

Base Integration Guide

Workspace ONE for iOS and iPadOS

Applications for iOS and iPadOS can be integrated with the Omnisia Workspace ONE® platform, by using its mobile software development kit. Complete the tasks below as a base for feature integration.

This document is part of the Workspace ONE Integration Guide set.

Table of Contents

Welcome.....	2
Downloads.....	2
Compatibility.....	3
Task: Configure application properties.....	4
Task: Add the software development kit package.....	16
Task: Initialize the software development kit runtime.....	20
Task: Declare supported features.....	35
Task: Demonstrate Basic Features.....	40
Next Steps.....	50
Appendix: Callback Scheme Sample Code.....	51
Appendix: Troubleshooting.....	54
Appendix: Keychain Clearance Sample Code.....	60
Document Information.....	62

Welcome

Welcome to the Workspace ONE Base Integration Guide for iOS and iPadOS.

The objective of this guide is for you to complete initial integration of your app with the Workspace ONE platform. This guide includes coding work.

Instructions in this guide assume that all the tasks in the Workspace ONE Integration Preparation Guide for iOS and iPadOS are already complete.

The Integration Preparation Guide is available

- in Markdown format, in the repository that also holds this guide:
github.com/euc-releases/ws1-sdk-integration-samples.
- in Portable Document Format (PDF), from the SDK home page on the Omnisia developer website:
developer.omnisia.com/ws1-uem-sdk-for-ios/integration/WorkspaceONE_iOS_BaseIntegration.

Warning: Don't change the bundle identifier. Your app must have the same bundle identifier as it did when the Integration Preparation Guide was being followed. This is because trust of the app by the Workspace ONE platform is based in part on the bundle identifier. The Workspace ONE unified endpoint manager (UEM) console and the Workspace ONE Intelligent Hub app are the components that must trust your app.

The Integration Preparation Guide has some discussion of Apple accounts as they pertain to Workspace ONE integration work, in an appendix.

Downloads

Omnissia provides this Software Development Kit (the “Software”) to you subject to the following terms and conditions. By downloading, installing, or using the Software, you agree to be bound by the terms of [SDK License Agreement](#). If you disagree with any of the terms, then do not use the Software.

For additional information, please visit the [Omnissia Legal Center](#).

Compatibility

Instructions in this document are compatible with the following software versions.

Software	Version
Workspace ONE software development kit for iOS	26.01
Workspace ONE UEM management console	2402 or later
Apple iOS and iPadOS	16 or later
Apple Xcode	16.4 or later

The SDK supports versions of the Swift language that are supported by the above Xcode versions.

Task: Configure application properties

Configuring application properties is a Workspace ONE platform integration task for mobile application developers. This task is dependent on all the tasks in the Integration Preparation Guide, as discussed in the [Welcome](#) section. The following instructions assume that the dependent tasks are complete already.

These are the required configurations.

- The Workspace ONE Intelligent Hub app will use a **custom URL scheme** to send enrollment details to your app. Therefore your app must declare a custom URL scheme.
- Particular **Queried URL Schemes** will be used to discover Hub communication channels and must be declared by all SDK apps.
- A **Camera Usage Description** must be declared by all SDK apps in order to support Workspace ONE QR code enrollment. If a description isn't declared then the operating system blocks the app from access to the device camera.
- A **Face ID Usage Description** must be declared in order to support face recognition for biometric authentication. If a description isn't declared then the operating system blocks the app from access to Face ID. Note that no declaration is needed for access to Touch ID.

Biometric authentication may be allowed in the Workspace ONE unified endpoint manager (UEM) console, as a security policy. Therefore it must be supported by all SDK apps.

- **Workspace ONE doesn't support multiple windows** at time of writing. If your app has a scene manifest then it must declare that multiple windows aren't supported.
- If you are developing more than one app, then add a **shared keychain group** for secure inter-application communication. The shared keychain group must be named **asdk** and be first in the Keychain Groups list.

You can set those configurations as you like if you are familiar with the Xcode project user interface already. Skip ahead to the screen captures at the end of the second and third sections for reference. Or you can follow these step-by-step instructions.

Declare a custom URL scheme

Start the [Task: Configure application properties](#) by declaring a custom URL (Uniform Resource Locator) scheme for your app.

If you haven't installed your application via Workspace ONE at least once, then do so now. If you don't, the application under development won't work when installed via Xcode. Instructions for installing via Workspace ONE can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.

To declare a custom URL scheme proceed as follows.

1. **Open the application project in Xcode.**
2. **In the navigator select the project itself, and then the target that you're using for integration.**

By default, the project itself will be the first item in the navigator, and each app project has only a single target.

3. **In the target editor, select the Info tab, and then expand the URL Types list.**

By default the URL Types list is empty.

