



Avaya In-App Messaging iOS SDK - Development Guide

Release 5.0.0
September 2022

Table of Contents

Introduction	4
Capabilities	7
Getting Started.....	10
What's included in the bundle?	10
API reference	10
QuickStart to your first conversation.....	11
Installation	12
In the folder “./distribution” contains the “AvMessagingSdk.xcframework”, if already contains the "ios-arm64_armv7_armv7s" and "ios-arm64_i386_x86_64-simulator".....	12
You just need the insert the xcframework manually in your project settings.....	12
After what you will be able to import header file.	12
Import the header file.....	12
Add Required Keys in your app's Info.plist	12
Initialize in your app	14
Displaying the conversation interface	15
Starting Text.....	15
Region configuration.....	16
Login with userId and JWT	17
Authentication delegate	17
Configuring push notifications.....	19
Localization	21
Enabling Localization in your app	21
Adding more languages	22
Customization	22
Strings customization.....	22
Styling the Conversation Interface	23
Menu items.....	24
Appendix A – Users Authentication	25
Appendix B- Managing Users.....	29
Appendix C – iOS FAQ.....	32
Appendix D – Localization strings.....	34
Appendix E – certificate configuration for file transfer feature	37



iOS SDK

Current version: v.5.0.0

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

- API reference
- Capabilities
- Getting Started
 - SDK bundle
 - API reference
- QuickStart to your first conversation
- Installation
- Initialize in your app
- Region configuration
- Authenticating users
- Configuring push notifications
- Rich notifications
- Localization
 - Enabling Localization in your app
 - Adding more languages
- Customization
 - Strings customization
 - Styling the Conversation Interface
 - Menu items
- Appendices
 - Appendix A – Users Authentication
 - Appendix B - Managing Users
 - Appendix C - Android FAQ
 - Appendix D – Localization strings

Enter IntegrationID

Enter the First name

Init SDK

Enter the Last Name

Start

Authenticate user

Custom Key Custom Value

Introduction

This document is a development guide and contains instructions and information for developers seeking to integrate In-app messaging iOS sdk, a highly customizable chat embeddable that can be added to any iOS app and is supported by Avaya IX Digital Connection platform.

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 apps.

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 iOS SDK release v.5.0.0

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 changed from previous release of SDK

In v.5.0.0

This major version contains a number of bug fixes and improvements including:

- Improved APIs related to setting up a conversation delegate. The delegate should now be set by using the `setConversationDelegate` method on the `Smooch` class. It is no longer possible to set a delegate by directly modifying a `SKTConversation` object.
- Fixed an issue that caused the `unreadCountDidChange` delegate method to not be called in a multi-conversation setting.
- Fixed an issue that caused a "Done" button to incorrectly display in the navigation bar on smaller devices.
- Fixed an observer-related crash that could occur infrequently when switching between conversations on slow connections.
- Migrated to updated API documentation tooling.

In v.4.1.3

- Improved APIs related to setting up a conversation delegate. The delegate should now be set by using the `setConversationDelegate` method on the `Smooch` class. It is no longer possible to set a delegate by directly modifying a `SKTConversation` object.
- Fixed an issue that caused the `unreadCountDidChange` delegate method to not be called in a multi-conversation setting.
- Fixed an issue that caused a "Done" button to incorrectly display in the navigation bar on smaller devices.
- Fixed an observer-related crash that could occur infrequently when switching between conversations on slow connections.
- Migrated to updated API documentation tooling.
- Fixed an issue that caused the keyboard to render over the text-input field.
- Fixed an app crash related to showing the conversation.
- Added support for Swift Package Manager.
- Fixed an issue where a crash could occur during login/logout.
- Fixed an issue where a user's location may fail to be sent.
- Fixed an issue where attempting to send an attachment could cause a crash on iPad.
- Fixed an issue where the typing indicator was not being shown.
- Fixed an issue where the SDK could block the main thread when sending messages and presenting and dismissing the conversation screen.
- Fixed an issue where the conversation header disappeared when cancelling a Quick Photo.
- Improved error logging in the event of a message sending failure.
- 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
- Added customizations for Dark Mode.
- Fixed a bug with the composer for multi-line replies.
- Fixed a minor bug for the attachment button when using UISceneDelegate.
- Fixed a bug with the seen indicator not refreshing.
- Xcode 12 Support
- Added ability to change the colour of text in a carousel.
- Hide/Disable keyboard.
- Fixed a bug for taking quick photos in iOS 14.
- "Messages" is displayed for conversations that do not have a name assigned.
- Added the ability to send "avatarUrl" when creating a conversation.

What you'll need

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

Capabilities

The Avaya Messaging iOS SDK supports a wide variety of capabilities. Below is a detailed view of each capability:

Text and Emoji

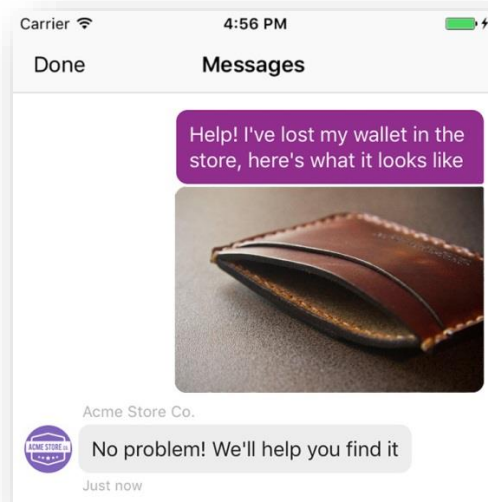
Plain text messages and Unicode Emojis ✨

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



Image

iOS SDK displays static images.



