Android SDK 5.x.x to 6.0.0 Migration Guide

This guide is intended for app developers who have been using the Java-based Alchemer Mobile (Formerly Apptentive) Android SDK and are migrating to the new Kotlin-based Android SDK.

> This guide is not intended for beta releases

Major Differences

The new Android SDK is a ground-up rewrite of the Java Alchemer Mobile Android SDK to Kotlin using modern programming practices.

A few quick notes for getting started on the migration:

- We now have a Kotlin codebase instead of Java
- The screen's `Activity` is now registered separately to the SDK.
  - This allows the `engage` function to be called without needing to provide an `Activity` `context`.
- Callback types are different when you engage events and when you register the SDK
- Modular-based SDK
- Minimum supported Android version is now Android 5.0 (API level 21)
- Minimum supported compileSDK version is now 31 as per google's requirement
- Message Center is in **full release** version of 6.0.0+

New Features

- Interaction Response Targeting

Improvements

- Interface customization
- ADA compliant interactions
- Dark mode support
- We are now using Material Design's Material Components within our SDK (designed to be used with Material Design 2, but Material 3 will also work)
  - If you haven't already, you will need to update your app to use Material Components and AndroidX
    - This should be a simple process and is highly recommended.
    - There are Bridge themes available if you cannot inherit from the `MaterialComponents` theme

Current limitations

Most styles in 5.8.4 and prior have **been deprecated and replaced** with new styles in 6.0.0.

Some styles have been moved over, such as many text size, text color, and typeface styles, but the vast majority of styles have been changed.

The new styles have much better **flexibility**, many more **options**, better **organization**, and better naming **patterns**.

If you are starting fresh, the possibilities are huge. Check out the UI Cookbook for examples.

Refer to the Interface Customization article for more help.

We do not recommend you migrate *yet* if you **need** any of the four features listed below.

These features will be implemented in the near future.

- Encryption is not yet supported
  - **Data migration will currently fail if you have used Encryption**
- Client Authentication (Multi-User / Login / Logout) is not yet supported
- Plugin wrappers
- Push notifications for Message Center

Java Apps

Due to Kotlin to Java interoperability limitations, our internal APIs are exposed when integrating from Java apps. We do not recommend you directly use our internal and `@InternalUseOnly` annotated fields as they are subject to change without notice.

Video Overview of How to Migrate to 6.0.0

Your browser does not support HTML5 video.

Dependency Implementation

In your `build.gradle` file, add the following dependency to integrate Alchemer Mobile SDK, replacing `APPTENTIVE_VERSION` with the most recent one from here:

```
dependencies {
  implementation "com.apptentive:apptentive-kit-android:APPTENTIVE_VERSION"
}
```

Registering the SDK

Shortly after the app launch, you'll want your app to call the register method. This should usually happen within your `Application` class. You'll create an `ApptentiveConfiguration` object to pass in the Alchemer Mobile **App Key** and **Alchemer Mobile (Formerly Apptentive) App Signature** values from the **API & Development** section of the **Settings** tab in your Alchemer Mobile **dashboard**. For more information on the optional `ApptentiveConfiguration` parameters, see the Configuration optional parameters section.

Kotlin example:

```
class MyApplication : Application() {
  override fun onCreate() {
    super.onCreate()
    val configuration = ApptentiveConfiguration(
    apptentiveKey = "<YOUR_APPTENTIVE_KEY>",
    apptentiveSignature = "<YOUR_APPTENTIVE_SIGNATURE>"
    ).apply {
      /**
      * Optional parameters:
      * shouldInheritAppTheme - Default is true
      * logLevel - Default is LogLevel.Info
      * shouldSanitizeLogMessages - Default is true
      * ratingInteractionThrottleLength - Default is TimeUnit.DAYS.toMillis(7)
      * customAppStoreURL - Default is null (Rating Interaction attempts to show Google In-App Review)
      */
    }
    Apptentive.register(this, configuration)
  }
}
```

Java example:

```
public class MyApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();
    /*
    * Optional parameters:
    * shouldInheritAppTheme - Default is true
    * logLevel - Default is LogLevel.Info
    * shouldSanitizeLogMessages - Default is true
    * ratingInteractionThrottleLength - Default is TimeUnit.DAYS.toMillis(7)
    * customAppStoreURL - Default is null (Rating Interaction attempts to show Google In-App Review)
    */
    ApptentiveConfiguration configuration = new ApptentiveConfiguration(
      "<YOUR_APPTENTIVE_KEY>",
      "<YOUR_APPTENTIVE_SIGNATURE>"
    );

    Apptentive.register(this, configuration);
  }
}
```

If you didn't already have an `Application` class defined in your app, you will need to create one and add it to your Manifest.

