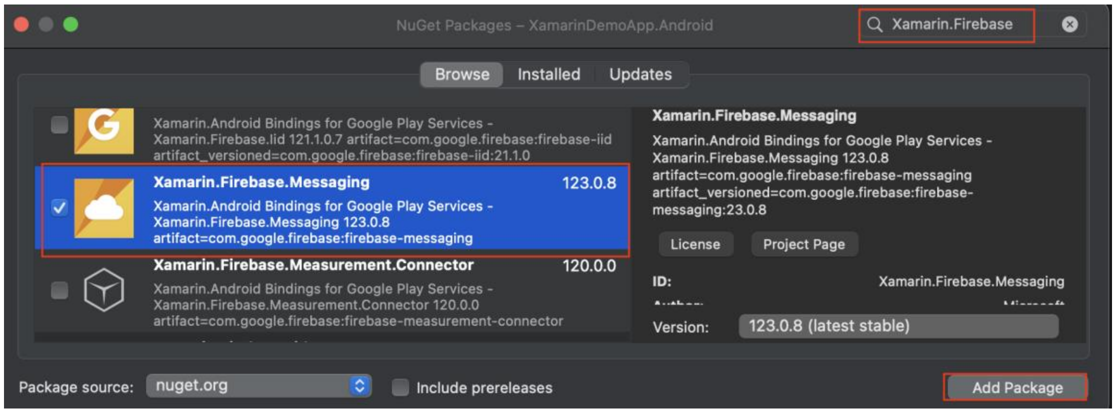
3. Right-click **google-services.json**, select **Build Action**, and select **GoogleServicesJson**.



4. 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*.

5. Right-click **XamarinDemoApp.Android** and select **Manage NuGet Packages**. In the NuGet Packages window, search for **`Xamarin.Firebase.Messaging`**, select it from the results list, and then click **Add Package**.


NuGet Packages – XamarinDemoApp.Android

Xamarin.Firebase

Browse | Installed | Updates

Xamarin.Android Bindings for Google Play Services - Xamarin.Firebase.Iid 121.1.0.7 artifact=com.google.firebase:firebase-iid artifact_versioned=com.google.firebase:firebase-iid:21.1.0

Xamarin.Firebase.Messaging 123.0.8
Xamarin.Android Bindings for Google Play Services - Xamarin.Firebase.Messaging 123.0.8 artifact=com.google.firebase:firebase-messaging

Xamarin.Firebase.Measurement.Connector 120.0.0
Xamarin.Android Bindings for Google Play Services - Xamarin.Firebase.Measurement.Connector 120.0.0 artifact=com.google.firebase:firebase-measurement-connector

Xamarin.Firebase.Messaging
Xamarin.Android Bindings for Google Play Services - Xamarin.Firebase.Messaging 123.0.8 artifact=com.google.firebase:firebase-messaging artifact_versioned=com.google.firebase:firebase-messaging:23.0.8

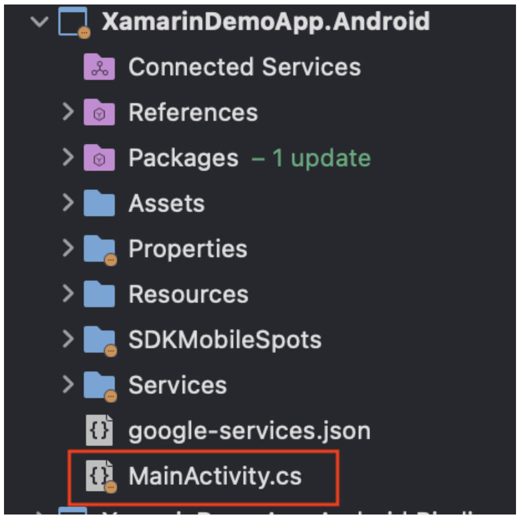
ID: Xamarin.Firebase.Messaging

Version: 123.0.8 (latest stable)

Package source: nuget.org

Include prereleases

6. Find MainActivity.cs under **XamarinDemoApp.Android**.



7. Add these `using` directives at the start of MainActivity.cs:

```
using Android.Gms.Tasks;
using Android.Content;
using Firebase;
using Firebase.Messaging;
using Com.Sas.Mkt.Mobile.Sdk.Iam;
using XamarinDemoApp.Services;
```

Note: If you see errors, copy the `using` directives from `XamarinDemoApp.Android/MainActivity.cs` in the mobile_sdk_xamarin project example.