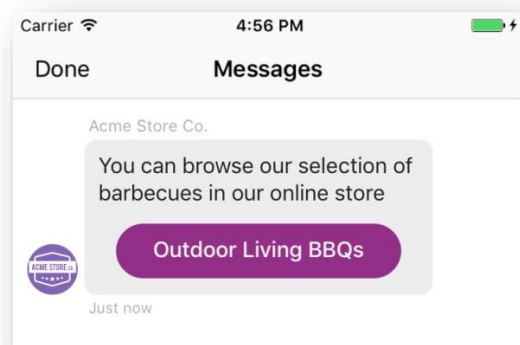
GIF

GIFs sent by the Agent (Business) as image messages will be displayed as a static image in the iOS SDK

GIFs sent as images by the user will also be displayed as a static image in the conversation.

Link

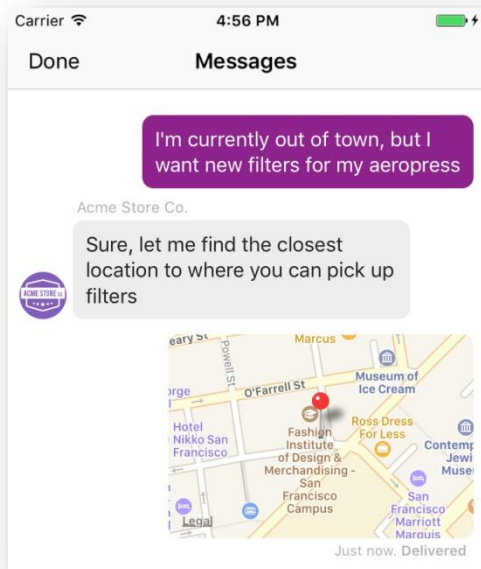
Display web links as buttons.
Transform links into clear calls to action.



Location

Send and receive geolocation messages

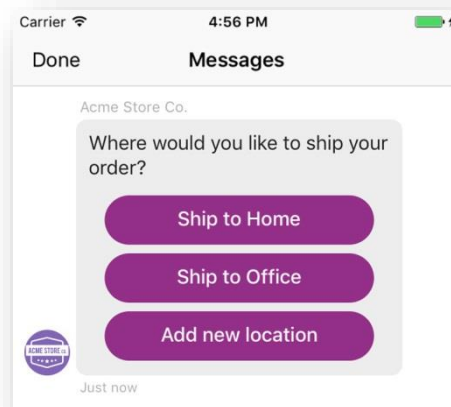
Upon sending a location, users will see a map of that location. Tapping on the map opens the Maps app centred on that location.



Postback

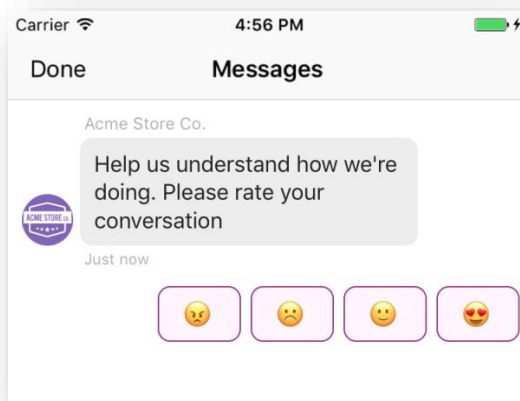
Send buttons to trigger events on your server

[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

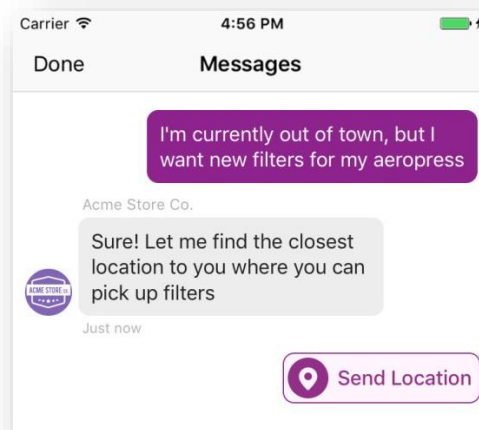
Suggest a few answers to reply to a message. When including replies with a message, the iOS SDK will display them at the bottom of the conversation. Users can quickly select one of them to send that reply.



Location Request

Request the current location of the user

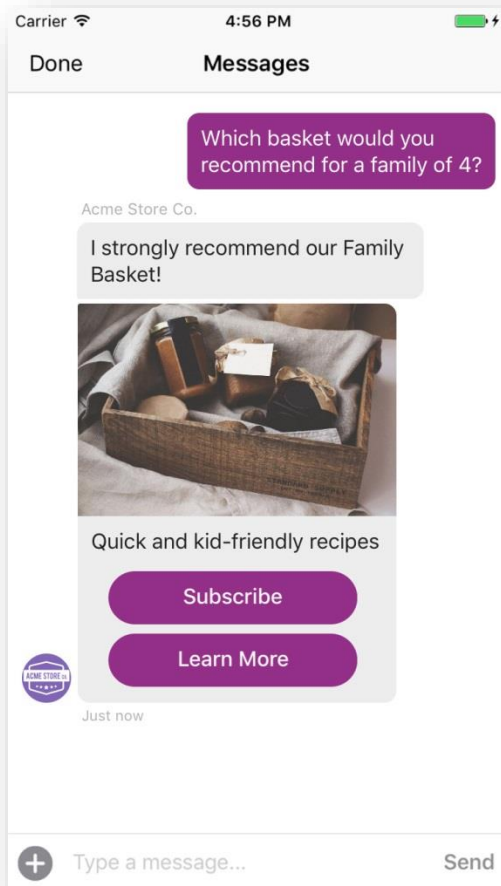
Once a user taps the request button, the iOS SDK will first ask for location permission and then send the user's location.



