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

SAS® Documentation
March 25, 2024

Contents

Overview

SAS Customer Intelligence 360 mobile SDK (also called *SASCollector*) enables 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 React Native, an open-source software development kit, to design a hybrid mobile application that uses only one codebase for both Android and iOS. The programming languages that are used to develop a mobile app with React Native are JavaScript or Typescript. In this document, Typescript is used.

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 React Native technology. This document shows how to create a project that adds the capabilities provided by SAS Customer Intelligence mobile SDKs.

In addition, there is a Mobile SDK React Native Package (.zip) that contains a sample React Native project (mobile-sdk-react-native).

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 React Native mobile apps and the programming languages that are used to design them.
- You understand the roles and responsibilities of the individuals who work with the 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 iOS and Working with Events for Android 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 SDK) to include location services and monitor location events. For more information, see Upload Location Data in *SAS Customer*

Intelligence 360: Administration 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*.

- **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:

- node.js. Go to nodejs.org and download the LTS version. The version used in this project is 16.15.1.

  Once node.js is installed, npm (Node Package Manager) is also installed. To install react-native-cli and react-native, open a terminal session, and type:

  `npm install -g react-native-cli` and `npm install -g react-native`.

- Android Studio and Xcode. Android Studio Chipmunk 2021.2.1 and Xcode 13.4.1.
- Visual Studio Code (VSCode) is used for most of the development work. Go to this link to download and install VSCode: https://code.visualstudio.com/download. In this cookbook, VSCode version 1.70.2 is used.
- Two extensions are needed in VSCode: Extension Pack for Java (v0.25.7) and Gradle for Java (v3.12.6). Both are from Microsoft.

Create a React Native Library Project

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

The React Native library works by using React's Bridge library that contains functions for converting native code to Javascript or Typescript, such as nativeModule, and React's Uimanager library for making the native views (such as inline ad view and interstitial ad view) that are in the SAS Customer Intelligence 360 SDKs available in React Native.

Create the Project

1. Open a terminal session and navigate to the desired location for this project.

2. Use the command shown in the example below to create the project (in this example, the project is called mobile-sdk-react-native):

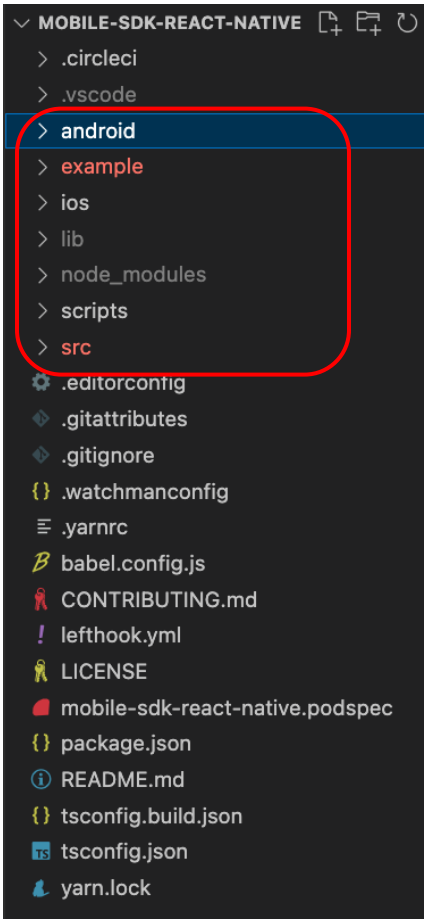
```
npx create-react-native-library mobile-sdk-react-native
```

You are asked several questions. The following figure shows an example:

Note: The URLs must be valid.

3. After the project is created, open it in VSCode. In VSCode, open an integrated terminal and type `npm install` to install the required node libraries, as shown in package.json.

In the project, these are the four folders that are used most often to build the React Native library: `android`, `ios`, `src`, and `example`. The project structure is shown in the figure below.



Here is a description of the folders:

- The `android` and `ios` folders contain code that exposes native functionality to the rest of the React Native app.
- The `example` folder contains a starter React Native app, sometimes referred to as the *example project*. It can be used for testing the React Native project.
- The `src` folder is where the native functionality for iOS and Android is translated into Typescript for a React Native app to use.

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*.

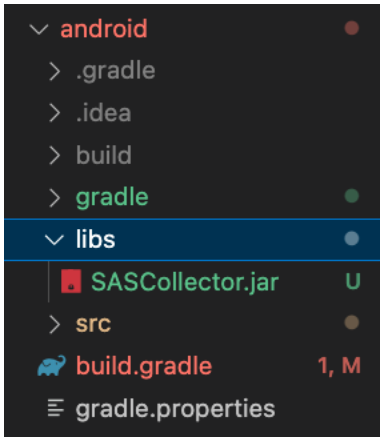
Note: A SASCollector.plist file (for iOS) and a SASCollector.properties file (for Android) contain necessary information to successfully implement the mobile SDKs, including the customer's selected tenant and mobile app ID. The files are not included in the public repository. The files must be obtained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

Add SAS Customer Intelligence 360 Mobile SDK Libraries

You need to add the SASCollector framework (library) to the React Native project that you created.

Android

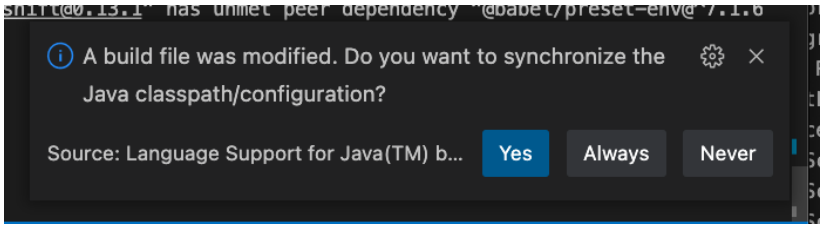
1. In the React Native project (mobile-sdk-react-native), under the `android` folder, create a folder called `libs`.
2. Navigate to the folder that contains the SAS Customer Intelligence 360 mobile SDK ZIP (SASCollector_<applicationID>.zip). Unzip SASCollector_<applicationID> and find SASCollector.jar inside the `android` folder. Copy SASCollector.jar into this folder:



3. In build.gradle (in the `android` folder shown in the previous figure), add the following JAR file dependency in the `dependencies` section:

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

4. Save build.gradle. In the pop-up that appears, choose YES to synchronize, as shown below.



5. Verify that the build is successful by navigating to the `example` folder in an integrated terminal and then typing `npm run android`.

iOS

1. Navigate to the folder that contains the SAS Customer Intelligence 360 mobile SDKs ZIP file (SASCollector_<applicationID>.zip).
2. Open the `iOS` folder and double-click the SASCollector.zip file to unzip it to xcframework named `SASCollector.xcframework`.
3. In the `ios` folder of the React Native project, add SASCollector.xcframework:

4. Navigate to mobile-sdk-react-native.podspec in the top project level and add this code after `s.dependency "React-Core"`:

```
s.vendored_frameworks = 'ios/SASCollector.xcframework'
```

5. In an integrated terminal, type `cd ios` and then `pod install && cd ..`
6. Verify that the build is successful by navigating to the `example` folder in an integrated terminal and then typing: `npm run ios`.

   If the build fails, you can open the example project in Xcode and run it from there to find out what caused the failure. Opening native code in Xcode can help to activate certain functionality. The error information can also be more meaningful in Xcode.

Basic Functionality

Some mobile app events, such as focus and defocus, do not need an explicit API call in the React Native project to make them work. The integration of SAS Customer

Intelligence mobile SDKs and the React Native app is sufficient.

Other basic functions, such as custom events, page loads, and identity, need to be converted to React Native functions to be used by a React Native 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 React Native 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 React Native App" at the end of this section. The "Configure Android", "Configure iOS", and "An implementation example that includes the exposed methods is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `ios/MobileSdkReactNative.mm` to find them.

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

Configure " sections describe how to use React Native's nativeModules function to expose methods from the native side, how to use React Native's requireNativeComponent, UIManager, and so on to create React Native components from the native side, and how to use event, event emitter and event listener to communicate between the native side and React Native side.

Configure Android

1. In the React Native project, navigate to the `android` folder. In the `android` folder, navigate to `src/main /java/com/mobilesdkreactnative` and find MobileSdkReactNativeModule.java.
2. In MobileSdkReactNativeModule.java:
   a. Replace the constructor with this code:

```
public MobileSdkReactNativeModule(
```

```
  ReactApplicationContext reactContext) {
  super(reactContext);
  if(!SASCollector.getInstance().isInitialized()){
    SASCollector.getInstance()
    .initialize(reactContext.getApplicationContext());
  }
}
```

b. Add these methods:

```
@ReactMethod
public void newPage(String uri) {
  SASCollector.getInstance().newPage(uri);
}

@ReactMethod
public void addAppEvent(

if (data == null) {
  SASCollector.getInstance().addAppEvent(eventKey, null);
  return;
}
  String eventKey, ReadableMap data) {
  HashMap<String, Object> rawData = data.toHashMap();
  HashMap<String, String> convertedData
    = new HashMap<>();
  for (Map.Entry<String, Object>
    entry : rawData.entrySet()) {
    if(entry.getValue() instanceof String) {
     convertedData.put(
       entry.getKey(), (String)entry.getValue());
    }
  }
  SASCollector.getInstance()
    .addAppEvent(eventKey, convertedData);
}

@ReactMethod
public void identity(
  String value, String type, Promise promise) {
  SASCollector.getInstance()
    .identity(value, type,
      new SASCollector.IdentityCallback() {
       @Override
       public void onComplete(boolean b) {
       Log.d("Identity", "Identity called with: "
        + (b ? "success" : "failure"));
        new Handler(Looper.getMainLooper())
          .post(new Runnable() {
            @Override
            public void run() {
```

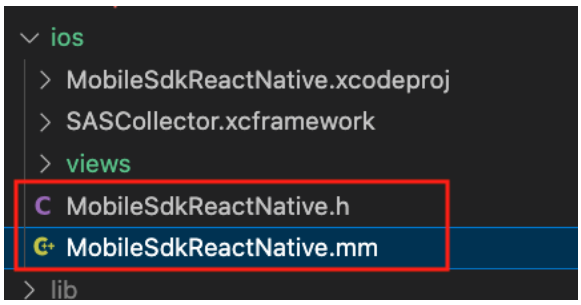
```
            promise.resolve(b);
          }
        });
      }
  });
}

@ReactMethod
public void detachIdentity(Promise promise) {
  SASCollector.getInstance()
    .detachIdentity(new SASCollector
     .DetachIdentityCallback() {
     @Override
     public void onComplete(boolean b) {
     new Handler(Looper.getMainLooper())
       .post(new Runnable() {
         @Override
         public void run() {
          promise.resolve(b);
         }
      });
    }
  });
}

@ReactMethod
public void startMonitoringLocation() {
  SASCollector.getInstance().startMonitoringLocation();
}

@ReactMethod
public void disableLocationMonitoring() {
  SASCollector.getInstance().disableLocationMonitoring();
}
```

An implementation example that includes these and other exposed methods that are referred to later in this documentation is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `android/src/main/java/com/mobilesdkreactnative/MobileSdkReactNativeModule.java` to find them.

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

Configure iOS

1. In the React Native project, navigate to the `ios` folder. Find the MobileSdkReactNative.h and MobileSdkReactNative.mm files:



2. In MobileSdkReactNative.h, add this import:

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

3. In MobileSdkReactNative.mm, add these methods:

```
RCT_EXPORT_METHOD(newPage:(NSString*)uri) {
    [SASCollector newPage:uri];
}

RCT_EXPORT_METHOD(addAppEvent: (NSString*)eventKey
  data:(NSDictionary*)data){
    [SASCollector addAppEvent:eventKey data:data];
}

RCT_EXPORT_METHOD(identity:(NSString*)value
  withType:(NSString*)type
  isSuccess:(RCTPromiseResolveBlock)successPromise
  isFailure:(RCTPromiseRejectBlock)failurePromise) {

  [SASCollector identity:value
    withType:type completion:^(BOOL success) {
    dispatch_async(dispatch_get_main_queue(), ^{
      if (success) {
      successPromise([NSNumber numberWithBool:success]);
      } else {
        failurePromise(@"Error", @"Identity failure",
         nil);
      }
    });
  }];
}

RCT_EXPORT_METHOD(detachIdentity:
  (RCTPromiseResolveBlock) successPromise
  isFailure:(RCTPromiseRejectBlock)failurePromise) {
  [SASCollector detachIdentity:^(BOOL success) {
    dispatch_async(dispatch_get_main_queue(), ^{
      if (success){
        successPromise([NSNumber numberWithBool:success]);
```

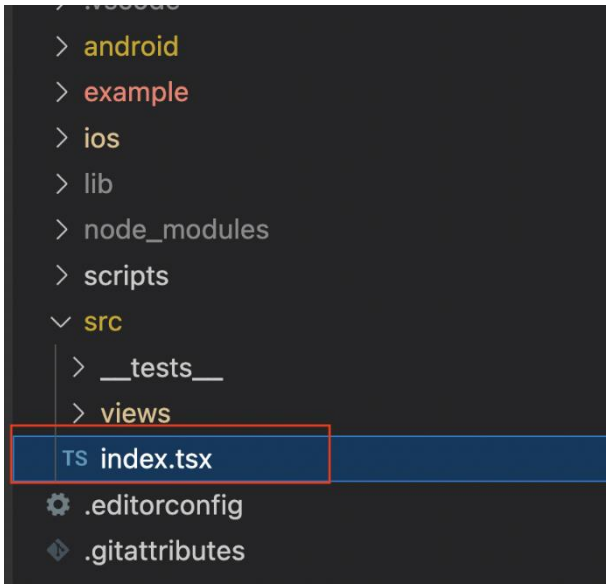
```
      } else {
        failurePromise(@"Error",
          @"Identity detach failure", nil);
      }
     });
  }];
}

RCT_EXPORT_METHOD(startMonitoringLocation) {
  [SASCollector startMonitoringLocation];
}

RCT_EXPORT_METHOD(disableLocationMonitoring) {
  [SASCollector disableLocationMonitoring];
}
```

An implementation example that includes the exposed methods is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `ios/MobileSdkReactNative.mm` to find them.

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

Configure React Native (Typescript)

1. In the `src` folder, find index.tsx:



2. In index.tsx, add these methods:

```
export function newPage(uri: string) {
```

```
  MobileSdkReactNative.newPage(uri);
}
export function addAppEvent(eventKey: string, data: Object) {
  MobileSdkReactNative.addAppEvent(eventKey, data);
}
export async function identity(value: string, type: string) {
  try {
    const isSuccess: boolean =
      await MobileSdkReactNative.identity(value, type);
    return isSuccess;
  } catch (e: any) {
    console.log(e);
    return false;
  }
}
export async function detachIdentity() {
  try {
    const isSuccess =
      await MobileSdkReactNative.detachIdentity();
    return isSuccess;
  } catch (e: any) {
    Console.log(e);
    return false;
  }
}
export function startMonitoringLocation() {
  MobileSdkReactNative.startMonitoringLocation();
}
export function disableLocationMonitoring() {
  MobileSdkReactNative.disableLocationMonitoring();
}
```

An implementation example that includes these and other exported methods that are referred to later in this documentation is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `src/index.tsx` to find them.

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

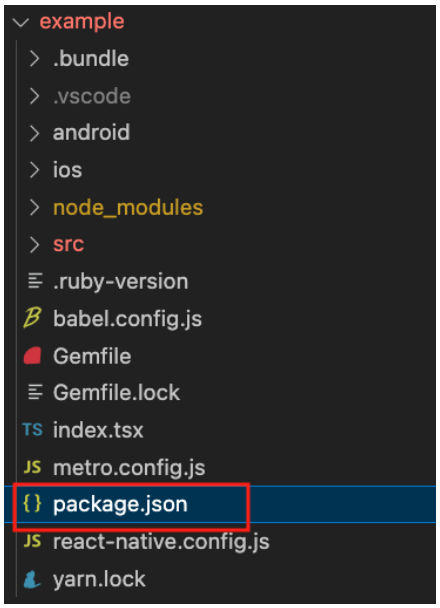
Configure the Example React Native App

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

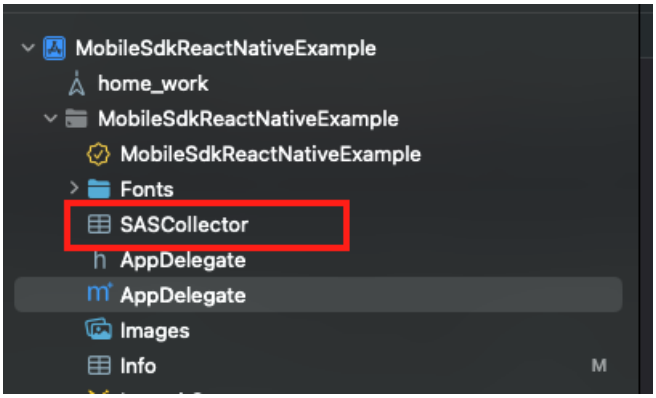
1. In the `example` folder, add a few dependencies for navigation. Follow this React Native documentation link to install these dependencies: React Navigation.

