Activeconnect

Release 1.2

Activeconnect

Oct 20, 2020

CONTENTS

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

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

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

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

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

1.1.1 Creating your First Application

- Once you have completed the registration process you are ready to create your first application.
- **Activeconnect Applications** are how you connect your application to Activeconnect.
- Click the **'ADD APPLICATION'** button on the Activeconnect 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 Activeconnect 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 Activeconnect: *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 Activeconnect. 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 Activeconnect

In the example below, the application has a **User** table that contains a **name** column and an **Activeconnect 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

Activeconnect 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 Activeconnect, 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 Activeconnect 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 Activeconnect 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 Activeconnect Authentication API *Authentication API*

Activeconnect 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

Activeconnect 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 Activeconnect session call the **logout_url** route in this data.

CHAPTER

TWO

MANAGEMENT API

The Activeconnect Management API is used to manage your *Activeconnect 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 Activeconnect API must include authentication data as described in *API Authentication*

2.1 Adding Users

URL:: https://activeapi.ninja/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://activeapi.ninja/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']
    }
}
```

(continues on next page)

(continued from previous page)

```
HTTP Status 200

{
    '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://activeapi.ninja/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://activeapi.ninja/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://activeapi.ninja/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 Activeconnect 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://activeapi.ninja/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://activeapi.ninja/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://activeapi.ninja/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://activeapi.ninja/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://activeapi.ninja/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 Activeconnect 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://activeapi.ninja/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)** Activeconnect will select appropriate methods.

type application_id: string.

return Authentication session data in json and http status code.