Compound Message

You can compose messages with multiple actions.

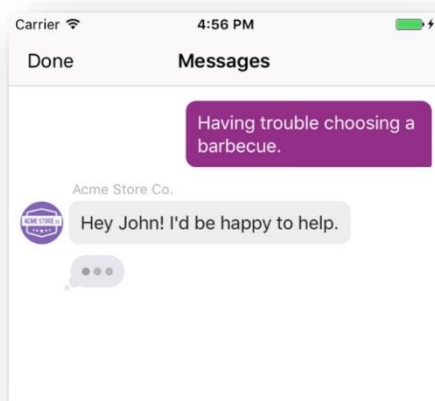
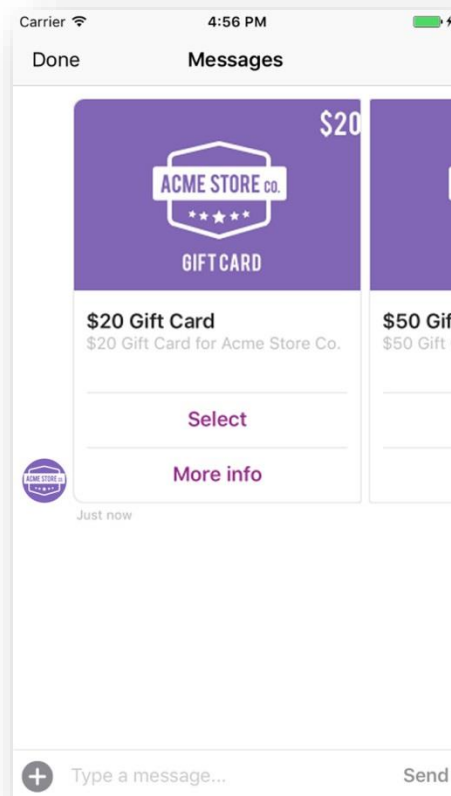
Compound message allows to send text, image and multiple buttons all in a single message.



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

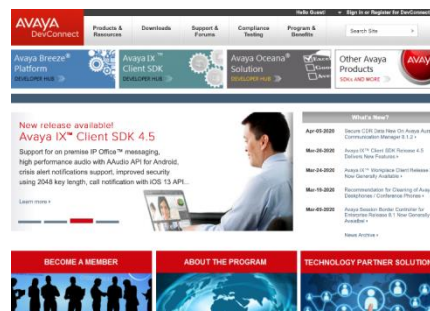
Display a typing indicator

The iOS SDK can show that the agent is typing with a typing animation. The API allows to specify an agent name and avatar. Agent is also notified when the user is typing.

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 a sample application to as a QuickStart to your first conversation using iOS Sdk. More details [here](#).

./distribution: contains framework file (sdk) which you can use in your iOS application to add messaging capabilities.

./docs/API_Reference:

API docs you can discover all the classes and methods available.

./docs/Development Guide: this guide.

API reference

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

QuickStart to your first conversation

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

Prerequisites

To complete the steps below, you must have [Xcode installed](#), as well as an Apple iOS Device to run the sample mobile app (physical device or simulator).

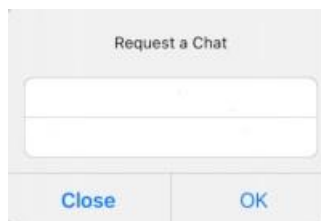
Steps

1. Launch Xcode or Click on `DemoAppSwift.xcodeproj` from the bundle.
2. Ensure your iOS device is connected (or the simulator has been created/configured)
3. From the Menu, select `Product > Run` (or `CMD+r`)
4. Enter the correct Integration ID. Will be easier to use the copy-paste operation for all fields. -> the **"Init SDK"** button will displayed
5. Press the **"Init SDK"** button. The **"Start"** button will displayed.
6. Press the **"Start"** button.
7. Send a test message!

NOTE: Also you may test the conversation with authenticate user.

Additional Steps:

1. Use the switcher - **"Authenticate user"** -> two new fields will display.
2. Paste the correct **JWT** and **User ID** and press **"Start"** button
3. Send a test message!



Installation

In the folder “./distribution” contains the “AvMessagingSdk.xcframework”, if already contains the "ios-arm64_armv7_armv7s" and "ios-arm64_i386_x86_64-simulator".

You just need the insert the xcframework manually in your project settings.

After what you will be able to import header file.

Import the header file

Import the file into your app delegate's .m file and any other places you plan to use it. Examples for Objective-C and Swift:

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

import AvMessagingSdk
```

Add Required Keys in your app's Info.plist

The SDK may need to ask users permission to use certain features. Depending on the feature, you must provide a description in your app's Info.plist to explain why access is required. These descriptions will be displayed the moment we prompt the user for permission.

Images

The SDK allows users to send images to you. To support this feature, you need to provide a description for the following keys:

- **NSCameraUsageDescription**: describes the reason your app accesses the camera (ex: Camera permission is required to send images to \${PRODUCT_NAME}). More information available [here](#).
- **NSPhotoLibraryUsageDescription**: describes the reason your app needs read access to the photo library (ex: Photo library permission is required to send images to \${PRODUCT_NAME}). More information available [here](#).
- **NSPhotoLibraryAddUsageDescription**: describes the reason your app needs write access to the photo library (ex: Photo library permission is required to send images to \${PRODUCT_NAME}). More information available [here](#).

Starting from iOS 10, these values are required. If they are not present in your app's Info.plist, the option to send an image will not be displayed.

Location

The iOS SDK also allows users to send their current location. To support this feature, you must provide a description for any of the following keys depending on your app's use of location services. Avaya In-app Messaging will ask the user for the location depending on the key you provide:

- `CLLocationWhenInUseUsageDescription`: describes the reason for your app to access the user's location information while your app is in use (ex: Location services is required to send your current location to `$(PRODUCT_NAME)`). This permission is recommended if your app does not use location services and the SDK will default to it if both keys are included. More information available [here](#).
- `CLLocationAlwaysUsageDescription`: describes the reason for your app to access the user's location information at all times (ex: Location services is required to send your current location to `$(PRODUCT_NAME)`). More information available [here](#).

If you don't provide one of these keys, any attempt from the user to send their current location will fail.

Initialize in your app

After following the steps above, your app is setup for working with the SDK. Before your code can invoke its functionality, you'll have to initialize the library using your **app id**.

You can contact Avaya DevOps Team to get the app Id for your **Avaya IX Digital Connection** app.

To initialize the SDK, add the following line of code to your `applicationDidFinishLaunchingWithOptions` method:

Objective-C:

```
[AvMessagingSdk initWithSettings: [AVASettings settingsWithAppId: @"YOUR_APP_ID"]
completionHandler:^(NSError * _Nullable error, NSDictionary * _Nullable userInfo) {
    if (error == nil) {
        // Your code after init is complete
    } else {
        // Something went wrong during initialization
    }
}];
```

Swift:

```
AvMessagingSdk.initWith (AVASettings (appId: "YOUR_APP_ID")) {
(error: Error?, userInfo: [AnyHashable : Any]?) in
    if (error == nil) {
        // Your code after init is complete
    } else {
        // Something went wrong during initialization
    }
}
```

Make sure to replace `YOUR_APP_ID` with your app id.

Displaying the conversation interface

Once you've initialized the SDK, 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 whenever you think that your user will need access to help or a communication channel to contact you. Examples below in Objective-C and Swift:

```
[AvMessagingSdk show];
```

```
AvMessagingSdk.show()
```

Starting Text

To prefill the text box in the conversation when opening it, simply supply a starting text string when calling the code to display the user interface. Obj-C and Swift examples below:

```
[AvMessagingSdk showWithStartingText: @"Hi there! Can you help me please?"];
```

```
AvMessagingSdk.show (withStartingText: "Hi there! Can you help me please?")
```

NOTE: To allow the SDK integrate to UI of your application you should implement UI using navigation controller. You may see details in [DemoAppSwift.xcodeproj](#)

Region configuration

The iOS SDK is supported in the following regions:

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

To target the EU region for example, set the region identifier in the `AVASettings` object:

Objective-C:

```
AVASettings settings = [AVASettings settingsWithAppId: @"YOUR_APP_ID"];
settings.region = @"eu-1";
[AvMessagingSdk initWithSettings:settings completionHandler:^(NSError * _Nullable error,
NSDictionary * _Nullable userInfo) {
    if (error == nil) {
        // Your code after init is complete
    } else {
        // Something went wrong during initialization
    }
}];
```

Swift:

```
let settings = AVASettings (appId: "YOUR_APP_ID")
settings.region = "eu-1"
AvMessagingSdk.init(settings) {(error: Error?, userInfo: [AnyHashable : Any]?) in
    if (error == nil) {
        // Your code after init is complete
    } else {
        // Something went wrong during initialization
    }
}
```

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

Login with userId and JWT

After the SDK has been initialized, your user can start sending messages right away. These messages will show up on the business side as 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 iOS SDK offers an `AVAAuthenticationDelegate` 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 `AVASettings` object when initializing the SDK. Examples below in

Objective-C:

```
AVASettings *settings = [[AVASettings alloc] init];
settings.authenticationDelegate = authDelegate;

[AvMessagingSdk initWithSettings:settings completionHandler: completionHandlerBlock];
```

Swift:

```
let settings = AVASettings(appId: "myAppId")
settings.authenticationDelegate = authDelegate

AvMessagingSdk.initWith(settings)
```

When `authDelegate` is an instance that conforms to the `AVAAuthenticationDelegate` protocol. For example:

```

@interface MyAuthenticationDelegate()

@end

@implementation MyAuthenticationDelegate
/**
 * Notifies the delegate of a failed request due to invalid credentials
 * @param completionHandler callback to invoke with a new token
 */
- (void)onInvalidToken:(void (^)(NSString *))completionHandler {
    // retrieve new token
    completionHandler(updatedToken);
}
@end

```

```

class AuthDelegate: NSObject, AVAAAuthenticationDelegate {
    func onInvalidToken (_error: Error, handler completionHandler: @escaping
        AVAAAuthenticationCompletionBlock) {
        // retrieve new token
        completionHandler(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. Enable Push Notifications and Generate the .p12 Certificate

For the steps below, you must be logged in to your Apple Developer account. If you don't have a developer account yet, you can enroll [here](#).

1. Log in to Apple Developer Member Center, and navigate to the [certificates list](#).
2. Click the + button at the top of the page to create a new certificate, and select Apple Push Notification service SSL (Sandbox & Production).
3. Select your app ID from the dropdown and click continue.
4. Follow the [instructions](#) to generate a Certificate Signing Request (CSR) using Keychain Access, and upload it to generate your certificate.
5. Once the certificate is ready, download it to your computer and double-click it to open it in Keychain Access.
6. Right click on the certificate you created, and select Export "Apple Push Services: {your-app-id}".
7. Choose a password, if desired, and save the .p12 file to your computer.