Note: The latest version of `react-native-safe-area-context` generates errors in iOS. Use version ^3.4.1 instead.

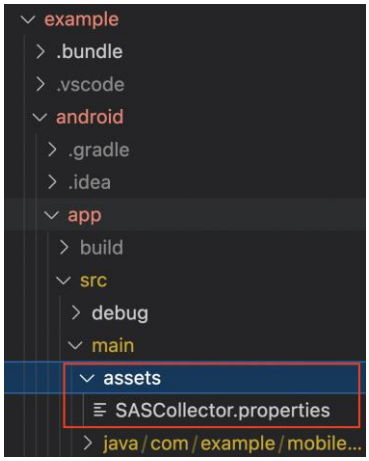
Refer to the `example` folder in package.json to see the included packages.



2. Add SASCollector.plist to iOS:
   a. In Xcode, navigate to the `ios` folder and find NativeSdkReactNativeExample.xcworkspace.
   b. Find the SASCollector.plist file. The file is included in the mobile SDK ZIP file for SAS Customer Intelligence 360 (SASCollector_<applicationID>.zip) in the `ios` folder.
   c. Drag SASCollector.plist into MobileSdkReactNativeExample target as shown below.



3. Add SASCollector.properties to Android:
    a. In VSCode, navigate to `app`/`src`/ `main` and create an `assets` folder.
    b. Find the SASCollector.properties file. (The file is in the mobile SDK ZIP file for SAS Customer Intelligence 360 (SASCollector_<applicationID>.zip) in the `android` folder.)
    c. Copy SASCollector.properties and paste it in the `assets` folder.

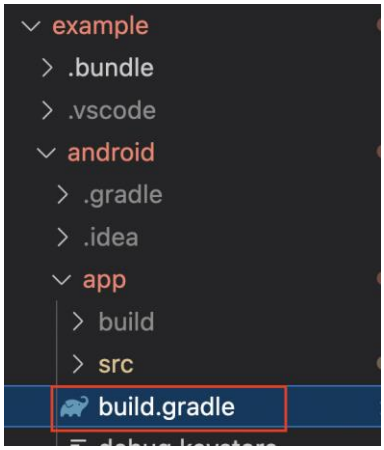


4. The Android SDK initialization occurs after the native Android's MainActivity finishes all its life cycle methods. To avoid issues (such as session not ready), the SDK's initialization should be added in MainApplication.

Note: The SDK only needs to be initialized once, as early as possible. Although SDK initialization can be added to MainActivity, it is not recommended because the result is multiple initializations.

---

    a. Add the location of the SAS Customer Intelligence 360 mobile SDK to the Android app's build.gradle file. Find build.gradle as shown below:



    b. In the dependencies section of the file, add the location of SASCollector.jar:

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

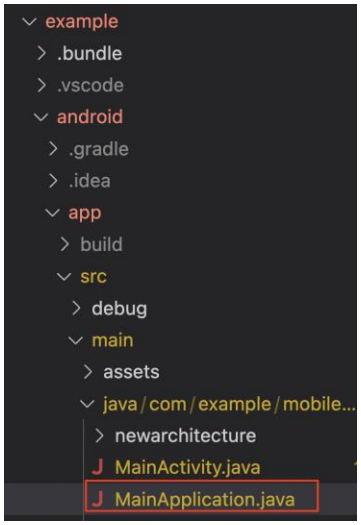
Because the SAS Customer Intelligence 360 mobile SDK's Android initialization needs google services and gson dependencies, add these dependencies:

```
implementation 'com.google.code.gson:gson:2.8.9'
implementation 'com.google.android.gms:play-services-
   location:19.0.1'
```

```
dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])

    //noinspection GradleDynamicVersion
    implementation "com.facebook.react:react-native:+"  // From node_modules

    implementation "androidx.swiperefreshlayout:swiperefreshlayout:1.0.0"
    implementation 'com.google.code.gson:gson:2.8.9'
    implementation 'com.google.android.gms:play-services-location:19.0.1'
    implementation platform('com.google.firebase:firebase-bom:30.3.1')
    implementation 'com.google.firebase:firebase-analytics'
    implementation 'com.google.firebase:firebase-core'
    implementation 'com.google.firebase:firebase-messaging'
    implementation files('../../../android/libs/SASCollector.jar')
    implementation project(':react-native-vector-icons')
```

Since build.gradle is changed, you are asked to sync the project to download the dependencies. Click **Yes**.

c. Find MainApplication.java in the Android project by navigating to `example/android/app/src/main/java/com/example/mobilesdkreactnative/`:

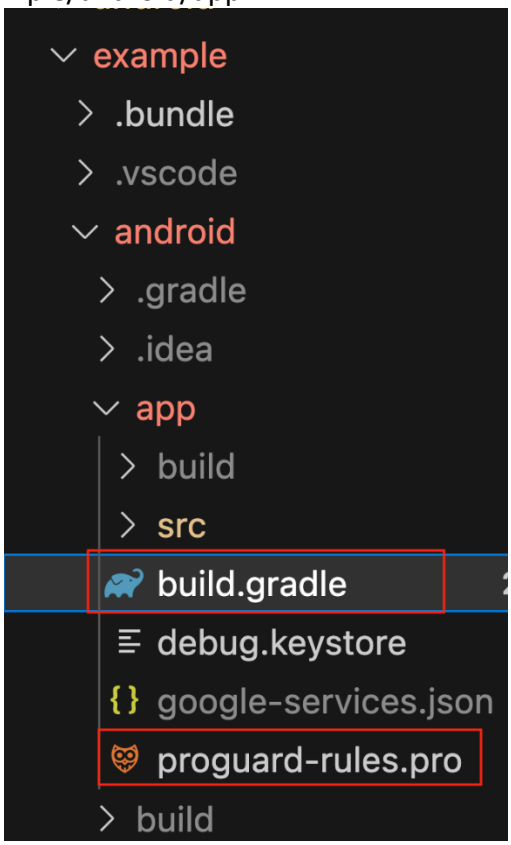


Add this initialization code in the onCreate method:

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

If you want to get all SASCollector's log information, you can also add `Slog.setLevel(Slog.ALL)`.

5. If the project is built for release, and you want to reduce the APK's size, some changes are needed in the build.gradle and proguard-rules.pro files under example/android/app:



a. Find def enableProguardInReleaseBuilds = false in build.gradle, and change it to def enableProguardInReleaseBuilds = true

b. In proguard-rules.pro add this code:

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

This completes the setup of Android for the example project.

6. In `src`, create a `screens` folder and add HomeScreen.tsx to this folder. In HomeScreen.tsx, add this import:

```
import * as MobileSdk  from 'mobile-sdk-react-native';
```

The code above only imports a few exposed functions. If more of these functions are needed, add them in this import inside the curly braces.

The following code shows how to use `addAppEvent`:

```
<CustomButton
        width={styles.buttonWidth}
        title="Add Event"
        onPress={() => addAppEvent(
          eventName,  {[attributeName]: attributeValue})}
/>
```

Note: `CustomButton` is a component that is created in the example project.

Mobile Spot Functionality

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

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

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

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

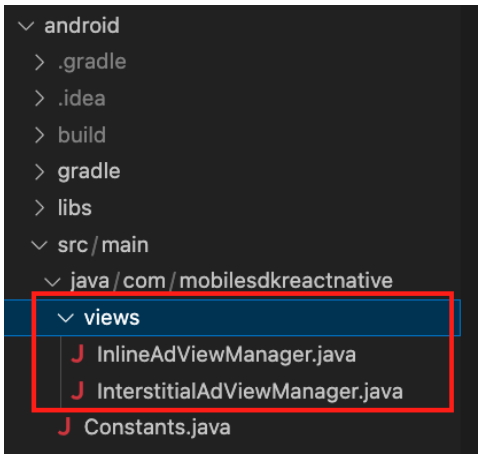
Currently, the implementation of spots in the React Native project requires only the spotID parameter. If other parameters of the spots are needed, developers can follow similar procedures to add them in the project.

This section describes how to implement mobile spot features in the React Native project example to be used in a React Native app. The creation of the React Native spots

functions is described in three sections: “Configure Android, “Configure iOS”, and “Configure React Native”. In Configure React Native, the typescript component classes are created that can be used by a React Native app to display mobile spots. Most of the work that is involved in constructing and presenting spots is in Android and iOS.

Configure Android

1. In the React Native project, navigate to `android/src/main/java/com/mobilesdkreactnative` and create a `views` folder.
2. In the `views` folder, create InlineAdViewManager.java and InterstitialAdViewManager.java:



3. In mobile-sdk-react-native.zip, navigate to `android/src/main/java/com/mobilesdkreactnative/views`. Copy the content of the applicable sample file and add it to the files that you created.

Note: The files use some literal strings that are defined in the Constants.java file outside of the `views` folder (shown in the previous figure). The files also import UseReactContext.java, which is in mobile-sdk-react-native.zip.

4. Unlike iOS, Android requires that the views be explicitly added in the package. To do this, find MobileSdkReactNativePackage.java in the `android/src/main/java/com/mobilesdkreactnative` folder:

5. Add this code in MobileSdkReactNativePackage.java:

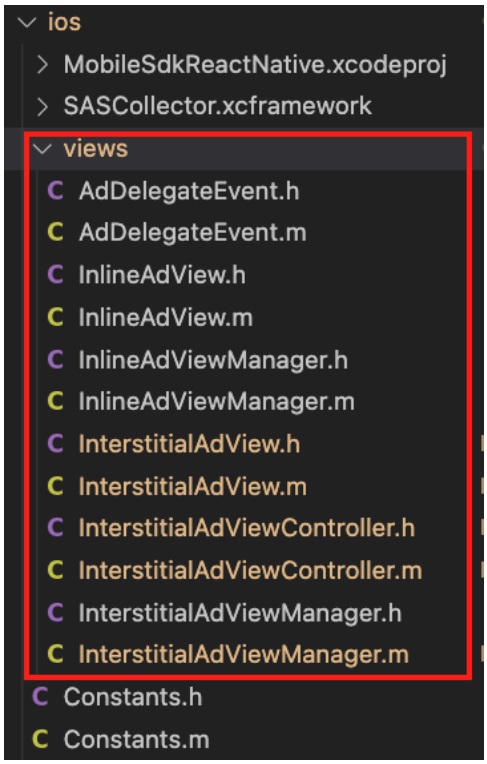
```
@NonNull
@Override
public List<ViewManager> createViewManagers(
  @NonNull ReactApplicationContext reactContext) {

  InlineAdViewManager inlineAdViewManager =
    new InlineAdViewManager(reactContext);

  InterstitialAdViewManager interstitialAdViewManager =
    new InterstitialAdViewManager(reactContext);

  return Arrays.<ViewManager>asList(
    inlineAdViewManager, interstitialAdViewManager);
}
```

Configure iOS

1. Navigate to the `ios` folder and create a `views` folder. In the `views` folder, create a few objective-C files, such as these:

   AdDelegateEvent.h
   AdDelegateEvent.m
   InlineAdView.h
   InlineAdView.m
   InlineAdViewManager.h
   InlineAdViewManager.m
   InterstitialAdView.h
   InterstitialAdView.m
   InterstitialAdViewController.h
   InterstitialAdViewController.m

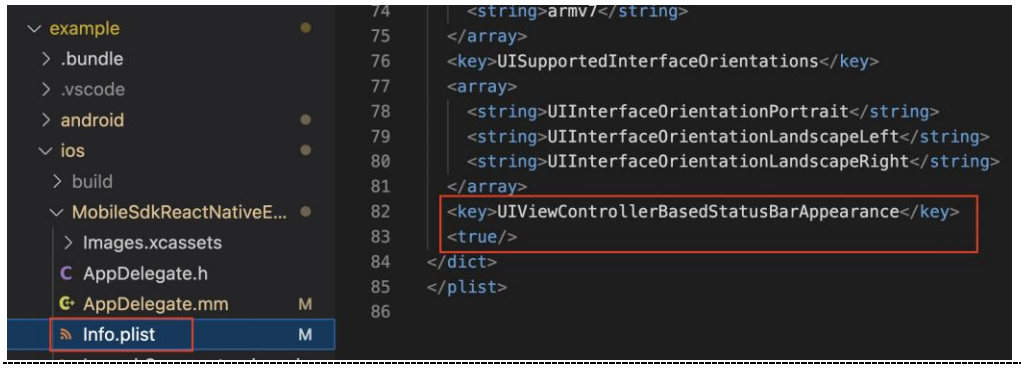
InterstitialAdViewManager.h
InterstitialAdViewManager.m



An implementation example of the classes is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to the `ios/views/` folder to find these class files.

2. After the files are created, in the VSCode integrated terminal, go to the `ios` folder and then run `pod install` for the new files to be logically added to the project.
3. Some of the string constants in the files are extracted in Constants.h and Constants.m, so create these two files, and the run `pod install`.

Note: In iOS, the close button for interstitial ad spots is hidden behind the status bar. Trying to hide the status bar using React Native's `StatusBar.setHidden(true)` or `<StatusBar hidden={true}/>` does not work. The solution is to set `UIViewControllerBasedStatusBarAppearance` to true in Info.plist under `example/ios` as shown below:



Configure React Native (Typescript)

1. In the `src` folder, create a `views` folder.
2. In the `views` folder, create InlineAdView.tsx and InterstitialAdView.tsx.

An implementation example of the ad views is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `src/views` and copy the content of InlineAdView.tsx and InterstitialAdView.tsx to the files that you created.

3. Outside the `views` folder, create Constants.tsx (as shown in the previous figure). The file contains a few string constants that correspond to those in iOS and Android.

   An implementation example of the Constants.tsx is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to the `src` folder and copy the content of Constants.tsx to the file that you created.

4. In index.tsx (shown in the previous figure), add these imports:

```
import InlineAdView from './views/InlineAdView';
import InterstitialAdView from './views/InterstitialAdView';
import * as Constants from './Constants';
```

   At the end of index.tsx, add these exports:

```
export { InlineAdView };
export { InterstitialAdView }
```

```
const AdDelegateEvent = NativeModules.AdDelegateEvent;
export { AdDelegateEvent };
export { Constants };
```

Configure the Example React Native App

1. For the Android version of the app to work, include the mobile SDK's implementation of the ad view activities in the example project's AndroidManifest.xml file. Do this for any React Native app that needs to include interstitial spots.

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

2. In the `example/src` folder, create SpotsScreen.tsx. To organize the different files, screens components can be put in a `screens` folder. SpotsScreen.tsx uses a custom button that is defined in the `components` folder. SpotsScreen.tsx is a functional component.
3. Add this code at the end of the function definition:

```
return (
  <ScrollView>
    <View style={styles.container}>
      <InlineAdView  spotId='your_inline_spot_id'
        style={styles.inlineView} />
      <CustomButton title='Show Interstitial Ad'
        onPress={() => {setShowInterstitial(true)}}
        width={{width: 200}} />
      {showInterstitial && <InterstitialAdView
        spotId='your_interstitial_spot_id'  />}

    </View>
   </ScrollView>
 );
```

As shown in the previous code, the inline spot takes a style that includes width and height. Developers can use a width and a height that ensure the entire image is displayed. Interstitial spot always fills the entire screen, so no style is used.

SpotsScreen.tsx also uses event listener to listen for inline and interstitial spots events and to show the corresponding toast messages. For example, it listens for onAdLoaded and onAdClosed events, and displays toast messages. The original event names are

different in the SAS Customer Intelligence 360 mobile SDKs for Android and iOS, but they are unified for React Native. Developers can replace toast messages with other operations they like to use. On the library's native Android side, `DeviceEventEmitter` is implemented in React Native, and it is used. However, for iOS, `DeviceEventEmitter` is deprecated. For this reason, `AdDelegateEvent` is created in the library's native iOS part to handle the ad's delegate events such as `onLoaded`, `onClosed`, which are SAS Customer Intelligence 360 SDK iOS library's ad event names. `AdDelegateEvent` is shown in the first figure in "Configure iOS**".** `AdDelegateEvent` is exported and can be used in a React Native app. SpotsScreen.tsx makes use of this `AdDelegateEvent`.

