# IAMPASS

*Release 1.3*

**IAMPASS**

**Mar 29, 2023**

# CONTENTS

IAMPASS provides a **passwordless** authentication system for web and native applications.

This document describes how to get started integrating IAMPASS into your application.

IAMPASS provides a REST API and libraries for popular languages. While it is possible to make direct calls to the IAMPASS REST API, we recommend that clients use our libraries.

CHAPTER

ONE

# GETTING STARTED

## 1.1 Introduction

The first stage in integrating IAMPASS into your application is to create an account using the IAMPASS Console.

### 1.1.1 Creating your First Application

- Once you have completed the registration process you are ready to create your first application.
- **IAMPASS Applications** are how you connect your application to IAMPASS.
- Click the **'ADD APPLICATION'** button on the IAMPASS Console and enter a name for your application.
- Once the application is created you will see a confirmation dialog, that contains the credentials your application will use to authenticate with the IAMPASS API.
- **This is the only time** you will be able to see the credentials.

### 1.1.2 Example Application

If you prefer to read code rather than documentation you can check out our Example Application. This is a Fork of Miguel Grinberg's Flask Mega-Tutorial.

### 1.1.3 Managing your Application

The IAMPASS: *Management API* is used to manage your application.

### 1.1.4 Adding Users

Before you can authenticate a user, you have to register the user with IAMPASS. To do this use the add_users route.

We recommend that you create a lookup table in your application that associates your Users, with a token used to authenticate with IAMPASS

In the example below, the application has a **User** table that contains a **name** column and an **IAMPASS Token** table that contains a **user_id** column. The **user_id** column has a foreign key constraint to **User.id**

The **token** field should be used as the id passed to the add_user route.

| User Table | |
|---|---|
| id | name |
| 1 | user1 |
| 2 | user2 |

| IAMPASS Token Table | | |
|---|---|---|
| id | user_id (FK User id) | token |
| 1 | 1 | token1 |
| 2 | 2 | token2 |

### 1.1.5 Registering Mobile Devices

Before a user can authenticate using IAMPASS, they have to register a mobile device.

To register a mobile device, the user must be provided with a registration link to open on their phone. We leave the decision as to how to share the link with your users up to you as IAMPASS does not store **any contact information** for your users.

To obtain a registration link, use the Management API **device_registration_link** route. Once you have the device registration link, you can: * Send it in an email * Send it in an SMS message (this will ensure the link is sent to a mobile device) * Render a QR code in your application. The IAMPASS mobile applications have the ability to scan QR codes and register devices.

### 1.1.6 Checking if a user has Registered a Mobile Device

If you need to know whether a user has registered a mobile device, you can use the Management API **has_registered_mobile_device** route.

### 1.1.7 Authenticating Users

Authentication is handled by the IAMPASS Authentication API *Authentication API*

**IAMPASS authentication is an asynchronous process**

- Call the **authenticate_user** route and store the returned data.
- Call the **status_url** endpoint in the authentication data until authentication completes (or fails)

### 1.1.8 Monitoring Status

**IAMPASS provides the ability to**

- Remotely log out users.
- Log users out if they leave the area where they logged in.

If you want your application to respond to these events, you should periodically call the **status_url** of the authentication data. You can then update your application state based on the response.

### 1.1.9 Logging Users Out

The **authenticate_user** route returns information that identifies the session. To end an IAMPASS session call the **logout_url** route in this data.

CHAPTER

TWO

# MANAGEMENT API

The IAMPASS Management API is used to manage your *IAMPASS Applications* You use the Management API to:

- Add users
- Delete users
- Registser mobile devices
- Detecting if users have registered a mobile device
- Handling lost devices

All calls to the IAMPASS API must include authentication data as described in *API Authentication*

## 2.1 Adding Users

**URL**:: https://main.iam-api.com/add_users/<application_id>

**param** application_id: The Application ID of the application.

**type** application_id: string

**return** The added users in json and http status code

**Example**:

```
curl -X POST https://main.iam-api.com/management/add_users/<application_id> -
↪H 'cache-control: no-cache' -H 'content-type: application/json' \
-d '{
    'users': ['user1', 'user2']
}'
```

**Expected Success Response**:

```
HTTP Status Code 201

{
    'status': True,
    'users': {
        'created': ['user1', 'user2'],
        'existing': ['existing-user1']
    }
}

HTTP Status 200
```

(continues on next page)

(continued from previous page)

```
{
    'status': False,
    'reason': (string)
}
```

**Expected Fail Response**:

```
HTTP Status Code 404

Client Application <application_id> not found
```

**Authentication**

HMAC using Application ID and Application Secret

## 2.2 Deleting Users

**URL**:: https://main.iam-api.com/delete_users/<application_id>

**param** application_id: The Application ID of the application.

**type** application_id: string

**return** Operation result as json and HTTP status code

**Example**:

```
curl -X POST https://main.iam-api.com/management/delete_users/<application_
↪id> -H 'cache-control: no-cache' -H 'content-type: application/json' \
-d '{
    'users': ['user1', 'user2']
}'
```

**Expected Success Response**:

```
HTTP Status Code 200

{
    'status': True,
}
```

**Expected Fail Response**:

HTTP Status Code 404

Client Application <application_id> not found

**Authentication**

HMAC using Application ID and Application Secret

## 2.3 Device Registration

This endpoint will obtain a device registration link that can be shared with users.

See *Getting Started* for information about registration links.

**URL**:: https://main.iam-api.com/device_registration_link/application_id/user_id?display_name=display_name

**param** application_id: The Application ID of the application.

**type** application_id: string

**param** user_id: The ID of the user. This value must be URL encoded.

**type** user_id: string

**param** display_name: (Optional) string that will be used by the IAMPASS Mobile App to display the user information. You can use something like *'user1@my_application'*. This value must be URL encoded.

**return** Operation result as json and HTTP status code

**Example**:

```
curl -X GET https://main.iam-api.com/management/device_registration_link/
↪<application_id>/<userID>?display_name="user1" -H 'cache-control: no-cache
↪' -H 'content-type: application/json'
```

**Expected Success Response**:

```
HTTP Status Code 200

{
    'register_url': (string)
    'status': True,
}
```

**Expected Fail Response**:

HTTP Status Code 404

Client Application *application_id* or User *user_id* not found.

**Authentication**

HMAC using Application ID and Application Secret

## 2.4 Checking for Registered Device

**URL**:: https://main.iam-api.com/has_registered_mobile_device/application_id/user_id

**param** application_id: The Application ID of the application.

**type** application_id: string

**param** user_id: The ID of the user. This value must be URL encoded.

**type** user_id: string

**return** Operation result as json and HTTP status code

**Example**:

```
curl -X GET https://main.iam-api.com/management/has_registered_mobile_device/
↪<application_id>/<userID> -H 'cache-control: no-cache' -H 'content-type:␣
↪application/json'
```

**Expected Success Response**:

```
HTTP Status Code 200

{
    'device_registered': True/False
    'status': True,
}
```

**Expected Fail Response**:

HTTP Status Code 404

Client Application *application_id* or User *user_id* not found.

**Authentication**

HMAC using Application ID and Application Secret

## 2.5 Dealing with Lost Devices

This endpoint will disable a user's mobile device and generate a new registration link.

See *Getting Started* for information about registration links.

**URL**:: https://main.iam-api.com/lost_user_mobile_device/application_id/user_id

**param** application_id: The Application ID of the application.

**type** application_id: string

**param** user_id: The ID of the user. This value must be URL encoded.

**type** user_id: string

**return** Operation result as json and HTTP status code

**Example**:

```
curl -X GET https://main.iam-api.com/management/lost_user_mobile_device/
↪<application_id>/<userID> -H 'cache-control: no-cache' -H 'content-type:␣
↪application/json'
```

**Expected Success Response**:

```
HTTP Status Code 200

{
    'register_url': (string)
    'status': True,
}
```

**Expected Fail Response**:

HTTP Status Code 404

Client Application *application_id* or User *user_id* not found.

**Authentication**

HMAC using Application ID and Application Secret

CHAPTER

THREE

# AUTHENTICATION API

The IAMPASS Authentication API is used to authenticate users.

**The client application is responsible for controlling access to protected resources**

For an overview of the authentication process see *Getting Started*

## 3.1 Initiating the Authentication Process

**URL**:: https://main.iam-api.com/authentication/authenticate_user/<application_id>/<user_id>?methods=methods

**param** application_id: The Application ID of the application.

**type** application_id: string

**param** user_id: The user to authenticate.

**type** user_id: string

**param** methods: (Optional) command separated list of authentication methods to use. If not present **(preferred)** IAMPASS will select appropriate methods.