Step 2. Upload the .p12 file to Avaya IX Digital Connection

You can contact your Avaya DevOps Team to configure the .p12 certificate to you Avaya IX Digital Connection app.

The certificate generated with the steps above will be interpreted as a **production** certificate by the Avaya IX Digital Conversation.

If you are testing push notifications with a development build of your app pushed from XCode, you will need to specify this when you request push notification configuration as they would need to upload your certificate with **production: false** instead.

Step 3. Re-create your Provisioning Profile

Now that you have enabled push notifications for your app, you must re-create your Provisioning Profile. You cannot use the one you've used in the past.

1. Go to Provisioning Profiles in the Apple Developer Member Center by [clicking here](#).

2. Click the `+` button to add a new provisioning profile and follow the on-screen instructions.
3. Notice that once you created the new provisioning profile, it shows that “Push Notifications” is an enabled service. Download the new profile.
4. Double click it to install it. It should now be selectable in Xcode for your app.
5. Build your app.

Step 4. Test it out!

You can't receive push notifications in the Xcode simulator, you must use a physical device.

1. Kill and restart your app.
2. Launch the Avaya iOS SDK.
3. Send a message to trigger the push notifications prompt. Important! If the user has not yet granted notification permissions for your app, you must resend a message after uploading the `.p12` file.
4. Reply to the message from the Business System integration of your choice

You'll receive a notification if you're in the app, or outside the app!

The iOS SDK automatically handles incoming push notifications by swizzling certain methods on your app's `UIApplicationDelegate`. To disable this behavior, you can set `enableAppDelegateSwizzling` to `false` on your `AVASettings` object. However, if you choose to do so, you **must** follow the instructions outlined in the [API documentation](#) to ensure that push notifications continue to be handled correctly.

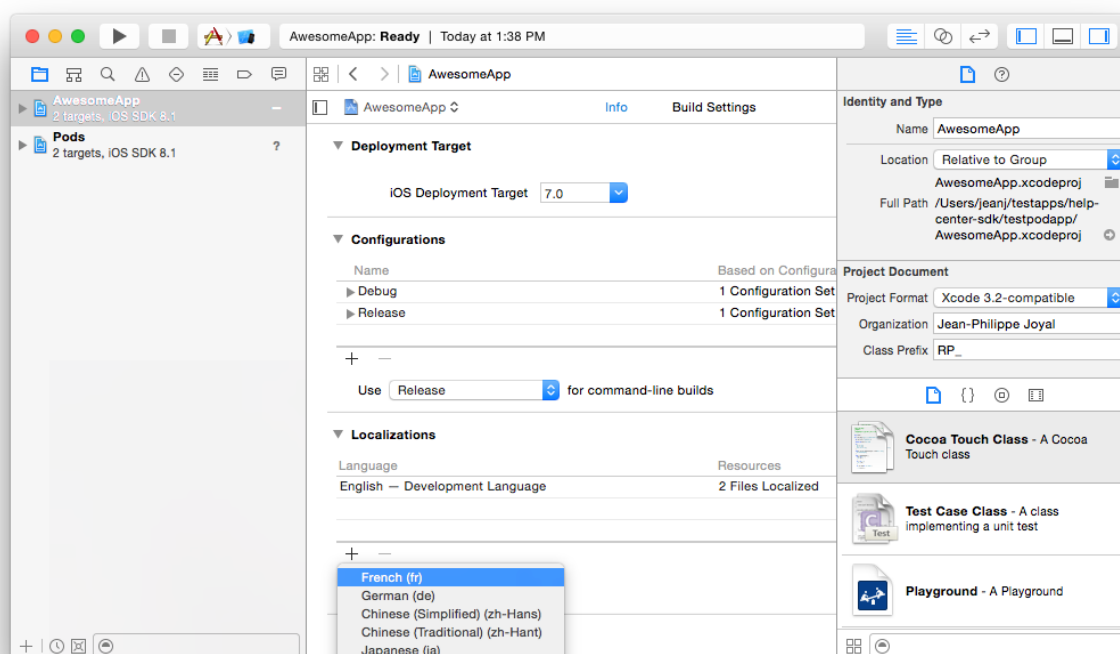
Additionally, on iOS 10 and above, the SDK will handle user notification presentation and on-tap handling by overriding the `UNUserNotificationCenterDelegate` for your application. To disable this behavior, you can set `enableUserNotificationCenterDelegateOverride` to `false` on your `AVASettings` object. As above, if you choose to disable automatic handling of notifications, you **must** follow the instructions outlined in the [API documentation](#).

Localization

Every string you see in the SDK can be [customized](#) and localized. A few languages are provided out of the box, but [adding new languages](#) is easy to do. When localizing strings, the SDK looks for `AvMessagingSdkLocalizable.strings` in your app bundle before checking the `AvMessagingSdk` bundle, enabling you to customize any strings and add support for other languages.

Enabling Localization in your app

To display a language other than English, your app needs to first enable support for that language. You can enable a second language in your Xcode project settings:



Once you have this, the UI will be displayed in the device language for the supported language.

These languages are included with the iOS SDK: Arabic, English, Finnish, French, German, Italian, Japanese, Korean, Mandarin Chinese (traditional and simplified), Persian, Portuguese (Brazil and Portugal), Russian, Slovenian, Spanish, and Swedish.