An implementation example of how to use the ad views is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `example/src/screens/SpotsScreen.tsx` to find details.

Location Functionality

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

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

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

- Beacon Discovered
- Geofence Entered
- Geofence Exit

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

To enable location features, these actions are required:

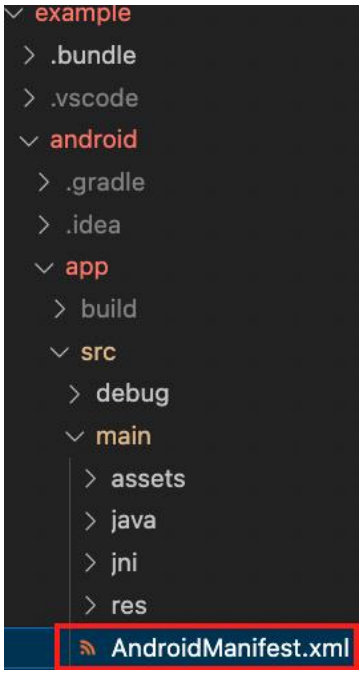
- **Add startMonitoringLocation and disableLocationMonitoring**. For geofences and beacons to work, these two functions are needed from the SDK.

Note: The startMonitoringLocation and disableLocationMonitoring functions were already added in MobileSdkReactNative.mm and MobileSdkReactNativeModule.java on the native side, and in index.tsx when you configured React Native (Typescript) 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*.

Configure Android

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



2. Add location and Bluetooth permissions:

```
<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" />
```

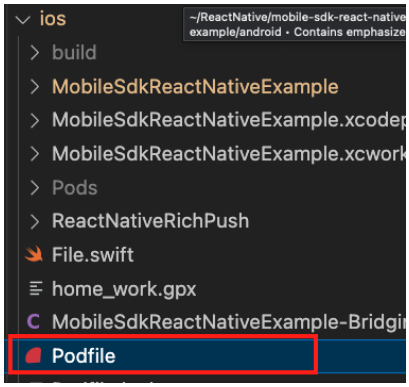
3. Inside `<application></application>`, add these lines:

```
<service android:name=
  "com.sas.mkt.mobile.sdk.SASCollectorIntentService" />

<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

Even though react-native-permissions was installed in **Configure React Native**, by default, no permission handler is installed on iOS.

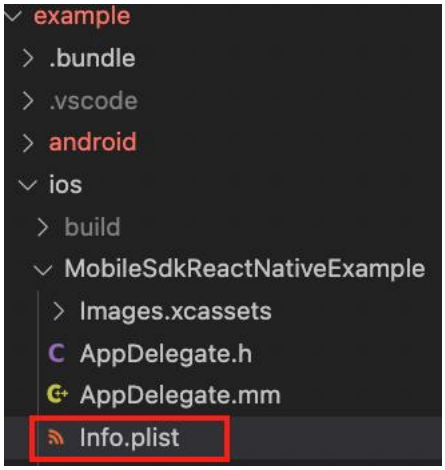
1. In the example project's `ios` folder, find Podfile:



2. In Podfile, add these lines:

```
permissions_path =
  '../node_modules/react-native-permissions/ios'
        pod 'Permission-LocationAccuracy',
  :path => "#{permissions_path}/LocationAccuracy"
        pod 'Permission-LocationAlways',
  :path => "#{permissions_path}/LocationAlways"
        pod 'Permission-LocationWhenInUse',
  :path => "#{permissions_path}/LocationWhenInUse"
```

3. In an integrated terminal, navigate to the `ios` folder in the example project, and run `pod install`.
4. When location permissions are requested from the user, the app must provide request descriptions. To do this, navigate to `example/ios/MobileSdkReactNativeExample` and find Info.plist.



5. In Info.plist, add these location permission request descriptions:

```
<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>
```

Configure React Native (Typescript)

The user's permission is required for you to track their location. To check if the user has granted the permission or to present a permission request, a third-party package must be installed.

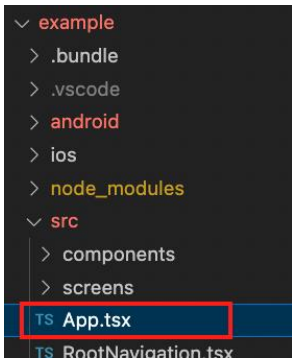
1. In the example project's `package.json` file, add: `"react-native-permissions": "^3.6.1"` under "dependencies":

2. Run `npm install` in an integrated terminal.

Note: `react-native-permissions` provides a wide range of permission checks and requests for React Native on iOS, Android, and Windows. However, SAS Customer Intelligence 360 uses only location permission.

---

3. Find App.tsx in the `src` folder:



4. At the start of App.tsx, add these imports:

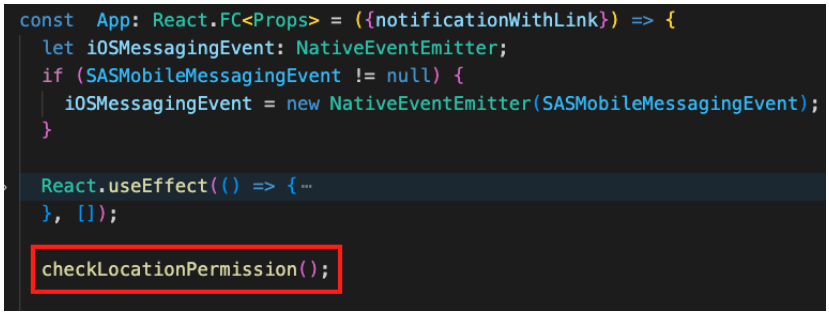
```
import { check, request, PERMISSIONS, RESULTS } from 'react-
native-permissions';

import { SASMobileMessagingEvent, Constants,
startMonitoringLocation } from mobile.sdk-react-native';
```

5. Add a function to check permission and start location monitoring if permission is granted:

```
async function checkLocationPermission(){
  if (Platform.OS === 'ios') {
    let statusIOS =
```

```
      await check(PERMISSIONS.IOS.LOCATION_ALWAYS);
    if (statusIOS === RESULTS.GRANTED){
      startMonitoringLocation();
      return;
    }
    statusIOS =
      await request('ios.permission.LOCATION_ALWAYS');
    if (statusIOS === RESULTS.GRANTED){
      startMonitoringLocation();
      return;
    }
    Toast.show('Not enough location permission. Please set
      location permission to always to enable geofence');

  } else if (Platform.OS == 'android'){
    let status1 = await
      check(PERMISSIONS.ANDROID.ACCESS_COARSE_LOCATION);
    let status2 = await
      check(PERMISSIONS.ANDROID.ACCESS_FINE_LOCATION);
    let status3 = await
      check(PERMISSIONS.ANDROID.ACCESS_BACKGROUND_LOCATION);
    if (status1 == RESULTS.GRANTED &&
        status2 === RESULTS.GRANTED &&
        status3 === RESULTS.GRANTED) {
      startMonitoringLocation();
      return;
    }
    if (status1 !== RESULTS.GRANTED) {
      status1 = await
       request('android.permission.ACCESS_COARSE_LOCATION');
    }
    if (status2 !== RESULTS.GRANTED) {
      status2 = await
        request('android.permission.ACCESS_FINE_LOCATION');
    }
    if (status3 !== RESULTS.GRANTED) {
      status3 = await
   request('android.permission.ACCESS_BACKGROUND_LOCATION');
    }
    if (status1 !== RESULTS.GRANTED &&
        status2 === RESULTS.GRANTED &&
        status3 === RESULTS.GRANTED) {
      Toast.show('Not enough location permission. \
        Please set location permission to always \
        to enable geofence');
    }
  }
}
```

6. Call the `checkLocationPermission` function after the `React.useEffect` block:



Test Geofencing and Beacon Functionality

Create a geofence CSV file with mobile application 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

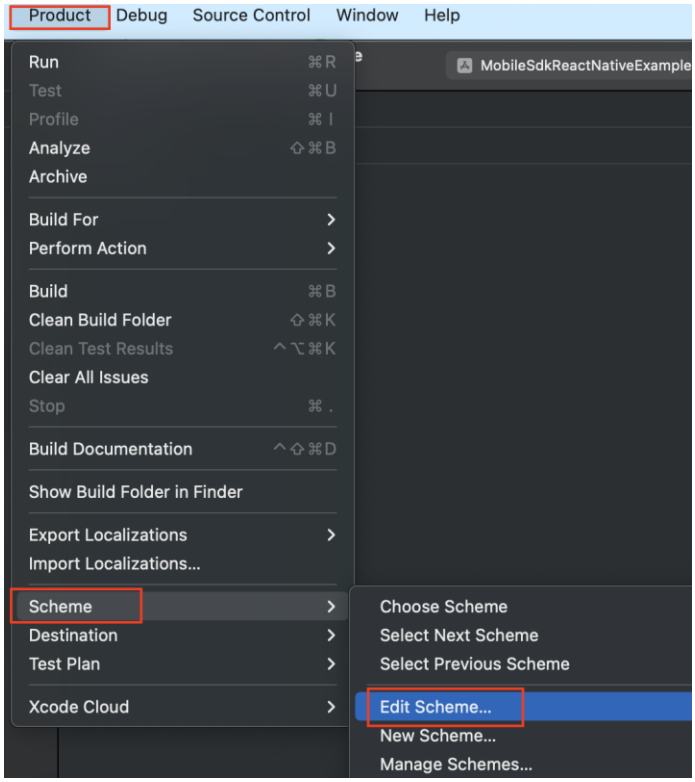
1. In the Android simulator, create a few location points. Make sure some, but not all, locations are also in the CSV file.

   Note: VSCode does not display any logs from native code. To see log information, start the example app from Android Studio, find a location in the simulator that is in the CSV file, and set the location. The logs from Slog should include an enter_geofence event.

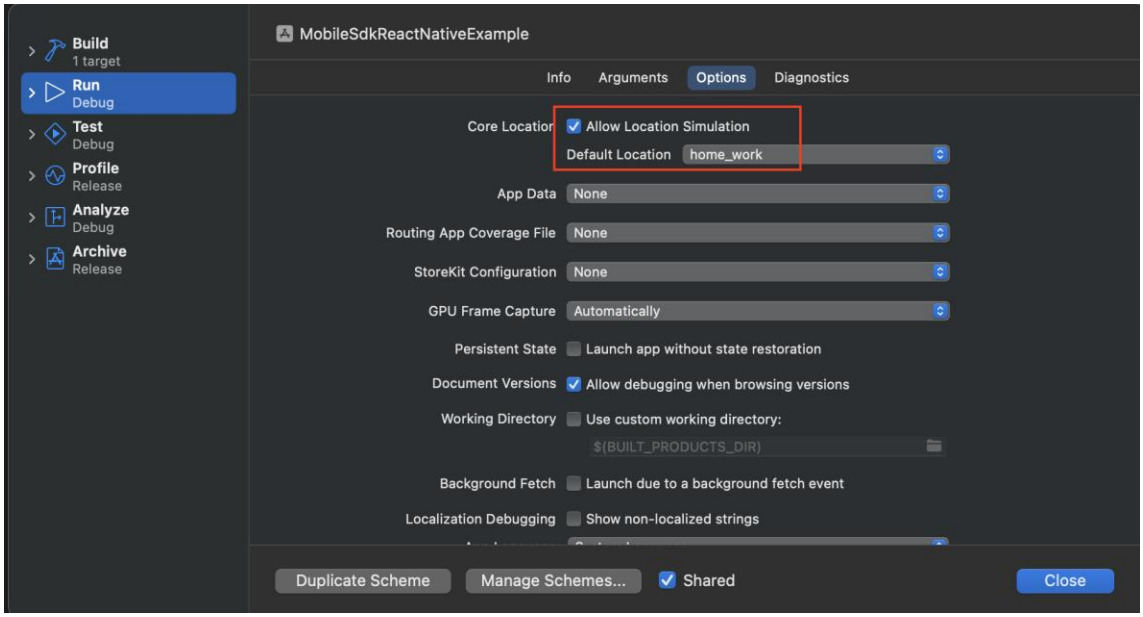
2. To test leaving a geofence, choose a location that is not in the CSV file, and set the location. The result is that an exit_geofence event is logged. Beacon events are also included in the logs.

iOS

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



The following window is displayed:



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

Note:

- To see location logs from SASCollector, the app needs to run from Xcode. VSCode does not display any logs from native code.
- Once the app is run, if a map is open, it moves from one location to another based on the setup in the GPX file.
- Logs of geofence information can be found in the output pane at the bottom of the Xcode window.

Mobile Message Functionality

Mobile message features include token registration, in-app messages, push notifications, rich push notification 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.

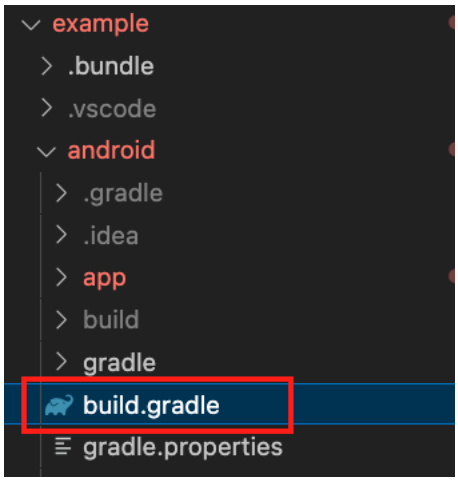
Like the configuration of location functionality, mobile messages require more native setup than Typescript or Javascript setup.

Note: There are third-party push notification plug-ins for React Native apps (such as react-native-notifications), but they do not provide the full functionality that SAS Customer Intelligence 360 mobile messaging delivers.

Configure Android

In Android, RCTDeviceEventEmitter is used, so a custom event emitter does not need to be created. Only the `example` app needs to be configured.

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



2. Add this line in the dependencies section of the project level build.gradle file:

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

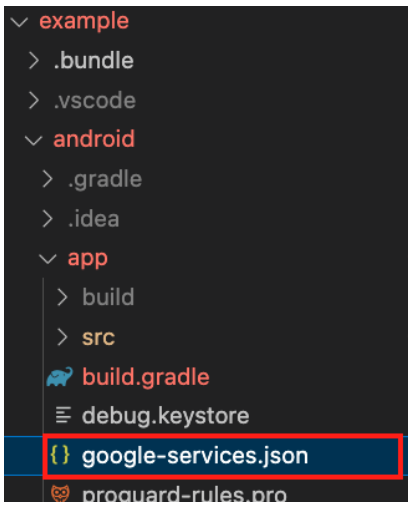
3. In the app level build.gradle file, add these lines in the dependencies section:

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

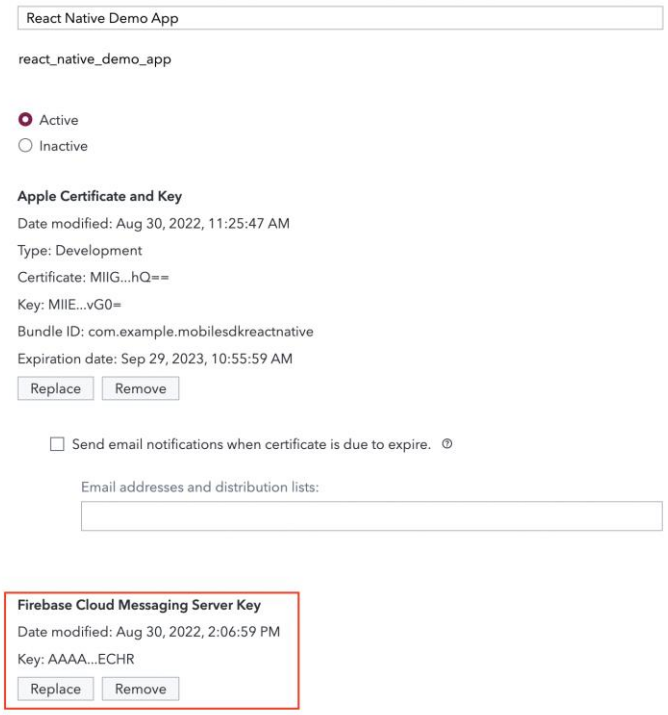
At the top of the file, under `apply:plugin: "com.android.application"`, add this line:

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

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

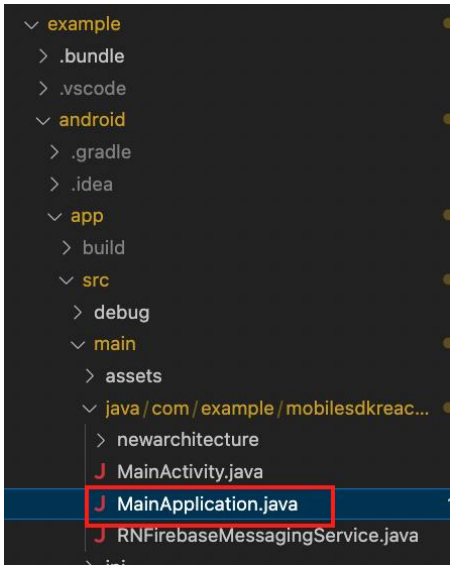


6. From the project in the Firebase console, get the server key and give it to the SAS Customer Intelligence 360 user. The user will add it to the SAS Customer Intelligence 360 mobile application created for the example project.



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

7. Find MainApplication.java in `example/android/app/src/java/com/example/mobilesdkreactnative/`:



8. In MainApplication.java, add the setPushChannel methods, as shown below.

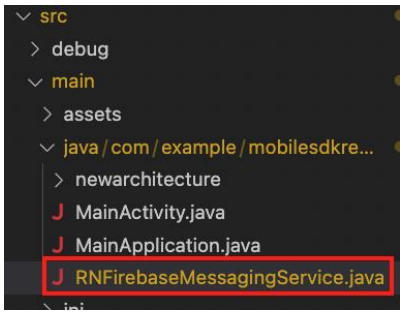
Note: Android version Oreo and above requires a push notification channel. By creating it in the application class, you can avoid having to recreate the channel.