4. **Click the plus button to add a URL Type.**

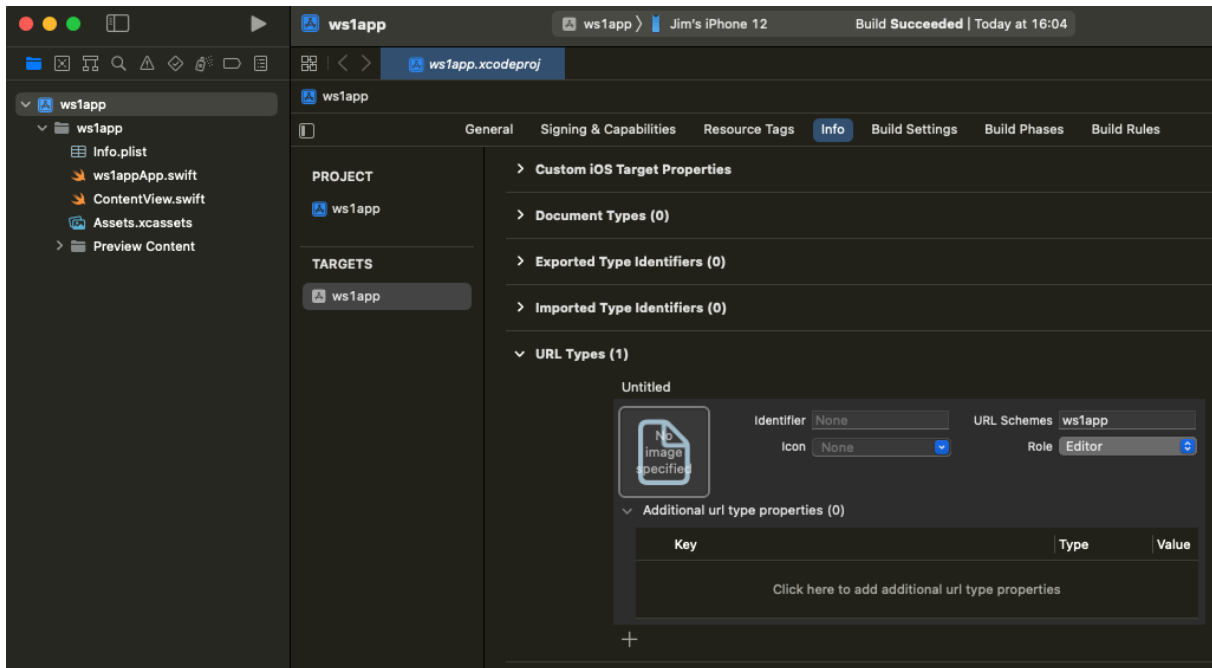
This adds an Untitled item to the list.

5. Enter a suitable value like **ws1app** in the URL Schemes text box.

The value will be used as the scheme in a URL (Uniform Resource Locator). This means that it cannot include characters such as space, underscore, and full stop (period).

None of the other fields matter for this task.

This screen capture shows how the added URL Type might appear in the Xcode user interface.



Screen Capture: Xcode Target URL Types

6. Install and run the app on your developer device to confirm that no mistakes have been made.

7. Check that the custom URL scheme invokes the app.

First, terminate the app, for example by using the device task switcher.

If your device is synchronized with your computer, open the Apple Notes app and create a new note. Add a URL like this to the note.

```
ws1app://dummpath
```

The URL has the value you entered in Xcode as the scheme, and has a dummy path.

Wait for the note to synchronise to your developer device, typically only a second or two, then open the mobile Notes app, open the note, and tap the URL. What should happen is that your app is opened.

If your app doesn't open then review the above instructions to check that no mistakes were made. Also check any settings related to Info.plist in Xcode.

If your Notes aren't synchronised you could instead open Safari on the device and type the URL into the address bar. Safari should open your app.

That completes declaration of a custom URL scheme.

Xcode might or might not have added a visible property list file named Info.plist to the project when you added an item to the URL Types list. It might or might not be possible to have added the declaration and to add the other required properties, below, by editing the Info.plist file. This guide assumes that it isn't possible.

Continue the task with the instructions [Add Queried URL Schemes and other required properties](#).

Add Queried URL Schemes and other required properties

After following the [Declare a custom URL scheme](#) instructions, add the other required Workspace ONE properties.

Here are some tips for adding properties in Xcode.

- App properties can be edited on the same screen as the custom URL scheme is declared, see above, but in the Custom iOS Target Properties list.
- When you right-click on a non-array key in the list, a context menu will appear. That menu is referred to here as the *properties context menu*.
- The list can show keys in either a raw form or in a wordy form. You can switch between the two in the properties context menu.
- Properties can be added by selecting the Add Row option in the properties context menu.
- After you add a row you first enter the key for the new property. The Xcode helper for key entry here is case sensitive.
- Properties of type Array or Dictionary can have sub-properties. Sub-properties can be added by expanding the top-level property and then clicking the plus button that appears.
- Changes made in the property list can be reversed using the usual undo interactions, Cmd+Z for example.

This is a list of the required Workspace ONE app properties and values. Keys are given in raw form and wordy form.

- **LSApplicationQueriesSchemes** Queried URL Schemes.
Array of String. Items can be in any order.
 - **airwatch**
 - **AWSS0Broker2**
 - **aws1enroll**
 - **wsoneSDK**
- **NSCameraUsageDescription** Privacy - Camera Usage Description.
String “Scan QR codes” or something similar.
- **NSFaceIDUsageDescription** Privacy - Face ID Usage Description.
String “Unlock the app” or something similar.
- **UIApplicationSceneManifest** Application Scene Manifest.
Dictionary.
 - **UIApplicationSupportsMultipleScenes** Enable Multiple Windows.
Boolean NO

You can add those properties as you like if you are familiar with the editor already, or follow these step-by-step instructions.