Example:

```
curl -X POST https://activeapi.ninja/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 Activeconnect will send authentication requests to the reference Activeconnect mobile application. Activeconnect 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 Activeconnect * Collecting the required authentication data in response to authentication requests.

The easiest way to provide this functionality in a custom iOS application is to integrate the iOS framework into the custom application. The iOS framework can be downloaded from GitHub.

4.1 Installation

The easiest way to integrate Activeconnect into your application is to use CocoaPods. To integrate ActiveconnectiOS into your xCode project using CocoaPods, specify it in your *podfile*. `` ` pod 'ActiveconnectiOS', '~>0.1.0' ` ``

4.2 Reference

The ActiveconnectiOS framework reference documentation can be found here.

4.3 Getting Started

4.3.1 Create an Activeconnect Account and Application

The first stage in integrating Activeconnect into your applicaton is to create an Activeconnect Application as described here. **Save the application id and application secret for your Activeconnec Application you will need them to communicate with Activeconnect.**

4.3.2 Add the ActiveconnectiOS framework to your project

To integrate ActiveconnectiOS into your xCode project using CocoaPods, specify it in your *podfile*. ` pod 'ActiveconnectiOS', '~>0.1.0' `

4.4 Application Configuration

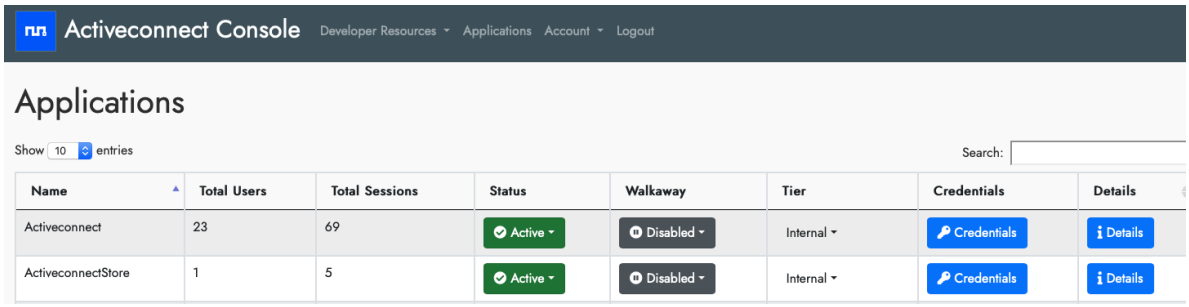
4.4.1 Enable Push Notifications

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

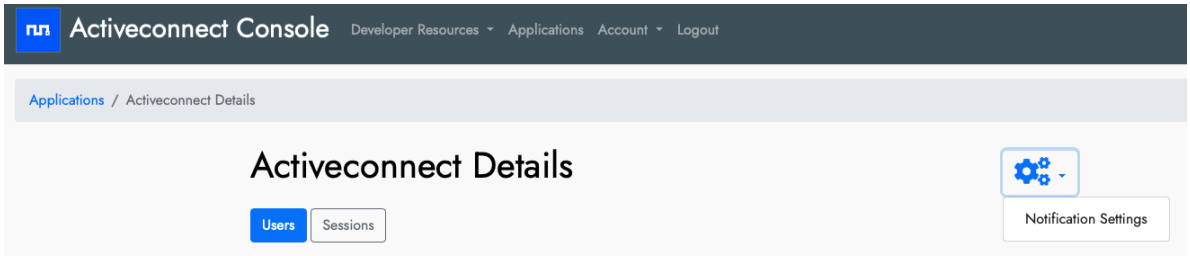
4.4.2 Configure Activeconnect Notifications Credentials

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

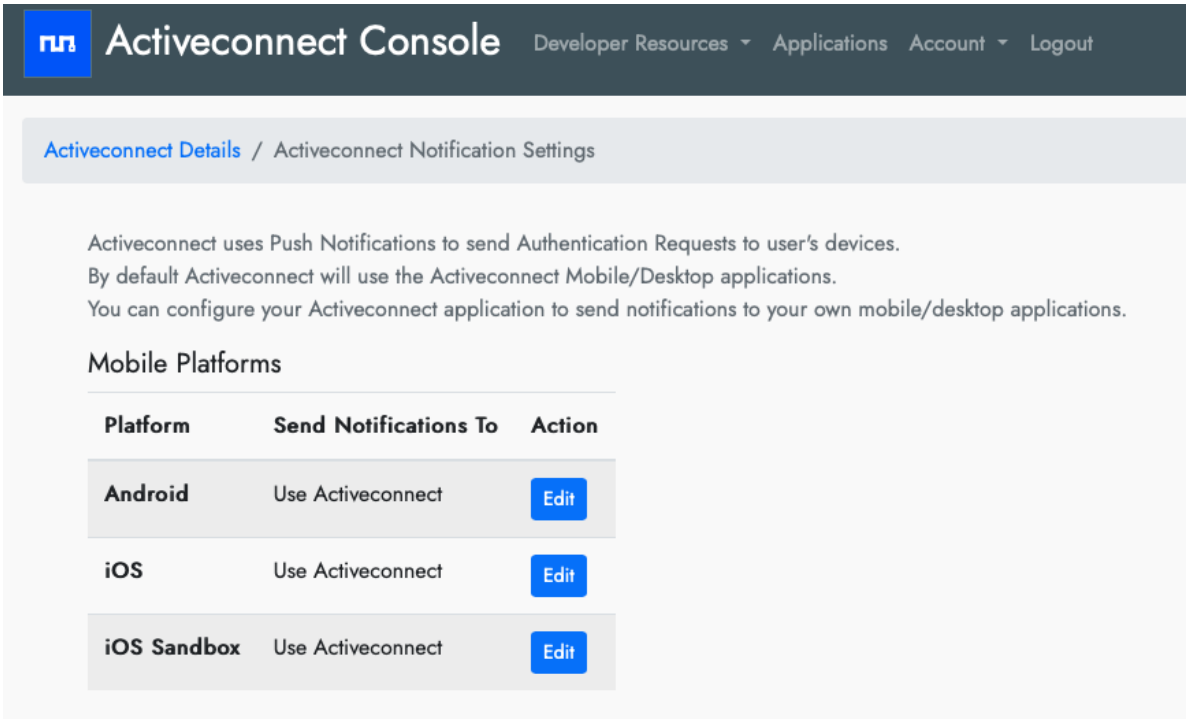
- Download the APNS certificates for your application from the Apple Developer Portal.