8. In the OnCreate method in MainActivity.cs, add the following code:

```
FirebaseApp.InitializeApp(this);
FirebaseMessaging.Instance?.GetToken()
    .AddOnSuccessListener(this,
        new FirebaseOnSuccessListener())
    .AddOnFailureListener(this,
        new FirebaseOnFailureListener());

SetPushChannel();
SetMobileMessageDelegate();


String notificationLink =
  Intent.GetStringExtra("notificationLink");

MobileMessageHandler.OnMsgLinkClicked(notificationLink,
  true);
```

Note: There are a few methods and classes that are shown in the code above that have not been defined yet. They will be defined in the next steps.

Note: The assignment and the method call at the end of the above code take care of handling push notifications that start the application. This happens when there are push notifications that appear on the device, but the application has not started yet. Clicking the push notifications takes the user to the diagnostics page.

9. If you use CI360 Android SDK 1.80.2 add this method in MainActivity class:

```
protected override void OnNewIntent(Intent intent)
{
   base.OnNewIntent(intent);

   String notificationLink =
     intent.GetStringExtra("notificationLink");

   MobileMessageHandler.OnMsgLinkClicked(notificationLink,
     true);
}
```

10. If you use CI360 Android SDK 1.80.3, skip step 9 and add this method in MainActivity class:

```
protected override void OnNewIntent(Intent intent)
{
   base.OnNewIntent(intent);

  Bundle bundle = intent.Extras;
  this.Intent.PutExtras(bundle);

   String notificationLink =
     intent.GetStringExtra("notificationLink");

   MobileMessageHandler.OnMsgLinkClicked(notificationLink,
     true);
}
```

11. If you use CI360 Android SDK 1.80.3, you also need to add this entryy in your SASCollector.properties file:

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

12. At the end of the content in MainActivity.cs, but still inside XamarinDemoApp.Droid namespace, add these classes:

```
public class FirebaseOnSuccessListener : Java.Lang.Object,
IOnSuccessListener
{
  public void OnSuccess(Java.Lang.Object result)
    {
        string token = result.ToString();
        SASCollector.Instance
            .RegisterForMobileMessages(token);

     }
}

public class FirebaseOnFailureListener : Java.Lang.Object,
IOnFailureListener
{
    public void OnFailure(Java.Lang.Exception e)
    {
        Console.WriteLine(e.LocalizedMessage);
    }
}

public class MobileMessageDelegate2 : Java.Lang.Object,
ISASMobileMessagingDelegate2
{
     private Context context;

     public MobileMessageDelegate2(Context context)
     {
          this.context = context;
     }

     public void Action(string link,
         SASMobileMessagingDelegate2SASMobileMessageType
           msgType)
     {
         MobileMessageHandler.OnMsgLinkClicked(link, false);
     }

     public void Dismissed()
     {
         MobileMessageHandler.OnMsgDismissed();
     }

     public Intent GetNotificationIntent(string s)
     {
          Intent intent = new Intent(context as Activity,
              typeof(MainActivity));
          intent.PutExtra("notificationLink", s);
          return intent;
     }
}
```

13. In MainActivity class, add these methods:

```
private void SetPushChannel()
{
    string channelId = "XamarinPushChannel1";
    string channelName = "Xamarin Push Channel1";
    NotificationManager notificationManager =
        (NotificationManager)GetSystemService(
             NotificationService);
    NotificationChannel channel = new NotificationChannel(
        channelId, channelName,
        NotificationImportance.High);
    channel.EnableLights(true);
    channel.EnableVibration(true);
    channel.SetShowBadge(true);
    notificationManager.CreateNotificationChannel(channel);

    SASCollector.Instance.SetPushNotificationChannelId(
        channelId);

    SASCollector.Instance.SetMobileMessagingIcon(
        Resource.Drawable.spotlight);
    }

    private void SetMobileMessageDelegate()
    {
         SASCollector.Instance.MobileMessagingDelegate2 =
             new MobileMessageDelegate2(this);
    }
```

Note: You can change the channel name and ID to whatever you like.