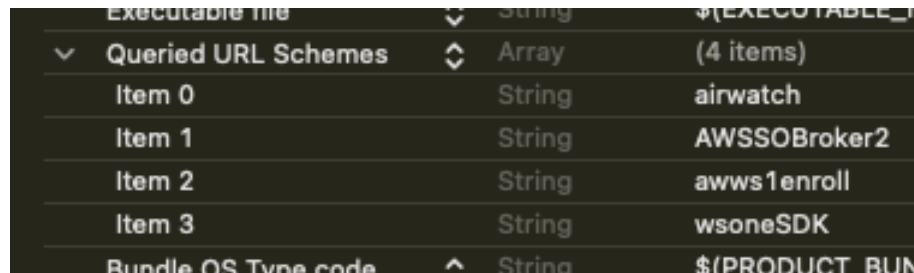
1. In Xcode, in the target editor, on the Info tab, expand the Custom iOS Target Properties list.
2. Add a top-level property row with the key: Queried URL Schemes.
Raw equivalent is **LSApplicationQueriesSchemes**.

Ensure that the type of the property is Array, and the type of items in the array is String. Those could be the defaults.

3. Add an item for each of these values to the new array.

- airwatch
- AWSS0Broker2
- awws1enroll
- wsoneSDK

This screen capture shows the new key and array as they appear in the Xcode user interface.



Executable file	String	\$(EXECUTABLE_1
▼ Queried URL Schemes	Array	(4 items)
Item 0	String	airwatch
Item 1	String	AWSS0Broker2
Item 2	String	awws1enroll
Item 3	String	wsoneSDK
Bundle OS Type code	String	\$(PRODUCT_BUN

Screen Capture: Xcode Queried URL Schemes

4. Add a top-level property row with the key: Privacy - Camera Usage Description.
Raw equivalent is **NSCameraUsageDescription**.

Ensure that the type of the property is String. This should be the default.

Set the value to something like: "Scan QR codes".

5. Add a top-level property row with the key: Privacy - Face ID Usage Description.
Raw equivalent is **NSFaceIDUsageDescription**.

Ensure that the type of the property is String. This should be the default.

Set the value to something like: "Unlock the app".

6. Check if your app has a top-level property row with the key: Application Scene Manifest.

Raw equivalent is **UIApplicationSceneManifest**.

If your app doesn't have this property you can skip the remaining steps.

This property will be of type Dictionary.

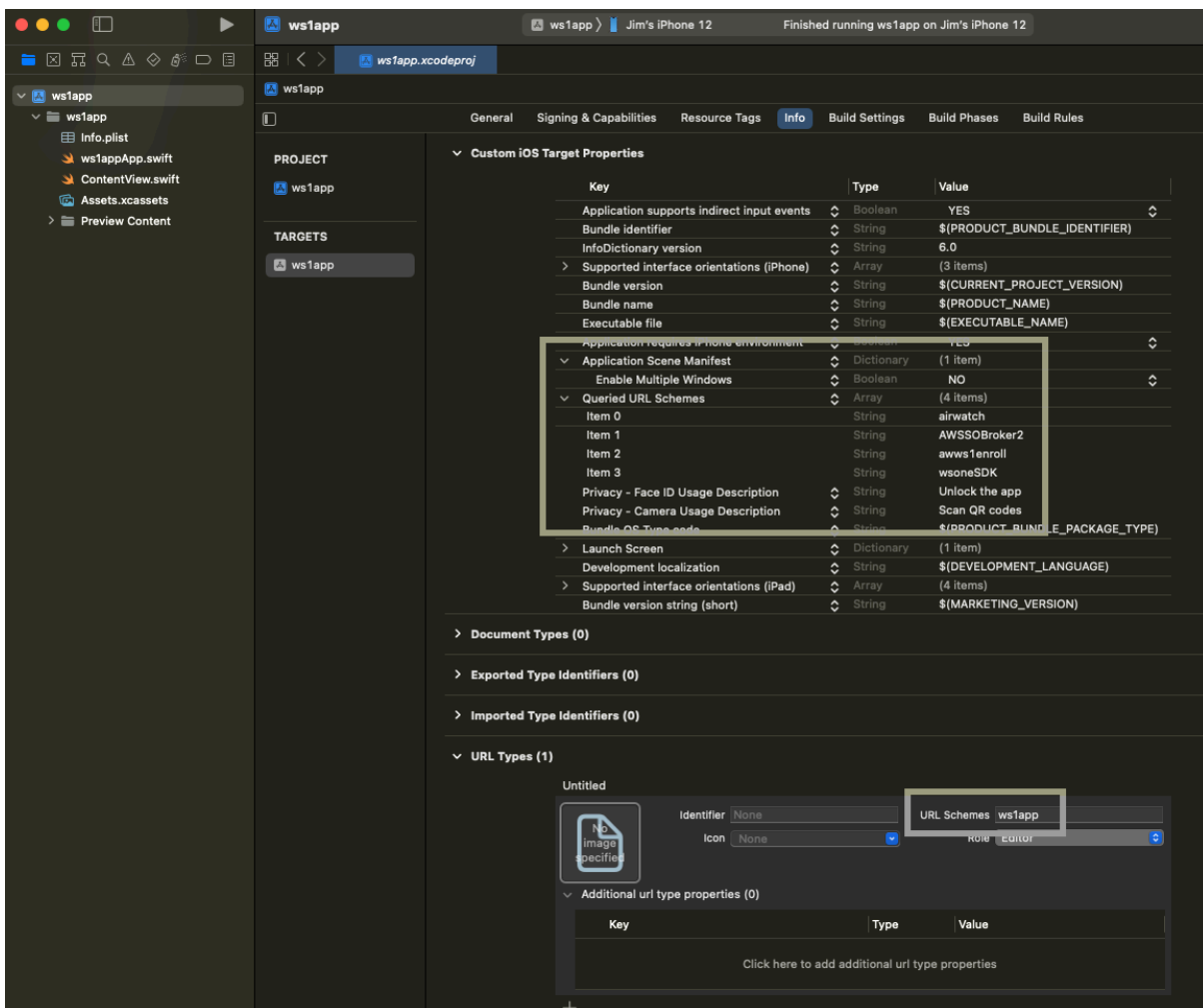
7. Expand the dictionary and find the sub-property row with the key: Enable Multiple Windows.