Using Registration Callbacks

Our registration callbacks help you debug the Alchemer Mobile SDK.

Kotlin example:

```
Apptentive.register(this, configuration) {
  when (it) {
    is RegisterResult.Success -> Log.i(
      "Apptentive Registration",
      "Registration successful"
    )
    is RegisterResult.Failure -> Log.w(
      "Apptentive Registration",
      "Registration failed with response code: ${it.responseCode} and error message: ${it.message}"
    )
    is RegisterResult.Exception -> Log.e(
      "Apptentive Registration",
      "Registration failed with exception",
      it.error
    )
  }
}
```

Java example:

```
Apptentive.register(this, configuration, result -> {
  if (result instanceof RegisterResult.Success) {
    Log.i("Apptentive Registration", "Registration successful");
  } else if (result instanceof RegisterResult.Failure) {
    RegisterResult.Failure resultFailure = (RegisterResult.Failure) result;
    Log.w(
      "Apptentive Registration",
      "Registration failed with response code: " + resultFailure.getResponseCode() +
      ", and error message: " + resultFailure.getMessage()
    );
  } else if (result instanceof RegisterResult.Exception) {
    RegisterResult.Exception resultError = (RegisterResult.Exception) result;
    Log.e("Apptentive Registration", "Registration failed with an exception", resultError.getError());
  }
});
```

Registering/Unregistering the Activity Callback to the SDK

With the re-write of the SDK, we leveraged a modern architecture using modules. This allows for greater flexibility in where the `engage` method can be called from.

The Alchemer Mobile SDK now needs to register the `Activity` to the SDK in order to show our Interactions in your application.

This will need to be done for every `Activity` within your application.

> We recommend that you implement this within a `BaseActivity` that your other Activities can extend from or use Android Jetpack Navigation. This will enable you to implement this a minimum number of times.

This is a 4 step process:

1. Extend the `ApptentiveActivityInfo` interface to your `Activity`
2. Override the `getApptentiveActivityInfo` function
   a. You will just return `this` (the `Activity`)
3. Register the callback with the Alchemer Mobile SDK using the `registerApptentiveActivityInfoCallback` function within your `onResume` function (after the `super`)
4. Unregister the callback using `unregisterApptentiveActivityInfoCallback` function within your `onPause` function(before the `super`)

Kotlin example:

```
class MainActivity : AppCompatActivity(), ApptentiveActivityInfo {
    override fun onResume() {
        super.onResume()
        Apptentive.registerApptentiveActivityInfoCallback(this)
    }

    override fun getApptentiveActivityInfo(): Activity {
        return this
    }

    override fun onPause() {
        Apptentive.unregisterApptentiveActivityInfoCallback()
        super.onPause()
    }
}
```

Java example:

```
public class MainActivity extends AppCompatActivity implements ApptentiveActivityInfo {
    @Override
    protected void onResume() {
        super.onResume();
        Apptentive.registerApptentiveActivityInfoCallback(this);
    }

    @NonNull
    @Override
    public Activity getApptentiveActivityInfo() {
        return this;
    }

    @Override
    protected void onPause() {
        Apptentive.unregisterApptentiveActivityInfoCallback();
        super.onPause();
    }
}
```

Engaging Events

At various points in your app's lifecycle, you will want to engage events with the Alchemer Mobile

SDK in order to allow the targeting and the launch of Alchemer Mobile interactions (Surveys, Prompts, Love Dialog, etc.).

`Events` can be added *almost** anywhere in the App. A few good places would be when an `Activity` comes to focus, on a button tap or when the App encounters an error.

> Avoid calling engage events in your `Application` class or before the SDK has a chance to get fully registered.

Kotlin example:

```
// Engaging
Apptentive.engage("my_event")

// Engaging with callback (optional)
Apptentive.engage("my_event") { result ->
    when (result) {
        is EngagementResult.InteractionShown -> { /* Interaction was shown */ }
        is EngagementResult.InteractionNotShown -> { /* Interaction was NOT shown */ }
        is EngagementResult.Error -> { /* There was an error during evaluation */ }
        is EngagementResult.Exception -> { /* Something went wrong */ }
    }
}
```

Java example:

```
// Engaging
Apptentive.engage("my_event");

// Engaging with callback (optional) & customData (optional)
Apptentive.engage("my_event", customData, result -> {
    if (result instanceof EngagementResult.InteractionShown) {
        /* Interaction was shown */
    } else if (result instanceof EngagementResult.InteractionNotShown) {
        /* Interaction was NOT shown */
    } else if (result instanceof EngagementResult.Error) {
        /* There was an error during evaluation */
    } else if (result instanceof EngagementResult.Exception) {
        /* Something went wrong */
    }
});
```

Alchemer Mobile Configuration optional parameters

The configuration options are enabled with defaults that are production-ready.