**type** application_id: string.

**return** Authentication session data in json and http status code.

**Example**:

```
curl -X POST https://main.iam-api.com/authentication/authenticate_user/
↪<application_id>/<user_id> -H 'cache-control: no-cache'
```

**Expected Success Response**:

```
HTTP Status Code 202 - Authentication Started

HTTP Status Code 200 - Authentication did not start

'authentication_status':
{
    'authenticated': True/False,
    'session_status': (string) The status of the session
    'reason': (string) Failure reason
    'status_url': (string) The URL to call to get the session status
    'logout_url': (string) The URL to call to end the session
    'session_token': (string) A token that identifies the session
```

(continues on next page)

(continued from previous page)

```
    'session_secret': (string) Secret used to authenticate calls to status_
↪url and logout_url
}
```

**Session Status Values**

Valid values for session status are defined in: *Session Status Values*

**Authentication Methods**

Valid values for session status are defined in: *Authentication Methods*

**Expected Fail Response**:

```
HTTP Status Code 404
Client Application <ApplicationID> not found
```

**Authentication**

HMAC using ApplicationID and Application Secret

## 3.2 Session Status Values

**Values of session status are:**

- "pending" - the authentication is in progress
- "timeout" - the authentication request has timed out (user did not respond)
- "closed" - the session has been closed
- "failed" - the authentication failed.
- "walkaway" - the mobile device used for authentication is no longer nearby.
- "active" - the user has been authenticated.
- "identifying" - the request is being processed by a mobile device.
- "cancelled" - the user cancelled the request.

**Any status other than:**

- "pending" - the authentication is in progress
- "walkaway" - the mobile device used for authentication is no longer nearby.
- "active" - the user has been authenticated.
- "identifying" - the request is being processed by a mobile device.

mean the user has not been authenticated or the session has ended.

## 3.3 Authentication Methods

**Current authentication methods are:**

- "acceptance" - user is prompted to confirm login attempt
- "device "- user needs to unlock mobile device
- "facial" - user needs perform facial recognition

## 3.4 Monitoring Session Status

The response to calls to the authenticate_user endpoint contain a url to monitor the status of the session.

**URL**:: Contained in the status_url of the authenticate_user response BODY.

**return** Authentication session status in json and http status code.

**Authentication** Calls to this enpoint must use the *session_token* and *session_secret* to construct the authentication headers.

See *API Authentication* for details.

**Expected Success Response**:

```
HTTP Status Code 200

{
    'authenticated': True/False,
    'session_status': (string) The status of the session
}
```

**Session Status Values**

Valid values for session status are defined in: *Session Status Values*

**Authentication**

HMAC using *session_token* and *session_secret*.

## 3.5 Ending Sessions

The response to calls to the authenticate_user endpoint contain a url to monitor the status of the session. **URL**:: Contained in the logout of the authenticate_user response BODY.

**return** Operation result in json and http status code.

**Authentication** Calls to this enpoint must use the *session_token* and *session_secret* to construct the authentication headers.

See *API Authentication* for details.

**Expected Success Response**:

```
HTTP Status Code 200

{
```

(continues on next page)

(continued from previous page)

```
    'status': True/False,
}
```

**Session Status Values**

Valid values for session status are defined in: *Session Status Values*

**Authentication**:

HMAC using *session_token* and *session_secret*.

CHAPTER

FOUR

# CUSTOM IOS APP GUIDE

By default IAMPASS will send authentication requests to the reference IAMPASS mobile application. IAMPASS Applications can be configured to send authentication requests to custom mobile applications. In this case the custom application is responsible for: * Registering the mobile device with IAMPASS * Collecting the required authentication data in response to authentication requests.

## 4.1 Installation

The easiest way to integrate IAMPASS into your application is to use CocoaPods. To integrate IAMPASS into your xCode project using CocoaPods, specify it in your *podfile*.:

```
pod 'IAMPASSiOS', '~>0.0.65'
```

## 4.2 Reference

The IAMPASSiOS framework reference documentation can be found here

## 4.3 Getting Started

### 4.3.1 Create an IAMPASS Account and Application

The first stage in integrating IAMPASS into your applicaton is to create an IAMPASS Application as described *here*.

**Save the application id and application secret for your IAMPASS Application you will need them to communicate with IAMPASS.**

### 4.3.2 Add the IAMPASSiOS framework to your project

To integrate IAMPASSiOS into your xCode project using CocoaPods, specify it in your *podfile*.:

```
pod 'IAMPASSiOS'
```

## 4.4 Application Configuration

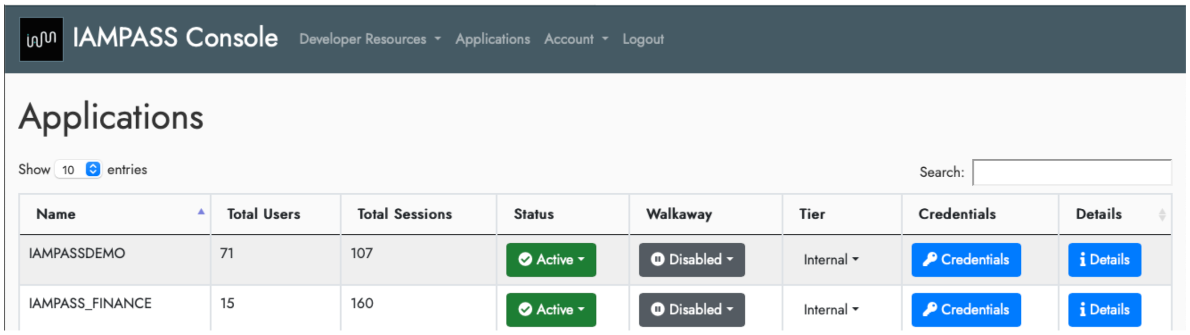
### 4.4.1 Enable Push Notifications

IAMPASS uses Push Notifications to notify users of Authentication Requests. Add the Push Notifications entitlement to your application.

### 4.4.2 Configure IAMPASS Notifications Credentials

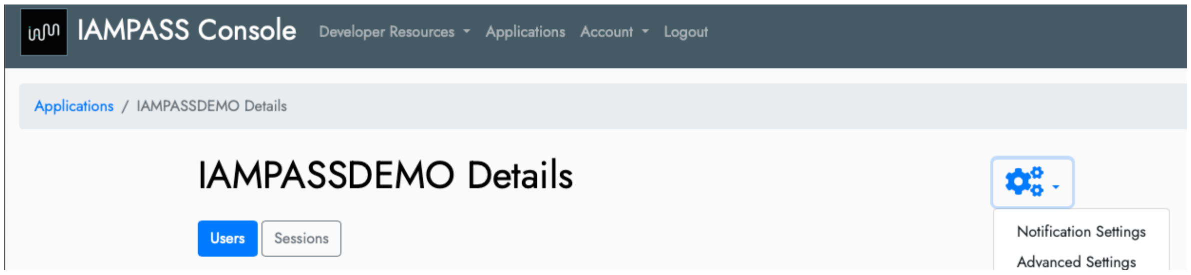
In order to send Push Notifications to your application you need to configure your IAMPASS: IAMPASS needs to be able to authenticate with the APNS servers. The easiest way to do this is to use Token Based Authentication. You can obtain the token information using the instructions here.

- Open the IAMPASS Console
- Select you application and click the **Details** button.

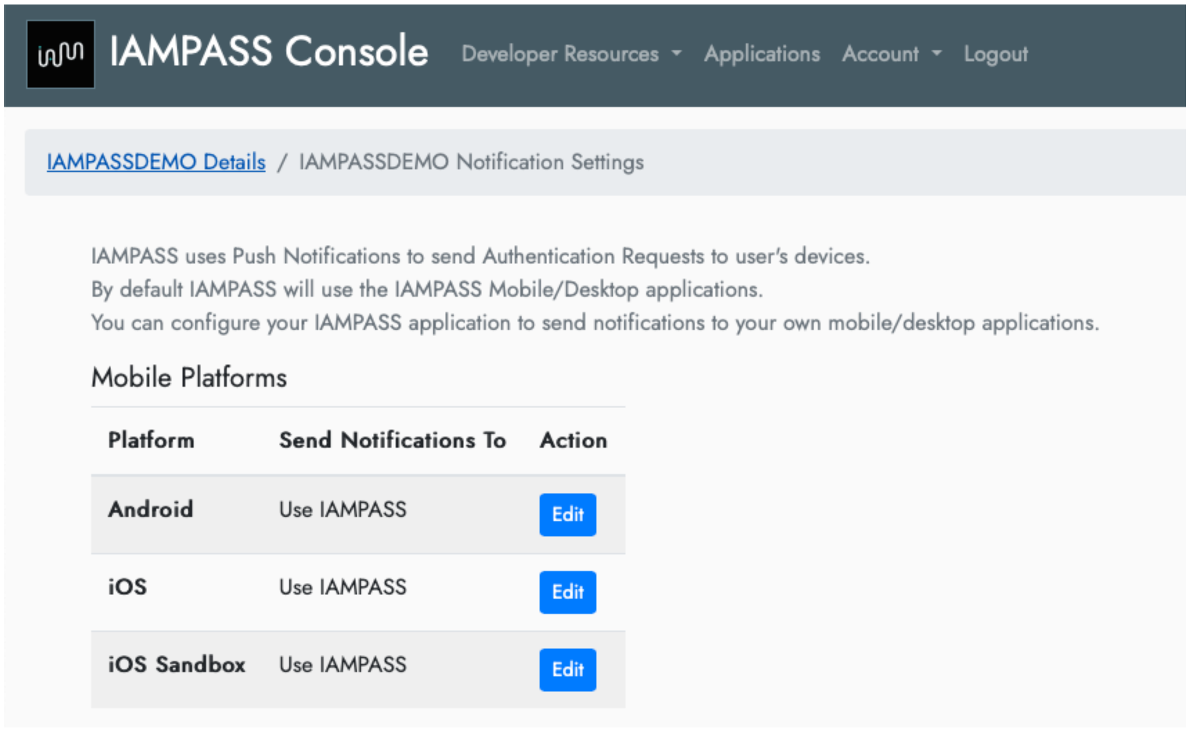


| Name | Total Users | Total Sessions | Status | Walkaway | Tier | Credentials | Details |
|---|---|---|---|---|---|---|---|
| IAMPASSDEMO | 71 | 107 | Active | Disabled | Internal | Credentials | Details |
| IAMPASS_FINANCE | 15 | 160 | Active | Disabled | Internal | Credentials | Details |

- In the details page click the settings icon and select **Notification Settings**.