```
@RequiresApi(api = Build.VERSION_CODES.O)
private void setPushChannel() {
  NotificationManager notificationManager =
    (NotificationManager)
    this.getSystemService(this.NOTIFICATION_SERVICE);

  String customAndroidChannel = "ReactNativePushChannel";
  CharSequence channelName = "React Native Channel";
  int importance = NotificationManager.IMPORTANCE_HIGH;
  NotificationChannel notificationChannel =
    new NotificationChannel(
      customAndroidChannel, channelName, importance);
  notificationChannel.enableLights(true);
  notificationChannel.setLightColor(Color.RED);
  notificationChannel.enableVibration(true);
  notificationChannel.setShowBadge(true);
  notificationChannel.setVibrationPattern(
   new long[]{100, 200, 300, 400, 500, 400, 300, 200, 400});
  notificationManager.createNotificationChannel(
    notificationChannel);

  SASCollector.getInstance().setPushNotificationChannelId(
    customAndroidChannel);
}
```

9. In MainApplication.java's onCreate method, call setPushChannel:

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

10. In the same folder where MainApplication.java resides, create RNFirebaseMessagingService.java:



11. Add this code in RNFirebaseMessagingService.java:

```
public class RNFirebaseMessagingService extends
FirebaseMessagingService {
  private static final String TAG = "RNFBMessageService";

  @Override
  public void onMessageReceived(RemoteMessage remoteMessage)
  {
    SLog.i(TAG, "From: " + remoteMessage.getFrom());
    SLog.i(TAG, "Data: " +
      remoteMessage.getData().toString());
    SASCollector.getInstance()
      .handleMobileMessage(remoteMessage.getData());
  }

  @Override
  public void onNewToken(String token) {
    super.onNewToken(token);

    SLog.e("NEW_TOKEN",token);

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

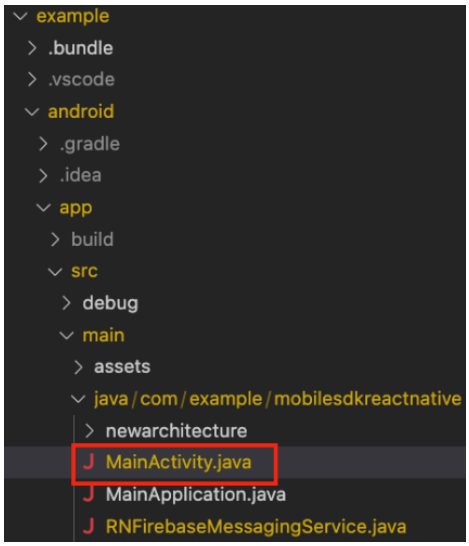
An implementation example of the import is provided in mobile-sdk-react-native.zip. In the mobile-sdk-react-native project example, navigate to `example/android/app/src/main/java/com/example/mobilesdkreactnative/RNFirebaseMessagingService.java` to find the imports.

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

```
<service android:name=
"com.example.mobilesdkreactnative.RNFirebaseMessagingService"
android:exported="false">
  <intent-filter>
```

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

13. Find MainActivity.java in the `example` project in the path `example/android/app/src/main/java/com/example/mobilesdkreactnative/`:



14. At the start of MainActivity.java in the class section, add these variables:

```
private DeviceEventManagerModule.RCTDeviceEventEmitter emitter
   = null;
   private static final String
     DEFAULT_NOTIFICATION_ACTION_LINK_NAME =
       "SASCollectorIntentServicenotificationActionLink";
```

Note: The static string value has to be exactly as in the above code. It is used to get the push notification action link when the application hasn't started yet but will be started by tapping the push notification.

15. For the onCreate method of MainActivity.java, add this code:

```
FirebaseMessaging.getInstance().getToken()
  .addOnSuccessListener(token -> {
    Log.d(TAG, "token="+token);
    if(!TextUtils.isEmpty(token)) {
      SASCollector.getInstance()
        .registerForMobileMessages(token);
```

```
    }
});
if (emitter == null) {
  ReactInstanceManager manager =
    getReactNativeHost().getReactInstanceManager();
  manager.addReactInstanceEventListener(
 (ReactInstanceManager.ReactInstanceEventListener) context ->
   emitter = context.getJSModule(
     DeviceEventManagerModule.RCTDeviceEventEmitter.class));
}

SASCollector.getInstance()
  .setMobileMessagingDelegate2(
    new SASMobileMessagingDelegate2() {
      @Override
      public void dismissed() {
        if (emitter != null) {
          emitter.emit("onMessageDismissed", null);
        }
      }
     @Override
     public void action(
      String s, SASMobileMessageType sasMobileMessageType){
        if (sasMobileMessageType.equals(
          SASMobileMessageType.IN_APP_MESSAGE)) {
          WritableMap args = Arguments.createMap();
          args.putString("link", s);
          args.putString("type", "InAppMsg");

          if (emitter != null) {
            emitter.emit("onMessageOpened", args);
          }
        }
     }
     @Override
```

```
    public Intent getNotificationIntent(String s) {
      SLog.i("getNotificationIntent", s);
      Intent intent = new Intent(
        MainActivity.this, MainActivity.class);
      intent.putExtra("notificationWithLink", s);
      return intent;
    }
  });
```

Note: You might need to call `FirebaseApp.initializeApp(this)` before `FirebaseMessaging.getInstance().getToken()`.

16. If you use CI360 Android SDK 1.80.2, add this method in MainActivity.java:

```
@Override
public void onNewIntent(Intent intent) {
   super.onNewIntent(intent);
   notificationLink =
   intent.getStringExtra("notificationWithLink");
   if (emitter != null && notificationLink != null &&
   !notificationLink.isEmpty()) {
     emitter.emit("onNotificationLinkReceived",
       notificationLink);
   }
 }
```

Note: The event name onNotificationLinkReceived is what the listener in React Native side is listening to. If you want to use other event name, you also need to change the name in React Native side. This also pertains to other strings (e.g. notificationWithLink) in this and other code. You can also create constants for these literal string values to avoid typos.

17. If you use CI360 Android SDK 1.80.3, add this method in MainActivity.java:

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

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

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


    notificationLink =
    intent.getStringExtra("notificationWithLink");
    if (emitter != null && notificationLink != null &&
    !notificationLink.isEmpty()) {
      emitter.emit("onNotificationLinkReceived",
        notificationLink);
    }
  }
```

18. If you use CI360 Android SDK 1.80.3, you also need to update your SASCollector.properties file to include this property:

    apprelaunch.disabled.on.notification.open=true

Note: CI360 Android SDKs 1.80.2 and 1.80.3 fixed an application relaunch bug. Because of this, you will not get the push notification's link in onCreat method, but you will in onNewIntent method if your application is in background. Thus, you need to override onNewIntent if you want to receive the link and send a message to the React Native (typescript) side for it to take actions such as displaying an alert. However, steps 14 and 15 are still valid.

Note: Steps 19 sets up deep linking to navigate to a specific screen dependent on action link contained in push notification data.

---

19. Find AndroidManifest.xml file in example/android/app/src folder and update the MainActivity as below (content added is in bold):

```
<activity
android:name="com.example.mobilesdkreactnative.MainActivity"
     android:label="@string/app_name"
android:configChanges="keyboard|keyboardHidden|orientation|
screenLayout|screenSize|smallestScreenSize|uiMode"
     android:launchMode="singleTask"
     android:windowSoftInputMode="adjustResize"
     android:exported="true">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
        <category
android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
```

```
    <intent-filter>
      <action android:name="android.intent.action.VIEW"/>
      <category
android:name="android.intent.category.DEFAULT"/>
      <category
android:name="android.intent.category.BROWSABLE"/>
       <data android:scheme="app://" />
    </intent-filter>
 </activity>
```

Note: In the example project, the push notification creative's action link is set up to be app://diagnostics. Thus, the scheme in intent-filter is set to be "app://"

20. In MainActivityDelegate class (inside of MainActivity.java), add two variables at the start of the class:

```
private final @Nullable Activity activity;
private Bundle initialProps = null;
```

21. In the constructor of MainActivityDelegate.java, add this line:

```
this.activity = activity;
```

22. Override the onCreate and getLaunchOptions methods of the MainActivityDelegate.java class.

Note: Create onCreate and getLaunchOptions if they are not already in your project.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  initialProps = new Bundle();
  final Bundle bundle =
    activity.getIntent().getExtras();
  if (bundle != null) {
    if (bundle.containsKey("notificationWithLink")) {
      initialProps.putString("notificationWithLink",
        bundle.getString("notificationWithLink"));
    } else if (bundle.containsKey(
          DEFAULT_NOTIFICATION_ACTION_LINK_NAME)) {
```

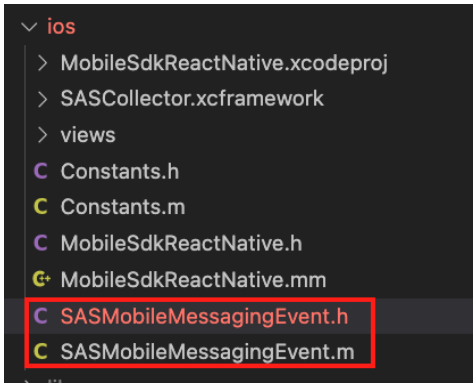
```
        initialProps.putString("notificationWithLink",
    bundle.getString(DEFAULT_NOTIFICATION_ACTION_LINK_NAME));
      }
    }
    super.onCreate(savedInstanceState);
}

@Nullable
@Override
protected Bundle getLaunchOptions() {
    return initialProps;
    }
}
```

Note: Steps 20-22 and part of step 15 are for setting up the Android native part for passing action link (notificationWithLink) as the initial prop to the React Native. When the app is started by tapping push notifications, the action link (notificationWithLink) is passed as an initial props to App.tsx.

Configure iOS

The SAS Customer Intelligence 360 mobile SDKs use mobile messaging delegate SASMobileMessagingDelegate2. Its methods are usually handled by developers to take whatever actions they would like. For this method to work on React Native, the native side must communicate with the Typescript/Javascript side. React Native uses event emitter to send event from the native to Typescript/Javascript side. However, the existing RCTDeviceEventEmitter in React's DeviceEventManagerModule is deprecated for iOS. For iOS, this means that a custom event emitter that extends RCTEventEmitter must be created. This custom event emitter is created as part of the project, not part of the example app.

1. Create SASMobileMessagingEvent.h and SASMobileMessagingEvent.m in the `ios` folder:



2. Add this code to SASMobileMessaginEvent.h:

```
#import <Foundation/Foundation.h>
#import <React/RCTBridgeModule.h>
#import <React/RCTEventEmitter.h>
@interface SASMobileMessagingEvent:
   RCTEventEmitter<RCTBridgeModule>
 +(void)emitMessageOpenedWithPayload:(NSDictionary *)payload;
 +(void)emitMessageDismissed;
@end
```

3. Add this code to SASMobileMessagingEvent.m:

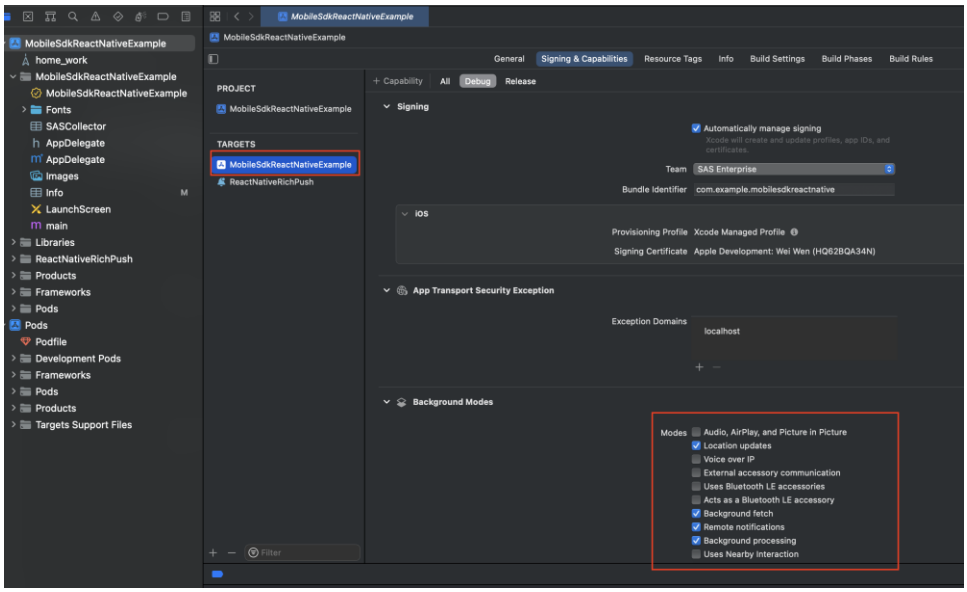
```
#import "SASMobileMessagingEvent.h"
#import "Constants.h"
@implementation SASMobileMessagingEvent
RCT_EXPORT_MODULE();
-(NSArray<NSString *> *)supportedEvents {
  return @[@"onMessageOpened", @"onMessageDismissed"];
}
-(void)onMessageOpened:(NSNotification*)notification {
  NSDictionary *args = notification.userInfo;
  [self sendEventWithName:MESSAGE_OPENED body:args];
}
-(void)onMessageDismissed {
  [self sendEventWithName:MESSAGE_DISMISSED body:nil];
}
-(void)startObserving {
 [[NSNotificationCenter defaultCenter] addObserver:self
   selector:@selector(onMessageOpened:) name:MESSAGE_OPENED
   object:nil];
```

```
  [[NSNotificationCenter defaultCenter] addObserver:self
   selector:@selector(onMessageDismissed)
   name:MESSAGE_DISMISSED object:nil];
}
-(void)stopObserving {
  [[NSNotificationCenter defaultCenter] removeObserver:self];
}
+(void)emitMessageOpenedWithPayload:(NSDictionary *)payload {
  [[NSNotificationCenter defaultCenter]
   postNotificationName:MESSAGE_OPENED object:self
   userInfo:payload];
}
+(void)emitMessageDismissed {
  [[NSNotificationCenter defaultCenter]
   postNotificationName:MESSAGE_DISMISSED object:self];
}
@end
```

4. In an integrated terminal, navigate to the `ios` folder and type `npm install` to include SASMobileMessagingEvent in the library.