Raw equivalent is **UIApplicationSupportsMultipleScenes**.

Ensure that the type of the sub-property is Boolean. This should be the default.

Set the value to: NO.

That completes the configuration of application properties. This screen capture shows how the target Info tab could look.



Screen Capture: Xcode target Info tab

If you haven't installed your application via Workspace ONE at least once, then do so now. If you don't, the application under development won't work when installed via Xcode. Instructions for installing via Workspace ONE can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.

Build and run the application from Xcode to confirm that no mistakes have been made.

You are now ready to proceed to either of the following.

- The instructions to [Add a shared keychain group](#) if you are developing more than one custom app.
- The next [Task: Add the software development kit package](#) if you are developing a single app at this time.

Add a shared keychain group

The instructions to add a shared iOS keychain group may be followed after those in the other sections in the [Task: Configure application properties](#), but are optional. An iOS shared keychain group is only required if you have multiple custom SDK apps that work together as a suite.

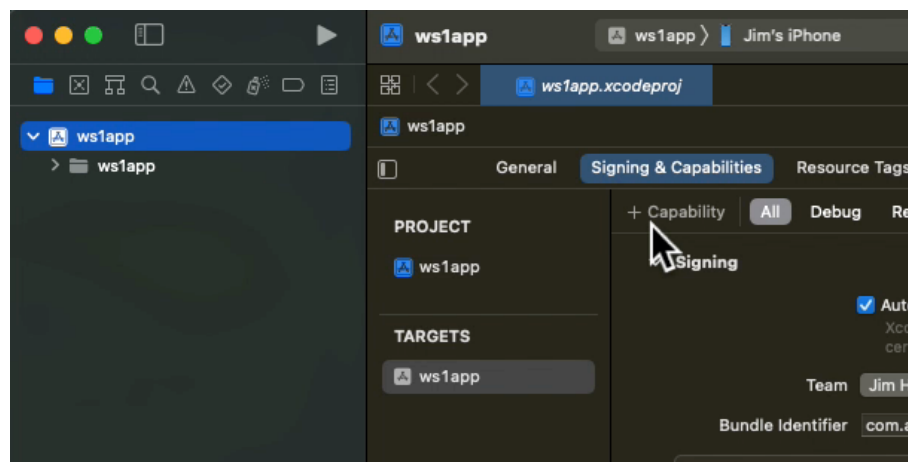
If you add a shared keychain group then the apps in the suite

- must be signed by the same app developer team.
- can share an app single sign-on (SSO) session, if configured in the management console security policies.
- can exchange data securely, via the pasteboard for example.

To add a shared keychain group proceed as follows.

1. In Xcode, in the target editor, select the **Signing & Capabilities** tab.
2. Click the plus button to add a capability.

This screen capture shows the location in the Xcode user interface.

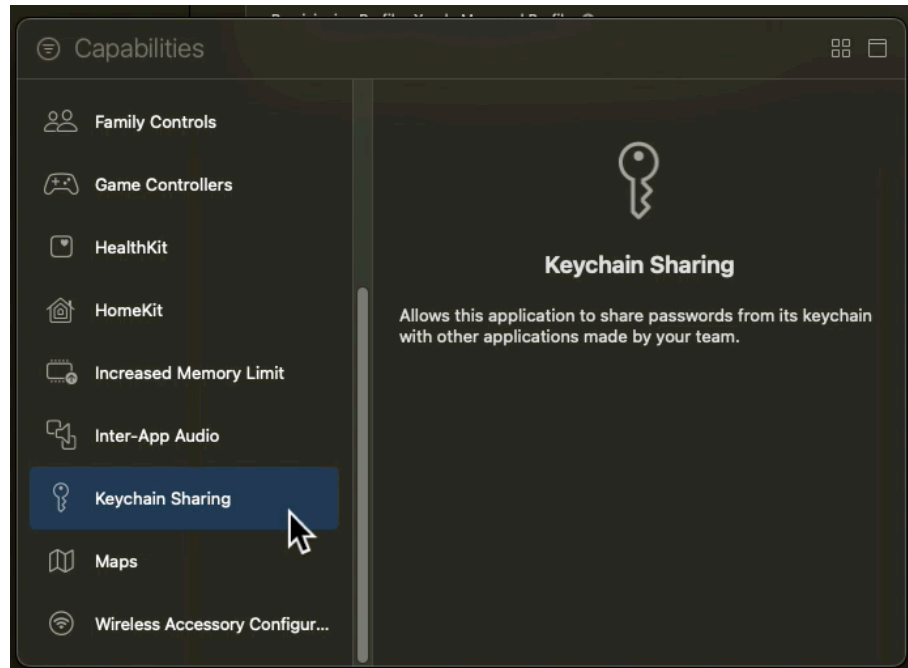


Screen Capture: Xcode Add App Capability

This opens a dialog on which you can select a capability.

