



Avaya In-App Messaging Android SDK - Development Guide

SDK Release 3.1.0
November 2022

Table of Contents

Introduction	5
What changed between SDK 1.0.3 and 3.1.0	5
Guidance on how to update settings during SDK upgrade	7
Color customization in v.3.1	8
Customisation of the header of the chat window	10
note and known limitations:	11
OS support notes:	11
Getting Started	15
API reference	15
QuickStart to your first conversation	16
Adding to your app	18
Initialize in your app	18
Initialize from an Activity class	19
Adding an Application class to your app	21
Displaying the User Interface	22
Location messages	22
Region configuration	23
Log in with userId and JWT	24
Authentication delegate	24
Configuring push notifications	25
Localization	27
Adding more languages	27
Customization	27
Strings customization	27
Menu items	28
Notification Action Intent Override	30
Notification Channel Settings Override	30
Starting Text	31
Extending ConversationActivity	31
Permissions	32
Appendix A – Users Authentication	33
Appendix B - Managing Users	37
Appendix C - Android FAQ	39
Appendix D – certificate configuration for file transfer feature	40

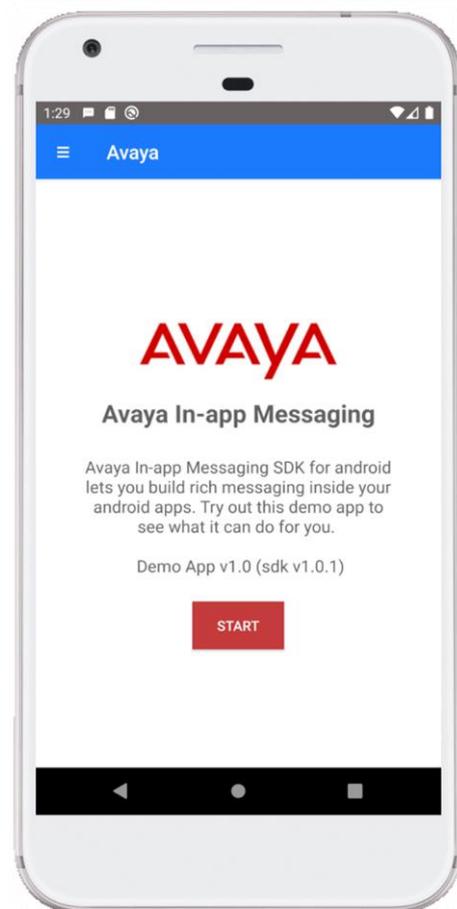


Android SDK

Current version: v3.1.0

The Avaya In-App Messaging android SDK is a highly customizable chat embeddable that can be added to any android app. It supports the Avaya IX Digital Connection API [capabilities](#), [push notifications](#) and is fully [localized](#).

- Introduction
- Capabilities
- Getting Started
 - SDK bundle
 - API reference
- QuickStart with Demo App
- Adding to your App
- Region Configuration
- Authenticating users
 - Authentication Delegate
- Configuring push notifications
- Localization
- Customization
- Permissions
- Appendices
 - Appendix A – Users Authentication
 - Appendix B - Managing Users
 - Appendix C - Android FAQ



Introduction

This document is a development guide and contains instructions and information for developers seeking to build In-app messaging applications on android platform which is supported by Avaya IX Digital Connection.

What is the Avaya IX Digital Connection?

The Avaya IX Digital Connection is a software platform that enables businesses to communicate with their customers across several popular messaging app.

Developers can use the Avaya IX Digital Connection along with the SDK to add messaging and conversational capabilities to their software. Avaya IX Digital Connection's rich APIs allow for conversation management, rich messaging, user metadata collection, account management and more.

Businesses can also use the Avaya IX Digital Connection to connect to their customers (with agents, bots) over messaging using Avaya contact center solution.

IMPORTANT NOTE: This Development guide is applicable to android sdk release v3.1.0

Who is this for?

- Product teams who want to add in-app messaging capabilities to their own software.
- Developers who want the richest in-app messaging available with a powerful, simple, and customizable SDK.
- Businesses who want to add in-app messaging to their app, and allow the conversation to live beyond the app and in any messaging channel.
- Bot builders who want to build a mobile app for their bot to live in using a powerful SDK.
- Customer success teams who want to proactively engage mobile users and build engaging relationships.
- Sales teams who want to do commerce and upsell from their mobile app.

What you'll need

- Technical expertise in Android development to add the SDK to your app.

What changed between SDK 1.0.3 and 3.1.0

- Fixed an issue related to Proguard rules
- Fixed an issue related to DexGuard
- Fixed a bug where the SDK would crash showing a notification in rare cases

- enhances the support for Multi-Conversations and introduces a new conversation list UI. We have also included updates to our conversation screen UI to show the conversation icon, conversation name and description. You can also configure your app to allow users to create multiple conversations from our conversation list screen. The release also includes support for Android X and we've fixed a bug relating to JWT expiry which invokes an auth delegate required to refresh the JWT and reattempt connection.
- Hide/Disable keyboard.
- Android 11 (API 30) support.
- Button text wrapping.
- "Messages" is displayed for conversations that do not have a name assigned.
- Added the ability to send "avatarUrl" when creating a conversation.
- Added the ability to set the notification color.
- Better Message Delivery tracking - enabling integrators to know when a message arrives on the device.
- We now return the participant's userExternalId when fetching the conversation information.
- Fixed a bug when opening a conversation from a list with no network connection.
- "Updated all dependencies to Android X.
- Storage permission changes for Android 11
- Improve performance when opening a conversation
- Allow copying/pasting of bot messages inside a conversation
- Fixed a crash when initializing the SDK without an integrationId.
- Fixed an issue with camera permissions.
- Android 12 support
- Upgrading firebase to the latest version
- Fixed a bug related to web hooks with wrong creation reason
- Fixed an issue with disappearing agent messages.
- Fixed an issue with push notifications not being received when the app is closed.
- Fixed an issue with metadata being ignored on createConversation for anonymous users.
- Add new ProGuard rules to core and UI modules
- Add new ProGuard rules to core for gson
- Added fix for null pointer exception
- Fixed an issue where new conversation could be pressed multiple times
- Fixed Javadocs gradle plugin not generating the docs.
- Fixed issue where removing the start of conversation and welcome string would result in a whitespace on conversation screen.
- Fixed issue with Java util threading concurrency exception.
- Bumped dependencies to resolve security issues.
- Fixed index out of bounds exception in conversations list.
- Fixed NPE for google maps API key.

- support for Android 13
- Updated targetSdkVersion and compileSdkVersion to 33.
- Added support for POST_NOTIFICATIONS permission.
- Updated minSdkVersion to 21.