- On the Notifications Settings page click the **Edit** button for iOS.



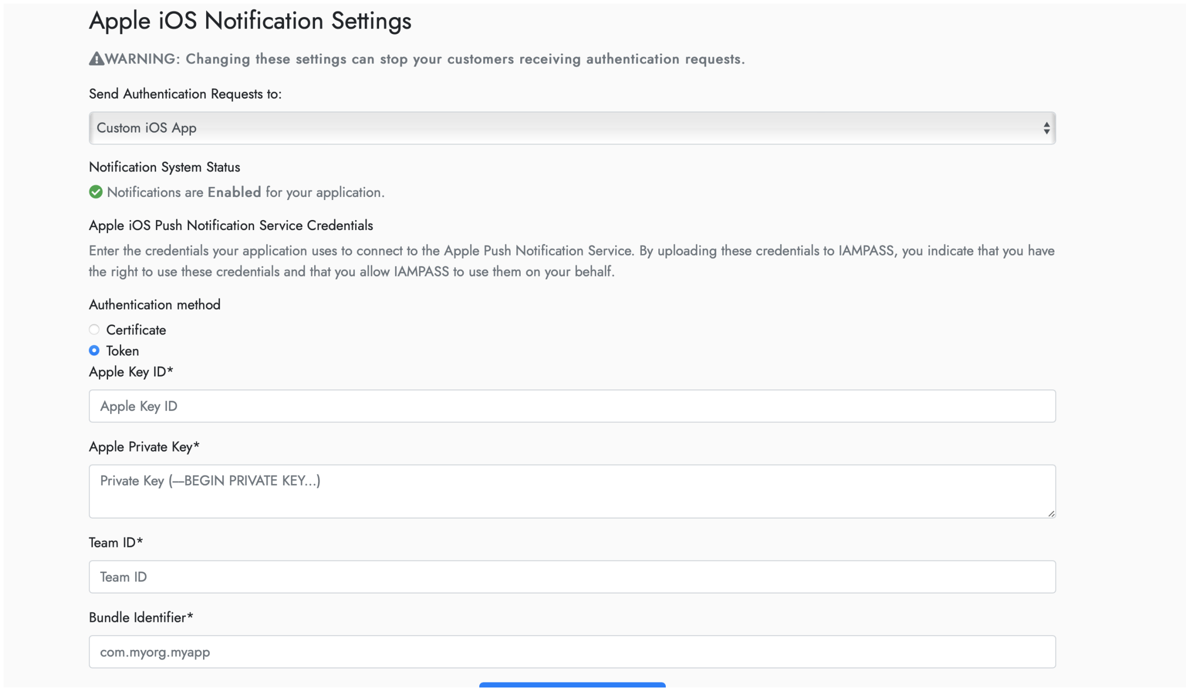
IAMPASSDEMO Details / IAMPASSDEMO Notification Settings

IAMPASS uses Push Notifications to send Authentication Requests to user's devices.
By default IAMPASS will use the IAMPASS Mobile/Desktop applications.
You can configure your IAMPASS application to send notifications to your own mobile/desktop applications.

Mobile Platforms

| Platform | Send Notifications To | Action |
|---|---|---|
| Android | Use IAMPASS | Edit |
| iOS | Use IAMPASS | Edit |
| iOS Sandbox | Use IAMPASS | Edit |

- On the **Apple iOS Notification Settings** page select **Custom iOS** app from the dropdown.
- Select token for the authentication method.



Apple iOS Notification Settings

WARNING: Changing these settings can stop your customers receiving authentication requests.

Send Authentication Requests to:

Custom iOS App

Notification System Status

Notifications are **Enabled** for your application.

Apple iOS Push Notification Service Credentials

Enter the credentials your application uses to connect to the Apple Push Notification Service. By uploading these credentials to IAMPASS, you indicate that you have the right to use these credentials and that you allow IAMPASS to use them on your behalf.

Authentication method

Certificate [✓ Token]

Apple Key ID*

Apple Key ID

Apple Private Key*

Private Key (—BEGIN PRIVATE KEY...)

Team ID*

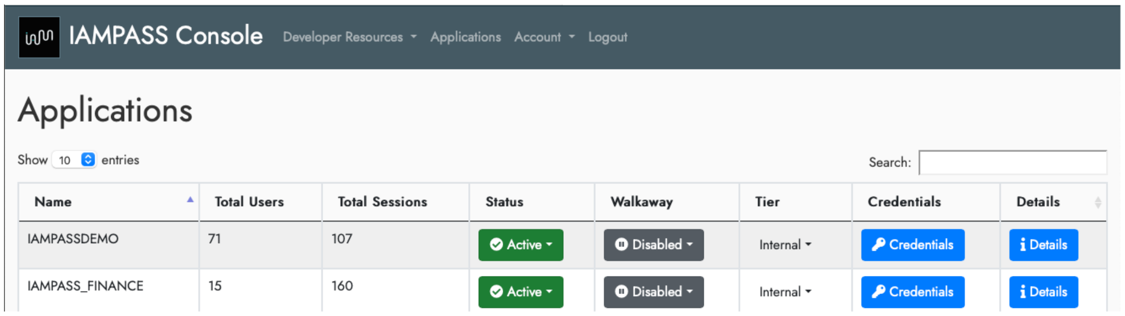
Team ID

Bundle Identifier*

com.myorg.myapp

- Enter the values obtained from the Apple Developer Portal

Alternatively you can use the certificate based authentication. * Download the APNS certificates for your application from the Apple Developer Portal. * Import the certificate and private key into KeyChain (both development and production keys). * Export the certificate and private key from KeyChain as .p12 files. * Open the IAMPASS Console * Select your application and click the **Details** button.