14. Under **XamarminDemoApp.Android** => **Services**, create a XamarinFirebaseMessagingService.cs file. Replace its content with this code:

```
using System;
using Android.App;
using Firebase.Messaging;
using Com.Sas.Mkt.Mobile.Sdk;
```

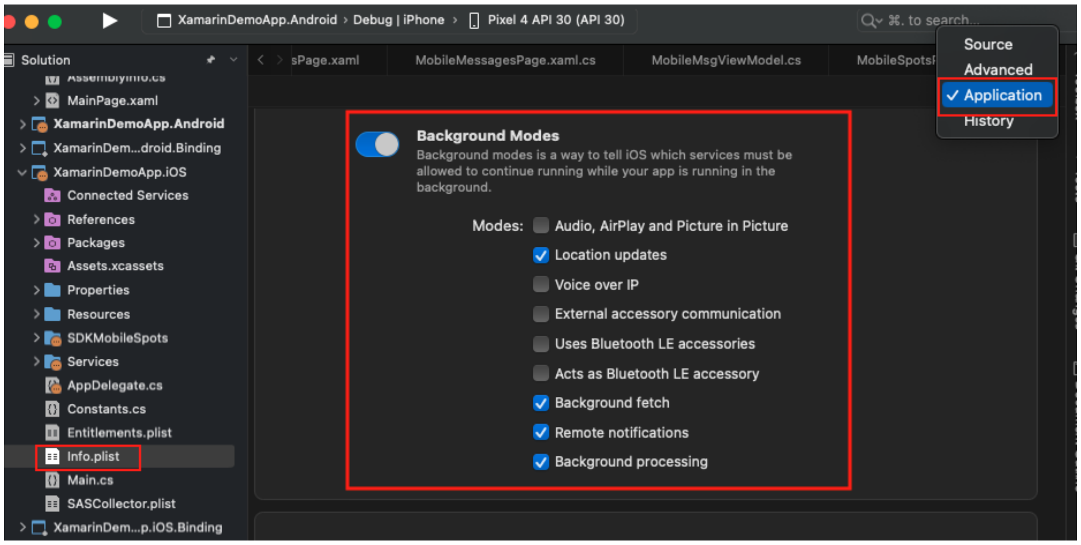
```
namespace XamarinDemoApp.Droid.Services
{
   [Service (Exported = false)]
   [IntentFilter(new[]
{"com.google.firebase.MESSAGING_EVENT"})]
   public class XamarinFirebaseMessagingService :
FirebaseMessagingService
   {
        public override void OnMessageReceived(
            RemoteMessage message)
        {
            base.OnMessageReceived(message);
              if (!SASCollector.Instance
                    .HandleMobileMessage(message.Data))
              {
                   // Handle non-SASCollector message
              }
        }

        public override void OnNewToken(string token)
        {
            base.OnNewToken(token);
            if (token != null)
            {
                SASCollector.Instance
                    .RegisterForMobileMessages(token);
            }
        }
    }
}
```

Note: This class does not need to be registered in AndroidManifest.xml. The custom attributes (square brackets and the content inside) that are added before the start of the class definition ensure it is added to AndroidManifest.xml during compilation.

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. Under **XamarinDemoApp.iOS**, find Info.plist and double-click to open it.
4. Info.plist might open as **Source** code. Make sure to change it to **Application**. Scroll down to find **Background Modes**. Select these modes:
   - **Location updates**

- **Background fetch**
- **Remote notifications**
- **Background processing**



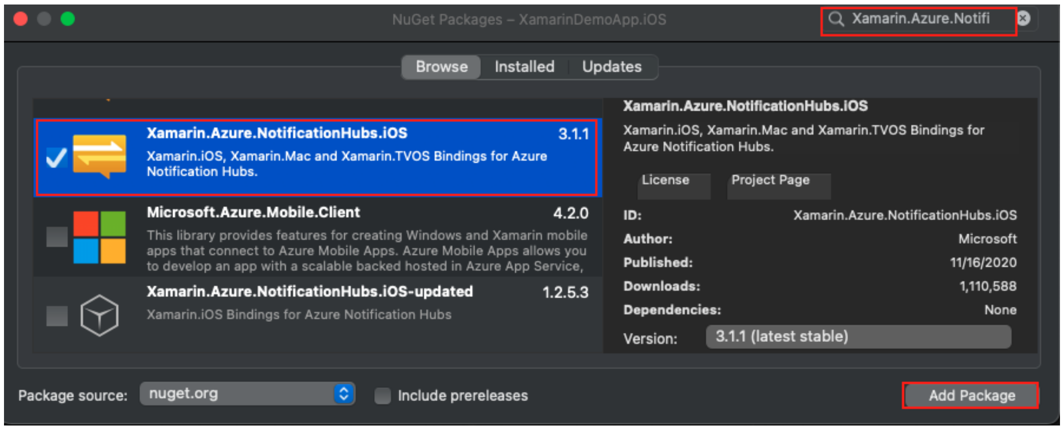
Background Modes
Background modes is a way to tell iOS which services must be allowed to continue running while your app is running in the background.