3. **Search for, or scroll to, the capability Keychain Sharing.**

This screen capture shows how it might appear in the Xcode user interface.



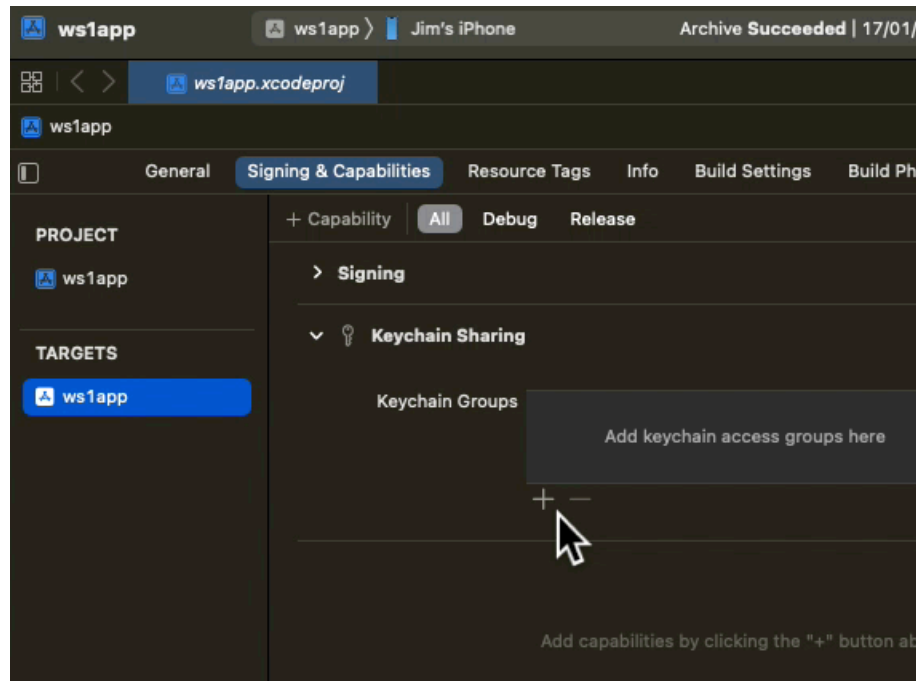
Screen Capture: Xcode Keychain Sharing

4. **Double click on Keychain Sharing.**

This closes the capability selection dialog. You will be returned to the Xcode target editor. The capability Keychain Sharing has now been added to your app.

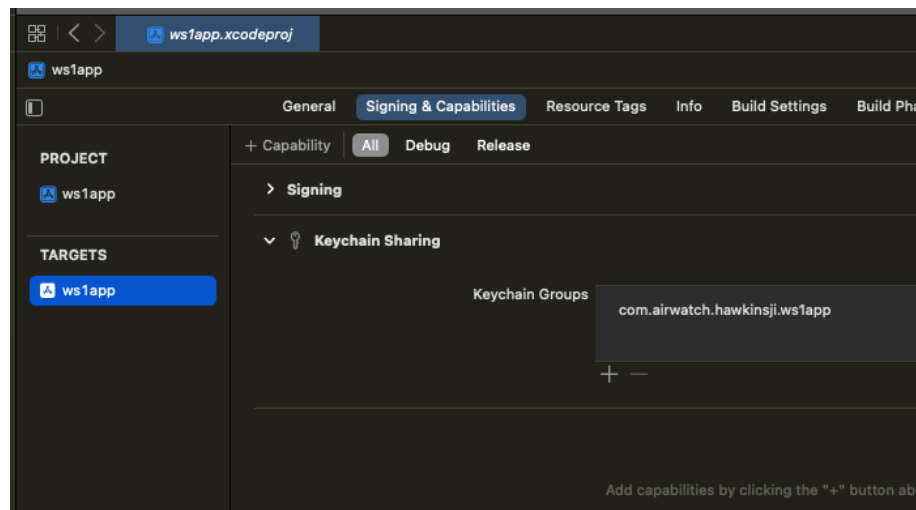
5. Click the plus button under the prompt to Add keychain access groups here.

This screen capture shows the location in the Xcode user interface.



Screen Capture: Xcode Add First Keychain Group

That adds a keychain group with the same name as your app's bundle identifier. This screen capture shows it might appear in the Xcode user interface.



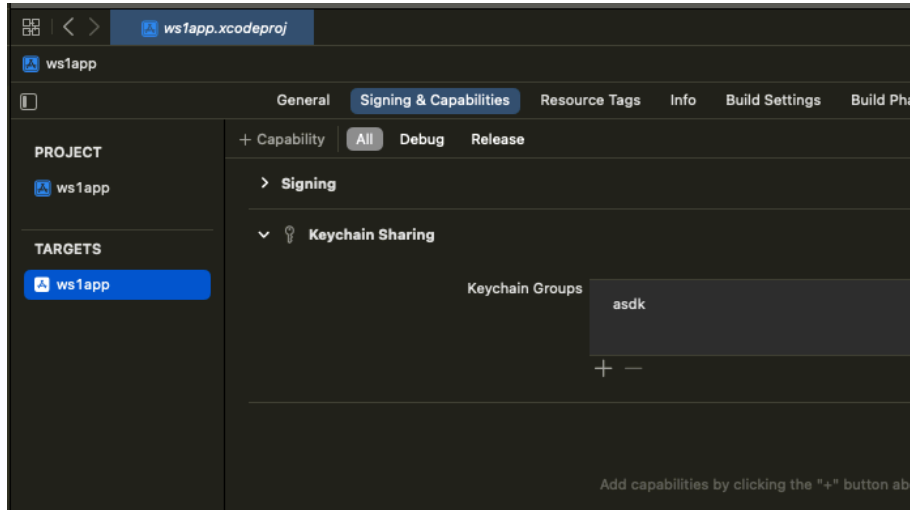
Screen Capture: Xcode Keychain Group Added

6. Change the name of the keychain group to **awsdk**.

You can double click on the name of the new group to edit it.