See how to support more languages in [Adding more languages](#).

Localization is subject to caching. If you can't see your changes, cleaning your project, resetting the simulator, deleting your app from your test devices are good measures.

Adding more languages

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

Customization

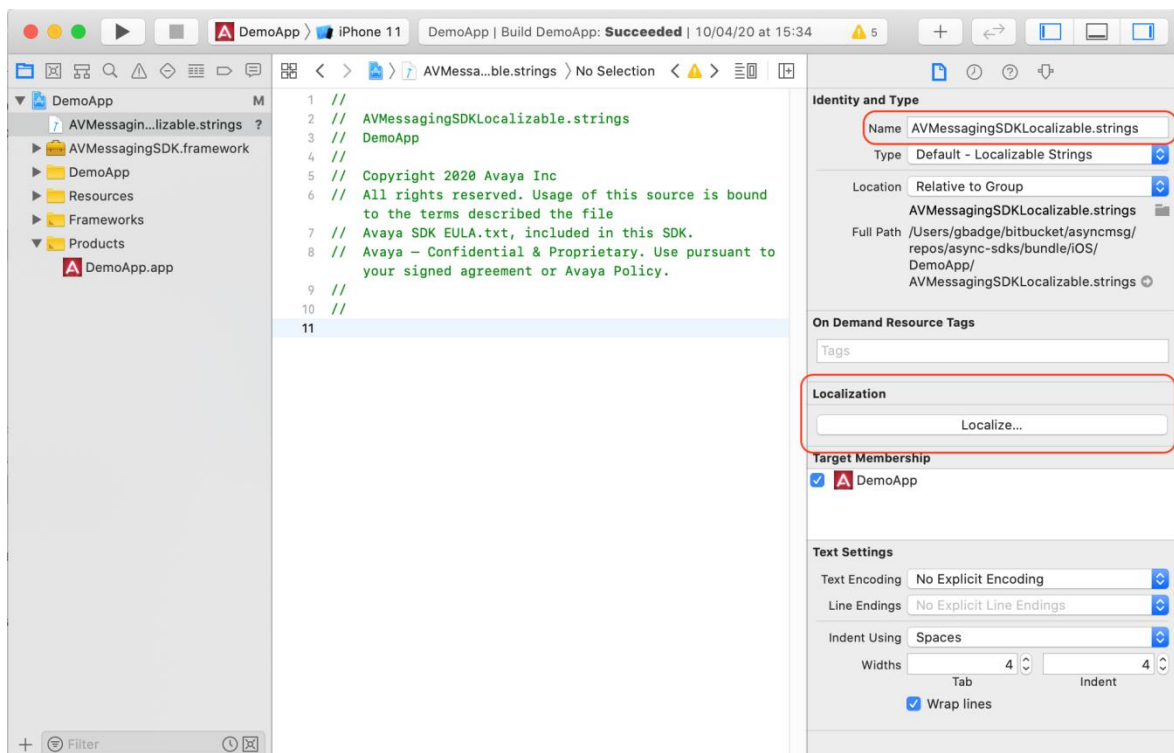
Strings customization

The iOS SDK lets you customize any strings it displays via Apple's localization mechanism. To override one or more strings, add an empty string file named `AvMessagingSdkLocalizable.strings` in your Xcode project and specify new values for the keys you would like to override. For example, to change the "Messages" header and the "Done" button, create a file with these contents:

```
"Messages" = "My Messages";

"Done" = "I'm Done";
```

The full set of keys is listed below. To enable string customization across languages, make sure you "Localize" your `AvMessagingSdkLocalizable.strings` file in Xcode.



Styling the Conversation Interface

The style of the conversation user interface can be controlled through two techniques:

- Using the `UIAppearance` proxy of `UINavigationController` to style the navigation bar's color and appearance.
- The `AVASettings` class provides access to the status bar and the color of the message bubbles and its text.

Suppose you wanted the conversation UI to have a black navigation bar and red message bubbles with white text. First, you'd use `UINavigationController`'s appearance proxy to set up the navigation bar. Then, you'd use `AVASettings` to finish styling the UI:

Objective-C:

```
AVASettings* settings = [AVASettings settingsWithAppId:@"YOUR_APP_ID"];
settings.conversationAccentColor = [UIColor redColor];
settings.userMessageTextColor = [UIColor whiteColor];
settings.conversationStatusBarStyle = UIStatusBarStyleLightContent;

[[UINavigationController appearance] setBarTintColor:[UIColor blackColor]];
[[UINavigationController appearance] setTintColor:[UIColor redColor]];
[[UINavigationController appearance] setTitleTextAttributes:@{ NSForegroundColorAttributeName :
[UIColor redColor] }];
```

Swift:

```
var settings = AVASettings (appId: "YOUR_APP_ID")
settings.conversationAccentColor = UIColor.redColor();
settings.userMessageTextColor = UIColor.whiteColor();
settings.conversationStatusBarStyle = UIStatusBarStyle.LightContent;

UINavigationController.appearance().barTintColor = UIColor.blackColor()
UINavigationController.appearance().tintColor = UIColor.redColor()
UINavigationController.appearance().titleTextAttributes = [ NSForegroundColorAttributeName :
UIColor.redColor()]
```

Menu items

The iOS 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, override the `allowedMenuItems` array in `AVASettings` to add the values of your choice.

To hide the menu completely, set the `allowedMenuItems` array to `nil`.