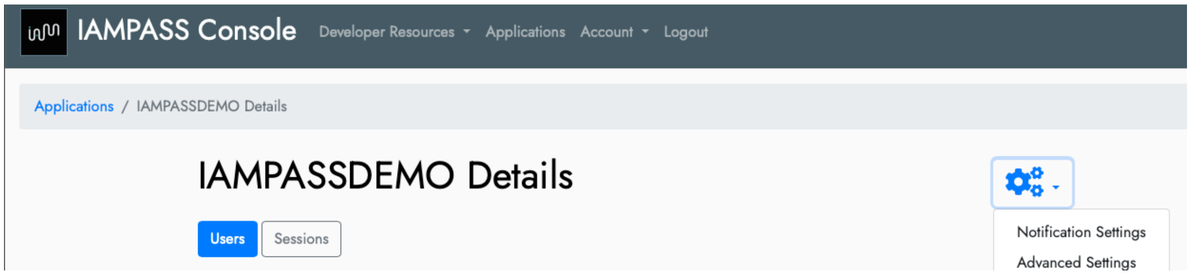
Applications

Show 10 entries

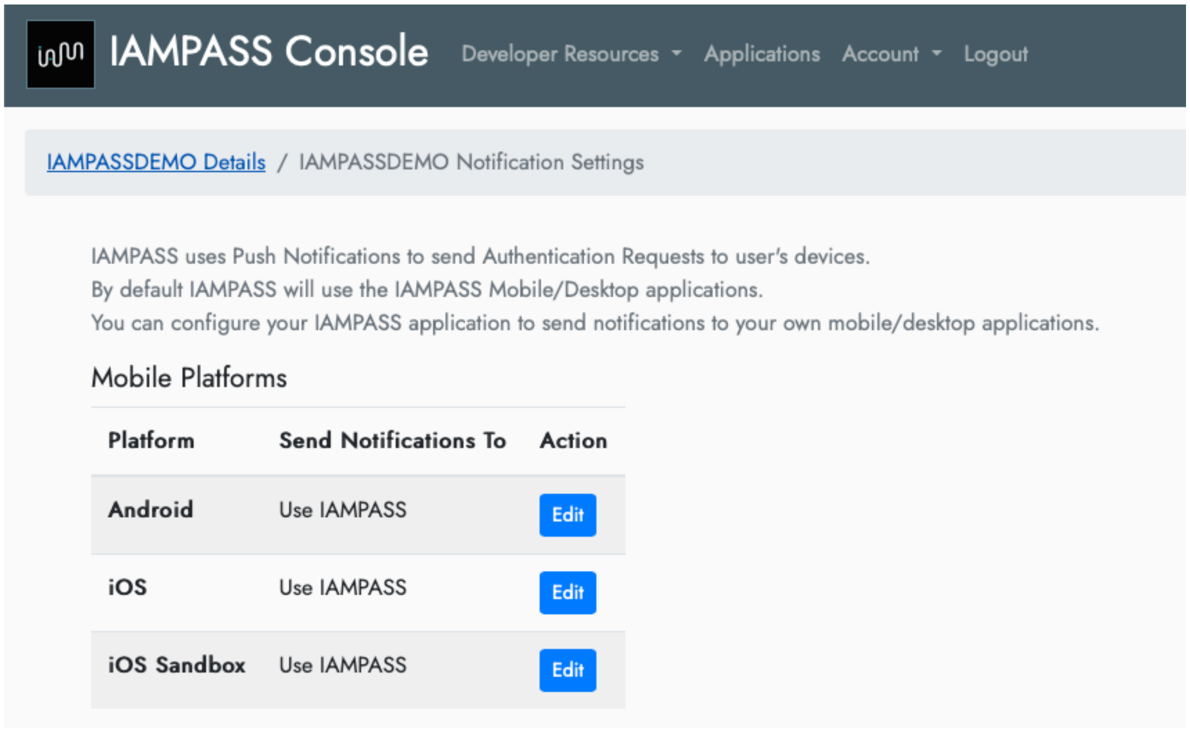
Search:

| Name | Total Users | Total Sessions | Status | Walkaway | Tier | Credentials | Details |
|---|---|---|---|---|---|---|---|
| IAMPASSDEMO | 71 | 107 | Active | Disabled | Internal | Credentials | Details |
| IAMPASS_FINANCE | 15 | 160 | Active | Disabled | Internal | Credentials | Details |

• In the details page click the settings icon and select **Notification Settings**.



• On the Notifications Settings page click the **Edit** button for iOS.

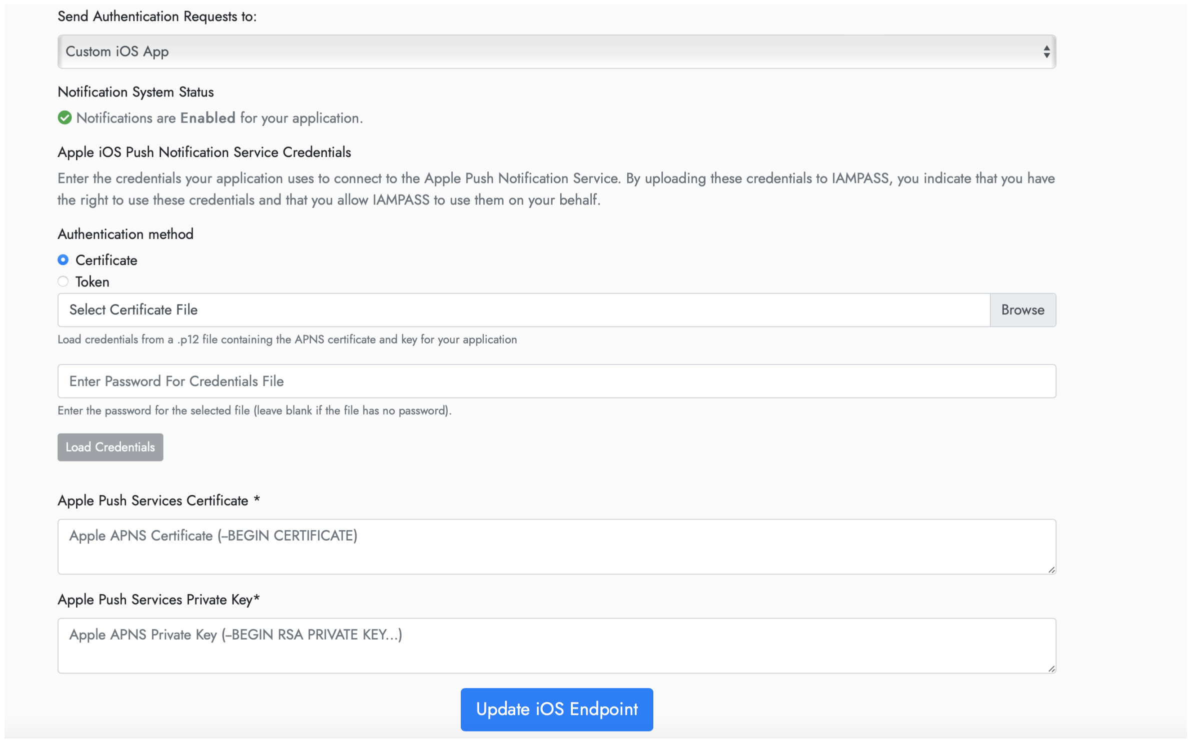


IAMPASS uses Push Notifications to send Authentication Requests to user's devices.
By default IAMPASS will use the IAMPASS Mobile/Desktop applications.
You can configure your IAMPASS application to send notifications to your own mobile/desktop applications.

Mobile Platforms

| Platform | Send Notifications To | Action |
|---|---|---|
| Android | Use IAMPASS | Edit |
| iOS | Use IAMPASS | Edit |
| iOS Sandbox | Use IAMPASS | Edit |

• On the **Apple iOS Notification Settings** page select **Custom iOS** app from the dropdown.



Send Authentication Requests to:

Custom iOS App

Notification System Status

Notifications are **Enabled** for your application.

Apple iOS Push Notification Service Credentials

Enter the credentials your application uses to connect to the Apple Push Notification Service. By uploading these credentials to IAMPASS, you indicate that you have the right to use these credentials and that you allow IAMPASS to use them on your behalf.

Authentication method

● Certificate

○ Token

Select Certificate File | Browse

Load credentials from a .p12 file containing the APNS certificate and key for your application

Enter Password For Credentials File

Enter the password for the selected file (leave blank if the file has no password).

Load Credentials

Apple Push Services Certificate *

Apple APNS Certificate (–BEGIN CERTIFICATE)

Apple Push Services Private Key*

Apple APNS Private Key (–BEGIN RSA PRIVATE KEY...)

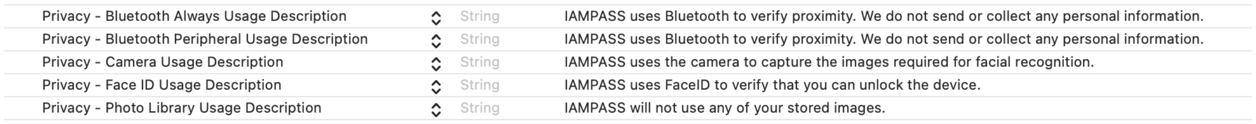
Update iOS Endpoint

- Select certificate for the authentication method.
- Update the credentials with your APNS certificate and Private Key.
- Repeat the process for iOS Sandbox using your development credentials.

IAMPASS will now route your users authentication requests to your iOS application.

## 4.4.3 Applications Entitlements

IAMPASS use Camera and Bluetooth services on your user's mobile device. You must add the following entries to your application's info.plist.



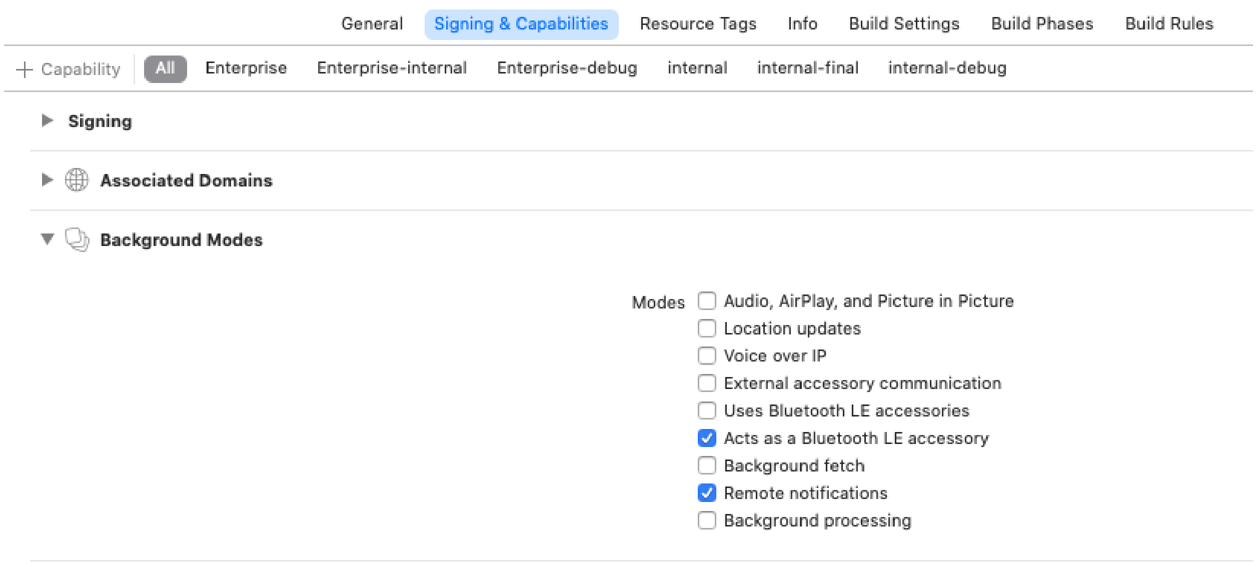
| | | |
|---|---|---|
| Privacy - Bluetooth Always Usage Description | String | IAMPASS uses Bluetooth to verify proximity. We do not send or collect any personal information. |
| Privacy - Bluetooth Peripheral Usage Description | String | IAMPASS uses Bluetooth to verify proximity. We do not send or collect any personal information. |
| Privacy - Camera Usage Description | String | IAMPASS uses the camera to capture the images required for facial recognition. |
| Privacy - Face ID Usage Description | String | IAMPASS uses FaceID to verify that you can unlock the device. |
| Privacy - Photo Library Usage Description | String | IAMPASS will not use any of your stored images. |