- Import the certificate and private key into KeyChain (both development and production keys).
- Export the certificate and private key from KeyChain as .p12 files.
- Open the Activeconnect Console
- Select your application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



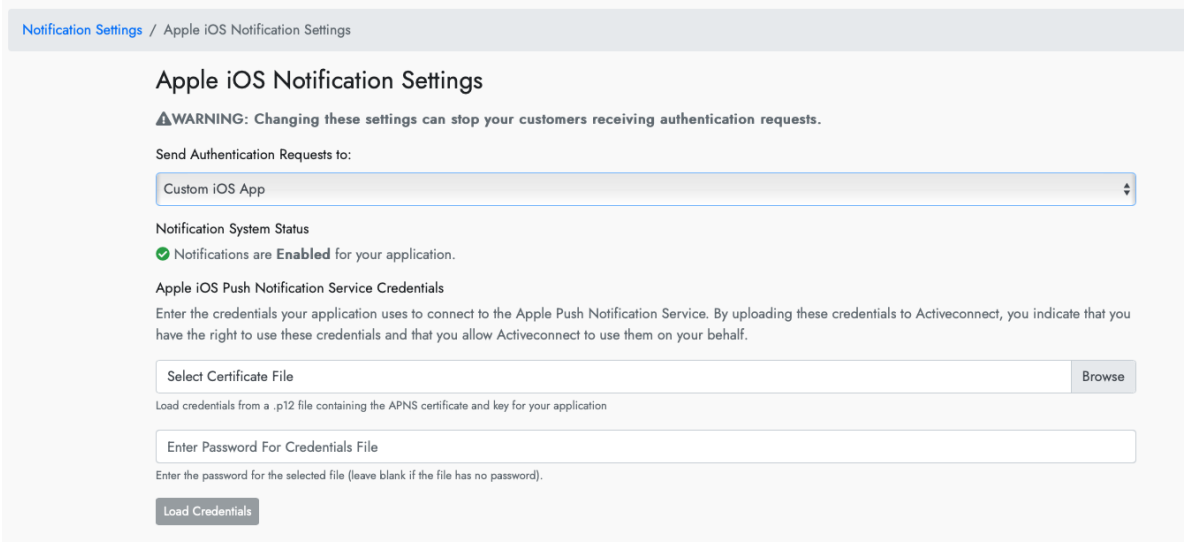
- In the details page click the settings icon and select **Notification Settings**.



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



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

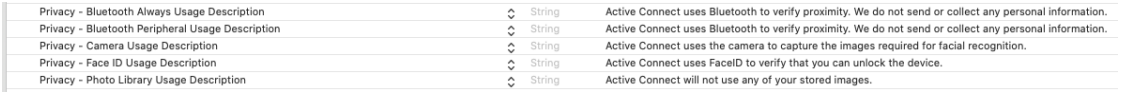


- Update the credentials with your APNS certificate and Private Key.
- Repeat the process for iOS Sandbox using your development credentials.

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

4.4.3 Applications Entitlements

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

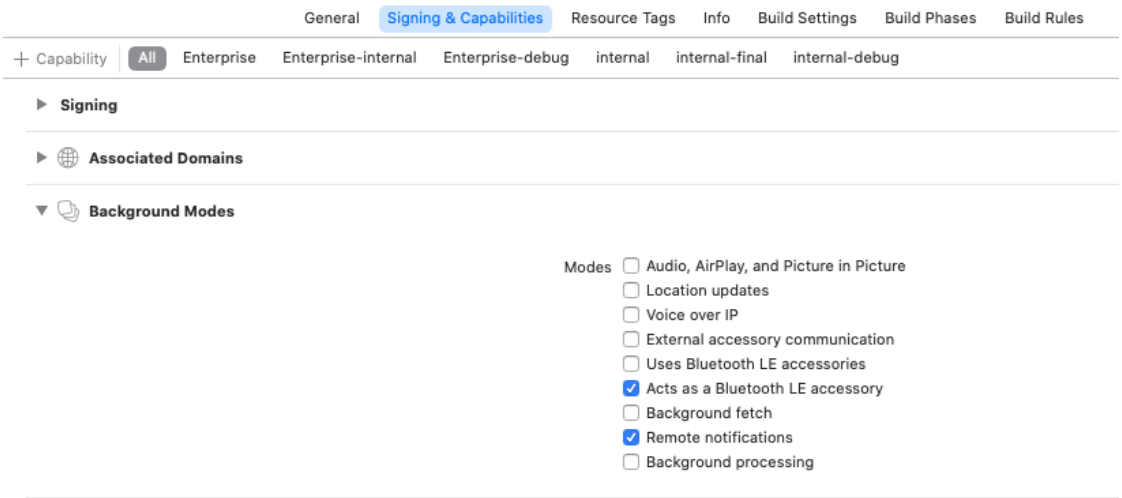


key	value
NSBluetoothAlwaysUsageDescription	uses Bluetooth to verify proximity
NSBluetoothPeripheralUsageDescription	uses Bluetooth to verify proximity
NSCameraUsageDescription	uses the camera to capture images required for facial recognition.
NSFaceIDUsageDescription	uses FaceID to verify that you can 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



4.5 Key Concepts

When a system uses Activeconnect to authenticate its users:

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

A custom Activeconnect mobile device is responsible for:

- Registering the user's device.
- Processing notifications from Activeconnect.

- Collecting the required data.

4.5.1 Registering Users

Activeconnect does not manage your users or replace your sign up flow. You register your user with Activeconnect 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

Mobile Application Registration

The Activeconnect iOS 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.