To configure the example project:

1. Go to developer.apple.com, enable push notifications for the app, and create a PEM file.
2. Copy the key and certificate and give them to a 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*.
3. Open the example ios project's MobileSdkReactNativeExample.xcworkspace in Xcode. Add push notifications and the checked capabilities in background modes:



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

```
#import <React/RCTBridgeDelegate.h>
#import <UIKit/UIKit.h>
#import <UserNotifications/UserNotifications.h>
#import <React/RCTBridge.h>
#import <React/RCTEventDispatcher.h>
#import <SASCollector/SASCollector.h>

@interface AppDelegate : UIResponder <UIApplicationDelegate,
UNUserNotificationCenterDelegate, RCTBridgeDelegate,
SASMobileMessagingDelegate2>

@property (nonatomic, strong) UIWindow *window;

@end
```

5. In AppDelegate.m:

   a. Add these imports before `#if RCT_NEW_ENABLED`:

```
#import <SASCollector/SASCollector.h>
#import <SASCollector/SASLogger.h>
#import <mobile-sdk-react-native/SASMobileMessagingEvent.h>
```

   b. Add the following method for getting push notification action link when the application is started by tapping the push notification. The action link will be used to redirect to the specific screen. In the example application, it is diagnostic screen.

```
-(NSDictionary*)getActionLinkFromMobileMessage:
  (NSDictionary *)notificationInfo {
```

```
  if (notificationInfo == nil) {
    return nil;
  }
  NSDictionary *aps = notificationInfo[@"aps"];
  NSDictionary *mobileMessageDictionary =
    aps[@"MobileMessage"];

  if (mobileMessageDictionary == nil) {
    return nil;
  }

  if (![mobileMessageDictionary[@"template"]
     isEqualToString:@"creative.pushNotification"]) {
    return nil;
  }

  NSArray *actions = mobileMessageDictionary[@"actions"];
  NSString *link = actions[0][@"link"];
  if (link == nil) {
    return nil;
  }
  return @{@"notificationWithLink": link};
}
```

c. In the didFinishLaunchingWithOptions method, add this code after `#RCT_NEW_ARCH_ENABLED/#endif` pair:

```
[SASLogger setLevel:SASLoggerLevelAll];


if (@available(iOS 10.0, *)) {
  UNUserNotificationCenter.currentNotificationCenter
    .delegate = self;
  [UNUserNotificationCenter.currentNotificationCenter
   requestAuthorizationWithOptions:(UNAuthorizationOptionSou
   nd | UNAuthorizationOptionAlert |
   UNAuthorizationOptionBadge) completionHandler:^(BOOL
   granted, NSError * _Nullable error) {
     if(error != nil) {
       [SASLogger error:error.localizedDescription];
         return;
       }
       dispatch_async(dispatch_get_main_queue(), ^{
         [application registerForRemoteNotifications];
       });
```

```
   }];
}
[SASCollector setMobileMessagingDelegate2:self];
```

d. In the didFinishLaunchingWithOptions method, delete the line that creates the rootView, and then add this code to get notification action link to pass to the rootView as an initial parameter:

```
NSDictionary *notificationInfo =
[launchOptions objectForKey:
   UIApplicationLaunchOptionsRemoteNotificationKey];


UIView *rootView = RCTAppSetupDefaultRootView(bridge,
 @"main", [self
    getActionLinkFromMobileMessage:notificationInfo]);
```

e. Add these methods:

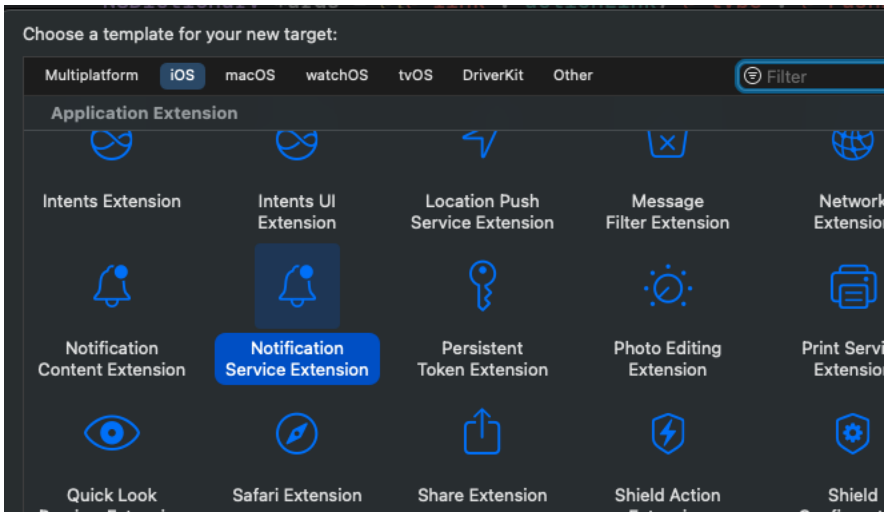
```
-(void)application:(UIApplication *)application
   didRegisterForRemoteNotificationsWithDeviceToken:(NSData
   *)deviceToken {
  [SASCollector registerForMobileMessages:deviceToken
   completionHandler:^{
    [SASLogger info:@"Registering for remote notifications
   is successful"];
  } failureHandler:^{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


-(void)application:(UIApplication *)application
   didReceiveRemoteNotification:(NSDictionary *)userInfo
   fetchCompletionHandler:(void
   (^)(UIBackgroundFetchResult))completionHandler {
  [SASCollector handleMobileMessage:userInfo
   WithApplication:application];
  completionHandler(UIBackgroundFetchResultNoData);
}
-(void)userNotificationCenter:(UNUserNotificationCenter
   *)center
   didReceiveNotificationResponse:(UNNotificationResponse
```

```
   *)response withCompletionHandler:(void
   (^)())completionHandler {
  [SASCollector handleMobileMessage:
    Response.notification.request.content.userInfo
    withApplication:UIApplication.sharedApplication];
  completionHandler();
}
```

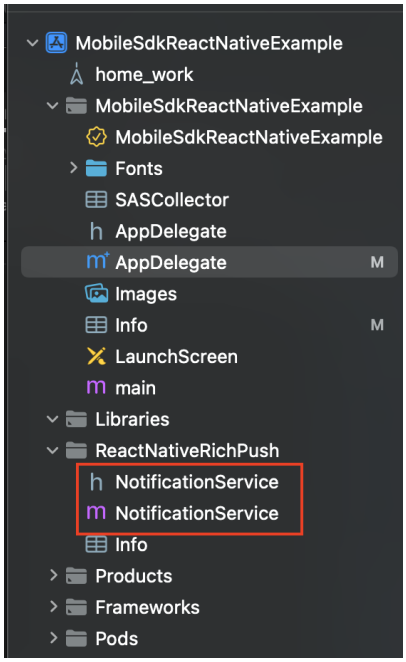
f. At the end of the file, add these `SASMobileMessagingDelegate2` methods:

```
#pragma mark SASMobileMessagingDelegate2
-(void)actionWithLink:(NSString * _Nonnull)link
   type:(SASMobileMessageType)type {
  NSMutableString* msgType = [NSMutableString
   stringWithString:@""];
  if (type == SASMobileMessageTypePushNotification) {
    msgType = [NSMutableString
   stringWithString:@"PushNotification"];
  } else if (type == SASMobileMessageTypeInAppMessage) {
    msgType = [NSMutableString
   stringWithString:@"InAppMsg"];
  }
  NSDictionary *args = @{@"type": msgType,
    @"link": link};
  [SASMobileMessagingEvent
    emitMessageOpenedWithPayload:args];
}
-(void)messageDismissed {
  [SASMobileMessagingEvent emitMessageDismissed];
}
```

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



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



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

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

  self.contentHandler = contentHandler;

  self.bestAttemptContent = [request.content mutableCopy];


  NSDictionary *notificationData =

    (NSDictionary*)request.content.userInfo[@"data"];

  if (notificationData == nil) {
```

```
  return;
}
NSString *urlStr = (NSString*)[notificationData
objectForKey:@"attachment-url"];
if (urlStr == nil) {
  return;
}
NSURL *fileUrl = [NSURL URLWithString:urlStr];
if (fileUrl == nil) {
  return;
}
NSURLSessionDownloadTask *downloadTask =
  [NSURLSession.sharedSession
    downloadTaskWithURL:fileUrl
    completionHandler:^(NSURL * _Nullable location,
      NSURLResponse * _Nullable response,
      NSError * _Nullable error) {

     if (location != nil && error == nil) {
       NSString *tempDir = NSTemporaryDirectory();
       NSString *suggestedName = [response
suggestedFilename];
       if (suggestedName != nil) {
         NSString *fileName = [NSString
stringWithFormat:@"file://%@%@", tempDir, suggestedName];
         NSString *tempFileName = [fileName
stringByReplacingOccurrencesOfString:@" " withString:@"_"];
         NSURL *tempUrl = [NSURL
URLWithString:tempFileName];
         NSError *removeFileError;

         if ([NSFileManager.defaultManager
fileExistsAtPath:tempUrl.path] &&
[NSFileManager.defaultManager
isDeletableFileAtPath:tempUrl.path]) {
           [NSFileManager.defaultManager
removeItemAtPath:tempUrl.path error:&removeFileError];
          }
```

```
            if (removeFileError != nil) return;


            NSError *moveFileError;
            [NSFileManager.defaultManager
  moveItemAtURL:location toURL:tempUrl error:&moveFileError];
            if (moveFileError != nil) return;


            NSError *attachmentError;
            UNNotificationAttachment *attachment =
  [UNNotificationAttachment
  attachmentWithIdentifier:@"ci360content" URL:tempUrl
  options:nil error:&attachmentError];
            self.bestAttemptContent.attachments =
  @[attachment];
            if (attachmentError != nil) return;
          }
        }
        self.contentHandler(self.bestAttemptContent);
    }];
    [downloadTask resume];
}
```

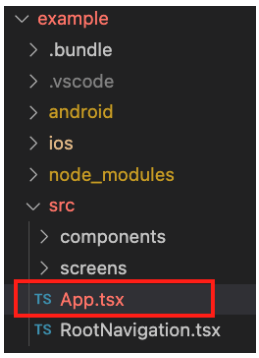
Configure React Native (Typescript)

Push Notifications

1. Open Integrated Terminal, cd to example folder, and type this command:

npm install events

2. Find App.tsx in the `example/src` folder.



3. Add these three imports:

```
import { SASMobileMessagingEvent, Constants } from 'mobile-
   sdk-react-native';

import {navigate, RootTabParameterList} from
   './RootNavigation';
```

import {EventEmitter as EvtEmitter} from 'events';

Note: RootNavigation.tsx was created in the same (src) folder. The file contains input parameter definitions of tab screens. The exported navigate is used in case navigation is not created yet to avoid crashes in the app.

4. Change the app type:

```
const App: React.FC<Props> = ({notificationWithLink})
```

Note: notificationWithLink is the parameter passed from native side when the app has not been running and tapping push notification starts the app.

5. Before the start of App definition, create an instance of event emitter.

const emitter = new EvtEmitter();

Note: The emitter will send event when push notification is received while the app is in the background. Listener will be set up in deep link in step 11 to listen for the event and navigate to the Diagnostic screen.

6. add an entry in src/Constants.tsx of the project (NOT in example folder):

export const MESSAGE_NOTIFICATION_LINK_RECEIVED =

"onNotificationLinkReceived";

7. At the start of App definition, create a mobile messaging event for iOS:

```
let iOSMessagingEvent: NativeEventEmitter;

if (SASMobileMessagingEvent != null) {

  iOSMessagingEvent = new
   NativeEventEmitter(SASMobileMessagingEvent);

}
```

8. Add this code to ensure that event listeners are in place to handle both push notifications and in-app messages:

```
React.useEffect(() => {

  // This happens when the app is not started yet.
```

```
//Tapping push notification starts the app.
//notificationWithLink is a parameter in the initial props
 passed from the native side.
  if (notificationWithLink &&
 notificationWithLink?.includes('diagnostics')) {
    Toast.show('User got push notification with link' +
 notificationWithLink, Toast.SHORT);
    setInitialTab('Diagnostics');

if (Platform.OS === 'android') {

 DeviceEventEmitter.addListener(Constants.MESSAGE_DISMISSED,
 () => {
   Toast.show('User dismissed message', Toast.SHORT);
 });
    DeviceEventEmitter.addListener(Constants.MESSAGE_OPENED,
 (obj: {[key: string]: string}) => {
   const link = obj['link'];
   const msg = 'User got in-app msg with link:' + link;
   Toast.show(msg, Toast.SHORT);
   if (link?.includes('diagnostic')) {
     navigate('Diagnostics', {link: link});
   }
 });
 DeviceEventEmitter.addListener(Constants
  .MESSAGE_NOTIFICATION_LINK_RECEIVED, (link: string) => {
    if (link.includes('diagnostic')) {
      Toast.show('User got push notification' +
        ' with link' + link, Toast.SHORT);

      //This event is for the listener in deep link to listen
      emitter.emit('PushLink', {link: link})
    }
  });

 } else if (Platform.OS === 'ios') {
```

```
    iOSMessagingEvent.addListener(Constants.MESSAGE_OPENED,
  (data) => {
      console.log('data: ' + data);
      console.log('message type: ' + data.type + 'link is: '
  + data.link);
      const msg = (data.type === 'InAppMsg' ? 'User got in-
  app msg with link:' + data.link : 'User got push
  notification with link:' + data.link);
      Toast.show(msg, Toast.SHORT);

      if (data.type === 'InAppMsg') {
         navigate('Diagnostics', {link: data.link});
      } else {
          //This event is for the listener in deep link to listen
         emitter.emit('PushLink', {link: data.link})
       }
   });

   iOSMessagingEvent.addListener(Constants.MESSAGE_DISMISSED,
   () => {
     Toast.show('User dismissed message', Toast.SHORT);
   });
  }

  return () => {
    DeviceEventEmitter.removeAllListeners();
    if (SASMobileMessagingEvent) {
      iOSMessagingEvent.removeAllListeners(
        SASMobileMessagingEvent);
    }
  }
}, []);
```

9. **Android only**: Right after `checkLocationPermission()`, add this code to specify the tab to display for a push notification. In this example, the app is set to display the **Diagnostics** tab when push notification is received.

Note: This step is only for Android because when a push notification is clicked by the user, the app restarts.

```
let tabName = 'Identity';
let pushLink = '';
if (notificationWithLink  &&
   notificationWithLink.includes('diagnostics')) {
    tabName = 'Diagnostics'
    pushLink = notificationWithLink;
}
```

10. If you use CI360 Android SDK 1.80.2 or 1.80.3, step 9 is not needed. Instead, this and step 11 will set up for deep link. After `checkLocationPermission()` , add this object to set up the screen when push notification is received:

```
const config = {
   screens: {
      Diagnostics: 'diagnostics'
   }
};
```

Note: Diagnostics is one of the screen name in the example project, 'diagnostics' is the part in the action link: app://diagnostics

11. After the config definition in step 10, add the linking options:

```
const linking: LinkingOptions<RootTabParameterList> = {
 prefixes: ['app://'],
 config: config,
 subscribe(listener: any) {
   const pushSub = emitter.addListener('PushLink',
           (data: any) => {
             listener(data.link)
           });
   return () => {
     pushSub.removeAllListeners();
   }
  }
};
```

Note: 'PushLink' event name that is passed in emitter.addListener is the event name that the emitter uses when sending the event with push notification's action link in both Android and iOS in step 8.

12. In <Tab.Navigator> add initialRouteName:

```
<Tab.Navigator initialRouteName={initialTab}
```

13. In <Tab.Screen> for Diagnostics, add initialParams:

```
<Tab.Screen name='Diagnostics' component={DiagnosticScreen}
  options={{tabBarLabel: 'diagnostics'}} />
```

14. If you use deep link, steps 12 and 13 are not needed. Instead update NavigationContainer by passing it the linking parameter:

<NavigationContainer linking={linking}>

Note: "linking" in {} is the parameter defined in step 11.

In-App Messages

In-app messages are sent to devices by invoking addAppEvent, which is exposed in the React Native library.

In the example project, a screen is created for sending an in-app message event and receiving it. Find MessageScreen.tsx in example/src/screens.

1. In MessageScreen, add this import:

```
import { addAppEvent } from 'mobile-sdk-react-native';
```

   For other imports, please see the attachment.

2. In the <View> container, add this code:

```
<View style={styles.spot}>
  <TextInput style={styles.textInput}
    onChangeText={setSmallMsg} value={smallMsg}/>
  <CustomButton title='Send Small In-App'
    width={{width: 200}} onPress={() => {
      addAppEvent(smallMsg, null); }} />
</View>
<View style={styles.spacer} />
```

```
<View style={styles.spot}>
  <TextInput style={styles.textInput}
    onChangeText={setLargeMsg} value={largeMsg} />
  <CustomButton title='Send Large In-App'
    width={{width: 200}} onPress={() => {
      addAppEvent(largeMsg, null); }} />
</View>
```

Note: The initial smallMsg and largeMsg values are the small and large in-app message events names created in Design Center.

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

Push notifications can be tested by sending external events from Postman.

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**.
3. For ID type, select your device ID and click Submit Test. You should receive a test push notification on your device.

When creating a push notification creative (and then using that creative in a push task) in SAS Customer Intelligence 360, if you have 'diagnostic' in the creative uri, clicking the notification brings the app to foreground and opens the diagnostic screen.

Test In-App Messages

To test in-app messages, select the **Messages** tab on the running app. Click the **Send Small In-App** button to open the small in-app message pop-up at the bottom of the screen. Click the **Send Large In-App** button to open the large in-app message pop-up at the center of the screen. Both message pop-ups have a Close (x) button. If a user clicks the button, a toast message briefly appears. The large in-app message pop-up has a button that opens the diagnostic screen.

Integrate the React Native Library with an Existing React Native App

1. To use the new library in an existing project, copy the library to a desired location. Note the absolute path of the library.

   Note: The location cannot be in the existing project's `node_modules` folder.

   For example, in the figure below, the app folder and the library folder are in the same folder. mobile_sdk_test is the React Native app that uses the mobile-sdk-react-native library.

2. Open your React Native app project in VSCode. In this example, the app is mobile_sdk_test.
3. Install the babel-plugin-module-resolver library and save it as a development dependency in package.json by running this command in integrated terminal:

   ```
   npm install babel-plugin-module-resolver --save-dev
   ```

   Note: babel-plugin-module-resolver is a library that adds a new resolver for the library (mobile-sdk-react-native) when compiling your Typescript-based project or project with newer ECMAScript2015+ Javascript using Babel.

4. In your app, find the babel.config.js file.

a. At the start of babel.config.js, add this code:

```
const path = require('path');

const pak = require('../mobile-sdk-react-
   native/package.json');
```

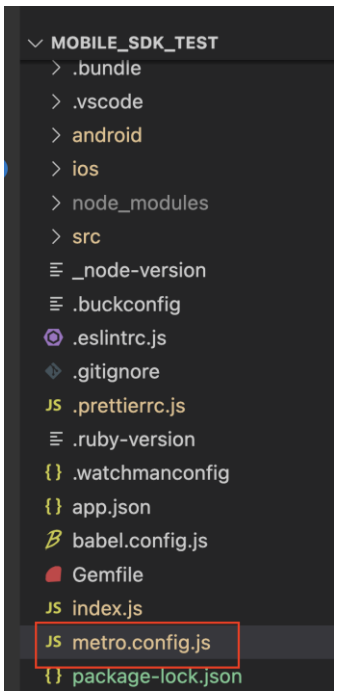
Note: The path in pak is based on folder structures in step 1. In the steps that follow, all paths are based on the same folder structures. If yours are different, you need to make modifications to the paths.

b. Add this code inside module.exports:

```
plugins: [
    [
      'module-resolver',
      {
        extensions: ['.tsx', '.ts', '.js', '.json'],
        alias: {
          [pak.name]: path.join(
            __dirname,
            '../mobile-sdk-react-native',
            pak.source,
          ),
```

```
        },
      },
    ],
```

5. Find the metro.config.js file.



   a. Add this code before module.exports:

```
const path = require('path');
const escape = require('escape-string-regexp');
const exclusionList = require(
  'metro-config/src/defaults/exclusionList');
const pak = require(
  '../mobile-sdk-react-native/package.json');
const sdkPath = path.resolve(__dirname,
  '../mobile-sdk-react-native');
const modules = Object.keys({
  ...pak.peerDependencies,
});
```

   b. Inside module.exports, add watchFolders and resolver:

```
watchFolders: [sdkPath],
  resolver: {
    blacklistRE: exclusionList(
```

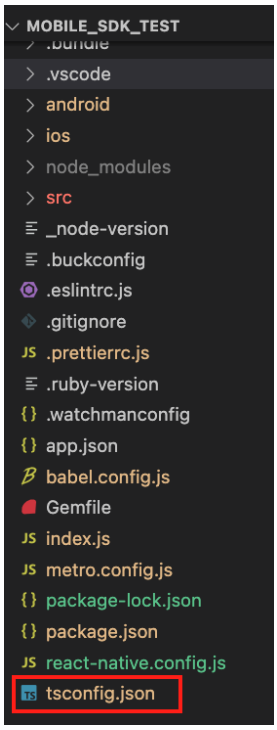
```
    modules.map(
      m =>
        new RegExp(`^${escape(path.join(sdkPath,
 'node_modules', m))}\\/.*$`),
    ),
  ),

extraNodeModules: modules.reduce((acc, name) => {
    acc[name] = path.join(__dirname, 'node_modules',
 name);
    return acc;
  }, {}),
},
```

6. Find react-native.config.js, and add this code:

```
const path = require('path');
module.exports = {
  dependencies: {
    'module-sdk-react-native': {
      root: path.join(__dirname, '../mobile-sdk-react-
 native'),
    },
  },
};
```

Note: If there is no react-native.config.js file in your React Native project, create it at the project level.

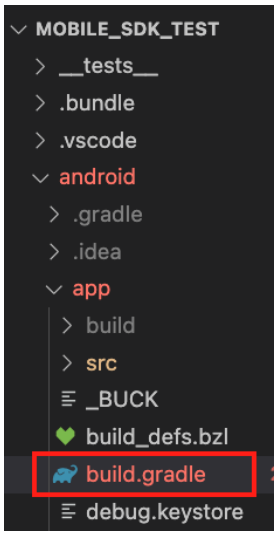
7. Find tsconfig.json, assuming the React Native project is based on Typescript.



In tsconfig.json, add this code inside “compilerOptions”:

```
"baseUrl": "./",
"paths": {
  "mobile-sdk-react-native":
  ["../mobile-sdk-react-native/src/index"]
},
```

8. The easiest way to make these changes take effect is to close the project and reopen it in VSCode. Alternatively, you can run `yarn tsc` if you have installed Yarn.
9. In the `android/app` folder, find build.gradle.



10. In the app level build.gradle file, add this line to the dependencies section:

```
implementation files('../../../mobile-sdk-react-
   native/android/libs/SASCollector.jar')
```

    After making the change, you will be asked to sync the project. Click **Yes**.

11. In an integrated terminal, go to the `ios` folder, and run `pod install`.
12. To use the library, look back in previous sections on how to add the functionalities in the `example` 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.

Updates

October 2023 Updates

The Inline mobile spot created in the example project is a simple use case. However, you may receive requirements to make it more dynamic. For example, if the mobile spot has no default content, it is desirable to not display the mobile spot. Another example is when the spot id is changed, we may need to update the mobile spot to show the new content. Such use cases of the inline mobile spots will be covered in this update.

Configure Android

1. Find InlineAdViewManager.java in android/src/main/java/com/mobilesdkreactnative/views, and create this property:

```
@ReactProp(name="notVisible")
  public void setHiden(InlineAdView inlineAdView,
    @Nullable boolean notVisible) {
    if (notVisible) {
      inlineAdView.setVisibility(View.GONE);
    } else {
      inlineAdView.setVisibility(View.VISIBLE);
    }
  }
```

Configure iOS

1. Find InlineAdView.h in ios/views and add a boolean property:

```
@property (nonatomic) BOOL notVisible;
```

2. Find InlineAdView.m in ios/views and make the following updates:

   a. Add an instance boolean variable inside InlineAdView interface:

```
@interface InlineAdView() {
  NSString* _spotID;
  BOOL _notVisible;
```

```
}
@end
```

b. Add notVisible property's getter and setter methods:

```
-(BOOL)notVisible {
    return _notVisible;
}
-(void)setNotVisible:(BOOL)notVisible {
  _notVisible = notVisible;
  [self setHidden:_notVisible];
}
```

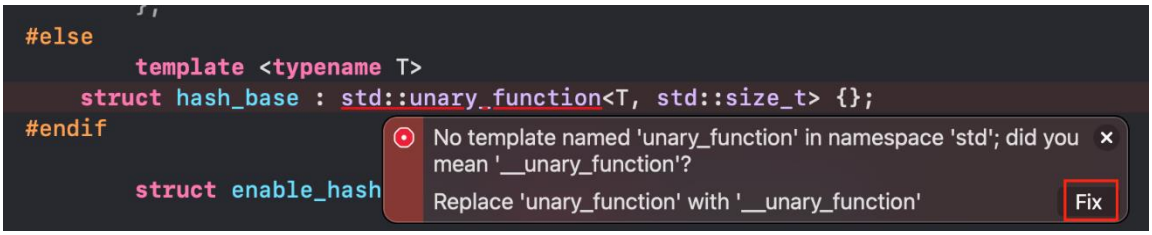
c. Find InlineAdViewManager.m and export notVisible property:

```
RCT_EXPORT_VIEW_PROPERTY(notVisible, BOOL)
```

Note: With the recent update of XCode to version 15, building the iOS application will generate compile time error. If this is the case when you run your iOS application, follow step 3 to fix the error.

---

3. Open the example project's iOS application in Xcode, build and run. You will see the following error in hash.hpp file. Click "Fix" to resolve the error.



Configure React Native (Typescript)

There are two places to update, with one in the library and one in the example application.

1. Find Constants.tsx in src and add the following variable:

```
export const AD_DEFAULT_LOADED = "onAdDefaultLoaded";
```

2. Find InlineAdView.tsx in src/views and update Props definition by adding the boolean variable notVisible:

```
type Props = {
  spotId: string;
  notVisible: boolean;
  style: ViewStyle;
};
```

3. Find SpotsScreen.tsx in example/src/screens and make the following updates:

a. Add an import to provide list selection functionality:

```
import SelectList from 'react-native-dropdown-select-list';
```

Note: The library in step a has been used in LoginScreen to provide users the option to choose the login type. It is used in SpotsScreen to provide the options to choose different mobile spots. You can also use other libraries that achieve the same functionality.

b. At the start of SpotsScreen definition, add this code:

```
const spotIdList = [
 {key:'cuteCatSpot_WW', value:'cuteCatSpot_WW'},
 {key:'cuteDogSpot_WW', value:'cuteDogSpot_WW'},
 {key:'flowerSpot_WW', value:'flowerSpot_WW'},
 {key:'noDefaultViewSpot_WW', value:'noDefaultViewSpot_WW'}
];

const [spotId, setSpotId] =
  React.useState('noDefaultViewSpot_WW');

const [inlineViewNotVisible, setInlineViewNotVisible] =
  React.useState(true);
```

Note: The content inside spotIdList are the spot Ids created for the example project. Use your own spot Ids. The last spot Id in spotIdList does not have any default content. The inclusion of it is to demonstrates that it will not show on the screen.

c. Inside React.useEffect, update addListener callbacks (Update is in bold as before):

```
if(Platform.OS === 'ios') {
   iOSMessagingEvent.addListener(Constants.AD_LOADED,
     (event: Event) => {
     if (event === Constants.TYPE_INTERSTITIAL_AD) {
       Toast.show('Interstitial Ad view is loaded', Toast.SHORT);
     } else if (event === Constants.TYPE_INLINE_AD) {
       Toast.show('Inline Ad view is loaded', Toast.SHORT);
       setInlineViewNotVisible(false);
     }
   });
   iOSMessagingEvent.addListener(Constants.AD_DEFAULT_LOADED,
     (event: Event) => {
     if (event === Constants.TYPE_INTERSTITIAL_AD) {
       Toast.show('Interstitial Ad default view is loaded',
       Toast.SHORT);
     } else if (event === Constants.TYPE_INLINE_AD) {
       Toast.show('Inline Ad default view is loaded', Toast.SHORT);
       setInlineViewNotVisible(true);
     }
   });
    //.....
 else if (Platform.OS == 'android') {
   DeviceEventEmitter.addListener(Constants.AD_LOADED,
     (event: Event) => {
     if (event === Constants.TYPE_INTERSTITIAL_AD) {
       Toast.show('Interstitial Ad view is loaded', Toast.SHORT);
     } else if (event === Constants.TYPE_INLINE_AD) {
       Toast.show('Inline Ad view is loaded', Toast.SHORT);
       setInlineViewNotVisible(false);
     }
   });
   DeviceEventEmitter.addListener(Constants.AD_DEFAULT_LOADED,
```

```
(event: Event) => {
  if (event === Constants.TYPE_INTERSTITIAL_AD) {
    Toast.show('Interstitial Ad default view is loaded',
    Toast.SHORT);
  } else if (event === Constants.TYPE_INLINE_AD) {
    Toast.show('Inline Ad default view is loaded', Toast.SHORT);
    setInlineViewNotVisible(true);
  }
});
```

c. Update InlineAdView and add SelectList component inside the last “return”. The layout of the views is re-designed. So please look in the zipped example project to view the details.

```
<InlineAdView
  spotId={spotId}
  notVisible={inlineViewNotVisible}
  style={ inlineViewNotVisible ? styles.inlineViewWithoutContent
          : styles.inlineViewWithContent} />

<SelectList
  boxStyles={{width: 200}}
  style={styles.selectionList}
  data={spotIdList}
  setSelected={setSpotId} />
```

Run the example applications, select different spot ids, and you will see that the content of the selected spot is displayed. If the id of the spot with no default content is selected, the view is not displayed.

Update with SASCollector SDK release

The latest SASCollector SDKs (iOS 1.72.3 and Android1.80.4) allows you to disable focus/defocus events and get feedback message from identity call. Here is what you can update to get the features.

Disable focus/defocus events:

1. Find SASCollector.properties in example/android/app/src/main/assets and add this entry:

   ```
   disable.focus.tracking=true
   ```

2. Find SASCollector.plist in example/ios and add this entry:

```
disableFocusTracking = "true";
```

Note: Steps 1 and 2 are optional. You only need them when you do not want to have focus/defocus events.

Receive feedback messages from identity call:

1. Find MobileSdkReactNativeModule.java in android/src/main/java/com/mobilesdkreactnative and update identity method:

```
@ReactMethod
public void identity(String value, String type, Promise promise) {
  SASCollector.getInstance().identity(value, type,
    new SASCollector.IdentityCallbackWithMessage() {
    @Override
    public void onComplete(boolean b, String message) {
 SLog.d("Identity callback message", message);
      SLog.d("Identity", "Identity called with: " +
        (b ? "success" : "failure"));

      new Handler(Looper.getMainLooper()).post(new Runnable() {
        @Override
        public void run() {
          promise.resolve(b);
        }
      });
    }
  });
}
```

2. Find MobileSdkReactNative.mm in ios and update identity method:

```
RCT_EXPORT_METHOD(identity:(NSString*)value
  withType:(NSString*)type
  isSuccess:(RCTPromiseResolveBlock)successPromise
  isFailure:(RCTPromiseRejectBlock)failurePromise) {
  [SASCollector identity:value withType:type
    withCompletion:^(BOOL success, NSString * _Nonnull message) {
    SLogInfo(@"Identity callback message: %@", message);

    dispatch_async(dispatch_get_main_queue(), ^{
```

```
        if (success) {
          successPromise([NSNumber numberWithBool:success]);
        } else {
          failurePromise(@"Error", @"Identity failure", nil);
        }
      });
    }];
}
```

Note: Steps 1 and 2 are optional. You only need them when you want to get feedback messages from the identity call. The messages can tell you the reason why you get failure or success.

November 2023 Updates

In October's update on mobile spots, the mobile spot screen only contains one inline mobile spot view. However, there may come the need to include multiple inline mobile spot views on one screen. And depending on whether the views have content, the views may need to render on screen or not. With the previous update, simply adding more views will not work as expected when we need to style the views differently or show/hide the views depending on which does not have content. This update addresses this issue.

In this update, in addition to changes in native Android and iOS mobile spots, a new screen is added to include one inline mobile spot view that does not have default content and another one that has default content (no content). In addition, for the view that has default content, the view is styled according to the spot id.

Configure Android

1. Find InlineAdViewManager.java in android/src/main/java/com/mobilesdkreactnative/views.

   a. Remove spotID property previously added in InlineAdViewManager.

   b. Add fields id and spotId in InlineAdView:

```
String id = "";
String spotId = "";
```

   c. Add two setter methods in InlineAdView:

```
public void setId(String id) {
  this.id = id;
}
public void setSpotId(String spotId) {
```

```
    this.spotId = spotId;
  }
```

d: Update sendEvent in InlineAdView:

```
  private void sendEvent(String eventName, String type) {
    ReactContext reactContext =
    UseReactContext.getReactContext(InlineAdView.this.getContext());
    boolean isLoadOrDefaultLoad = false;
    // Need to import WritableMap at the start of the class
    WritableMap data = new WritableNativeMap();

    if (eventName.equals(Constants.AD_LOADED) ||
        eventName.equals(Constants.AD_DEFAULT_LOADED)) {
        data.putString("eventType", type);
         data.putString("spotId", spotId != null ? spotId : "");
         data.putString("viewId", id != null ? id : "");
         isLoadOrDefaultLoad = true;
     }
     reactContext.getJSModule(
       DeviceEventManagerModule.RCTDeviceEventEmitter.class)
     .emit(eventName, isLoadOrDefaultLoad ? data : eventName);
  }
```

e. Update these property setter methods in InlineAdViewManager:

```
  @ReactProp(name = "spotId")
  public void setSpotId(InlineAdView inlineAdView,
    @Nullable String spotID) {
    if (spotID != null) {
       inlineAdView.setSpotId(spotID);
       inlineAdView.load(spotID, null);
    }forLoad
  }
  @ReactProp(name = "viewId")
  public void setViewId(InlineAdView inlineAdView,
    @Nullable String viewId) {
    inlineAdView.setId(viewId);
  }
```

```
@ReactProp(name="notVisible")
public void setHiden(InlineAdView inlineAdView,
  @Nullable boolean notVisible) {
  if (notVisible) {
     inlineAdView.setVisibility(View.GONE);
  } else {
    inlineAdView.setVisibility(View.VISIBLE);
  }
}
```

Note: For completeness, you can update InterstitialAdViewManager.java similarly. For details, please see the example project.

Configure iOS

1. Find InlineAdView.h in ios/views and add viewId property:

```
@property (nonatomic, copy) NSString *viewId;
```

2. Find InlineAdView.m in ios/views and do the following update:

   a. Define local storage for viewId:

```
@interface InlineAdView() {
  NSString* _spotID;
  NSString* _viewID;
  BOOL _notVisible;
}
```

   b. Define getter and setter methods of viewId:

```
-(NSString*)viewId {
  return _viewID;
}
-(void)setViewId:(NSString *)viewId {
  _viewID = viewId;
}
```

3. Find InlineAdViewManager.m in ios/views and do the following update:

   a. Remove the definition of InlineAdView storage, i.e. remove the variable inside "@implementation"

b. Add viewId as an exported property:

```
RCT_EXPORT_VIEW_PROPERTY(viewId, NSString)
```

c. Update didLoad and didLoadDefault method:

```
- (void)didLoad:(SASIA_AbstractAd *)ad {
    InlineAdView *adView = (InlineAdView*)ad;
    [AdDelegateEvent emitAdLoadedEventWithType:TYPE_INLINE_AD
      withSpotId:adView.spotId withViewId:adView.viewId];
 }
 - (void)didLoadDefault:(SASIA_AbstractAd *)ad {
    InlineAdView *adView = (InlineAdView*)ad;
    [AdDelegateEvent
      emitAdDefaultLoadedEventWithType:TYPE_INLINE_AD
      withSpotId:adView.spotId withViewId:adView.viewId];
 }
```

d. Find AdDelegateEvent.h in ios/views and update the following two method declarations:

```
+(void)emitAdLoadedEventWithType:(NSString*)adType
  withSpotId:(NSString*)spotId withViewId:(NSString*)viewId;
+(void)emitAdDefaultLoadedEventWithType:(NSString*)adType
  withSpotId:(NSString*)spotId withViewId:(NSString*)viewId;
```

e. Find AdDelegateEvent.m in ios/views and update the following two methods:

```
+(void)emitAdLoadedEventWithType:(NSString*)adType
  withSpotId:(NSString *)spotId withViewId:(NSString *)viewId{
  [[NSNotificationCenter defaultCenter]
   postNotificationName:AD_LOADED object:nil
   userInfo:@{@"type": adType, @"spotId": spotId,
    @"viewId": viewId}];
}
+(void)emitAdDefaultLoadedEventWithType:(NSString*)adType
  withSpotId:(NSString *)spotId withViewId:(NSString *)viewId {
  [[NSNotificationCenter defaultCenter]
    postNotificationName:AD_DEFAULT_LOADED object:nil
    userInfo:@{@"type": adType, @"spotId": spotId,
    @"viewId": viewId}];
 }
```

f. Create the following method:

```
-(void)composeAndSendEvent:(NSNotification *)notification
  withEventName:(NSString*)eventName {
  // can be inline or interstitial
  NSString *type = [notification.userInfo objectForKey:@"type"];
  NSString *spotId = [notification.userInfo
                      objectForKey:@"spotId"];
  NSString *viewId = [notification.userInfo
                      objectForKey:@"viewId"];
  if (!spotId)
     spotId = @"";
  if (!viewId)
     viewId = @"";
  if (![eventName isEqualToString:AD_DEFAULT_LOADED] &&
      ![eventName isEqualToString:AD_LOADED]) {
     return;
  }
  [self sendEventWithName:eventName
   body:@{@"eventType": type, @"spotId": spotId,
          @"viewId": viewId}];
}
```

g. Update the following methods to call the method defined in step f:

```
-(void)onAdLoaded:(NSNotification*)notification {
  [self composeAndSendEvent:notification withEventName:AD_LOADED];
}
-(void)onAdDefaultLoaded:(NSNotification*)notification {
  [self composeAndSendEvent:notification
       withEventName:AD_DEFAULT_LOADED];
}
```

Note: For completeness, you can update InterstitialAdView.h, InterstitialAdView.m, InterstitialAdVIewController.m similarly. For details, please see the example project.

Configure React Native (Typescript)

This part of the update is only in the example folder. In order not to affect previously added functionality, a new screen is created.

1. Create Spots2Screen.tsx file in example/src/screens. The following only contains important part related to the newly added properties (spotId, viewId) of the spot views.

a. Add the following at the start of SpotsScreen definition. They will be used in the InlineAdViews:

```
const spotId = 'noDefaultViewSpot_WW';
const spotId2 = 'cuteDogSpot_WW';
const [inlineViewNotVisible, setInlineViewNotVisible] =
      React.useState(true);
const [inlineViewNotVisible2, setInlineViewNotVisible2] =
      React.useState(true);
const [spotViewWithBorder, setSpotViewWithBorder] =
      React.useState(false);
const inlineViewId1 = "inlineViewId1";
const inlineViewId2 = "inlineViewId2";
```

b. Inside React.useEffect, update event listener like below:

```
if (Platform.OS === 'ios') {
   iOSMessagingEvent.addListener(Constants.AD_LOADED,
   (obj: {[key: string]: string}) => {
      const event = obj['eventType'];
      const receivedSpotId = obj['spotId']; //not used
      const receivedViewId = obj['viewId'];
      if (event === Constants.TYPE_INTERSTITIAL_AD) {
        Toast.show('Interstitial Ad view is loaded', Toast.SHORT);
      } else if (event === Constants.TYPE_INLINE_AD) {
        Toast.show('Inline Ad view is loaded', Toast.SHORT);
        if (receivedViewId === inlineViewId1) {
           setInlineViewNotVisible(false);
        }
        if (receivedViewId === inlineViewId2) {
           setInlineViewNotVisible2(false);
        }
        if (receivedSpotId === 'cuteDogSpot_WW') {
```

```
            setSpotViewWithBorder(true);
          }
        }
      });
      iOSMessagingEvent.addListener(Constants.AD_DEFAULT_LOADED,
      (obj: {[key: string]: string}) => {
        const event = obj['eventType'];
        const receivedSpotId = obj['spotId']; // not used
        const receivedViewId = obj['viewId'];
        if (event === Constants.TYPE_INTERSTITIAL_AD) {
           Toast.show('Interstitial Ad default view is loaded',
             Toast.SHORT);
        } else if (event === Constants.TYPE_INLINE_AD) {
          Toast.show('Inline Ad default view is loaded',
             Toast.SHORT);
        if (receivedViewId === inlineViewId1) {
           setInlineViewNotVisible(true);
        }
        if (receivedViewId === inlineViewId2) {
           setInlineViewNotVisible2(true);
        }
      }
    });
  } else if (Platform.OS == 'android') {
    DeviceEventEmitter.addListener(Constants.AD_LOADED,
    (obj: {[key: string]: string}) => {
      const event = obj['eventType'];
      const receivedViewId = obj['viewId'];
      const receivedSpotId = obj['spotId'];
      if (event === Constants.TYPE_INTERSTITIAL_AD) {
         Toast.show('Interstitial Ad view is loaded', Toast.SHORT);
      } else if (event === Constants.TYPE_INLINE_AD) {
      if (receivedViewId === inlineViewId1) {
         setInlineViewNotVisible(false);
       }
```

```
        if (receivedViewId === inlineViewId2) {
           setInlineViewNotVisible2(false);
        }
        if (receivedSpotId === 'cuteDogSpot_WW') {
           setSpotViewWithBorder(true);
        }
        Toast.show('Inline Ad view is loaded', Toast.SHORT);
      }
    });
    DeviceEventEmitter.addListener(Constants.AD_DEFAULT_LOADED,
    (obj: {[key: string]: string}) => {
      const event = obj['eventType'];
      const receivedViewId = obj['viewId'];
      if (event === Constants.TYPE_INTERSTITIAL_AD) {
         Toast.show('Interstitial Ad default view is loaded',
         Toast.SHORT);
      } else if (event === Constants.TYPE_INLINE_AD) {
        Toast.show('Inline Ad default view is loaded', Toast.SHORT);
        if (receivedViewId === inlineViewId1) {
           setInlineViewNotVisible(true);
        }
        if (receivedViewId === inlineViewId2) {
           setInlineViewNotVisible2(true);
        }
      }
    });
    // The rest is omitted
```

c. Inside <View></View>, add two InlineAdView:

```
  <View style={inlineViewNotVisible ?
    styles.inlineViewContainerNoContent :
    spotViewWithBorder? styles.inlineViewContainerWithBorder :

    styles.inlineViewContainerWithContent}>
    <InlineAdView
      spotId={spotId}
```

```
    viewId={inlineViewId1}
    notVisible={inlineViewNotVisible}
    style={ inlineViewNotVisible ? styles.inlineViewNoContent :
      spotViewWithBorder? styles.inlineViewContainerWithBorder :
      styles.inlineViewWithContent} />
</View>
{!inlineViewNotVisible && <View style={styles.spacer} />}
<View style={inlineViewNotVisible2 ?
  styles.inlineViewContainerNoContent :
  spotViewWithBorder? styles.inlineViewContainerWithBorder :
  styles.inlineViewContainerWithContent}>
  <InlineAdView
     spotId={spotId2}
     viewId={inlineViewId2}
     notVisible={inlineViewNotVisible2}
     style={ inlineViewNotVisible2 ? styles.inlineViewNoContent :
       styles.inlineViewWithContent} />
 </View>
```

Note: For styling of the views, please see the example project.

2. Update the existing SpotsScreen.tsx to add viewId to InlineAdView and InterstitialAdView, update event emitter's addListener function to change the listener function's parameter to be an object as in Spots2Screen.tsx.

3. Update App.tsx to include Spots2Screen.

4. Add useIFocused hook in both SpotsScreen.tsx and Spots2Screen.tsx. This is because both SpotScreen and Spot2Screen listen for mobile spots events. But we only want the active (focused) screen to listen. useIsFocused actually does not work in Android. In iOS, it is not completely accurate. So, you should find a better solution for this issue. Since this is not the focus of the cookbbook, no further action is taken.

Note: Consult the finished example project for how to perform steps 2-4.

March 2024 Updates

Mobile Spot

The native SASCollector iOS SDK (1.74.0) and Android SDK (1.82.0) added a new feature that allows developers to use resources such as fonts and icons from inside their apps to style their mobile spots. However, because React native properties passed to the native side are a map data structure, we cannot assume an order of setting the properties. But SASCollector SDK needs to know if there is the intention to use local resources before loading the spots to ensure the local resources are used. For this reason, and to also keep the original functionality (i.e., the functionality before the update), a new mobile spot view will be created in this update.

Configure Android

1. Create InlineAdViewWithLocalResourcesManager.java in android/src/main/java/com/mobilesdkreactnative/views. This class and its inner InlineAdViewWithLocalResources class are similar to the original classes. In the following steps, mostly only the differences will be pointed out. Check out the example project for the complete code.

2. In InlineAdViewWithLocalResourcesManager, make these changes:

a. Add these fields:

```
private boolean isSpotIdSet;
private boolean isUseLocalResourcesSet;
private boolean isViewLoaded;
private boolean isResourcePathSet;
public static final String NAME =
"InlineAdViewWithLocalResources";
```

b. Add this method to set spotId property:

```
@ReactProp(name = "spotId")
public void
setSpotId(InlineAdViewManagerLocalResources.InlineAdViewWithLocal
Resources inlineAdView, String spotID) {
  isViewLoaded = false;
  inlineAdView.setSpotId(spotID);
  isSpotIdSet = true;
  if (isSpotIdSet && isUseLocalResourcesSet && isResourcePathSet

    && !isViewLoaded) {
    inlineAdView.loadWithLocalResources();
    isViewLoaded = true;
```

```
    }
}
```

c. Add this method to set useLocResource property:

```
@ReactProp(name="useLocResources")
public void setUseLocResources(InlineAdViewWithLocalResources

  inlineAdView, boolean useLocalResources) {
  isViewLoaded = false;
  inlineAdView.setUseLocalResources(useLocalResources);
  isUseLocalResourcesSet = true;
  if (isOKToLoadSpot()) {
    inlineAdView.loadWithLocalResources();
    isViewLoaded = true;
  }
}
```

d. Add this method to set resourcePath property:

```
@ReactProp(name="resourcePath")
public void setResourcePath(InlineAdViewWithLocalResources
inlineAdView, String path) {
  isViewLoaded = false;
  inlineAdView.setResourcePath(path);
  isResourcePathSet = true;
  if (isOKToLoadSpot()) {
    inlineAdView.loadWithLocalResources();
    isViewLoaded = true;
  }
}
```

e. Add this method to determine when to load mobile spot:

```
private boolean isOKToLoadSpot() {
  if (isSpotIdSet && isUseLocalResourcesSet && isResourcePathSet

  && !isViewLoaded)
    return true;
  return false;
}
```

3. In InlineAdViewWithLocalResources, make these changes:

a. Add these fields in addition to those that are in InlineAdView:

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

b. Add these methods:

```
public void setUseLocalResources(boolean useLocalResources) {
  this.useLocalResources = useLocalResources;
}
public void setResourcePath(String path) {
  this.resourcePath = path;
}
public void loadWithLocalResources() {
  this.useLocalResources(useLocalResources, resourcePath);
  this.load(spotId, null);
}
```

Configure iOS

1. Create the .h and .m files of InlineAdViewWithLocalResources in ios/views. InlineAdViewWithLocalResources is similar to InlineAdView, but has two more properties useLocalResources and resourcePath. In the following steps, mostly only the differences are shown. Check out the example project for the complete code.

2. In InlineAdViewWithLocalResources.h, declare these five properties:

```
@property (nonatomic, copy) NSString *spotId;

@property (nonatomic, copy) NSString *viewId;

@property (nonatomic) BOOL notVisible;

@property (nonatomic) BOOL useLocResources;

@property (nonatomic, copy) NSString *resourcePath;
```

3. In InlineAdViewWithLocalResources.m, make these changes:

a. In @interface section add these variables:

```
NSString* _spotID;
NSString* _viewID;
BOOL _notVisible;
BOOL _useLocResources;
NSString *_resourcePath;
BOOL isSpotIdSet;
BOOL isUseLocResourcesSet;
BOOL isViewLoaded;
```

b. Add this method:

```
-(void)setSpotId:(NSString *)spotID {
  _spotID = spotID;
  isViewLoaded = NO;
  super.spotID = _spotID;
  isSpotIdSet = YES;
```

```
    if (isSpotIdSet && isUseLocResourcesSet && !isViewLoaded) {
      [self setLocResourcesAndLoad];
      isViewLoaded = YES;
    }
  }
```

c. Add these methods to get and set useLocResources property:

```
-(BOOL)useLocResources {
  return _useLocResources;
}
-(void)setUseLocResources:(BOOL)useLocalResources {
  _useLocResources = useLocalResources;
  isViewLoaded = NO;
  isUseLocResourcesSet = YES;
  if (isSpotIdSet && isUseLocResourcesSet && !isViewLoaded) {
    [self setLocResourcesAndLoad];
    isViewLoaded = YES;
  }
}
```

d. Add these methods to get and set resourcePath property. This property is not used but is for compatibility with Android. In iOS, files in different folders are flattened in the same project, so there is no notion of folders in the compiled project.

```
-(NSString*)resourcePath {

  return _resourcePath;

}

-(void)setResourcePath:(NSString *)path {

  _resourcePath = path;

}
```

4. Create the .h and .m files of InlineAdViewWithLocalResourcesManager in ios/views. InlineAdViewWithLocalResourcesManager is similar to InlineAdView,Manger but has two more exported properties.

5. In InlineAdViewWithLocalResourcesManager.m, add these exported properties:

```
RCT_EXPORT_VIEW_PROPERTY(spotId, NSString)

RCT_EXPORT_VIEW_PROPERTY(useLocResources, BOOL)

RCT_EXPORT_VIEW_PROPERTY(resourcePath, NSString)

RCT_EXPORT_VIEW_PROPERTY(viewId, NSString)
```

```
RCT_EXPORT_VIEW_PROPERTY(notVisible, BOOL)
```

6. After the above files are created, in VSCode integrated terminal, go to example/ios folder and run the following code to make them logically added in MobileSdkReactNative library:

```
pod install
```

Configure React Native (Typescript)

Update the mobile-sdk-react-native library

1. Create InlineAdViewWithLocalResources.tsx file in src/views folder and add this code in the file.