Guidance on how to update settings during SDK upgrade

- You need to update the gradle to 7.3.3 or later.
- Remove the old artifacts folders and add new one from “distribution” folder. For Example: “core-3.1.0” and “ui-3.1.0”
- Register new folders in the project “settings.gradle” file, like this – “include ':core-3.1.0', ':ui-3.1.0’”
- Update or create the “build.gradle” file in the new artifacts folders. It should look like – “configurations.maybeCreate(“default”) artifacts.add(“default”, file(‘core-3.1.0.aar’))”
- Update the dependencies versions in the application “build.gradle” file. You should use the following:

```

android {
    compileSdkVersion 33

    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 33
    }

    dependencies {

        implementation 'androidx.appcompat:appcompat:1.4.2'
        implementation 'androidx.annotation:annotation:1.4.0'

        implementation project(':core-3.1.0')

        implementation 'com.google.firebase:firebase-messaging:23.0.5'
        implementation 'com.google.code.gson:gson:2.9.0'
        implementation 'com.squareup.okhttp3:okhttp:4.10.0'
        implementation 'com.squareup.retrofit2:retrofit:2.6.2'
        implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
        implementation 'com.google.dagger:dagger:2.30.1'
        implementation 'androidx.annotation:annotation:1.4.0'

        annotationProcessor 'com.google.dagger:dagger-compiler:2.30.1'

        implementation project(':ui-3.1.0')
    }
}

```

```

implementation 'com.github.bumptech.glide:glide:4.11.0'
implementation 'androidx.legacy:legacy-support-v4:1.0.0'
implementation 'androidx.appcompat:appcompat:1.4.2'
implementation 'androidx.exifinterface:exifinterface:1.3.3'
implementation 'androidx.recyclerview:recyclerview:1.2.1'
implementation 'androidx.media:media:1.6.0'
implementation 'com.google.android.gms:play-services-location:20.0.0'
implementation 'com.davemorrissey.labs:subsampling-scale-image-view:3.10.0'
}

```

- Add in the project "gradle.properties" file the following parameters:

```

android.enableJetifier=true
android.useAndroidX=true
org.gradle.jvmargs=-Xmx1536m

```

- Add in the "AndroidManifest.xml" file the FcmService. Like this –

```

<service
    android:name="com.avaya.core.FcmService"
    android:exported="true">
</service>

```

Color customization in v.3.1

To be sure you color customization will persist after upgrade from v.1.0.3, you need to consider the following points:

- 1) color customization should be implemented using actual resource variables from SDK. You need to override the needed variables in **res/values/colors.xml** in your project.
- 2) if your app should support the dark theme and several languages, you need to define needed variables in correspond resource files. For example, in **res/values-night/colors** for dark theme and in **res/values-ru** for Russian language. Otherwise, your color customisation will change to default after applied the dark theme or other languages.
- 3) please note - the variables for header of the chat window named the following: background is "**AvMessagingSdk_actionBarColor**", text color is "**AvMessagingSdk_actionBarTextColor**"
- 4) list of the variables with default values:

```

<color name="AvMessagingSdk_accent">#0072EE</color>
<color name="AvMessagingSdk_accentDark">#76008a</color>
<color name="AvMessagingSdk_accentDarker">#66008a</color>
<color name="AvMessagingSdk_accentLight">#be7cca</color>
<color name="AvMessagingSdk_accentFailure">#800072EE</color>

<color name="AvMessagingSdk_btnSendHollow">#7fc0c0c0</color>
<color name="AvMessagingSdk_btnSendHollowBorder">#c0c0c0</color>

<color name="AvMessagingSdk_btnActionButton">#eeeeee</color>
<color name="AvMessagingSdk_btnActionButtonPressed">#cccccc</color>
<color name="AvMessagingSdk_btnActionButtonRipple">#c0c0c0</color>

<color name="AvMessagingSdk_header">#a0a0a0</color>

```

```

<color name="AvMessagingSdk_inputTextBackground">#ffffff</color>
<color name="AvMessagingSdk_inputTextColor">#212121</color>
<color name="AvMessagingSdk_inputTextColorHint">#bdbdbd</color>

<color
name="AvMessagingSdk_messageDate">@color/AvMessagingSdk_header</color>

<color name="AvMessagingSdk_conversationBackground">#ffffff</color>
<color
name="AvMessagingSdk_conversationListBackground">#ffffff</color>
<color
name="AvMessagingSdk_conversationListTextColor">#99000000</color>
<color
name="AvMessagingSdk_conversationListTitleTextColor">#DE000000</color>
<color
name="AvMessagingSdk_conversationListErrorTextColor">#ffffff</color>
<color
name="AvMessagingSdk_conversationListErrorBackgroundColor">#212b35</co
lor>
<color
name="AvMessagingSdk_conversationListErrorRetryIconColor">#ffffff</col
or>
<color
name="AvMessagingSdk_conversationListTitleTimestampTextColor">#9900000
0</color>

<color
name="AvMessagingSdk_remoteMessageAuthor">@color/AvMessagingSdk_header
</color>
<color name="AvMessagingSdk_remoteMessageBackground">#ecebeb</color>
<color name="AvMessagingSdk_remoteMessageBorder">#d9d9d9</color>
<color name="AvMessagingSdk_remoteMessageText">#212121</color>

<color
name="AvMessagingSdk_userMessageBackground">@color/AvMessagingSdk_acce
nt</color>
<color
name="AvMessagingSdk_userMessageBorder">@color/AvMessagingSdk_accentDa
rk</color>
<color
name="AvMessagingSdk_userMessageUnsentBackground">@color/AvMessagingSdk
_userMessageFailedBackground</color>
<color
name="AvMessagingSdk_userMessageUnsentBorder">@color/AvMessagingSdk_us
erMessageFailedBorder</color>
<color
name="AvMessagingSdk_userMessageFailedBackground">@color/AvMessagingSdk
_accentFailure</color>
<color
name="AvMessagingSdk_userMessageFailedBorder">@color/AvMessagingSdk_ac
centFailure</color>
<color name="AvMessagingSdk_userMessageText">#ffffff</color>
<color name="AvMessagingSdk_descriptionText">#FFC0C0C0</color>
<color
name="AvMessagingSdk_unreadBadge">@color/AvMessagingSdk_accent</color>
<color name="AvMessagingSdk_unreadBadgeText">#ffffff</color>
<color name="AvMessagingSdk_statusBarColor">#C6C6C6</color>
<color name="AvMessagingSdk_actionBarColor">#ffffff</color>
<color name="AvMessagingSdk_actionBarTextColor">#000000</color>
<color
name="AvMessagingSdk_errorSnackBarBackgroundColor">#ff222222</color>

<color

```

```

name="AvMessagingSdk_btnNewConversationBackground">@color/AvMessagingS
dk_accent</color>
<color
name="AvMessagingSdk_btnNewConversationTextColor">#ffffff</color>
<color
name="AvMessagingSdk_btnNewConversationIconColor">#ffffff</color>

<color name="text_helper_color">#a8a8a8</color>
<color name="text_failure_color">#ff2851</color>

```

Customisation of the header of the chat window

In SDK v 3.0.0 and following by default will using the icon from your account from <https://app.smooch.io/> and name from using app integration from your account from a same site.

If you want to update the icon or\and header text you should implement it in your app.

Please use the following method:

```

private void loadConvo() {
    Log.d("titleTest", "Grabbing user's conversations");
    AvMessagingSdk.getConversationsList(new
    AvMessagingSdkCallback<List<Conversation>>() {
        @Override
        public void run(@NonNull Response<List<Conversation>> response) {
            List<Conversation> conversations = response.getData();
            //for new conversation immediately after creation this
            instance will null.
            //update is possible after first messages
            if(conversations != null) {
                Conversation firstConvo = conversations.get(0);

                if (firstConvo != null) {
                    AvMessagingSdk.loadConversation(firstConvo.getId(), new
                    AvMessagingSdkCallback<Conversation>() {
                        @Override
                        public void run(@NonNull Response<Conversation>
                        response) {
                            Log.d("titleTest", "Loaded conversation,
                            attempting to update title and icon.");
                            updateConvo();
                        }
                    });
                }
            }
        }
    });
}

```

```

private void updateConvo() {
    if (AvMessagingSdk.getConversation().getId() != null) {
        Map metadata = new HashMap<String, Object>();
        metadata.put("oceana_intrinsic_name", "any name");
        String convId = AvMessagingSdk.getConversation().getId();
        String title = "Updated conversation title";
        ConversationActivity.close();
        AvMessagingSdk.updateConversationById(convId, title, null,
        "https://web-assets.zendesk.com/images/p-sunshine-conversations/img-sunco-
        footer@2x.jpg", metadata, new AvMessagingSdkCallback() {
            @Override
            public void run(@NonNull Response response) {
                Log.i("response", "response is - " + response);
                ConversationActivity.builder()
            }
        });
    }
}

```

```

        .withFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
        .show(getActivity());
    }
});
Log.d("titleTest", "Conversation title updated.");
}
}

```

note and known limitations:

1) for new conversation immediately after creation the conversations list will null. Updating is possible after the first messages. You have two ways. Update header after the second opening of the conversation or track messages and update header after the first delivered message.