Modes: ☐ Audio, AirPlay and Picture in Picture
☑ Location updates
☐ Voice over IP
☐ External accessory communication
☐ Uses Bluetooth LE accessories
☐ Acts as Bluetooth LE accessory
☑ Background fetch
☑ Remote notifications
☑ Background processing

Note: You might need to set your Apple account and signing provision while in Info.plist.

---

5. Right-click **XamarinDemoApp.iOS** and select **Manage NuGet Packages**.
6. In the NuGet Packages window, search for **`Xamarin.Azure.NotificationHubs`**, select **Xamarin.Azure.NotificationHubs.iOS** from the results list, and then click **Add Package**.



Xamarin.Azure.NotificationHubs.iOS 3.1.1
Xamarin.iOS, Xamarin.Mac and Xamarin.TVOS Bindings for Azure Notification Hubs.

Microsoft.Azure.Mobile.Client 4.2.0
This library provides features for creating Windows and Xamarin mobile apps that connect to Azure Mobile Apps. Azure Mobile Apps allows you to develop an app with a scalable backed hosted in Azure App Service,

Xamarin.Azure.NotificationHubs.iOS-updated 1.2.5.3
Xamarin.iOS Bindings for Azure Notification Hubs

Xamarin.Azure.NotificationHubs.iOS
Xamarin.iOS, Xamarin.Mac and Xamarin.TVOS Bindings for Azure Notification Hubs.

ID:	Xamarin.Azure.NotificationHubs.iOS
Author:	Microsoft
Published:	11/16/2020
Downloads:	1,110,588
Dependencies:	None
Version:	3.1.1 (latest stable)

7. Create an administrator account or log in to your existing administrator account in the Azure portal.
8. In the Azure portal, create a Notification Hub resource, and configure it with APNS. Get the notification hub name and connection string from Azure portal.

9. Create a Constants.cs file in XamarinDemoApp.iOS and add the hub name and connection string in the file:

```
using System;
namespace XamarinDemoApp.iOS
{
   public class Constants
   {
        // Azure app-specific connection string and hub path
        public const string ConnectionString =
            "your_conn_str";
        public const string NotificationHubName =
            "your_hub_name";
    }
}
```

Note: Replace *ConnectionString* and *NotificationHubName* with your app's connection string and hub name.

---

10. Under **XamarinDemoApp.iOS**, double-click **AppDelegate.cs** to open it.
11. In AppDelegate.cs, add these `using` directives at the start of the file:

```
using UserNotifications;
using WindowsAzure.Messaging;
using System.Diagnostics;
using XamarinDemoApp.Services;
```

12. Add this class to AppDelegate.cs, after the AppDelegate class definition, but inside XamarinDemoApp.iOS namespace:

```
public class XamarinMobileMessagingDelegate :
SASMobileMessagingDelegate2
{
    public override void ActionWithLink(string link,
        SASMobileMessageType type)
    {
         Boolean isPushNotification =
             type == SASMobileMessageType.PushNotification;
         MobileMessageHandler.OnMsgLinkClicked(link,
             isPushNotification);
    }
    public override void MessageDismissed()
    {
        MobileMessageHandler.OnMsgDismissed();
    }
}
```

13. Add this code at the start of the AppDelegate class definition, before the FinishedLaunching method:

```
private SBNotificationHub Hub { get; set; }
private XamarinMobileMessagingDelegate messagingDelegate;
```

14. To the AppDelegate class definition, add this override method to register for remote notifications:

```
public override void
RegisteredForRemoteNotifications(UIApplication application,
NSData deviceToken)
{
     Hub = new SBNotificationHub(Constants.ConnectionString,
         Constants.NotificationHubName);
     Hub.UnregisterAll(deviceToken, (error) => {
         if (error != null)
     {
         Debug.WriteLine("Error calling Unregister: {0}",
            error.ToString());
         return;     }
     NSSet tags = null; // create tags if you want
        Hub.RegisterNative(deviceToken, tags, (errorCallback)
=>{
             If (errorCallback !=null)
                 Debug.WriteLine(
        "RegisterNative error: " + errorCallback.ToString());
        });
     });

     SASCollector.RegisterForMobileMessages(deviceToken,
         () => {
                Debug.WriteLine("Registration successful");
                },
          () => {
                Debug.WriteLine("Registration failed");
               }
     );
  }
```