1. `shouldInheritAppTheme`
    a. A `boolean` that determines if Alchemer Mobile Interactions will use the host app's theme or Alchemer Mobile's theme.

    b. If `true`, Alchemer Mobile Interactions will use the host app's theme and colors.

    c. If `false`, Alchemer Mobile Interactions will use the `Theme.Apptentive` theme and colors, set

in the styles.xml file within the `apptentive-core-ui` package.

d. `ApptentiveThemeOverride` will always take priority over all themes.

e. See ThemeHelper.kt within the `apptentive-core-ui` package for more info.

2. `logLevel`
   a. An `enum` used to define what level of logs we will show in Android Studio's Logcat

   b. Default is `LogLevel.Info`

   c. Log level options are
      i. `LogLevel.Verbose`
         i. Any relevant info not shown in other log levels

      ii. `LogLevel.Debug`
         i. Processes with more technical information

      iii. `LogLevel.Info`
         i. General processes and non-technical results

      iv. `LogLevel.Warning`
         i. Non-breaking / handled issues

      v. `LogLevel.Error`
         i. Breaking / unhandled issues (Throwables)

3. `shouldSanitizeLogMessages`
   a. A `boolean` that declares whether or not to convert the data of variables using the `@SensitiveDataKey` annotation to `<REDACTED>`

   b. When `true` (default)
      i. Redacts sensitive information from Logcat

      ii. Redacted information includes:
         i. Alchemer Mobile (Apptentive) Key & Signature

         ii. Conversation Token

         iii. Server call
            i. Headers

            ii. Request Body

            iii. Response Body

         iv. mParticle IDs

v. Custom Data

vi. Legacy data while being converted

vii. SDK Author name and email

c. When set to `false`
   i. All info above is available to be logged to Android Studio's Logcat
      i. Whether or not it is logged depends on the set `logLevel`

4. `ratingInteractionThrottleLength`
   a. A millisecond based rating interaction throttle

   b. Default is 7 days
      i. We are using `TimeUnit` of the `java.util.concurrent` library to help with Day to Millisecond conversions.
         i. `TimeUnit.DAYS.toMillis(7)`

   c. Hard limits the frequency that the user will see the Rating Interaction in a specified period of time

   d. This is an internal throttle safeguard outside of the defined limitations set in the dashboard

5. `customAppStoreURL`
   a. A `String` based variable that is used for alternate app store ratings.

   b. If `null` (default), the user will see the Google In-App Review and will be able to leave an app rating directly to the Play Store without leaving the app.

   c. If set to a `String`, the SDK will ignore the Google In-App Review Interaction for the Alchemer Mobile Rating Dialog Interaction and will attempt to send the user to the specified URL if the user chooses to rate the app.

Message Center

Message Center is largely the same functionality as before, with the current limitation of no push notifications (coming soon in a follow-up release)

```
 // Opening Message Center
Apptentive.showMessageCenter()

// Engaging with callback (optional)
Apptentive.showMessageCenter() { result ->
    when (result) {
        is EngagementResult.InteractionShown -> { /* Interaction was shown */ }
        is EngagementResult.InteractionNotShown -> { /* Interaction was NOT shown */ }
        is EngagementResult.Error -> { /* There was an error during evaluation */ }
        is EngagementResult.Exception -> { /* Something went wrong */ }
    }
}
```

Styling

Migrating styling will be familiar, yet different. We have opened up more options for styling and kept most of the styling parameters while removing others in order to better support a wider range of styles.

Here's an article on our interface customization currently available and how the default interaction UIs look.

Prerequisites

- If you haven't already, you will need to update your app to use Material Components and AndroidX
    - This should be a simple process and is highly recommended.
    - There are Bridge themes available if you cannot inherit from the MaterialComponents theme.

What's new?