2) By default, for each end user will exist only one conversation. But we have the possibility to create several. Please look at the article:

<https://docs.smooch.io/guide/multi-party-conversations/>

There are several good use cases for a user having (or being a participant of) multiple conversations, which we cover in the article. However, if you're not explicitly creating new conversations for your users and adding them as participants you should not need to worry about `getConversationsList` returning more than one conversation.

3) In example you can see the close of activity before update and reopen after update. This is necessary. The update may not work correctly without it.

4) you may add or update the metadata for the user using a same method. The title and the `isonUrl` may be null in this case for `AvMessagingSdk.updateConversationById`

OS support notes:

You may launch the demo app and your app based on SDK on Chrome OS. All base functions should work, but you may face the OS specific issues, for example with resize the app window.

No commitments to support Chrome OS are provided, even though it could be using Android OS as core/base

Only Android is tested and supported.

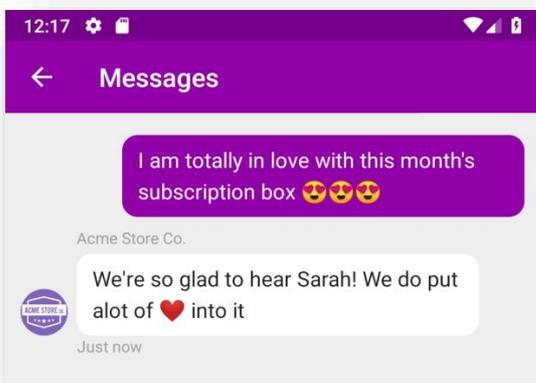
Capabilities

Native Android SDK supports a wide variety of capabilities. Below is a detailed view of each capability:

Text and Emoji

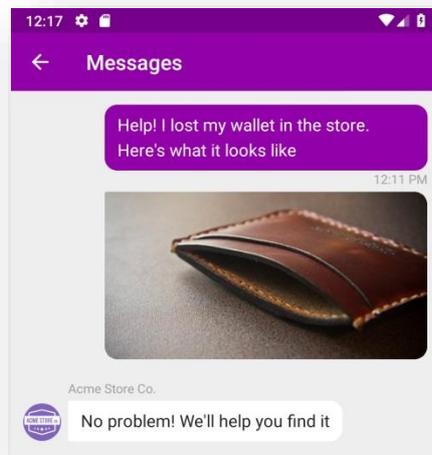
Plain text messages and Unicode Emojis 🐼

The Android SDK displays any Unicode emoji sent in text messages. Mobile users can use the emoji keyboard on their device to send them.



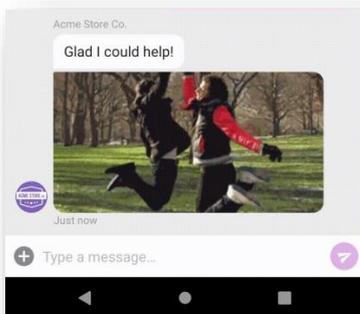
Image

Display static images. The conversation interface supports the display of various image types. Tapping on an image will open it in an image viewer.



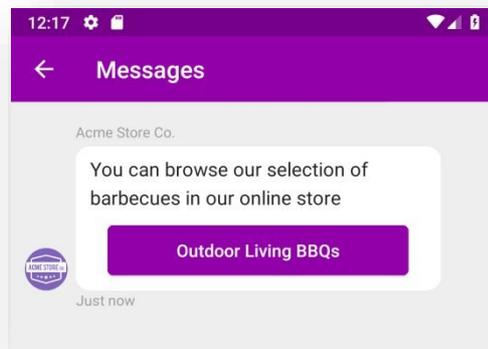
GIF

GIFs sent via the API will be animated on Android devices. Tapping on a Gif will open it in an image viewer. Note that for animated gifs, the image viewer will display the first frame only. The Android SDK does not include a way for users to choose GIFs to send.



Link

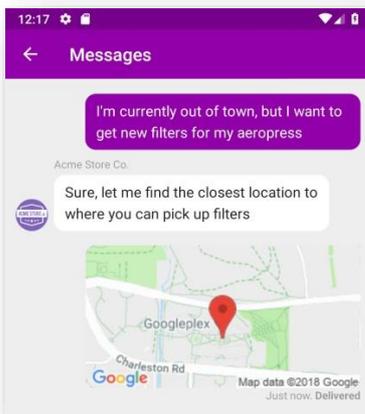
Display web links as buttons, Transform links into clear calls to action.



Location

Send and receive geolocation messages

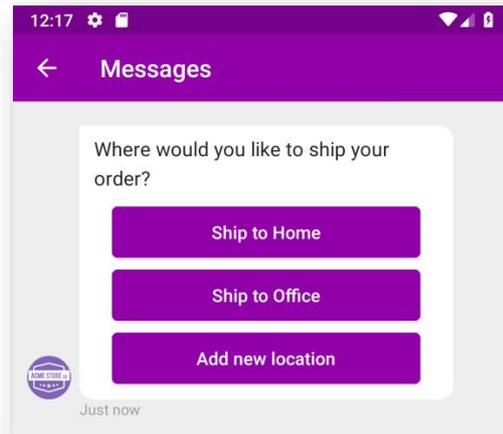
Upon sending a location, users will see a map of that location. Tapping on the map opens the Maps app centered on that location. Location messages include longitude and latitude coordinates along with a Google Maps link (as text) in the API. Location messages are rendered as text messages when a Maps API Key is not present.



Postback

Send buttons to trigger events on your server.

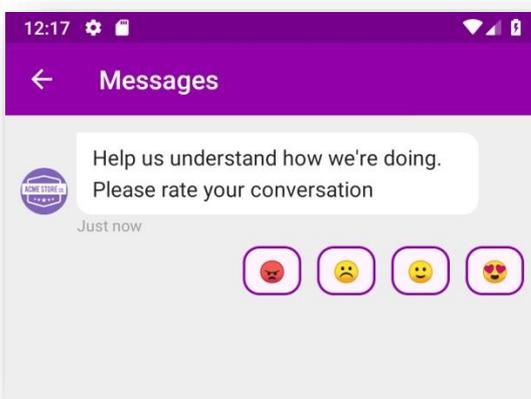
Postback buttons notify the server by webhook when clicked. The server can then act on the click and post messages back to the user in response to the click.



Reply

You can suggest a few answers to reply to a message.

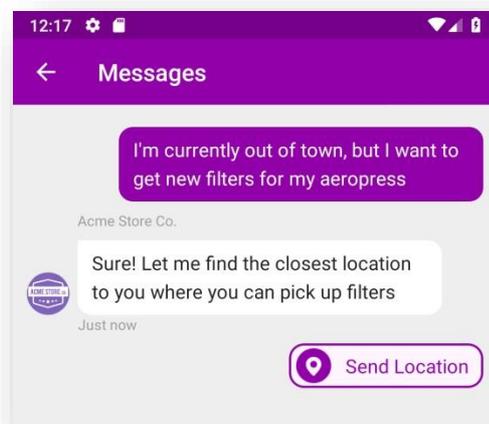
When including replies with a message, the android SDK will display them at the bottom of the conversation. Users can quickly select one of them to send that reply.