```
import ActiveconnectiOS
...
// user: client generated identifer for the client applications user.
// NOTIFICATION_TOKEN: The token received by the application when it␣
↪registers for Push Notifications
// MY_APPLICATION_ID: The ID of the Activeconnect application (from␣
↪Activeconnect Console)
// MY_APPLICATION_SECRET: The Application Secret of the Activeconnect␣
↪application (from Activeconnect Console)

// Create an Activeconnect Management API instance using credentials for␣
↪client application
let management_api = ManagementAPI(application_id: MY_APPLICATION_ID,
↪application_secret: MY_APPLICATION_SECRET)

// Register the user with Activeconnect and register this device.
management_api.create_user_and_register_device(user: user, notification_
↪token: NOTIFICATION_TOKEN) {(user, device) in
    // user: The identifier of the user (same as passed to create_user_and_
↪register_device)
    // device: ACMobileDevice that contains user and device information␣
↪required for subequent Activeconect calls.

    // This example encodes the ACDeviceData as JSON and stores it in user␣
↪defaults.
    let encoder = JSONEncoder()

    if let encoded = try? encoder.encode(device){
        let defaults = UserDefaults.standard
        defaults.set(encoded, forKey: "user_data")
        defaults.set(user, forKey: "user_name")
    }
} failure: { (user, error) in
```

(continues on next page)

(continued from previous page)

```
    // user: The identifier of the user (same as passed to create_user_and_
↪register_device)
    // error: Error that describes failure reason.
}
```

External Registration

Activeconnect 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 Activeconnect iOS framework provides a method to get a registration link for the user

```
import ActiveconnectiOS
...
// 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 Activeconnect application (from␣
↪Activeconnect Console)
// MY_APPLICATION_SECRET: The Application Secret of the Activeconnect␣
↪application (from Activeconnect Console)


// Create an Activeconnect 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 ActiveconnectiOS
...
ACMobileDevice.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: ACMobileDevice that contains user and device information␣
↪required for subequent Activeconect calls.
            print("registered")

        } failure: { (identifier, error) in
            // identifier: identifier for user (same as value passed to␣
↪registerDevice)
            // error: Error indicating failure reason
```

(continues on next page)

(continued from previous page)

```
            print("failed")
        }
```

Storing Device Information

The client application should store the ACMobileDevice instance returned by device registration. ACMobileDevice 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.

4.5.2 Training

After a device has been registered Activeconnect may have to collect some training data for the device. After registration check the *training_required* property of *ACMobileDevice* to determine if Activeconnect needs to perform training.

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

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

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

    func doTraining()->Void{
        let device = self.get_device()
        let user_id = self.get_user_id()

        var training_vc = ACTrainingViewController()
        training_vc.device = device
        training_vc.identifier = user_id
        training_vc.delegate = self

        // Present the view controller
        self.present(training_vc, animated: true, completion: nil)
    }

    func get_device()->ACMobileDevice{
        // Return registered device information
    }

    func get_user_id()->String{
        // Return the user id
    }

    func save_device(user: Any?, device: ACDeviceData)->Void{
        // Store the update device data.
    }

}
// Implementation of ACTrainingDelegate
extension MainViewController: ACTrainingDelegate{
```

(continues on next page)

(continued from previous page)

```
    // Training is complete, update the stored device information
    func didCompleteTraining(identifier: Any?, device: ACMobileDevice) {
        print("completed training")
        self.save_device(...)
    }

    func didFailToCompleteTraining(identifier: Any?, device: ACMobileDevice,␣
↪error: Error?) {
        print("training failed")
    }
}
```

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

4.5.3 Updating Device Information

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

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

    // 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")
        }

    }

}
```

4.5.4 Handling Authentication Requests

Activeconnect 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" = "Activeconnect® 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 Activeconnect iOS framework, which will decode the payload and carry out the required authentication steps.