| key | value |
|---|---|
| NSBluetoothAlwaysUsageDescription | uses Bluetooth to verify proximity |
| NSBluetoothPeripheralUsageDescription | uses Bluetooth to verify proximity |
| NSCameraUsageDescription | uses the camera to capture images required for facial recognition. |
| NSFaceIDUsageDescription | uses FaceID to verify that you can unlock your phone. |
| NSPhotoLibraryUsageDescription | will not use any of your stored images. |

### 4.4.4 Background Modes

Add the Background Modes entitlement to your application in xCode. Enable the following modes:

- Acts as a Bluetooth LE accessory
- Remote Notifications



Modes
- [ ] Audio, AirPlay, and Picture in Picture
- [ ] Location updates
- [ ] Voice over IP
- [ ] External accessory communication
- [ ] Uses Bluetooth LE accessories
- [✓] Acts as a Bluetooth LE accessory
- [ ] Background fetch
- [✓] Remote notifications
- [ ] Background processing

## 4.5 Key Concepts

**When a system uses IAMPASS to authenticate its users:**

- A notification is sent to the users registered device.
- The registered device collects the required information and send it to IAMPASS.
- IAMPASS processes the collected data and makes an authentication decision.

**A custom IAMPASS mobile device is responsible for:**

- Registering the user's device.
- Processing notifications from IAMPASS.
- Collecting the required data.

## 4.6 Registering Users

IAMPASS does not manage your users or replace your sign up flow. You register your user with IAMPASS by providing a token that you can relate back to your user See *Getting Started* for information about user management. There are 2 common registration flows: * User registers using your mobile application. * User registers externally (in a web browser for example).

**You must wait until your application has completed the Push Notification registration process before registering devices**

### 4.6.1 Mobile Application Registration

The IAMPASS iOS framework provides an interface for registering users and their mobile device. See the IAMPASS example application on GitHub for details.

### 4.6.2 External Registration

IAMPASS generates custom registration links for associating a device with a user. The external client application can get the registration link and share it with the mobile application. It is the client's responsibility to share these links with the mobile application. Alternatively, the mobile application can provide an interface for the user to enter their username and obtain a registration link. The IAMPASS iOS framework provides a method to get a registration link for the user

```
import IAMPASSiOS
...
// user_id: client generated identifer for the client applications user.
// display_name: A readable name for the user (user_id may be a random token)
// MY_APPLICATION_ID: The ID of the IAMPASS application (from IAMPASS Console)
// MY_APPLICATION_SECRET: The Application Secret of the IAMPASS application (from␣
↪IAMPASS Console)

// Create an IAMPASS Management API instance using credentials for client application
let management_api = ManagementAPI(application_id: MY_APPLICATION_ID,application_
↪secret: MY_APPLICATION_SECRET)

management_api.get_registration_link(user_id: user, display_name: user) { (reg_link)␣
↪in
            // reg_link: URL that can be used to register the device.
        } failure: { (error) in
            // Failed to get registration link.
        }
    }
```

Whichever method is used to obtain the registration link, the link can now be used to register the device

```
import IAMPASSiOS
...
IPUser.registerDevice(identifier: user_id, registration_link: reg_link, notification_
↪token: NOTIFICATION_TOKEN) { (identifier, device) in
            // Device registered
            // identifier: identifier for user (same as value passed to␣
↪registerDevice)
            // device: IPUser that contains user and device information required for␣
↪subequent IAMPASS calls.
            print("registered")
        } failure: { (identifier, error) in
            // identifier: identifier for user (same as value passed to␣
↪registerDevice)
            // error: Error indicating failure reason
            print("failed")
        }
```

### 4.6.3 Storing Device Information

The client application should store the IPUser instance returned by device registration. IPUser implements the Codeable interface and can be persisted using Swift encoding (JSONEncoder for example). The iOS UserDefaults can be used to store the data, however the iOS KeyChain main be a more secure option.

## 4.7 Training

After a user has been registered IAMPASS may have to collect some training data. After registration check the *training_required* property of *IPUser* to determine if IAMPASS needs to perform training.

```
if registeredUser.training_required{
    // Perform training...
}
```

The IAMPASS iOS framework provides a default UI for performing training. The example code below shows a view controller that presents the training UI.

```
func doTrainingForUser(identifier: Any, user: IPUser) {
    // It is important that the code to display UI components happens
    // on the main thread.
    DispatchQueue.main.async {

        // Get the ViewController to use to present the UI.
        // If this code is part of a ViewController, just use self.
        let presentingVC = getViewController()

        // Create the IAMPASS training view controller.
        if let vc = IPTrainingViewController.create(user: user, identifier:␣
↪identifier, success: { sender, identifier, device in
            // The training process completed successfuly so save the user␣
↪information.
            if let id = identifier as? String{

                //TODO: Save the user data.

            }
            // sender is the training UI view controller, so dismiss it.
            sender.dismiss(animated: true)
        }, failure: { sender, identifier, device, error in
            // TODO: Training failed - provide user feedback.
            sender.dismiss(animated: true)
        }){
            // Present the training UI.
            vc.modalPresentationStyle = .fullScreen
            vc.modalTransitionStyle = .crossDissolve
            presentingVC.present(vc, animated: true, completion: nil)

        }
    }
}
```

If you prefer to use Storyboards and Segues you can create a new IPTrainingViewController derived class and instantiate an instance in Interface Builder.

### 4.7.1 Updating Device Information

Every time your application starts up or the user changes Notification Settings, you should update the device stored IPUser. Use the *update* method of *IPUser* to update the device information. When the device is updated IAMPASS may need to perform training. See *training* for information about training.

```
// Class implements IPTrainingDelegate, which processes training results.
class MainViewController : UIViewController, IPTrainingDelegate{

    // Update the stored mobile device.
    func updateDevice()->Void{
        let device = self.get_device()
        let user_id = self.get_user_id()

       device.update(identifier: user_id, notification_token: NOTIFICATION_TOKEN) {␣
↪(identifier, updated_device) in
           print("Updated Device")

           // Save the device
           self.save_device(identifier: user_id, device: device)
           if mobile_device.training_required{
               DispatchQueue.main.async {
                   self.doTraining()
               }
           }
       } failure: { (identifier, error) in
           print("Update error")
       }

    }

}
```

### 4.7.2 Handling Authentication Requests

IAMPASS delivers authentication requests to mobile applications using Push Notifications. The notification title and description use string IDs so the client application must have the following strings in its string resource.

```
/*
  Localizable.strings
    ...
*/
...
/*Content of the login notification*/
"NOTIFICATION_TITLE" = "IAMPASS AuthenticationRequest";
"NOTIFICATION_BODY" = "TAP TO ACCEPT or clear to cancel.";
"ACTION_LOGIN_ALERT_NOTIFICATION" = "VERIFY";
```

You can change the values of the strings but the keys must exist.