Objective-C:

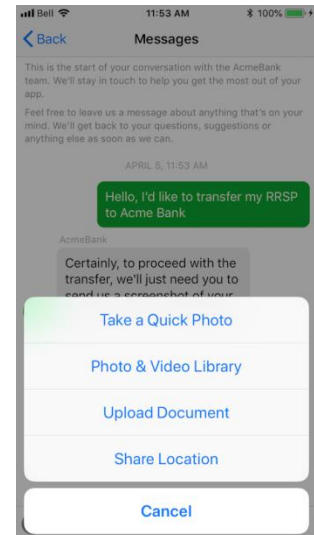
```
AVASettings *settings = [AVASettings settingsWithAppId:@" YOUR_APP_ID "];
settings.allowedMenuItems = @[
    SKTMenuItemCamera,
    SKTMenuItemGallery,
    SKTMenuItemDocument,
    SKTMenuItemLocation
];
```

Swift:

```
AVASettings *settings = [AVASettings settingsWithAppId:@"your_app_id"];
settings.allowedMenuItems = nil;
```

Even with menu items `AVAMenuItemCamera` and `AVAMenuItemGallery` enabled, they may not be displayed depending on your app's configuration, see [Add Required Keys in your app's Info.plist](#) for more detail.

Even with menu item `AVAMenuItemDocument` enabled, it will not be displayed to users on iOS 10 or lower with iCloud drive disabled.



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` for Objective-C and Swift:

```

[AvMessagingSdk login:@"user-id" jwt:@"jwt" completionHandler:^(NSError *
_Nullable error, NSDictionary * _Nullable userInfo) {
  if (error == nil) {
    // Your code after login is complete
  } else {
    // Something went wrong during login. Your JWT might be invalid
  }
}];

```

```

AvMessagingSdk.login("user-id", jwt:"jwt") { ( error:Error? ,
userInfo:[AnyHashable : Any]?) in
  if (error == nil) {
    // 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. Sample code for Objective-C and Swift below:

```
[AvMessagingSdk logoutWithCompletionHandler:^(NSError * _Nullable error, NSDictionary
* _Nullable userInfo) {
    if (error == nil) {
        // Your code after logout is complete
    } else {
        // Something went wrong during logout
    }
}];
```

```
AvMessagingSdk.logout { (error:Error? , userInfo:[AnyHashable : Any]?) in
    if (error == nil) {
        // 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.

Objective-C:

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

[AVAUUser currentUser].firstName = @"Steve";
[AVAUUser currentUser].lastName = @"Brule";
[AVAUUser currentUser].email = @"steveb@channel5.com";
[AVAUUser currentUser].signedUpAt = [NSDate date];
[[AVAUUser currentUser] addProperties:@{
    @"premiumUser": @YES,
    @"numberOfPurchases": @20,
    @"itemsInCart": @3,
    @"couponCode": @"PREM_USR"
}];
```

Swift:

```
AVAUUser.current()?.firstName = "Steve"
AVAUUser.current()?.lastName = "Brule"
AVAUUser.current()?.email = "steveb@channel5.com"
AVAUUser.current()?.signedUpAt = NSDate() as Date
AVAUUser.current()?.addProperties([
    "premiumUser": true,
    "numberOfPurchases": 20,
    "itemsInCart": 3,
    "couponCode": "PREM_USR"
])
```

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 – iOS FAQ

<p>How does Avaya IX Digital Connection handle push notifications?</p>	<p>We can handle push notifications in 2 ways:</p> <p>If the enableAppDelegateSwizzling property of your AVASettings object is set to YES, then Avaya IX Digital Connection will automatically handle any push notifications that originated from IX Digital Connection before forwarding to your app delegate's didReceiveRemoteNotification method. If you set that property to NO, then you will have to call [AvMessagingSdk handlePushNotification:userInfo] in your didReceiveRemoteNotification callback. To check the origin, you use the following code:</p> <pre>BOOL AvMessagingSdkNotification = userInfo[AVAPushNotificationIdentifier] != nil</pre> <p>You can find official documentation about this in the IX Digital Conversation header files, or in our API docs:</p> <p>.../Resources/Documents/Classes/AVASettings.html #//api/name/enableAppDelegateSwizzling</p> <p>.../Resources/Documents/Classes/AvMessagingSdk.html#//api/name/setPushToken</p> <p>.../Resources/Documents/Classes/AvMessagingSdk.html#//api/name/handlePushNotification</p>
<p>How do I present Avaya Messaging Conversations inside a View Controller?</p>	<p>The iOS SDK provides a method in API docs to present Avaya Messaging Conversations in a custom ViewController:</p> <p>.../Resources/Documents/Classes/AvMessagingSdk.html#//api/name/newConversationViewController</p>
<p>How can I perform custom handling when a message action is tapped?</p>	<p>The delegate method conversation:shouldHandleMessageAction: is called whenever a user taps an AVAMessageAction. You</p>

	<p>can perform custom handling in this method, and return false to cancel AvMessagingSdk's default handling of the tap. Read more about it in the iOS API docs</p> <p>.../Resources/Documents/Protocols/AVAConversationDelegate.html#//api/name/conversation:shouldHandleMessageAction:</p>
How can I determine whether or not the conversation view controller is active or inactive?	<p>It's best to implement AVAConversationDelegate in your code and track state of the conversation view controller this way. The two methods you'll want to use are described in the following links:</p> <p>.../Resources/Documents/Protocols/AVAConversationDelegate.html#//api/name/conversation:willShowViewController:</p> <p>.../Resources/Documents/Protocols/AVAConversationDelegate.html#//api/name/conversation:willDismissViewController:</p> <p>Read more about it in the iOS API docs</p>
How can I customize the iOS SDK's appearance?	Read more about this in our documentation .
Is it possible to switch between app IDs during a session?	Currently, we don't provide support for switching between app IDs during the same application session. You can however switch users during the same session (by calling AvMessagingSdk.login with a different userId).
Xcode build fails with Library not loaded: @rpath/Frameworks/AvMessagingSdk.framework. How can I fix this ?	In your project build settings, make sure Runtime Search Paths (LD_RUNPATH_SEARCH_PATHS) is set to \$(inherited), @executable_path/Frameworks.

Appendix D – Localization strings