That completes the addition of a shared iOS keychain group.

This screen capture shows it might appear in the Xcode user interface.



Screen Capture: Xcode Keychain Groups

Build and run the application to confirm that no mistakes have been made.

Warnings:

- Never delete the awsdk group.
- Don't use the awsdk group in your apps. The group must be left for exclusive use by the SDK.
- If you add another keychain group for your apps to use, the awsdk group must always be first in the list.
- Keychain group access can only be shared between apps signed by the same developer team.

You are now ready to proceed to the next [Task: Add the software development kit package](#).

Task: Add the software development kit package

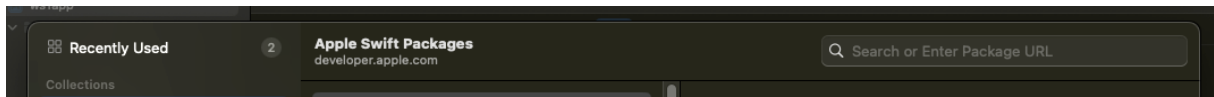
Adding the SDK package is a Workspace ONE platform integration task for mobile application developers. This task can be done after the [Task: Configure application properties](#). These instructions assume that task is complete already.

If you haven't installed your application via Workspace ONE at least once, then do so now. If you don't, the application under development won't work when installed via Xcode. Instructions for installing via Workspace ONE can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.

Proceed as follows to add the SDK package.

1. **Open the application project in Xcode.**
2. **In the Xcode menu select File, Add Packages...**

This opens a first add package dialog. The dialog has a search control at the top. This screen capture shows the location of the search control.



Screen Capture: Xcode Swift Package Search

The prompt is Search or Enter Package URL. (URL is an abbreviation for Uniform Resource Locator.)

3. **Select the search control and paste in the URL for the Workspace ONE SDK Swift Package.**

<https://github.com/euc-releases/iOS-WorkspaceONE-SDK>

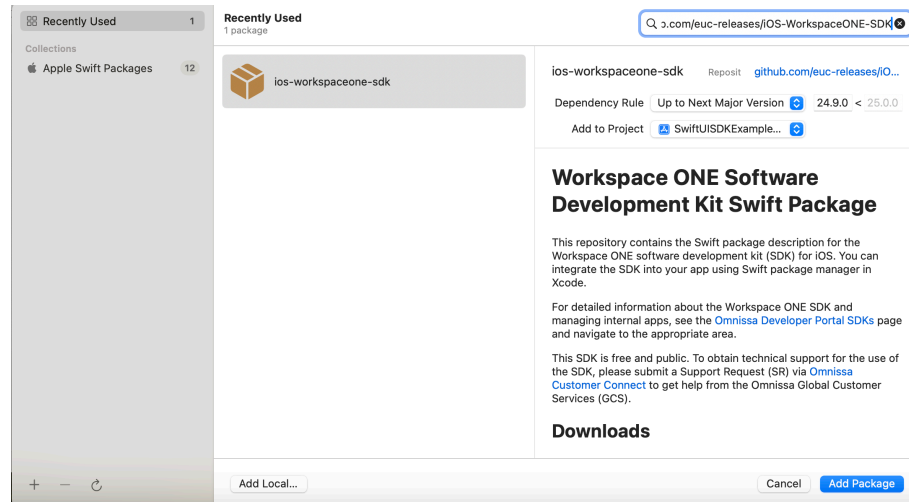
The add package dialog changes to show controls for specifying the package version.

4. **Select Dependency Rule: Exact Version and 23.3.0 as the parameter, or other values if preferred.**

You can check the latest version by opening the package URL and then opening the list of tags. The package uses Git tags for SDK versions.

5. Select **Add to Project: ws1app** or your app name if it isn't selected by default, and then select **Add Package**.

This screen capture shows the location of the selections in the Xcode user interface.

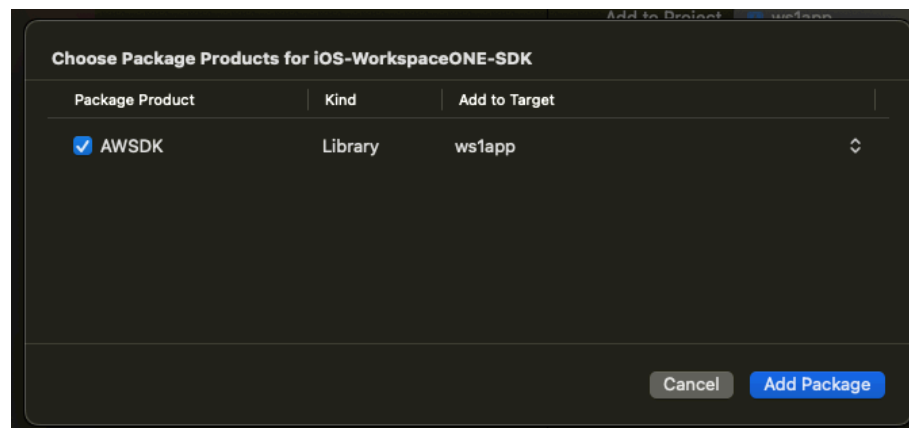


Screen Capture: Xcode Swift Package Add

Xcode will fetch and verify the contents of the package. When it finishes, a package product selection dialog is displayed.