Location Request

Request the current location of the user

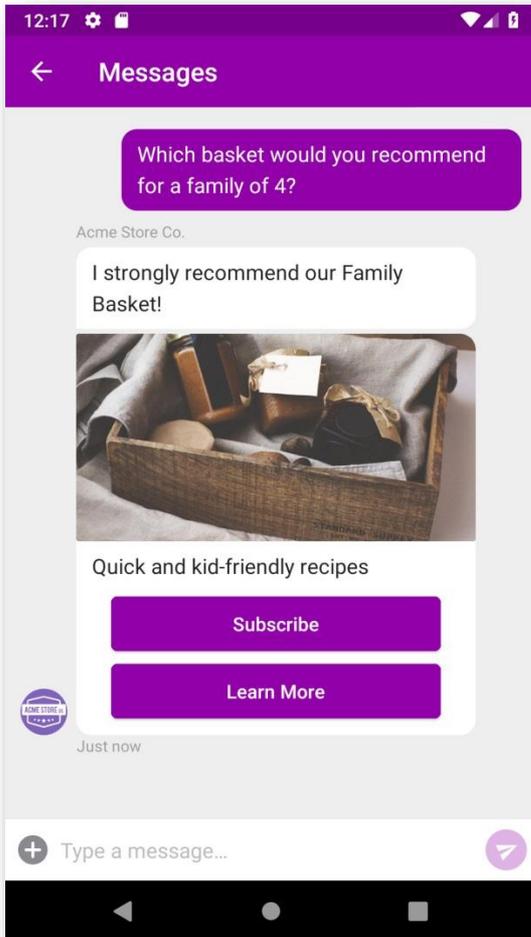
Once a user taps the request button, the android SDK will first ask for location permission and then send the user's location.



Compound Message

You can compose messages with multiple actions.

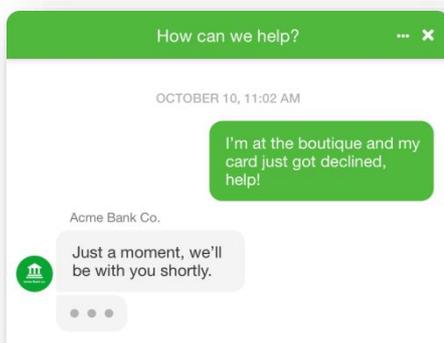
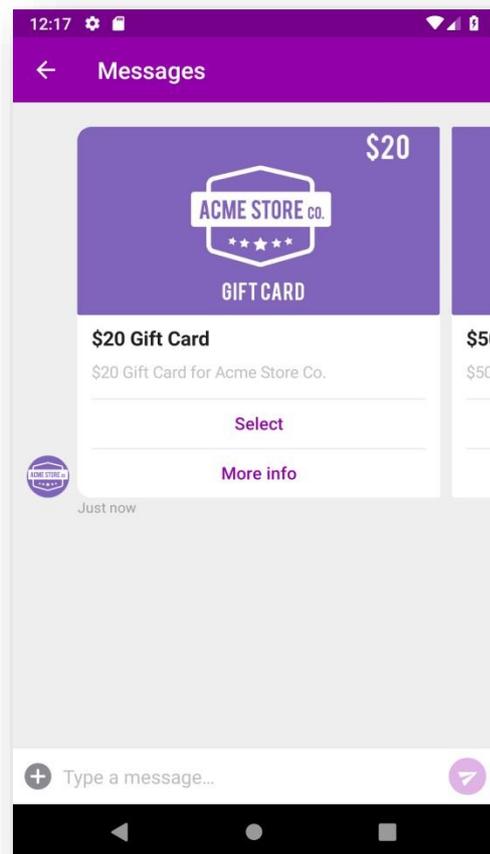
Compound message allows to send text, image and multiple buttons all in a single message.



Carousel

Send a horizontally scrollable set of cards that can contain text, image, and action buttons.

Carousels support up to a maximum of 10 message items. Each message item must include a title and at least one supported action.



Typing Status

Display a typing indicator. Web Messenger can show that the agent is typing with a typing animation. The API allows to specify an agent name and avatar.

Getting Started

Avaya DevConnect offers a bundle Avaya In-app messaging SDK. This bundle includes SDK for each platform (iOS, android), API reference docs and a basic implementation of a mobile messaging app and Web Messenger that uses the SDK to send messages from an android Device to your Avaya contact center Solution.

To get started, download the bundle from Avaya DevConnect <http://www.avaya.com/devconnect>



What's included in the bundle?

./DemoApp: contains project you can open in Android Studio, configure settings and run it on android device or AVD. read [more](#).

./distribution: contains the ready to use SDK binaries that you can add as a dependency in your Android app.

. android\distribution\docs\: contains API reference documentation in javadoc.jar format.

./docs: this guide.

API reference

The Avaya Messaging SDK for Android includes a client side API to initialize, customize and enable advanced use cases of the SDK. See the [Android API reference](#) included in the bundle to discover all the classes and methods available.

The screenshot shows the API reference for the `AvMessagingSdk` class. It includes a package list on the left, a class hierarchy, and a detailed method summary table.

Modifier and Type	Method and Description
static void	<code>destroy()</code> Terminate the AvMessagingSdk facilities.
static AvMessagingSdkConnectionStatus	<code>getAvMessagingSdkConnectionStatus()</code> Accessor method for the current AvMessagingSdkConnectionStatus, or Settings, AvMessagingSdkCallback hasn't been called yet.
static Config	<code>getConfig()</code> Accessor method for the currently connected app configuration.
static Conversation	<code>getConversation()</code> Accessor method for the current conversation.
static void	<code>getConversationById(java.lang.String conversationId, AvMessagingSdkCallback conversationCallback)</code>

QuickStart to your first conversation

The DevConnect bundle for android includes a `/DemoApp` subfolder with a basic implementation of a mobile messaging app that uses the Avaya Messaging android SDK to send messages from an android Device to your contact center.

Prerequisites

To complete the steps below, you must have [Android Studio installed & updated](#), as well as an Android Device to run the sample mobile app (Physical device or [Android Virtual Device \(AVD\)](#)).

Steps

1. Launch Android Studio
2. From the Menu, use `File > Open` to open the project's `/DemoApp` subfolder
3. From the Menu, select `Build > Clean Project` (if the project was built/run and code changes were made)
4. Ensure your Android device is connected (or AVD has been created/configured)
5. From the Menu, select `Run > Run App`
6. Open the menu and press "Settings" . See [Figure 1](#)
7. Setup the correct Integration ID and press "RE-INITIALIZE SDK"
8. Open the menu and press "Conversation" . See [Figure 1](#)
9. Send a test message!
10. you can also configure user credentials (userId, jwt) in the same settings screen to test for an authenticated user, see [Figure 2](#).
11. From the Menu, select `Build > Build APK(s)` if you want to run the demo app on a Physical device.

Please note that the **Demo App** and **APK** provided in the bundle is only for reference, you will need to embed the SDK into your app as per your business requirements.

The minimum supported SDK version is API level **15**, and your app must be compiled with at least API version **26**. If your app needs to support earlier versions of Android, you may still try to integrate, but it is untested and we cannot guarantee compatibility.

Figure 1 - Android SDK Setting in menu

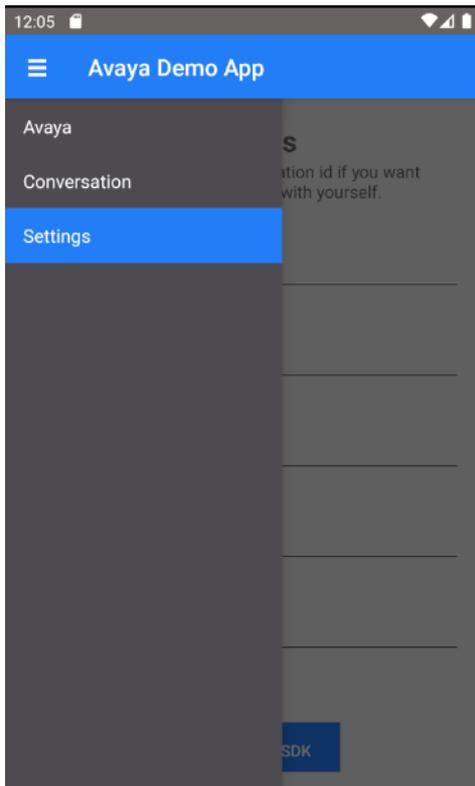


Figure 2 - Android SDK: Set user credentials on settings screen



Adding to your app

The `AvMessagingSdk` library is distributed in both AAR and JAR format. If you are using Android Studio, Manually add the SDK dependencies right into your `build.gradle` file.