```
import { requireNativeComponent, UIManager, ViewStyle, } from
"react-native";

type Props = {

spotId: string;

useLocResources: boolean;

resourcePath: string;

viewId: string;

notVisible: boolean;

style: ViewStyle;

};

const ComponentName = 'InlineAdViewWithLocalResources';

const LINKING_ERROR = `The ${ComponentName} does not seem to be
linked`;

const InlineAdViewWithLocalResources =

UIManager.getViewManagerConfig(ComponentName) != null

? requireNativeComponent<Props>(ComponentName)

: () => {
```

```
throw new Error(LINKING_ERROR);

};

export default InlineAdViewWithLocalResources;
```

2. In src/index.tsx file, add this code:

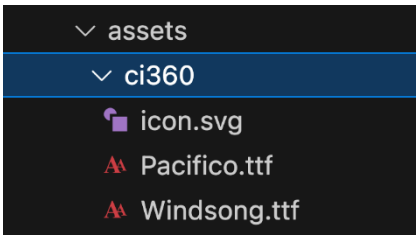
```
import InlineAdViewWithLocalResources from
'./views/InlineAdViewWithLocalResources';

export {InlineAdViewWithLocalResources};
```

Update the example app

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

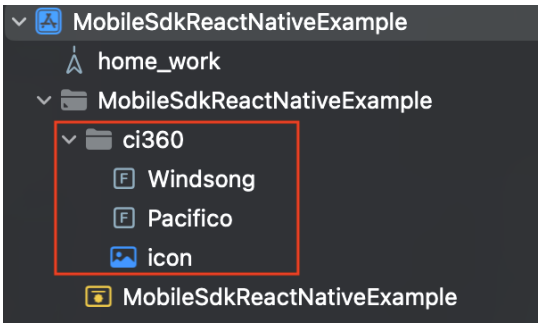
```
<style>
  @font-face {
    font-family: Pacifico;
    src: url('Pacifico.ttf')
  }
  @font-face {
    font-family: Windsong;
    src: url('Windsong.ttf')
  }
...
</style>
```

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

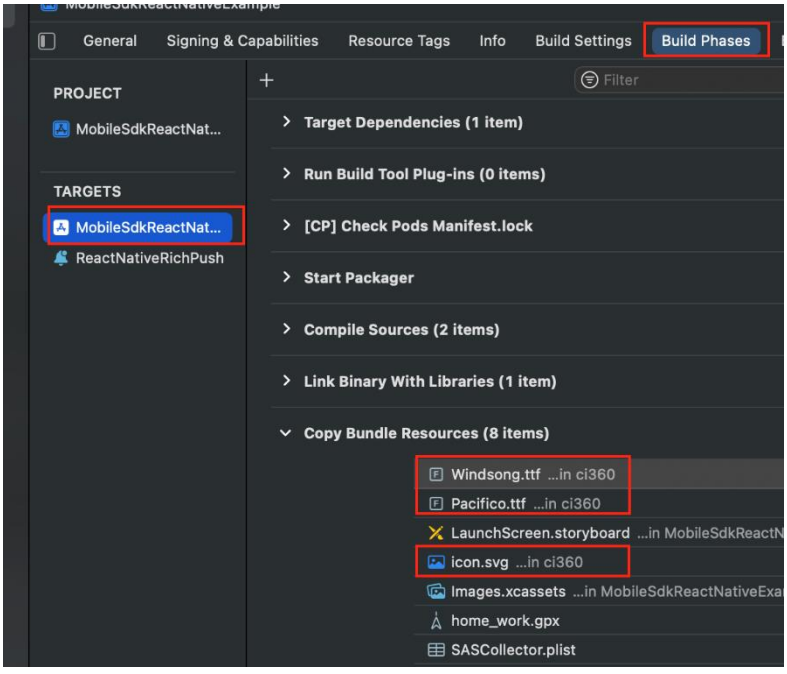


3. Open the example app project in Xcode and makes these updates:

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



b. In MobileSdkReactNativeExample target, choose build phases and make sure the fonts and icon files are in “Copy Bundle Resources”. If not, then click “+” button to add them.



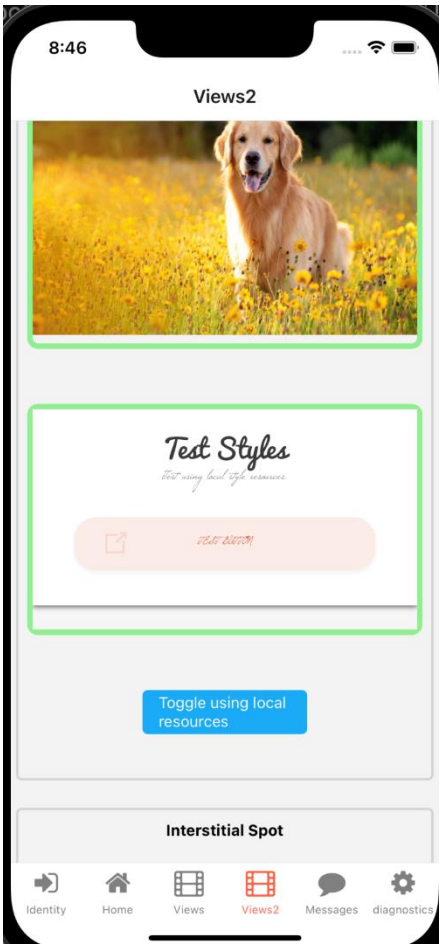
4. Add import and set state for the toggle button:

```
import { InlineAdViewWithLocalResources} from
  'mobile-sdk-react-native';
const [isUseLocResource, setIsUseLocResource] =
  React.useState(true);
```

5. Add this code after the second InlineAdView in example/src/screens/Spots2Screens.tsx. Please check the example project for the styles.

```
<View style={styles.inlineViewShortContainerWithBorder}>
  <InlineAdViewWithLocalResources
    spotId={spotId3}
    useLocResources={isUseLocResource}
    resourcePath='ci360'
    viewId={inlineViewId3}
    notVisible={false}
    style={styles.inlineViewWithContentShort}/>
</View>
<CustomButton title='Toggle using local resources'
  onPress={() => {
    setIsUseLocResource(!isUseLocResource);
  }}
  width={{width: 200}} />
```

After running the example, the mobile spot screen looks like below. The center mobile spot that has title “Test Styles” uses the local fonts and icon. Clicking the button below will toggle using the local resources.



Note: This update only provides instructions on how to create an inline mobile spot. If you are interested in creating an interstitial mobile spot, you can follow the same steps.

Setting Application Version Programmatically

In addition to the mobile spot update, the iOS SDK also fixed a bug that made it impossible to set application version programmatically. Currently the cookbook also does not include instructions on how to set application version programmatically either. This is addressed below. Please note that the goal of the following steps is not only to set application versions, but also to have application versions updated in UDM session_details table. Updating session_details table is not an explicit step in the following steps, but an implicit one performed in the backend.

Configure the mobile-sdk-react-native library

This will include changes on Android, iOS and React Native (typescript).

Configure Android

1. Make sure SASCollector.jar 1.82.0 is included in android/libs folder.

2. In android/build.gradle, add one or more of these dependencies:

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

3. Remove setPushChannel method and the call to setPushChannel in example/android/app/src/java/com/example/mobilesdkreactnative/MainApplication.java.

4. In android/src/main/java/com/mobilesdkreactnative/MobileSdkReactNativeModule.java, make these changes:

a. Add Firebase import:

```
import com.google.firebase.messaging.FirebaseMessaging;
```

b. Update the constructor:

```
public MobileSdkReactNativeModule(ReactApplicationContext

        reactContext) {
  super(reactContext);
}
```

c. Move setPushChannel method from step 3, and make a small change as shown in bod below:

```
@RequiresApi(api = Build.VERSION_CODES.O)
private void setPushChannel() {
  NotificationManager notificationManager =(NotificationManager)

    getCurrentActivity().getSystemService(

    getCurrentActivity().NOTIFICATION_SERVICE);

  String customAndroidChannel = "ReactNativePushChannel";
  CharSequence channelName = "React Native Channel";
  int importance = NotificationManager.IMPORTANCE_HIGH;
  NotificationChannel notificationChannel =

    new NotificationChannel(
```

```
    customAndroidChannel, channelName, importance);
  notificationChannel.enableLights(true);
  notificationChannel.setLightColor(Color.RED);
  notificationChannel.enableVibration(true);
  notificationChannel.setShowBadge(true);
  notificationChannel.setVibrationPattern(

    new long[]{100, 200, 300, 400, 500, 400, 300, 200, 400});
  notificationManager.createNotificationChannel(

    notificationChannel);
  SASCollector.getInstance().setPushNotificationChannelId(

    customAndroidChannel);
}
```

c. Add this method:

```
@ReactMethod
public void setAppVersionAndInitSDK(String appVersion) {
  if (appVersion.matches("^\\d+\\.\\d+\\.\\d+$")) {
    SASCollector.getInstance().setApplicationVersion(appVersion);
  } else {

    //if appVersion does not have the correct format, use 0.0.1

    // as the default version. Change it to whatever you like
    SASCollector.getInstance().setApplicationVersion("0.0.1");
  }
  SASCollector.getInstance().initialize(getCurrentActivity());
  FirebaseMessaging.getInstance().getToken()

    .addOnSuccessListener(token -> {
    Log.d("SASModule", "token="+token);
    if(!TextUtils.isEmpty(token)) {
     SASCollector.getInstance().registerForMobileMessages(token);
    }
  });

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

Configure iOS

1. Make sure SASCollector.xcframework 1.74.0 is included in ios folder.

2. In ios/MobileSdkReactNative.mm, add this code:

```
RCT_EXPORT_METHOD(setAppVersionAndInitSDK:(NSString*)appVersion) {
  NSRegularExpression *regExpr = [NSRegularExpression
    regularExpressionWithPattern:@"^[0-9]+\\.[0-9]+\\.[0-9]+$"
    options:NSRegularExpressionCaseInsensitive error:nil];
  NSRange range = [regExpr rangeOfFirstMatchInString:appVersion
    options:NSMatchingProgress
    range:NSMakeRange(0, appVersion.length)];
  if (range.location != NSNotFound) {
     [SASCollector setApplicationVersion:appVersion];
  } else {
     [SASCollector setApplicationVersion:@"0.0.1"];
  }
  [SASCollector initializeCollection];
}
```

3. In ios/Constants.h, add this code:

```
FOUNDATION_EXPORT NSString *const REGISTER_DEVICE_TOKEN;
```

4. In ios/Constants.m, add this code:

```
NSString *const REGISTER_DEVICE_TOKEN = @"onRegisterDeviceToken";
```

5. In ios/SASMobileMessagingEvent.h, add this method declaration:

```
+(void)emitMessageDeviceToken:(NSDictionary *)payload;
```

6. In ios/SASMobileMessagingEvent.m, update supportedEvents methdo:

```
- (NSArray<NSString *> *)supportedEvents {
  return @[@"onMessageOpened", @"onMessageDismissed",
           @"onRegisterDeviceToken"];
}
```

7. Still in ios/SASMobileMessagingEvent.m, add these methods:

```
-(void)onRegisterDeviceToken:(NSNotification*)notification {
  NSDictionary *args = notification.userInfo;
  NSData *token = args[@"deviceToken"];
  NSString *tokenStr = [SASMobileMessagingEvent
                        dataToHexStr:token];
  NSDictionary *tokenInfo = @{@"deviceToken": tokenStr};
  [self sendEventWithName:REGISTER_DEVICE_TOKEN body:tokenInfo];
}
+(void)emitMessageDeviceToken:(NSDictionary *)payload {
   [[NSNotificationCenter defaultCenter]
     postNotificationName:REGISTER_DEVICE_TOKEN
     object:self  userInfo:payload];
}
+(NSString*)dataToHexStr:(NSData*)data {
   NSMutableString *str = [NSMutableString stringWithCapacity:64];
   NSUInteger length = [data length];
   char *bytes = malloc(sizeof(char)*length);
   [data getBytes:bytes length:length];
```

```
    for(int i=0; i<length; i++) {
      [str appendFormat:@"%02.2hhX", bytes[i]];
    }
    free(bytes);
    return str;
}
```

8. Still in ios/SASMobileMessagingEvent.m, update `startObserving` to add this code:

```
[[NSNotificationCenter defaultCenter] addObserver:self
  selector:@selector(onRegisterDeviceToken:)
  name:REGISTER_DEVICE_TOKEN object:nil];
```

Configure React Native (Typescript)

1. In src/index.tsx, add this code:

```
export function setAppVersionAndInitSDK(appVersion: string){
  MobileSdkReactNative.setAppVersionAndInitSDK(appVersion);
}
```

2. In src/Constants.tsx, add this constant definition:

```
export const REGISTER_DEVICE_TOKEN = "onRegisterDeviceToken";
```

Configure the example application

This will include changes in Android, iOS and React Native (typescript).

Configure Android

1. In example/android/app/src/main/java/com/example/mobilesdkreactnative, update MainApplication.java's onCreate method to remove this code:

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

2. In In example/android/app/src/main/java/com/example/mobilesdkreactnative, update MainActivity.java's onCreate method to remove this code:

```
FirebaseMessaging.getInstance().getToken()

  .addOnSuccessListener(token -> {
  Log.d(TAG, "token="+token);
  if(!TextUtils.isEmpty(token)) {
    SASCollector.getInstance().registerForMobileMessages(token);
  }
 });
```

Configure iOS

1. In example/ios/SASCollector.plist, update `developerInitialized` as below. You can also update in Xcode.

```
developerInitialized = true;
```

2. In example/ios/MobileSdkReactNativeExample/AppDelegate.mm, make the following changes:

a. At the start of `@implementation AppDelegate`, add these variables:

```
NSData *deviceTokenForNotification;

BOOL hasRegisteredDeviceToken;
```

b. Update didRegisterForRemoteNotificationWithDeviceToken as below:

```
- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData
*)deviceToken {

    hasRegisteredDeviceToken = NO;

    deviceTokenForNotification = deviceToken;

}
```

Note: In b, we removed the previous SASCollector method call because SASCollector SDK is not initialized, and so it cannot register device token. For this reason, we have to keep track of the

device token and pass it to SASCollector later in step c. In addition, to not repeatitively register device token, we used the boolean variable hasRegisteredDeviceToken.

c. Implement the below method:

```
- (void)applicationWillResignActive:(UIApplication *)application
{
 if (!hasRegisteredDeviceToken) {
    NSDictionary *args = @{@"deviceToken":
                          deviceTokenForNotification};
    [SASMobileMessagingEvent emitMessageDeviceToken:args];
     hasRegisteredDeviceToken = YES;
   }
 }
```

Note: The iOS side completely finishes initialization in (AppDelegate) before the React Native side starts. For this reason, we cannot send the event to the React Native side in any of the initialization method, such as didFinishLaunchingWithOptions, didBecomeActive, but have to send the event when the application is going to become inactive. You may also choose to send the event in other lifecycle methods, as long as the React Native side can receive it.

Configure React Native (typescript)

1. In example/src/App.tsx, make these changes:

a. update imports from mobile-sdk-react-native as below:

```
import { SASMobileMessagingEvent, Constants,
       startMonitoringLocation, setAppVersionAndInitSDK,
       registerForMobileMessage } from 'mobile-sdk-react-native';
```

b. In App function, add the bolded code in React.useEffect:

```
  React.useEffect(() => {

    //...

    if (Platform.OS === 'android') {

      setAppVersionAndInitSDK("1.0.1");

      //...

    } else if (Platform.OS === 'ios') {
setAppVersionAndInitSDK("1.2.2");

     iOSMessagingEvent.addListener(Constants.REGISTER_DEVICE_TOKEN,

       data => {

         console.log('tokenData: ' + data);

         const token = data.deviceToken;

         registerForMobileMessage(token);

       });

       //...

     }

    //...

    });
```

Note: In the above code, I assume the Android and iOS apps have different versions. Change the versions to whatever you want.

Optional SASMobileMessagingDelegate2

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

Configure iOS

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

```
@interface AppDelegate : UIResponder <UIApplicationDelegate,

   UNUserNotificationCenterDelegate, RCTBridgeDelegate>

     @property (nonatomic, strong) UIWindow *window;

@end
```

2. In In example/ios/MobileSdkReactNativeExample/AppDelegate.mm, remove SASMobileMessagingDelegate2's delegate methods:

```
- (void)messageDismissed

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