6. In the dialog select to add the library **AWSDK** to the target for your app.

Those selections could be the default and are shown in this screen capture.

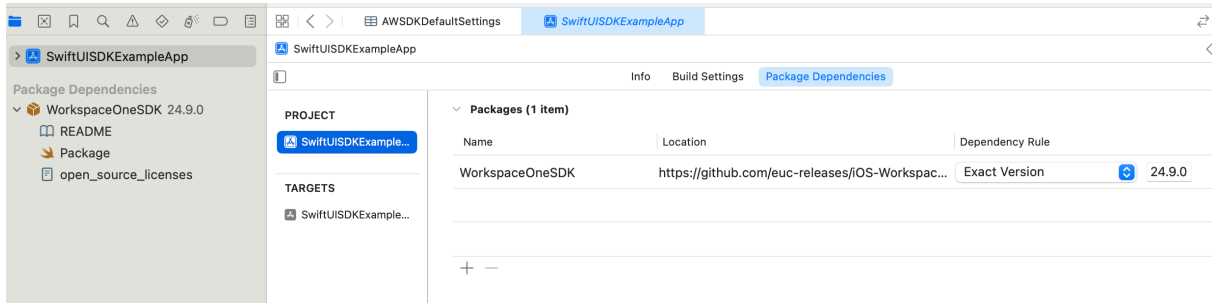


Screen Capture: Xcode Choose Package Products

When the selections are as required, click the **Add Package** button.

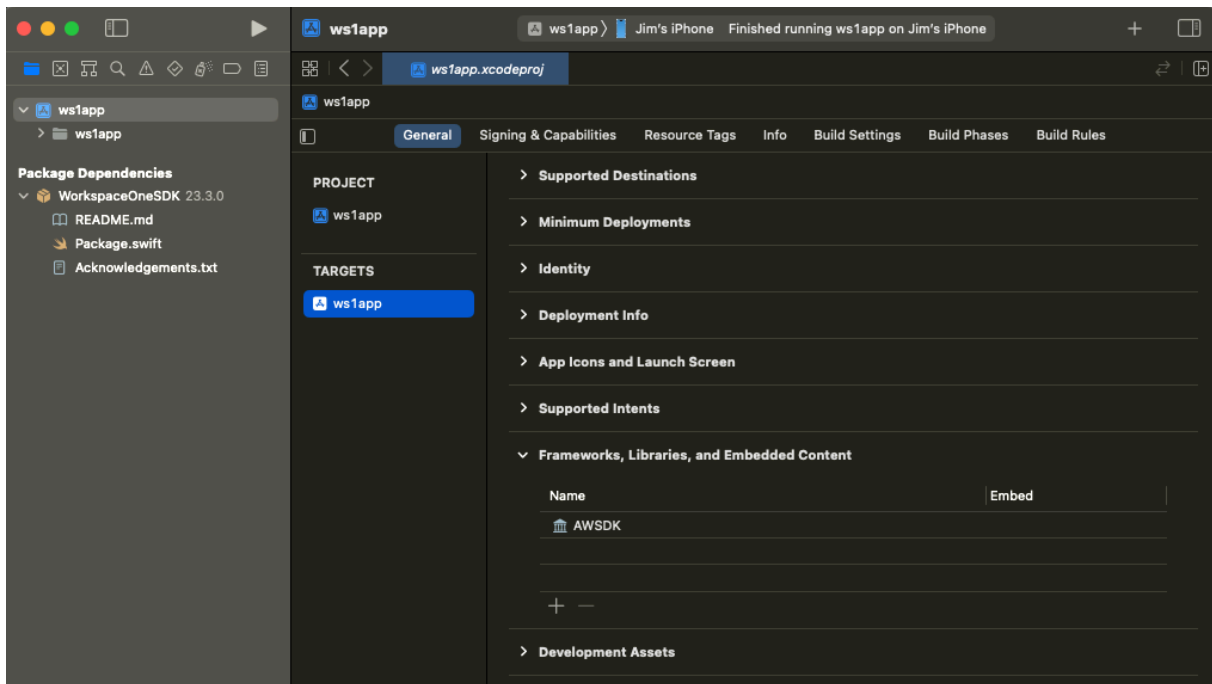
7. Check that the package has been added to the app project and to the target.

In the Xcode project navigator select the project itself and then the tab Package Dependencies. Check that the Workspace ONE SDK appears. This screen capture shows how this appears in the Xcode user interface.