- We are now using Material Design's Material Components within our SDK (designed to be used with Material Design 2, but Material 3 will also work)
- `shouldInheritAppTheme` will let you decide if you want to use Alchemer Mobile's colors or if we should inherit your app's colors
    - Default is `true`, which will inherit your app's colors
- Every layout, text, button, widget, and most other views on the screen now have their own customization attribute associated with it
- We have removed all hard-coded stylings from the views themselves in order to give greater freedom when overriding our styles
- We now have an alpha attribute that allows us to give text better visual priorities without needing separate similar text colors
    - These attributes are `apptentiveHeaderAlpha` and `apptentiveSubheaderAlpha`

Android UI Cookbook

The UI Cookbooks will share some custom designs and examples and how to translate them into

Alchemer Mobile interactions using styles.

The code for all 3 cookbook designs can also be found within the example app styles file.

- UI Cookbook Overview
  - UI Cookbook 1
  - UI Cookbook 2
  - UI Cookbook 3

Deprecated Styles

Since our UI was updated, not all styles were able to migrate over or they did not translate well into the new architecture with new customization options.

> **Most styles** in 5.8.4 and prior have **been deprecated and replaced** with new styles in 6.0.0.
>
> Some styles have been moved over, such as many text size, text color, and typeface styles, but the vast majority of styles have been changed.
>
> The new styles have much better **flexibility**, many more **options**, better **organization**, and better naming **patterns**.
>
> If you are starting fresh, the possibilities are huge. Check out the UI Cookbook for examples.
>
> Refer to the Interface Customization article for more help.

> Styles used within the `ApptentiveThemeOverride` style will only affect **Alchemer Mobile's** Interactions. You won't need to worry about style items like `colorPrimary` affecting your app's styles if it is placed within the `ApptentiveThemeOverride` style.
>
> Good alternatives to the deprecated styles are to override the view's style individually and set any style items you want to it.

Alchemer Mobile Code Resources

- Example app styles.xml file
  - Check here for samples on how to use `ApptentiveThemeOverride` and all the style items available.
- Alchemer Mobile default UI styles.xml file
  - Check here for the defaults that Alchemer Mobile is using to style the views
    - A good starting point for creating your custom UI
- Alchemer Mobile Attributes (apptentive-attrs.xml) file
  - Check here for all available attributes (the other two styles files above should also have all)

How to override an Alchemer Mobile style

1. Go to your `styles.xml` file (or create one)
2. Add an `ApptentiveThemeOverride` style
3. Include any `<item>` overrides within that style

```
<style name="ApptentiveThemeOverride">
  // Override items go here
</style>
```

Style Information

Style / Theme Hierarchy

1. `ApptentiveThemeOverride` (set only by the developer in their `styles.xml` file) takes priority over all other stylings
2. Second priority, we disable the style item `android:background`
   a. This is a problem style for many of our Material Design widgets
   b. We use `android:colorBackground` and `colorSurface` for background colors
   c. If you do need to use this, you can re-enable with `ApptentiveThemeOverride`
3. Third priority, we apply the host app's theme if `shouldInheritAppTheme` is set to true
4. Last priority, we apply Alchemer's theme
   a. If `shouldInheritAppTheme` is false and you don't have anything set in `ApptentiveThemeOverride`, you will be able to see Alchemer Mobile's default theme

See ThemeHelper.kt for more info.

Important styles we inherit

> We are following Material Theme guidelines for color styling
>
> These do not need to be set within `ApptentiveThemeOverride`, we will inherit them automatically

Dialog colors (Love Dialog, Prompts, Rating Dialog)

- Background uses `colorSurface`
- Non-button text uses `colorOnSurface`
- Button text uses `colorSecondary`

Survey colors

- Toolbar (top and bottom) background uses `colorPrimary`
- Toolbar (top and bottom) text uses `colorOnPrimary`

- Activity main background uses `android:colorBackground`
- Question Text uses `colorOnBackground`
- Question Widgets use a combination of `colorOnBackground` and `colorSecondary`
- Question Error Text uses `colorError`
- Submit button uses `colorPrimary`
- Submit button text uses `colorOnPrimary`

Logging

To set the severity of log messages shown in the logs, set the Alchemer Mobile `LogLevel`. The default is set to `LogLevel.Info`.

Kotlin example:

```
 val configuration = ApptentiveConfiguration(...)
configuration.logLevel = LogLevel.DEBUG
Apptentive.register(this, configuration)
```

Java example:

```
 ApptentiveConfiguration configuration = new ApptentiveConfiguration(...)
configuration.setLogLevel(LogLevel.Debug);
Apptentive.register(this, configuration);
```

Example App

example

A simple app to help customers integrate with the Alchemer Mobile Android SDK

Instructions

1. Download and open the example app in Android Studio
2. Click the TODO tab at the bottom left of Android Studio to follow step-by-step instructions on where and how to create a very basic Alchemer Mobile integrated Android app.

Related Articles