15. Add these two methods to the AppDelegate class definition for handling when mobile in-app messages and push notifications are received, respectively:

```
public override void
DidReceiveRemoteNotification(UIApplication application,
NSDictionary userInfo, Action<UIBackgroundFetchResult>
completionHandler)
{
     if (!SASCollector.HandleMobileMessage(userInfo,
         UIApplication.SharedApplication))
     {
         Console.WriteLine(
             "Handle non-SASCollector message");
     }
     completionHandler(UIBackgroundFetchResult.NoData);
}


public override void ReceivedRemoteNotification(UIApplication
application, NSDictionary userInfo)
{
    if (!SASCollector.HandleMobileMessage(
        userInfo, application))
    {
        Console.WriteLine(
            "Handle non-SASCollector message");
    }
}
```

16. Call RegisterForRemoteNotifications() in the FinishLaunching method in the AppDelegate class definition:

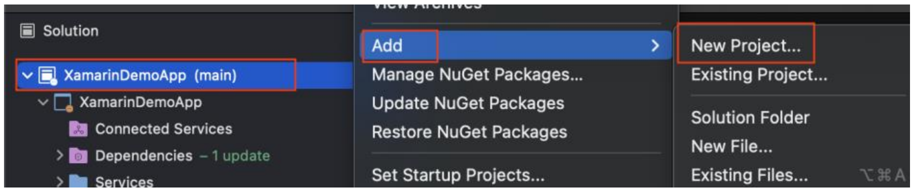
```
private void RegisterForRemoteNotifications()
```

```
{
    if (UIDevice.CurrentDevice.CheckSystemVersion(10, 0))
    {
      UNUserNotificationCenter.Current.RequestAuthorization(
          UNAuthorizationOptions.Alert |
          UNAuthorizationOptions.Badge |
          UNAuthorizationOptions.Sound,
          (granted, error) =>
          {
              if (granted)
                  InvokeOnMainThread(
                      UIApplication.SharedApplication
                        .RegisterForRemoteNotifications);
       });
     }
     else if (UIDevice.CurrentDevice
         .CheckSystemVersion(8, 0))
     {
        var pushSettings =
            UIUserNotificationSettings.GetSettingsForTypes(
                UIUserNotificationType.Alert |
                UIUserNotificationType.Badge |
                UIUserNotificationType.Sound,
                new NSSet());

         UIApplication.SharedApplication
             .RegisterUserNotificationSettings(
                 pushSettings);

         UIApplication.SharedApplication
             .RegisterForRemoteNotifications();
     }
     else
     {
          UIRemoteNotificationType notificationTypes =
              UIRemoteNotificationType.Alert |
              UIRemoteNotificationType.Badge |
              UIRemoteNotificationType.Sound;

          UIApplication.SharedApplication
              .RegisterForRemoteNotificationTypes(
                   notificationTypes);
      }
}
```

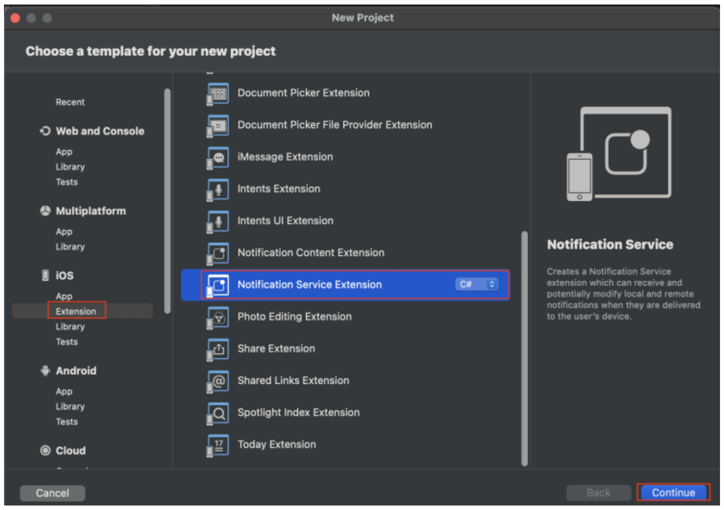
17. In the FinishedLaunching method, add this code before the return statement:

```
RegisterForRemoteNotifications();
messagingDelegate = new XamarinMobileMessagingDelegate();
SASCollector.SetMobileMessagingDelegate2(messagingDelegate);
```

18. To enable rich push notifications, right-click **XamarinDemoApp** (the solution folder), select **Add,** and then select **New Project**.