When a mobile application receives a Push Notification it should pass it to the IAMPASS iOS framework, which will decode the payload and carry out the required authentication steps.

```
func application(
      _ application: UIApplication,
      didReceiveRemoteNotification userInfo: [AnyHashable: Any],
```

(continues on next page)

(continued from previous page)

```
      fetchCompletionHandler completionHandler:
      @escaping (UIBackgroundFetchResult) -> Void
    ) {

        // IAMPASS authentcation request notifications contain data to␣
↪identify the user the request targets.
        // We have to provide the notification handler with a list of user␣
↪data that is stored on this device so that it can determine whether the␣
↪notification should be handled.

        // Do we have a user?
        // Need to implent getRegisteredUser
        if let user = self.getRegisteredUser(){
            let registeredUser = [user]
            let  notificationHandler = IPNotificationHandler()
            notificationHandler.processNotification(userInfo: userInfo,␣
↪registeredUsers: registeredUser) { request, user in
                // This is an authentication request for a user of this␣
↪device so show the authentication UI.
                let vc = IPAuthenticationViewController.create(request:␣
↪request, device: user) { sender in
                    // Authentication was successful
                    sender.dismiss(animated: false) {
                        // We send a notification now that the UI has been␣
↪cleaned up so that interested
                        // parties (ViewController) can update their state.
                        NotificationCenter.default.post(name: AUTHENTICATION_
↪UI_COMPLETE_MESSAGE, object: nil)
                    }
                } failure: { sender, error in
                    // Authentication failed.
                    sender.dismiss(animated: false){
                        // We send a notification now that the UI has been␣
↪cleaned up so that interested
                        // parties (ViewController) can update their state.
                        NotificationCenter.default.post(name: AUTHENTICATION_
↪UI_COMPLETE_MESSAGE, object: nil)
                    }
                }
                vc?.modalPresentationStyle = .fullScreen
                vc?.modalTransitionStyle = .crossDissolve

                self.topViewController()?.present(vc!, animated: true)

            } onStatusChanged: { status in
                // The notification is a session status change notification.
                completionHandler(.noData)
            } onError: { error in
                // There was an error handling the notification.
                completionHandler(.noData)
            } onIgnore: {
                // The notification is an IAMPASS notification but should␣
↪be ignored.
                completionHandler(.noData)
            } defaultHandler: { userInfo in
                // The notification is not an IAMPASS notification.
                // The application should continue with its normal␣
↪notification handling.
```

(continues on next page)

(continued from previous page)

```
                completionHandler(.noData)
            }

        }else{
            completionHandler(.noData)
        }

    }
```

### 4.7.3 Logging

The IAMPASS iOS framework uses system logging to display diagnostic information.

CHAPTER

FIVE

# CUSTOM ANDROID APP GUIDE

By default IAMPASS will send authentication requests to the reference IAMPASS mobile application. IAMPASS Applications can be configured to send authentication requests to custom mobile applications. In this case the custom application is responsible for: * Registering the mobile device with IAMPASS * Collecting the required authentication data in response to authentication requests.

## 5.1 Installation

The easiest way to integrate IAMPASS into your application is to use the IAMPASS Libray, which is available from the Maven repositiory. Add the folowing to your project's build.gradle:

```
dependencies {
    implementation 'com.iampass.iampass:iampass:1.1.3'
}
```

## 5.2 Getting Started

### 5.2.1 Create an IAMPASS Account and Application

The first stage in integrating IAMPASS into your applicaton is to create an IAMPASS Application as described *here*.

**Save the application id and application secret for your IAMPASS Application you will need them to communicate with IAMPASS.**

### 5.2.2 Add the IAMPASSframework to your project

Add the folowing to your project's build.gradle:

```
dependencies {
    implementation 'com.iampass.iampass:iampass:1.1.3'
}
```

## 5.3 Application Configuration

### 5.3.1 Enable Push Notifications

IAMPASS uses Push Notifications to notify users of Authentication Requests. Use the Android Studio Firebase Assistant to add In App Messaging to your application (Tools -> Firebase).

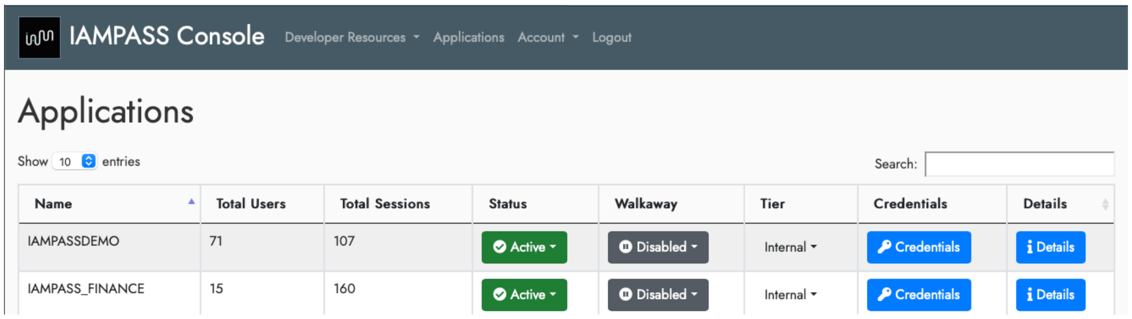
### 5.3.2 Configure IAMPASS Notifications Credentials

**In order to send Push Notifications to your application you need to configure your IAMPASS application:**

- Get the FCM API Key for your application. This can be found in the file google-services.json that was added to your project when you enabled FCM.:

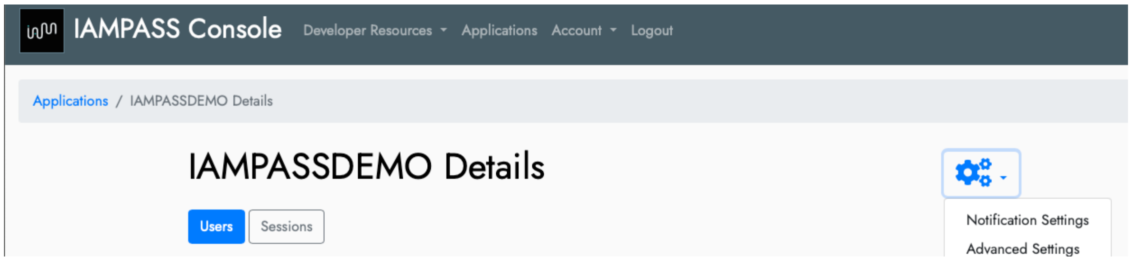
```
"api_key": [
{
  "current_key": "XXXXXXXXXXXXXXXXXXXX"
}]
```

- Open the IAMPASS Console
- Select your application and click the **Details** button.

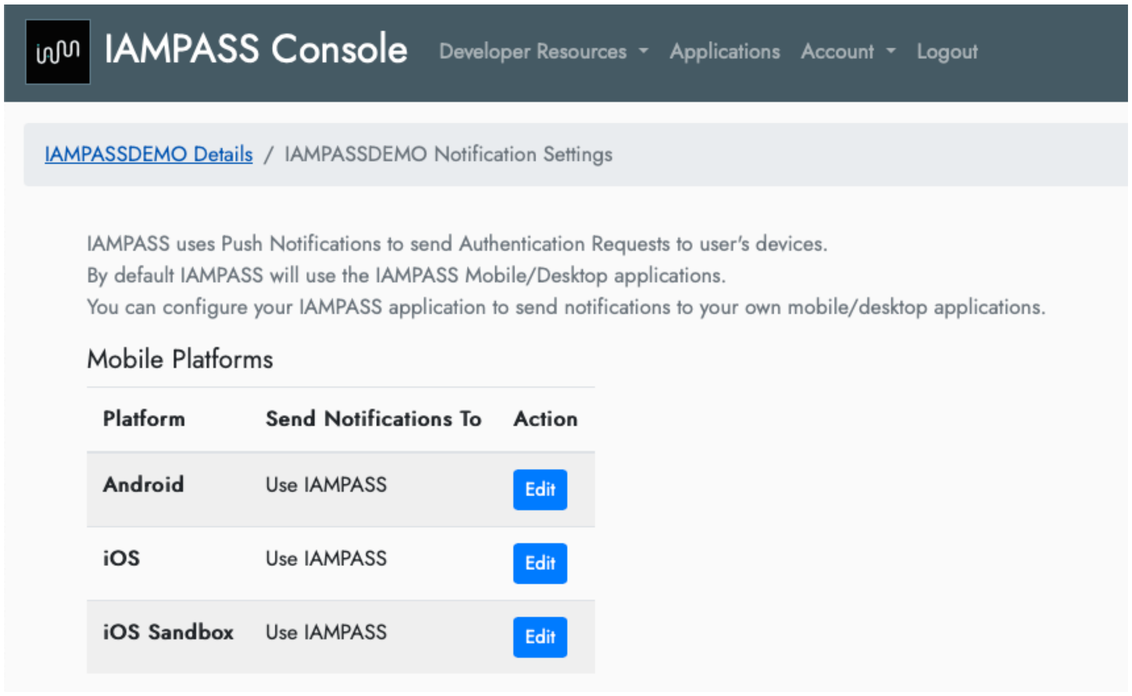


| Name | Total Users | Total Sessions | Status | Walkaway | Tier | Credentials | Details |
|---|---|---|---|---|---|---|---|
| IAMPASSDEMO | 71 | 107 | Active | Disabled | Internal | Credentials | Details |
| IAMPASS_FINANCE | 15 | 160 | Active | Disabled | Internal | Credentials | Details |

- In the details page click the settings icon and select **Notification Settings**.



- On the Notifications Settings page click the **Edit** button for iOS.