- `core-3.1.0.aar`
- `ui-3.1.0.aar`

You can refer to `../android/distribution/README.md` in the bundle for more information.

Initialize in your app

After following the steps above, your app is set up for working with the SDK. Before your code can invoke its functionality, you'll have to initialize the library using your app's ID. This ID uniquely identifies your app and links it to the backend (Avaya IX Digital Connection) that does the heavy lifting necessary to bridge the gap between you and your users.

You can contact Avaya DevOps Team to get the integration ID for Android Channel Integration to your **Avaya IX Digital Connection** app.

Once you've received your integration ID, you can initialize in two different ways: from your `Application` class (suggested), or from an `Activity` class:

Add the following lines of code to your `onCreate` method on your `Application` class:

```
AvMessagingSdk.init(this, new Settings("YOUR_INTEGRATION_ID"), new
AvMessagingSdkCallback <InitializationStatus>() {
    @Override
    public void run(@NonNull Response<InitializationStatus> response) {
        // Handle the response, no casting required...
        if (response.getData() == InitializationStatus.SUCCESS) {
            // Your code after init is complete
        } else {
            // Something went wrong during initialization
        }
    }
});
```

Make sure to replace `YOUR_INTEGRATION_ID` with your integration ID.

If you don't have an Application class, you **must** create one to make sure It is always initialized properly. You can follow the instructions [here](#).

Initialize from an Activity class

You can also initialize from an `Activity`. This is useful if you don't know your integration ID at app launch or if you want to run many apps in the same Android app.

Add the following line of code to your `onCreate` method on your `Application` class:

```
AvMessagingSdk.init(this);
```

If you don't have an Application class, you **must** create one to make sure is always initialized properly. You can follow the instructions [here](#).

Add the following line of code where you want to initialize in your `Activity` class:

```
AvMessagingSdk.init(this, new Settings("YOUR_INTEGRATION_ID"), new
AvMessagingSdkCallback <InitializationStatus>() {
    @Override
    public void run(@NonNull Response<InitializationStatus> response) {
        // Handle the response, no casting required...
        if (response.getData() == InitializationStatus.SUCCESS) {
            // Your code after init is complete
        } else {
            // Something went wrong during initialization
        }
    }
});
```

For example, to initialize when a button is tapped, you can do the following:

Remember to replace `your.package`, `YourActivity`, `YOUR_INTEGRATION_ID` by the appropriate names and your integration ID.

```

package your.package;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import com.avaya.core.InitializationStatus;
import com.avaya.core.Settings;
import com.avaya.core.AVMessagingSdk;
import com.avaya.core.AVMessagingSdkCallback;
import com.avaya.ui.ConversationActivity;

public class YourActivity extends AppCompatActivity implements View.OnClickListener {
    @Override
    protected void onCreate(final Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button showConversationActivityButton =
        findViewById(R.id.button_show_conversation_activity);
        showConversationActivityButton.setOnClickListener(this);
    }
    public void onClick(View v) {
        final int id = v.getId();
        if (id == R.id.button_show_conversation_activity) {
            AvMessagingSdk.init(getApplicationContext(), new Settings("YOUR_INTEGRATION_ID"), new
            AvMessagingSdkCallback <InitializationStatus>() {
                @Override
                public void run(@NonNull Response<InitializationStatus> response) {
                    if (response.getData() == InitializationStatus.SUCCESS) {
                        ConversationActivity.show(getApplicationContext());
                    } else {
                        // Something went wrong during initialization
                    }
                }
            });
        }
    }
}

```

If you initialize from an Activity, you'll need to handle initialization when receiving a push notification. To do so, [override the activity that is opened when tapping a notification](#) and initialize before opening ConversationActivity. For example:

```

package your.package;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import com.avaya.core.InitializationStatus;
import com.avaya.core.Settings;
import com.avaya.core.AVMessagingSdk;
import com.avaya.core.AVMessagingSdkCallback;
import com.avaya.ui.ConversationActivity;

public class YourNotificationHandlerActivity extends AppCompatActivity {
    @Override
    protected void onCreate(final Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        AVMessagingSdk.init(getApplication(), new Settings("YOUR_INTEGRATION_ID"), new
        AVMessagingSdkCallback <InitializationStatus>() {
            @Override
            public void run(@NonNull Response<InitializationStatus> response) {
                if (response.getData() == InitializationStatus.SUCCESS) {
                    ConversationActivity.show(getApplicationContext());
                } else {
                    // Something went wrong during initialization
                }
            }
        });
    }
}

```

Adding an Application class to your app

If you don't have an `Application` class in your app, you can copy the following and save it to your application package.

```

package your.package;
import android.app.Application;

public class YourApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        // Initialize here
    }
}

```

You also need to **declare** your newly created `Application` class in the `<application>` tag in your `AndroidManifest`.

```

<application
    android:name="your.package.YourApplication">
    ...
</application>

```

Remember to replace `your.package`, `YourApplication` by the appropriate names.

Displaying the User Interface

Once you've initialized , you're ready to try it out.

Find a suitable place in your app's interface to invoke and use the code below to display the user interface. You can bring up the Conversation User Interface whenever you think that your user will need access to help or a communication channel to contact you.

```
ConversationActivity.show(this);
```

You should also take the time to [configure the push notifications setup](#).

Location messages

IX Digital Connection offers the option to render location messages with a Google Maps preview. To do so, you must provide your [Google Maps API Key](#) when initializing the SDK.

```
Settings settings = new Settings("YOUR_INTEGRATION_ID");
settings.setMapsApiKey("YOUR_MAPS_API_KEY");
AvMessagingSdk.init(this, settings, new AvMessagingSdkCallback<InitializationStatus>() {
    @Override
    public void run(@NonNull Response<InitializationStatus> response) {
        if (response.getData() == InitializationStatus.SUCCESS) {
            // Your code after init is complete
        } else {
            // Something went wrong during initialization
        }
    }
});
```

- Location messages are rendered as text messages when a Maps API Key is not present.
- If you provide an invalid Maps API Key, messages will render with an error message.
- On Google Cloud Platform, you need to enable Maps Static API.

Region configuration

The Android SDK is supported in the following regions:

Name	Region identifier
United States	Leave unspecified
European Union	eu-1

To target the EU region for example, set the region identifier with `Settings.setRegion()`:

```
Settings settings = new Settings("YOUR_INTEGRATION_ID");
settings.setRegion("eu-1");
AvMessagingSdk.init(this, settings, new AvMessagingSdkCallback<InitializationStatus>() {
    @Override
    public void run(@NonNull Response<InitializationStatus> response) {
        if (response.getData() == InitializationStatus.SUCCESS) {
            // Your code after init is complete
        } else {
            // Something went wrong during initialization
        }
    }
});
```

You can contact Avaya DevOps Team to verify region setting for your **Avaya IX Digital Connection** app.

Log in with userId and JWT

After the SDK is initialized, your user can start sending messages right away. These messages will show up on the business side under a new appUser. However, these appUsers will not yet be associated to any user record you might have in an existing user directory.

If your application has a login flow, or if a user needs to access the same conversation from multiple devices, this is where the `login` method comes into play. You can associate users with your own user directory by assigning them a `userId`. You will then issue each user a `jwt` credential during the login flow.

You can read more about this in the [Authenticating users](#) section.