```
var notificationHandler: ACNotificationHandler?


class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication,␣
↪didFinishLaunchingWithOptions launchOptions: [UIApplication.
↪LaunchOptionsKey: Any]?) -> Bool {

        self.notificationHandler = ACNotificationHandler(delegate: self)
        return true
    }

    // Client implemented function that gets any device information stored␣
↪on this device.
    func get_registered_devices()->[ACMobileDevice]{

    }

    // Application has received a Push Notification.
    func application(   _ application: UIApplication,
                        didReceiveRemoteNotification userInfo: [AnyHashable:␣
↪Any],
                        fetchCompletionHandler completionHandler:
                        @escaping (UIBackgroundFetchResult) -> Void) {

        // Get the device information stored on this device
        let registeredDevices: self.get_registered_devices()

        // Has the notification handler been initialized?
        if let notificationHandler = self.notificationHandler{

            // Pass the notification to the notification handler along with␣
↪the list of registered devices.
            // The notification handler will check that the notification is␣
↪for the device.
            // The notification handler will call either␣
↪presentAuthenticationUI or sessionStatusChanged
```

(continues on next page)

(continued from previous page)

```
            // methods of its delegate if it handles the notification.
            let activeconnectResponse = notificationHandler.
↪processNotification(userInfo: userInfo, registeredDevices:␣
↪registeredDevices)

            if activeconnectResponse.activeconnectNotification{
                completionHandler(activeconnectResponse.
↪suggestedCompletionResult)
                return
            }else{
                // This is not an Activeconnect notification so continue␣
↪regular notification handling.
            }
        }
        // Perform regular notification handling.
        completionHandler(.noData)
    }
}

extension AppDelegate: ACNotificationHandlerDelegate{
    func sessionStatusChanged(status: ACSessionStatus) {
        print("session status has changed")
    }

    func presentAuthenticationUI(request: ACAuthenticationRequest, device:␣
↪ACMobileDevice) {
        // Present the authentication flow.
    }
}
```

Presenting the Authentication UI

The Activeconnect iOS framework provides the class *ACAuthenticationViewController* to display the authentication UI. The *ACNotificationHandler.presentAuthenticationUI* should create an instance of *ACAuthenticationViewController*, set the *authenticationRequest* and *mobileDevice* properties and display the view controller.

In iOS applications, notifications are received by the AppDelegate but there is no 'recommended' way of presenting a ViewController from the AppDelegate.

When the *ACAuthenticationViewController* completes it will call either the *didAuthenticate* or *didFailToAuthenticate* member of its *authenticationDelegate*. The client application is responsible for dismissing the *ACAuthenticationViewController*.

```
class MainViewController: UIViewController{
    ...
    func presentAuthenticationUI( request: ACAuthenticationRequest, device:␣
↪ACMobileDevice ) -> Void{
        let authenticationController = ACAuthenticationViewController()
        authenticationController.authenticationRequest = request
        authenticationController.mobileDevice = device
        authenticationController.authenticationDelegate = self
        authenticationController.modalPresentationStyle = .fullScreen
        self.present(authenticationController, animated: true, completion:␣
↪nil
    }
```

(continues on next page)

(continued from previous page)

```
}

extension MainViewController: ACAuthenticationDelegate{
    func didAuthenticate() {
        print("authenticated")
        // return UI to previous state.
    }

    func didFailToAuthenticate(error: Error?) {
        print("failed to authenticate")
        // return UI to previous state.
    }
}
```

You can create a ACAuthenticationViewController in a StoryBoard or Nib file rather than creating it programatically.

4.5.5 Logging

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

CHAPTER

FIVE

REST AUTHENTICATION

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

5.1 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.

5.1.1 One-time-use Token

Activeconnect 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)
```

5.1.2 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 = 92233720368547758O7
application ID = ABCD
request_uri = https://activeconnect.activeapi.ninja/management/add_users/ABCD
time_stamp = 1234567890
key = string(nonce) + request_uri + string(time_stamp)
```

This would yield: key=9223372036854775807https://activeconnect.activeapi.ninja/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)

5.1.3 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-Activeconnect-Authentiaction-Timestamp header is used. The following is a non-normative example where 1234567890 represents the time of the request. X-Activeconnect-Authentiaction-Timestamp: 1234567890 The X-Activeconnect-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-Activeconnect-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