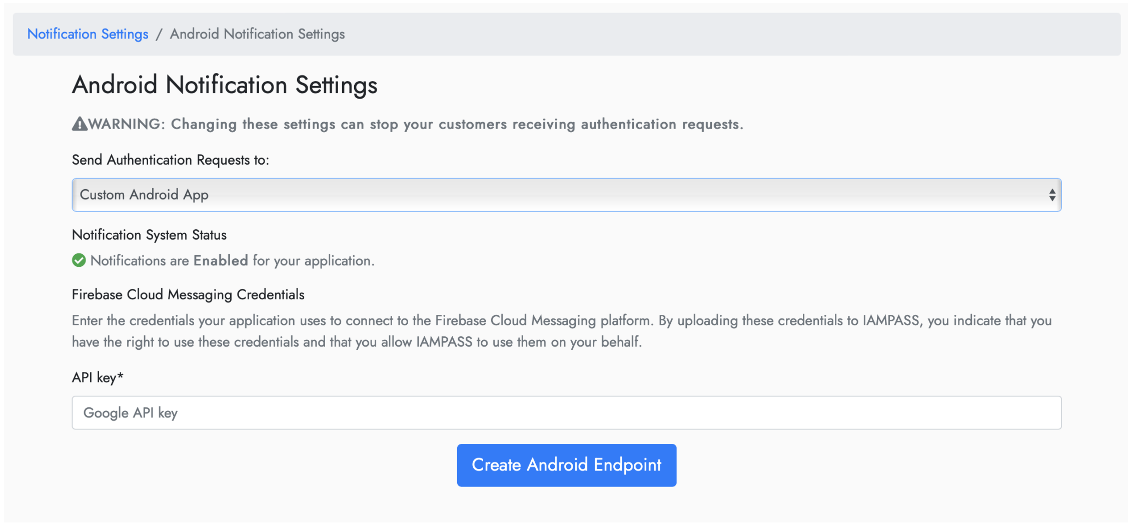
IAMPASSDEMO Details / IAMPASSDEMO Notification Settings

IAMPASS uses Push Notifications to send Authentication Requests to user's devices.
By default IAMPASS will use the IAMPASS Mobile/Desktop applications.
You can configure your IAMPASS application to send notifications to your own mobile/desktop applications.

Mobile Platforms

| Platform | Send Notifications To | Action |
|---|---|---|
| Android | Use IAMPASS | Edit |
| iOS | Use IAMPASS | Edit |
| iOS Sandbox | Use IAMPASS | Edit |

- On the **Android Notification Settings** page select **Custom Android App** from the dropdown.



Notification Settings / Android Notification Settings

Android Notification Settings

WARNING: Changing these settings can stop your customers receiving authentication requests.

Send Authentication Requests to: Custom Android App

Notification System Status

Notifications are **Enabled** for your application.

Firebase Cloud Messaging Credentials

Enter the credentials your application uses to connect to the Firebase Cloud Messaging platform. By uploading these credentials to IAMPASS, you indicate that you have the right to use these credentials and that you allow IAMPASS to use them on your behalf.

API key*: Google API key

- Update the credentials with your FCM API Key.

IAMPASS will now route your users authentication requests to your Android application.

### 5.3.3 Applications Entitlements

IAMPASS use Camera and Bluetooth services on your user's mobile device. You must add the following permissions to your application's manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="XXX">

    ...

    <uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
    <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.USE_BIOMETRIC" />
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-feature
        android:name="android.hardware.bluetooth_le"
        android:required="true" />
    ...
```

### 5.3.4 Setup Main Activity

The IAMPASS library provides an Activity `IPMainActivity` that performs most IAMPASS operations. You should derive your applications main activity from `IPMainActivity`.:

```
import com.iampass.iampass.ui.IPMainActivity
...
class MainActivity : IPMainActivity() {

private lateinit var appBarConfiguration: AppBarConfiguration
private lateinit var binding: ActivityMainBinding

override fun iampassInitialized() {

    Toast.makeText(applicationContext,"Initialized", Toast.LENGTH_SHORT).show()
}

override fun handleApplicationNotification(remoteMessage: com.google.firebase.
↪messaging.RemoteMessage) {
}
```

`IPMainActivity` has two abstract members::

```
override fun iampassInitialized() {
}
```

This method is called when IAMPASS has finished updating the user information stored on the device.:

```
override fun handleApplicationNotification(remoteMessage: com.google.firebase.
↪messaging.RemoteMessage) {

}
```

`IPMainActivity` has code to handle IAMPASS notifications. If the application receives a notification which is not an IAMPASS notification the application can handle them here.

### 5.3.5 User Storage

IAMPASS needs to store information about the users regeistered on the device. The class `IPUserManager` is used to store the data. The `IPUserManager` instance must be created before other IAMPASS functions are used. The recommended way to do this is to create a custom Application object and call `IPUserManager.create` in its `init` method.:

```
import com.iampass.iampass.IPDefaultSerializer
import com.iampass.iampass.IPUserManager

class ExampleApplication: Application() {

    init {
        IPUserManager.create(IPDefaultSerializer(application = this, "IAMPASS_EXAMPLE_
↪DATA"))
    }
}
```

To create the custom Application it is necessary to modify the app's manifest file (AndroidManifest.xml).:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...>

    <application
        android:name=".ExampleApplication"
```

**`IPUserManager` uses an object that implements `IPUserSerializer` to store information. The provided `IPDefaultSeri`**

- Create a new class derived from `IPDefaultSerializer`.
- Pass an instance of the custom serializer to `IPUserManager.create`.

## 5.4 Key Concepts

**When a system uses IAMPASS to authenticate its users**

- A notification is sent to the users registered device.
- The registered device collects the required information and send it to IAMPASS.
- IAMPASS processes the collected data and makes an authentication decision.

A custom IAMPASS mobile device is responsible for: * Registering the user's device. * Processing notifications from IAMPASS. * Collecting the required data.

## 5.5 Registering Users

IAMPASS does not manage your users or replace your sign up flow. You register your user with IAMPASS by providing a token that you can relate back to your user See *Getting Started* for information about user management. There are 2 common registration flows:

- User registers using your mobile application.
- User registers externally (in a web browser for example).

**You must wait until your application has completed the Push Notification registration process before registering devices**

## 5.5.1 Mobile Application Registration

The IAMPASS Android framework provides an interface for registering users and their mobile device. This example assumes that the mobile application has: * Registered for Push Notifications and stored the returned token in **NOTIFICATION_TOKEN** * Registered the user **user** with its own system.

**User registration is performed by an `Activity`, which is launched using `IPRegisterUserContract`**

```
import com.iampass.iampass.ui.IPRegisterUserContract
import com.iampass.iampass.ui.IPRegisterUserParams

// Create the registration launcher.
// The launcher must be created before the calling activities onCreate is called.

val registerLauncher = registerForActivityResult(IPRegisterUserContract())
↪{result->
    if (result.success){
        // Registration complete
    }else{
        // Registration failed
    }
}

/**
* Register a user
* @param userID an identifier for the user.
*/
fun registerUser(userID: String){
    val applicationID = getString(R.string.TEST_APPLICATION_ID)
    val applicationSecret = getString(R.string.TEST_APPLICATION_SECRET)
    val serverURL = getString(R.string.TEST_APPLICATION_SERVER)

    val urParams = IPRegisterUserParams(
        userName=userName,
        applicationID = XXXXX_APPLICATION_ID_XXXX,
        applicationSecret=XXXX_APPLICATION_SECRET_XXXX)
        )
    )

    registerLauncher.launch(urParams)

}
```

The IAMPASS Android framework includes a UI that is displayed when the user is being displayed. The UI used standard Android colors and fonts. You can customize the appearance using Android Themes or by providing a layout in your application called activity_register_user.xml. The layout must contain a TextView and a ProgressBar:

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
↪android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent">
```

(continues on next page)

(continued from previous page)

```
    <ProgressBar
        android:id="@+id/ip_register_user_progress"
        ...
    />

    <TextView
        android:id="@+id/ip_register_user_label"
        ... />
    ...
</androidx.constraintlayout.widget.ConstraintLayout>
```

### 5.5.2 External Registration

If you prefer for your users to register on a website, your users will have to register their mobile device. IAMPASS provides an API that can be used to obtain a registration link that can be used to register a mobile device. There are 2 ways to register the device: 1) Obtain a registration link in a web application and share it with your user. Call the registration URL from your mobile application, providing the information required to register the device.

```
import com.iampass.iampass.*

// Get the FCM token
FirebaseMessaging.getInstance().token
    .addOnCompleteListener(OnCompleteListener { task ->
        if (task.isSuccessful) {

            val notificationToken = task.result

            val registrationLink = URL("XXXX_EXTERNAL_REGISTRATION_LINK_XXXX")

            val userID: String = "IDENTIFIER FOR USER"

            // Successfully obtained a registration link so now register this device.
            IPUser.register(
                appContext = this.applicationContext,
                identifier = userID,
                registration_link = registrationLink,
                notification_token = notificationToken,
                completion = IPUser.RegisterUserCallback(
                    success = { identifier, user ->
                        IPUserManager.instance.addUser(identifier, user)
                        IPUserManager.instance.save()
                        // DONE
                    },
                    failure = { identifier, error ->
                        // Failed to register device.
                        // DONE
                    })
            )
        }
    })
```

2) Obtain the registration link in your mobile application, collect the required information and call the registration URL.