Authentication delegate

The Android SDK offers an `AuthenticationDelegate` which will be called when an invalid authentication token has been sent to IX Digital Connection. It allows you to provide a new token for all subsequent requests. The request that originally failed will be retried up to five times. To set the delegate, set it in the `Settings` object when initializing the SDK:

```
Settings settings = new Settings("myIntegrationId");
settings.setAuthenticationDelegate(authDelegate);
AvMessagingSdk.init(application, settings, initCallback);
```

Where `authDelegate` is an instance that implements the `AuthenticationDelegate` interface. For example:

```
class MyAuthDelegate implements AuthenticationDelegate {

    /**
     * Notifies the delegate of a failed request due to invalid credentials
     *
     * @param error detail about the authentication error
     * @param callback callback to invoke with a new token
     */
    void onInvalidAuth(AuthenticationError error, AuthenticationCallback completionHandler) {
        // retrieve new token
        completionHandler.updateToken(updatedToken);
    }
}
```

See [Expiring JWTs on SDKs](#) for more information.

Configuring push notifications

Push notifications are a great, unobtrusive way to let your users know that a reply to their message has arrived.

Step 1. Generate a FCM configuration file for your Android project

Following these steps will enable cloud messaging for your app and create a server API key.

1. Go to Google's [Firebase console](#).
2. If you do not have a project already, create a new one.
3. Once created, click on "Add Firebase to your Android app" and follow the instructions to generate your `google-services.json` file (for your package name, copy and paste the package used in your application's `AndroidManifest.xml` file).
4. Save the `google-services.json` file generated by the tool at the root of your Android application (`<project>/<app-module>/google-services.json`).
5. Follow the Gradle instructions and modify your `build.gradle` files accordingly:
 1. Project-level `build.gradle` (`<project>/build.gradle`):

```
buildscript {
    dependencies {
        // Add this line
        classpath 'com.google.gms:google-services:4.3.3'
    }
}
```

2. App-level `build.gradle` (`<project>/<app-module>/build.gradle`):

```
// Add to the bottom of the file
apply plugin: 'com.google.gms.google-services'
```

3. Press `Sync now` in the bar that appears in the IDE.

Note: the SDK adds its own implementation of `FirebaseMessagingService` to your `AndroidManifest` automatically. If your app needs to handle Firebase messages from different providers then you must create your own implementation of `FirebaseMessagingService`. In that case, you need to notify the Avaya Messaging SDK about token changes with `setFirebaseCloudMessagingToken` method on `AvMessagingSdk` class and use `triggerAvMessagingSdkNotification` method on `FcmService` class whenever a new `RemoteMessage` is received to trigger push notifications from Avaya IX Digital Connection.

Step 2. Configure push notifications in

You will need to share your *Server API Key* and *Sender ID* (these can be retrieved by clicking on the cogwheel (Settings) next to `Project Overview` on the sidebar, then clicking on the `CLOUD MESSAGING` tab).

You can contact your Avaya DevOps Team to configure *Server API Key* and *Sender ID* in Avaya IX Digital Connection.

Step 3. Test it out!

To test push notifications in the Android emulator, you must use a system image with Google APIs.

1. Kill and restart your app.
2. Launch .
3. Send a message.
4. Press the home button or navigate away from the conversation.
5. Reply to the message from your choice of integrated service.

You'll receive a notification on the phone!

Localization

Strings can be [customized](#) and localized. The SDK provides a few languages out of the box, but [adding new languages](#) is easy to do. When localizing strings, look for values in the `strings.xml` in your app first then in the ui bundle, enabling you to customize any strings and add support for other languages.

Adding more languages

To enable other languages beside the provided ones, first copy the `English strings.xml` file from the ui bundle to the corresponding values folder for that language. Then, translate the values to match that language.

Customization

Strings customization

String Customization lets you customize any strings it displays by overwriting its keys. In order to do so, simply add `res/values-<your-language-code>/strings.xml` file in your Android project and specify new values for the keys used in . You can find all available keys by browsing to the `ui-x.x.x/res/values/values.xml` file in the External Libraries in Android Studio.

Dates shown in the conversation view are already localized to the user's device.

For example, if you wanted to override strings for English, you would create a file `res/values-en/strings.xml` and include the following in that file:

```

<resources>
  <color name="AvMessagingSdk_accent">#9200aa</color>
  <color name="AvMessagingSdk_accentDark">#76008a</color>
  <color name="AvMessagingSdk_accentLight">#be7cca</color>
  <color name="AvMessagingSdk_backgroundInput">#ffffff</color>
  <color name="AvMessagingSdk_btnSendHollow">#c0c0c0</color>
  <color name="AvMessagingSdk_btnSendHollowBorder">#303030</color>
  <color name="AvMessagingSdk_header">#989898</color>
  <color name="AvMessagingSdk_messageDate">@color/AvMessagingSdk_header</color>
  <color name="AvMessagingSdk_messageShadow">#7f999999</color>
  <color name="AvMessagingSdk_conversationBackground">#ececbeb</color>
  <color name="AvMessagingSdk_remoteMessageAuthor">@color/AvMessagingSdk_header</color>
  <color name="AvMessagingSdk_remoteMessageBackground">#ffffff</color>
  <color name="AvMessagingSdk_remoteMessageBorder">#d9d9d9</color>
  <color name="AvMessagingSdk_remoteMessageText">#000000</color>
  <color name="AvMessagingSdk_userMessageBackground">@color/AvMessagingSdk_accent</color>
  <color name="AvMessagingSdk_userMessageBorder">@color/AvMessagingSdk_accentDark</color>
  <color name="AvMessagingSdk_userMessageFailedBackground">@color/AvMessagingSdk_accentLight</color>
  <color name="AvMessagingSdk_userMessageText">#ffffff</color>
</resources>

```

Menu items

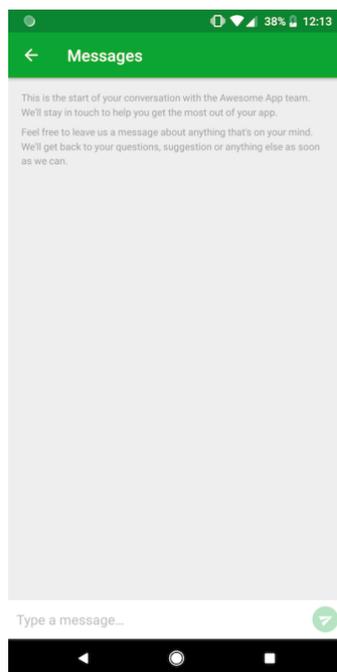
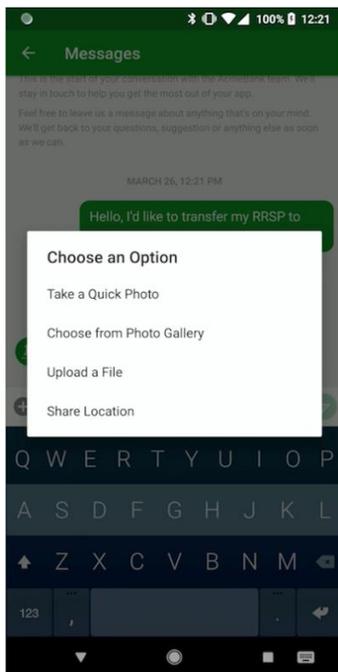
The Android SDK features a tappable menu icon that allows the user to send various message types. The types displayed in this menu can be customized, or the menu can be hidden altogether.