```
/* Nav bar button, action sheet cancel button */
"Cancel" = "...";

/* Conversation title */
"Messages" = "...";

/* Conversation header. Uses CFBundleDisplayName */
"This is the start of your conversation with the %@ team. We'll stay in touch to help you get the most out of your app.\nFeel free to leave us a message about anything that's on your mind. We'll get back to your questions, suggestions or anything else as soon as we can." = "...";

/* Conversation header when there are previous messages */
"Show more..." = "...";

/* Conversation header when fetching previous messages */
"Retrieving history..." = "...";

/* Error message shown in conversation view */
"No Internet connection" = "...";

/* Error message shown in conversation view */
"Could not connect to server" = "...";

/* Error message shown in conversation view */
"An error occurred while processing your action. Please try again." = "...";

/* Error message shown in conversation view */
"Reconnecting..." = "...";

/* Error message shown in conversation view */
"Invalid file" = "...";

/* Error message shown in conversation view */
"A virus was detected in your file and it has been rejected" = "...";

/* Error message shown in conversation view. Parameter represents the max file size formatted by NSByteCountFormatter */
"Max file size limit exceeded %@." = "...";

/* Fallback used by the in app notification when no message author name is found */
"%@ Team" = "...";

/* Conversation send button */
"Send" = "...";

/* Conversation text input place holder */
"Type a message..." = "...";

/* Conversation nav bar left button */
"Done" = "...";

/* Failure text for chat messages that fail to upload */
"Message not delivered. Tap to retry." = "...";

/* Status text for chat messages */
```

```

"Sending..." = "...";
/* Status text for sent chat messages */
"Delivered" = "...";

/* Status text for chat messages seen by the appMaker */
"Seen" = "...";

/* Timestamp text for recent messages */
"Just now" = "...";

/* Timestamp text for messages in the last hour */
"%0fm ago" = "...";

/* Timestamp text for messages in the last day */
"%0fh ago" = "...";

/* Timestamp text for messages in the last week */
"%0fd ago" = "...";

/* Action sheet button label */
"Take Photo" = "...";

/* Action sheet button label */
"Photo & Video Library" = "...";

/* Action sheet button label */
"Use Last Photo Taken" = "...";

/* Action sheet button label */
"Share Location" = "...";

/* Photo confirmation alert title */
"Confirm Photo" = "...";

/* Action sheet button label */
"Resend" = "...";

/* Action sheet button label */
"View Image" = "...";

/* Error displayed in message bubble if image failed to download */
"Tap to reload image" = "...";

/* Error displayed as message if location sending fails */
"Could not send location" = "...";

/* Error title when user selects "use latest photo", but no photos exist */
"No Photos Found" = "...";

/* Error description when user selects "use latest photo", but no photos exist */
"Your photo library seems to be empty." = "...";

/* Error title when user attempts to upload a photo but Photos access is denied */
"Can't Access Photos" = "...";

/* Error description when user attempts to upload a photo but Photos access is denied */
"Make sure to allow photos access for this app in your privacy settings." = "...";

```

```

/* Error title when user attempts to take a photo but camera access is denied */
"Can't Access Camera" = "...";
/* Error description when user attempts to take a photo but camera access is denied */
"Make sure to allow camera access for this app in your privacy settings." = "...";

/* Generic error title when user attempts to upload an image and it fails for an unknown reason */
"Can't Retrieve Photo" = "...";

/* Generic error description when user attempts to upload an image and it fails for an unknown reason */
"Please try again or select a new photo." = "...";

/* Error title when user attempts to send the current location but location access is denied */
"Can't Access Location" = "...";

/* Error description when user attempts to send the current location but location access is denied */
"Make sure to allow location access for this app in your privacy settings." = "...";

/* UIAlertView button title to link to Settings app */
"Settings" = "...";

/* UIAlertView button title to dismiss */
"Dismiss" = "...";

/* Title for payment button */
"Pay Now" = "...";

/* Title for message action when payment completed */
"Payment Completed" = "...";
/*
Instructions for entering credit card info. Parameters are as follows:
1. Amount (e.g. 50.45)
2. Currency (e.g. USD)
3. App name (Uses CFBundleDisplayName)
*/
"Enter your credit card to send $%@ %@ securely to %@" = "...";

/* Error text when payment fails */
"An error occurred while processing the card. Please try again or use a different card." = "...";

/* Button label for saved credit card view */
"Change Credit Card" = "...";
/*
Information label for saved credit card view. Parameters are as follows:
1. Amount (e.g. 50.45)
2. Currency (e.g. USD)
3. App name (Uses CFBundleDisplayName)
*/
"You're about to send $%@ %@ securely to %@" = "...";
/* Title for user notification action */
"Reply" = "...";
/* Date format used for message grouping headers on the conversation screen */
"MMMM d, h:mm a" = "MMMM d, h:mm a";

/* Date format used for message timestamps on the conversation screen */
"hh:mm a" = "hh:mm a";

/* Error message when the content of a webview fails to load */
"Failed to open the page" = "...";

```

Appendix E – 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