```
import com.iampass.iampass.*


// Get the FCM token
FirebaseMessaging.getInstance().token
    .addOnCompleteListener(OnCompleteListener { task ->
        if (task.isSuccessful) {

            val notificationToken = task.result

            val managementAPI = IPManagementAPI(
                applicationContext = this.applicationContext,
                application_id = XXXX_APPLICATION_ID_XXXX,
                application_secret = XXXX_APPLICATION_SECRET_XXXX
            )

            val userID: String = "IDENTIFIER FOR USER"
            val displayName: String = "VALUE TO DISPLAY FOR USERNAME"

            // Get a registration link for the user.
            managementAPI.getRegistrationLink(userID,
                displayName,
                IPManagementAPI.GetRegistrationLinkCallback(
                    success = { registration_link ->
                        // Successfully obtained a registration link so now register␣
↪this device.
                        IPUser.register(
                            appContext = this.applicationContext,
                            identifier = userID,
                            registration_link = registration_link,
                            notification_token = notificationToken,
                            completion = IPUser.RegisterUserCallback(
                                success = { identifier, user ->
                                    IPUserManager.instance.addUser(identifier, user)
                                    IPUserManager.instance.save()
                                    // DONE
                                },
                                failure = { identifier, error ->
                                    // Failed to register device.
                                    // DONE
                                })
                        )
                    },
                    failure = { error ->
                        // Failed to get registration link
                        // DONE
                    }
                ))
        }
    })
```

### 5.5.3 Storing Device Information

The client application should store the IPUser instance returned by device registration. `IPUserManager` can be used to store the user data. Before using `IPUserManager`, `IPUserManager.create` must be called passing in an `IPUserSerializer` instance that is responsible for serializing the data. IAMPASS includes the class `IPDefaultSerializer` which stores the user data using `EncryptedSharedPreferences`.

## 5.6 Training

After a device has been registered IAMPASS may have to collect some training data for the user. After registration check the *trainingRequired* property of `IPUser` to determine if IAMPASS needs to perform training.

```
if user.trainingRequired{
    // Perform training...
}
```

The IAMPASS Android framework includes the *IPFacialTrainingActivity* which provides the UI for performing training. To launch the activity us *IPFacialTrainingContract*. The IAMPASS iOS framework provides a default UI for performing training.

```
// Launcher for the facial training activity.
// This must be created before this Activity's `onCreate` is called.

private val facialTrainingLauncher =␣
↪registerForActivityResult(IPFacialTrainingContract()) { result->
    if(result.failed.isNotEmpty()){
        // Some users failed to train.
        // result.failed contains a Map<String,IPUser> containing the failed users.
    }
    }else{
        // Training completed.
    }
}

// Launch the training activity
val usersToTrain: Map<String,IPUser> = ...
facialTrainingLauncher.launch(IPFacialTrainingParams(usersToTrain))
```

The IAMPASS Android framework contains a standard UI for training. You can customize the apperance using Android Themes or by creating the following layouts:

1) activity_facial_training.xml This displays a message informing the user that the app need to perform training.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
↪android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    >

    <TextView
        android:id="@+id/ip_training_instructions_label"
        />

    <Button
        android:id="@+id/ip_training_start_button"
```

(continues on next page)

(continued from previous page)

```
        />

    <ProgressBar
        android:id="@+id/ip_upload_progress"
        />
</androidx.constraintlayout.widget.ConstraintLayout>
```

2) activity_face_capture.xml This actually does the image capture. The CameraView provides the video overlay. The ImageView draws the head outline over the video preview. If you provide your own layout you will have to add api 'com.otaliastudios:cameraview:2.7.2' to the dependencies of your application build.gradle.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
↪android.com/apk/res/android"
android:background="#FF444444">
    <com.otaliastudios.cameraview.CameraView
        android:id="@+id/camera"
        android:keepScreenOn="true"
        app:cameraAudio="off"
        app:cameraEngine="camera2"
        app:cameraFacing="front"
        >

        <ImageView
            android:id="@+id/image_overlay"
            android:contentDescription="@string/ip_background_face_overlay"
            android:scaleType="fitXY"
            app:layout_drawOnPictureSnapshot="false"
            app:layout_drawOnPreview="true"
            app:layout_drawOnVideoSnapshot="false"
             />
    </com.otaliastudios.cameraview.CameraView>

    <Button
        android:id="@+id/button_Cancel"
     />

    <ProgressBar
        android:id="@+id/faceProgress"
        style="@android:style/Widget.ProgressBar.Horizontal"
         />
</androidx.constraintlayout.widget.ConstraintLayout>
```

## 5.7 Logging

The IAMPASS Android framework uses the Timber logging library .

CHAPTER

SIX

# REST AUTHENTICATION

Client applications must pass authentication information with all API calls. The IAMPASS libraries have this functionality built into them and make integration easier. This document describes the authentication data required.

## 6.1 Authentication Protocol 1

In addition to connections be encrypted, they must also be authenticated. Authentication is performed using the Hash Message Authentication Code (HMAC) in conjunction with the SHA-256 algorithms as defined RFC 4868, specifically HMAC-SHA-256-128.

The HTTP Authentication Header is used to define the credentials. Further, additional headers are used to increase entropy, limit the life space of a HMAC and also link an specific authentication to the requested URL.

### 6.1.1 One-time-use Token

IAMPASS will assign each client application a unique client ID and a 192 bit (24 byte) Shared Secret.

The token is derived by creating a 64 bit (8 byte) nonce and pre-pending it to the 192 bit Shared Secret to create a 256 bit key.

The nonce must not be reused within a reasonable amount of time.

Further, the nonce should not be constructed only from printable characters. All 64 bits should have equal probability of being set to a 0 or 1.

A non-normative method for doing this is to use a pseudo-random number generator that is properly seeded with sufficient entropy and choose an unsigned number between 0 and 264 – 1. Section 2.1.1 of RFC 4868 prohibits the use of keys that are not identically 256 bit. Authentication will be rejected if the nonce is not 64 bits.

The 256 bit key (nonce + Shared Secret) are passed to a SHA-256 function. This will produce a 256 bit output and the left most 128 bits (see RFC 2104) are selected as the token.

In pseudo-code the procedure to produce the token would be:

```
nonce = PRG(0, 264 - 1, uniform distribution)
key = concatenate(nonce, Shared Secret)
SHA256_Output = SHA-256(key)
token = left_truncate(SHA256_Output, 128 bits)
```

### 6.1.2 HMAC Signature

After computing the token, the HMAC signature needs to be determined. This is done in a three step process. First the key is computed, followed by the digest and then the signature. This key consists of the concatenation of nonce, the Request URI and the authentication time stamp (represented as a string). The nonce must be represented as a string without any leading zeros removed as it must represent 64 bits of data. The Request URI must be exactly what is in the header, including scheme, resource and query string. The pseudo-code for producing the key: key = concatenate(string(nonce), Request URI, authentication time stamp) A non-normative example would be:

```
nonce = 9223372036854775807
application ID = ABCD
request_uri = https://main.iam-api.com/management/add_users/ABCD
time_stamp = 1234567890
key = string(nonce) + request_uri + string(time_stamp)
```

This would yield: key=9223372036854775807https://main.iam-api.com/management/add_users/ABCD

The digest is determined by computing the HMAC-SHA-256 using the token as the secret and the above key. This digest is than truncated to the left most 128 bits. The pseudo-code to compute the digest is: digest_256=HMAC-SHA-256(token, key) digest = left_truncate(digest_256, 128 bits) The final step is to determine the signature. The signature is the base-64 encoded representation of the digest. In pseudo-code this would be: signature = base64encode(digest)

### 6.1.3 HTTP Headers

Several headers are used to transmit the parameters needed for the authenticating system to reconstruct the digest it was sent. These headers are also used to define what authentication scheme is being used to create the HMAC digest, limit the time frame that a specific HMAC can be used. The time at which the API authentication request is being made is encoded in a header. The format of the time stamp is the number of seconds since January 1, 1970 00:00:00 GMT; known as ctime or linux time. The X-IAMPASS-Authentiaction-Timestamp header is used. The following is a non-normative example where 1234567890 represents the time of the request. X-IAMPASS-Authentiaction-Timestamp: 1234567890 The X-IAMPASS-Authentiaction-Version header is used to represent the version of the authentication scheme that is being used. The version described in this section would have the header: X-IAMPASS-Authentiaction-Version: 1 The Authentication header consists of the authentication scheme, hmac, and the authentication data. The authentication data has three components that are separated by a colon and no white space. The first portion is the identifier used by the client. The second component is the nonce used in generating the token and the signature and it must be converted from an integer to a string. Care must be taken that any leading zeros are not lost as the nonce must represent 64 bits. The final section is the signature. It has the following format: 1 Authentication: hmac client:nonce:signature