If you want to control this menu, simply override the following resource in `settings.xml` to the value of your choice:

```

<string-array name="AvMessagingSdk_settings_showMenuOptions"
  type="array">
  <item>@string/AvMessagingSdk_settings_takePhotoMenuKey</item>
  <item>@string/AvMessagingSdk_settings_chooseMediaMenuKey</item>
  <item>@string/AvMessagingSdk_settings_uploadFileMenuKey</item>
  <item>@string/AvMessagingSdk_settings_shareLocationMenuKey</item>
</string-array>

```



To hide the menu completely, override the resource as follows:

```
<string-array name=" AvMessagingSdk_settings_showMenuOptions" type="array"/>
```

Notification Action Intent Override

The default behaviour of tapping on a push or in-app notification is to open the `ConversationActivity` intent. If you want to change this behaviour, simply override the following resource in `settings.xml` to the value of your choice.

```
<resources>
  <string name="AvMessagingSdk_settings_notificationIntent">com.avaya.ui.ConversationActivity</string>
  <string name="AvMessagingSdk_settings_notificationTriggerKey">trigger</string>
  <string name="AvMessagingSdk_settings_notificationTrigger">AvMessagingSdkNotification</string>
</resources>
```

When launched, the intent will include an extra with the key `trigger` which can be accessed through `intent.getStringExtra`. You can override the key / value of this extra if you want. Use this extra to know when your intent has launched as a result of a notification.

```
@Override
protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String notifTriggerKey =
        getString(R.string.AvMessagingSdk_settings_notificationTriggerKey);
    String notifTrigger = getIntent().getStringExtra(notifTriggerKey);

    if (notifTrigger != null && notifTrigger.equals(getString(
        R.string.AvMessagingSdk_settings_notificationTrigger ))) {
        // Intent was launched by a notif tap
    }
}
```

Note that the specified intent must extend the `Activity` class.

Notification Channel Settings Override

Apps targeting Android Oreo (SDK 26) or higher will have push notifications delivered to a [notification channel](#). If you wish to override the channel settings for notifications, set the following resources in `settings.xml` to the values of your choice:

```
<resources>
  <string name="AvMessagingSdk_settings_notificationChannelId">your_channel_id</string>
  <string name="AvMessagingSdk_settings_notificationChannelName">Your Channel
  Name</string>
</resources>
```

Starting Text

Allows to you to prefill the text box in the conversation when opening it. To do so, simply supply a `startingText` string when calling `ConversationActivity.show`:

```
ConversationActivity.show(this, "Hi there! Can you help me please?");
```

Extending ConversationActivity

In some cases it may be desirable to use your own custom `ConversationActivity` in place of the one provided. One such case is if you want to modify or remove the action bar. This can be achieved if you extend the `ConversationActivity`.

To do so, do the following:

1. Create a new activity that extends `ConversationActivity`
2. Add the theme to your activity in your `AndroidManifest.xml` file (or create one which specifies `<item name="android:windowActionBarOverlay">true</item>`)

At this point, your `Activity` should look something like this:

```
package your.package;
import android.os.Bundle;
import com.avaya.ui.ConversationActivity;

public class CustomConversationActivity extends ConversationActivity {
    @Override
    protected void onCreate(final Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setTitle("Whatever You Want");
        getSupportActionBar().setDisplayHomeAsUpEnabled(false);
    }
}
```

And your manifest should look something like this:

```
<activity
    android:name="your.package.CustomConversationActivity"
    android:theme="@style/Theme.AvMessagingSdk"
"/>
```

3. When showing in your Android code, instead of calling `ConversationActivity.show(this)`, call `startActivity(new Intent(this, CustomConversationActivity.class))`;
4. To ensure the correct activity is launched when a notification is tapped, override the setting `AvMessagingSdk_settings_notificationIntent` to provide the full path of your new activity.

```
ex: <string name=" AvMessagingSdk_settings_notificationIntent">
    <your.package.CustomConversationActivity
```

```
</string>
```

Permissions

The library includes the following permissions by default:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

- `WRITE_EXTERNAL_STORAGE` is used to take photos and to store downloaded pictures locally to avoid needless re-downloading.
- `ACCESS_FINE_LOCATION` is used in order to access the customer's location when requested using [location request buttons](#) (see [capabilities section](#)).

If you do not intend to request the user's location at any point, it is safe to remove the `ACCESS_FINE_LOCATION` using the following override:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
tools:node="remove" />
```

All other permissions are necessary to function as intended.

Appendix A – Users Authentication

If your software or application has an existing user authentication method then you can optionally federate those identities with **Avaya IX Digital Connection** by issuing a [JSON web token](#) (JWT). A JWT is required to protect the identity and data of these users. This option requires your app to be connected to your own secure web service. There are JWT libraries available supporting a wide variety of popular languages and platforms.

First, you must assign a `userId` to each of your users. The `userId` will uniquely identify your users within Avaya IX Digital Connection and the JWTs you issue serve as a signed proof that your software or app has successfully authenticated that user.

A `userId` is a string that can have any value you like, but must be unique within a given **Avaya IX Digital Connection** app. Examples of `userId`s include usernames, GUIDs, or any existing ID from your own user directory. The `userId` should map to a unique identity in your existing user directory. The `userId` should always reference an external entity; in other words you should not reuse any id that was assigned by **Avaya IX Digital Connection** as a `userId`. When choosing a `userId` you should also ideally avoid using user properties that change, like a phone number.

To log in with a JWT:

1. Generate an API key for your **Avaya IX Digital Connection** app.

You can contact your Avaya DevOps Team to generate *key ID* and *secret* for your **Avaya IX Digital Connection** app.

2. Implement server side code to sign new JWTs using the **key ID** and **secret provided**. The JWT header must specify the key ID (`kid`). The JWT payload must include a `scope` claim of `appUser` and a `userId` claim which you've assigned to the app user. Make sure the `userId` field is formatted as a String. If you use numeric ids, the `userId` must be a String representation of the number - using a number directly will result in an invalid auth error.
3. Issue a JWT for each user. You should tie-in the generation and delivery of this JWT with any existing user login process used by your app.
4. Initialize Avaya In-App Messaging SDK in your website or app. See instructions for [Android](#) in this document.

A node.js sample is provided below using `jsonwebtoken >= 6.0.0`

```

var jwt = require('jsonwebtoken');
var KEY_ID = 'app_5deaa3531c7f940010cc4ba4';
var SECRET = 'BFJJ88naxc5PZnamu9KpBNTR';

var signJwt = function(userId) {
  return jwt.sign(
    {
      scope: 'appUser',
      userId: userId
    },
    SECRET,
    {
      header: {
        alg: 'HS256',
        typ: 'JWT',
        kid: KEY_ID
      }
    }
  );
};

```

5. Call `AvMessagingSdk.login` with your `userId` and `jwt`:

Android (Java):

```

AvMessagingSdk.login("user-id", "jwt", new AvMessagingSdkCallback() {
  @Override
  public void run(Response response) {
    if (response.getError() == null) {
      // Your code after login is complete
    } else {
      // Something went wrong during login. Your JWT might be invalid
    }
  }
});

```

If your API key is ever compromised you can generate a new one. Avaya IX Digital Connection will accept a JWT as long as it contains all required fields and is signed with any of your Avaya IX Digital Connection Conversations app's valid API keys. Deleting an API key will invalidate all JWTs that were signed with it.

Expiring JWTs on SDKs

If you desire to generate credentials that expire after a certain amount of time, using JWTs is a good way to achieve this.

The exp (expiration time) property of a JWT payload is honoured by the IX Digital Connection API. A request made with a JWT which has an exp that is in the past will be rejected.

Keep in mind that using JWTs with exp means that you will need to implement regeneration of JWTs, which demands additional logic (Android, iOS or Web Messenger) in your software.

JWTs are required to identify your users by a custom identifier (userId) in SDKs. In this case, JWTs are signed with an app API key with a scope of appUser, and an additional payload property userId.

Sample JWT Structure with expiry

Header:

```
{
  "alg": "HS256",
  "typ": "JWT",
  "kid": "<app-key-id>"
}
```

Payload:

```
{
  "scope": "appUser",
  "exp": "1542499200",
  "userId": "<user-id>"
}
```

Note: expiry field - timestamp representing **2018-11-18T00:00:00+00:00**

Users on multiple clients

You may have a single user logging in as the same `userId` from multiple clients. For example, they have your app installed on both their iPhone and their iPad or multiple android devices. You might also have Avaya IX Digital Connection integrated in both your mobile app as well as on your web site.

Once a user has been logged in to Avaya IX Digital Connection, they will see the same conversation across each of these clients.

Omitting the userId

Avaya IX Digital Connection will work perfectly fine without a `userId`. Profile information can still be included, and the user can take advantage of all rich messaging features, but the user will only be able to access the conversation from the client they're currently using. Without a `userId`, if the same individual opens Avaya IX Digital Connection on a new client, or runs your web app in an incognito browser session, they will see a newly created empty conversation when they open Avaya IX Digital Connection, and on the contact center

(business) side they will be represented as two distinct `appUsers`. This will happen even if you specify the same profile information in both cases.

A `userId` can also be omitted at first and added at a later time. If you deploy an update to your app that assigns an existing user with a new `userId` that they didn't have before, any existing conversation history they have will be preserved and their messages will start being synchronized across all clients where that `userId` is being used. This is particularly useful if a user opens Avaya IX Digital Connection and starts a conversation before having logged in to your app or website.

Switching users

If your app allows a shared client to switch between multiple user identities you can call the `login` API multiple times to switch between different `userIds`.

Logging out

Your app may have a logout function which brings users back to a login screen. In this case you would want to revert IX Digital Connection to a pre-login state. You can do this by calling the `logout` API.

Calling `logout` will disconnect your user from any `userId` they were previously logged in with and it will remove any conversation history stored on the client. Logging out will *not* disable Avaya IX Digital Connection. While logged out, the user is free to start a new conversation but they will show up as a different `appUserId` on the business end.

Android (Java):

```
AvMessagingSdk.logout(new AvMessagingSdkCallback() {
    @Override
    public void run(Response response) {
        if (response.getError() == null) {
            // Your code after logout is complete
        } else {
            // Something went wrong during logout
        }
    }
});
```

Appendix B - Managing Users

In addition to the information automatically collected and stored for each of a user's clients, an **appUser** itself can have metadata and profile information attached to it, in order to better understand the context and the history of the user.

The appUser

In the IX Digital Connection lexicon, user (usually referred to as an app user or appUser) refers to an end-user of your platform or a customer of your business. The following are all examples of what IX Digital Connection refers to as an **appUser**:

- A visitor to your Website
- The holder of an SMS number
- A user of your mobile app
- A member of the public on Facebook Messenger

Profile information can be added at runtime with the mobile and web SDKs. There are two types of profile information fields: *structured* and *unstructured*.

Structured Fields

Structured fields are properties that IX Digital Connection has identified as common across many use cases, and has exposed as common properties across all users, when present. The currently supported structured fields are:

- **givenName**, also referred to as **firstName** in some contexts, which represents the user's given name
- **surname**, also referred to as **lastName** in some contexts, which represents the user's surname
- **signedUpAt**, which is the date when the user first started using your service, or when they first became a customer. If not customized, this field is automatically populated to be the date the user was created in , which is most likely the moment when the user messaged you for the first time.
- **email**, which represents the user's email address.

Unstructured Fields

Unstructured fields, also referred to as "custom properties", are a set of key/value pairs that are specific to your application domain, or use case. These fields are stored under the **properties** field of an appUser, and can have values of type **Number**, **String**, or **Boolean**.

Custom properties are limited to 4KB per users. Each custom property 'key' is limited to 100B and each custom property 'value' is limited to 800B. Exceeding characters will be truncated.

An error will be returned if an **appUser** is in the process of being created, or updated, and the sum of all custom properties' sizes is over the 4KB limit.

Adding properties using the SDKs

Each of IX Digital Connections' web and mobile SDKs support attaching properties to a user at runtime. The details of when and how these properties are uploaded to the server is handled automatically by the SDKs, so in general you should not need to worry about this detail. However, the process is documented here for completeness.

On Android and iOS, when a user property is set using one of the SDK methods, the properties are immediately serialized to disk until they can be uploaded to the server. Changes to user properties are uploaded in batches at regular intervals while the app is in the foreground, as well as just before the app is sent to the background, or immediately before a message is sent by the user. If the application exits unexpectedly, or the user has intermittent internet connection or no internet connection, the properties will remain on disk until the upload eventually succeeds (even across app launches). If the user does not yet exist (i.e. the user has not yet sent their first message, and the [startConversation](#) method has not been called), the properties are still tracked and stored on disk until the user is eventually created, and they will be uploaded as part of the user creation flow.

On Web, the user properties are stored in memory, and uploaded in batches at regular intervals, or immediately before a message is sent by the user. If the user does not yet exist (i.e. the user has not yet sent their first message, and the [startConversation](#) method has not been called), the properties are still tracked and stored in memory until the user is eventually created, and they will be uploaded as part of the user creation flow. In contrast to the Android and iOS SDKs, the Web Messenger does not store any user property information on disk - if the browser window is closed before the user is created, then the properties will be discarded.

```
import com.avaya.core.User;
User.getCurrentUser().setFirstName("John");
User.getCurrentUser().setLastName("Doe");
User.getCurrentUser().setEmail("steveb@test.com");
User.getCurrentUser().setSignedUpAt(new Date(1420070400000));

final Map<String, Object> customProperties = new HashMap<>();
customProperties.put("premiumUser", true);
customProperties.put("numberOfPurchases", 20);
customProperties.put("itemsInCart", 3);
User.getCurrentUser().addProperties(customProperties);
```

The `addProperties` method accepts a `Map` containing the properties to add. This dictionary must have keys that are type `String` and values that are either `String`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, or `Date`. If your map contains any other data type as a value, then `toString` will be called on the object and the resulting `String` will be added as a property.

Appendix C - Android FAQ

<p>Why is your SDK only compatible with Android 16+?</p>	<p>Android SDK supports Android API level 16 and above. At the time we made this decision, only 0.6% of the Android Market is below this API level.</p> <p>You can ignore the limitation, but we can't guarantee that it will work properly and do not officially support or test the SDK in this configuration ourselves.</p>
<p>How do I set the ConversationActivity to Portrait-only mode?</p>	<p>To do so, you'll need to create your own activity that extends <code>com.avaya.ui.ConversationActivity</code></p> <p>Once you've created your own activity, ensure that it's launched when a push notification is received.</p>
<p>What is Push Notification?</p>	<p>Android SDK uses Google's Firebase Cloud Messaging (FCM) for Push notification which provides a reliable and battery-efficient connection between your server and devices that allows you to deliver and receive messages and notifications on iOS, Android, and the web at no cost.</p> <p>https://firebase.google.com/products/cloud-messaging</p>

Appendix D – certificate configuration for file transfer feature

You need to generate client cert using **openssl**. To do it you need to run the following commands (in order). In addition, for testing it is better to have the same password (6 and more characters):

```
openssl genrsa -aes256 -out client.key 2048
```

```
openssl pkey -in client.key -out client_test_fixed.key
```

(after this command you can delete **client.key**. **client_test_fixed.key** which was generated after this command should be left and renamed to **client.key**)

```
openssl req -x509 -sha256 -new -key client.key -out client.csr
```

(during command execution you need to answer the following questions)

Country Name (2 letter code) [AU]:

State or Province Name (full name) [Some-State]:

Locality Name (eg, city) []:

Organization Name (eg, company) [Internet Widgits Pty Ltd]:

Organizational Unit Name (eg, section) []:

Common Name (e.g. server FQDN or YOUR name) []:

Email Address []:

```
openssl x509 -sha256 -days 3652 -in client.csr -signkey client.key -out client.crt
```