19. In the New Project window, in the left-hand pane, under **iOS**, select **Extension**, in the center pane, select **Notification Service Extension**, and then click **Continue**.


New Project
Choose a template for your new project

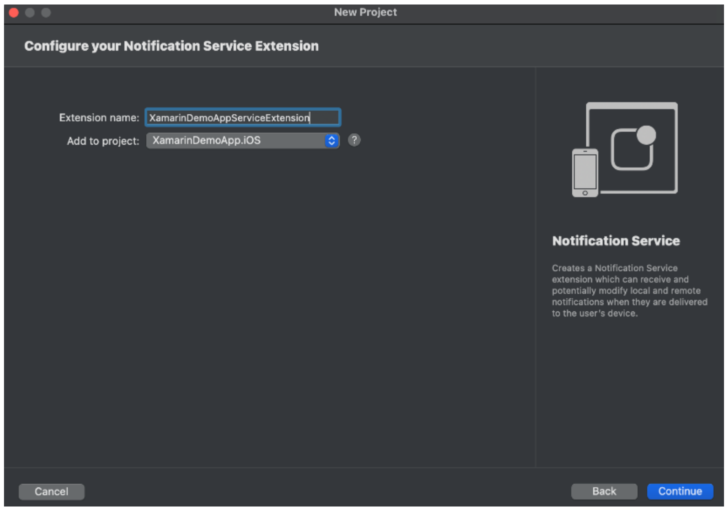
Recent
Web and Console: App, Library, Tests
Multiplatform: App, Library
iOS: App, Extension, Library, Tests
Android: App, Library, Tests
Cloud

Document Picker Extension
Document Picker File Provider Extension
iMessage Extension
Intents Extension
Intents UI Extension
Notification Content Extension
Notification Service Extension C#
Photo Editing Extension
Share Extension
Shared Links Extension
Spotlight Index Extension
Today Extension

Notification Service
Creates a Notification Service extension which can receive and potentially modify local and remote notifications when they are delivered to the user's device.

Cancel Back Continue

20. In the configuration window, enter a project name, and select **Continue** to finish creating the extension project.


New Project

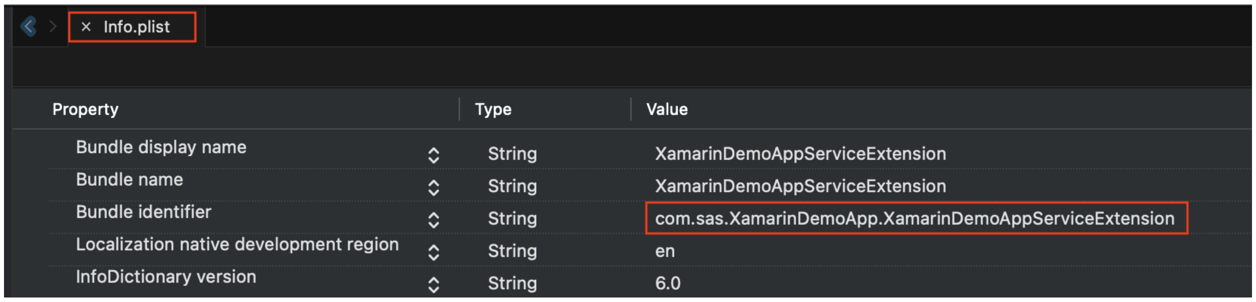
Configure your Notification Service Extension

Extension name: XamarinDemoAppServiceExtension

Add to project: XamarinDemoApp.iOS

Notification Service
Creates a Notification Service extension which can receive and potentially modify local and remote notifications when they are delivered to the user's device.
Cancel
Back
Continue

21. Log in to your Apple developer account. Using the bundle identifier that was created for the extension project, create an identifier for the push notification extension project. Then, create a provisioning profile for that extension. You can find the bundle identifier in Info.plist.


Info.plist

Property	Type	Value
Bundle display name	String	XamarinDemoAppServiceExtension
Bundle name	String	XamarinDemoAppServiceExtension
Bundle identifier	String	com.sas.XamarinDemoApp.XamarinDemoAppServiceExtension
Localization native development region	String	en
InfoDictionary version	String	6.0

22. While you are still logged in to your Apple developer account, create a push service certificate for the Xamarin project. Install the certificate in your Keychain Access application on your Mac.

23. Find NotificationService.cs in XamarinDemoAppServiceExtension, and then replace the DidReceiveNotificationRequest method with the one in `XamarinDemoAppServiceExtension/NotificationService.cs` in the mobile_sdk_xamarin project example.

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.

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.