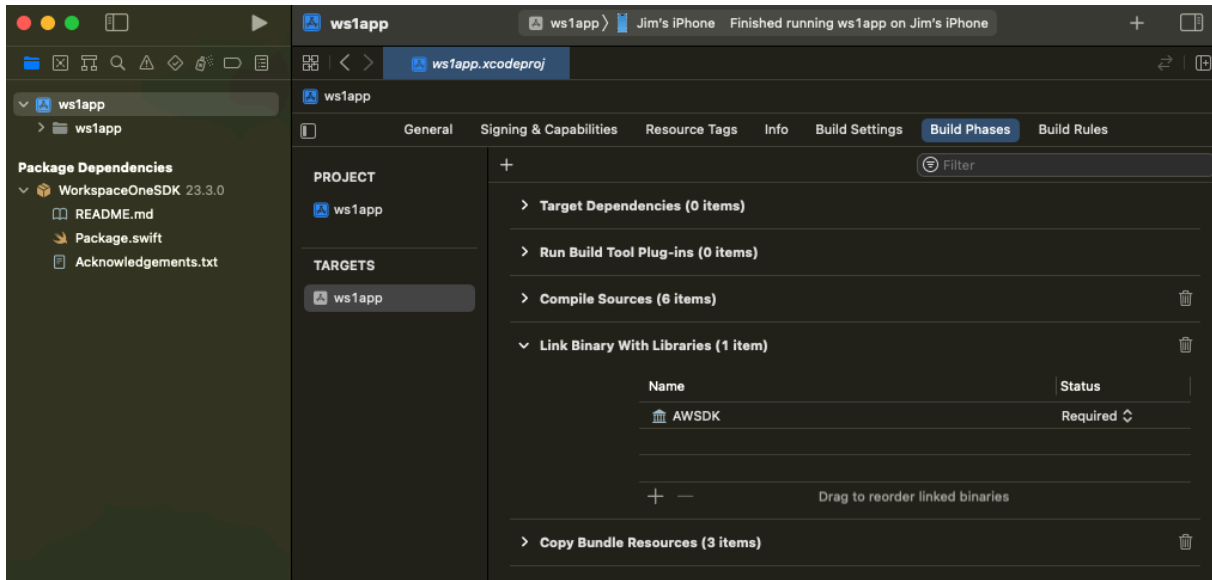
Screen Capture: Xcode Package Dependencies

Then, in the Xcode project navigator select the app target that you're using for integration, and then the tab General and expand the Frameworks, Libraries, and Embedded Content list. Check that AWSDK appears. This screen capture shows how this appears in the Xcode user interface.



Screen Capture: Xcode Target Link Frameworks, Libraries, and Embedded Content

Still in the app target, select the tab Build Phases and expand the Link Binary With Libraries list. Check that AWSDK appears. This screen capture shows how this appears in the Xcode user interface.



Screen Capture: Xcode Target Link Build Libraries

This completes adding the software development kit to the application. Build the application to confirm that no mistakes have been made, and to force Xcode to recognize the package's module for import. After that, continue with the next [Task: Initialize the software development kit runtime](#).

Task: Initialize the software development kit runtime

Initializing the SDK runtime is a Workspace ONE platform integration task for mobile application developers. This task is dependent on the [Task: Add the software development kit package](#). The following instructions assume that the dependent task is complete already.

Before you begin, you will need to know the Team ID of your Apple developer account. Instructions for locating the required value can be found on the Apple developer website, for example here.

developer.apple.com/help/account/manage-your-team/locate-your-team-id

Note that your app must have the same bundle identifier as it did when the Integration Preparation Guide was being followed, as discussed in the [Welcome](#) section. This could mean that you must continue to use the same Team ID also.

How you initialize the SDK runtime depends on how the app user interface is implemented.

- If the app user interface is implemented in SwiftUI, follow the [Initialize from SwiftUI](#) instructions.
- If the app user interface is implemented in a storyboard, follow the [Initialize from Storyboard](#) instructions.

Initialize from SwiftUI

If the app user interface is implemented in SwiftUI, you can follow these instructions to start the [Task: Initialize the software development kit runtime](#). The instructions assume that all dependencies of that task are completed already.

These instructions are intended to meet general requirements for Workspace ONE integration. They should be easy to adapt to your own specific requirements if needed.

Proceed as follows.

Add a helper class

Start by adding a helper class that can interface between SwiftUI and the Workspace ONE SDK.

1. Open the application project in Xcode and add a new Swift class.

For example by right-clicking on the project and selecting New File... in the context menu that appears. Then selecting Swift File in the template chooser.

2. Add the helper class.

This can be a singleton class. It should conform to these interfaces.

- NSObject as usual.
- AWControllerDelegate for the SDK side of the interface.
- ObservableObject for the SwiftUI side of the interface.

The top of the class file could look like this.

```
import AWSDK

class WorkspaceONEHelper:
    NSObject, AWControllerDelegate, ObservableObject
{
    // Singleton class.
    private override init() { super.init() }
    static var shared: WorkspaceONEHelper = .init()

    // Rest of the class will go here.
}
```

3. Implement a helper callback that is invoked when the SDK finishes initialization.

The callback is the `controllerDidFinishInitialCheck(error:)` function and is required by the declared SDK protocol.

You may add `@Published` variables to the helper so that the initialized status can be made visible in the app user interface.

The code could look like this.

```
@Published var sdkMessage: String = "SDK uninitialised."
func controllerDidFinishInitialCheck(error: NSError?) {
    // Good spot for a debug checkpoint.
    if let nsError = error {
        sdkMessage = "controllerDidFinishInitialCheck failed \(nsError)."
    }
    else {
        sdkMessage = "controllerDidFinishInitialCheck OK."
    }
    print(sdkMessage)
}
```

4. Add a helper function to start the SDK, and an instance variable to enforce that the SDK is only started once.

To start the SDK, do the following.

1. Access the **AWController** singleton through the **clientInstance()** class function.
2. Set the **callbackScheme** to the URL Type added in the [Declare a custom URL scheme](#) instructions.

You can do this by adding code to read the app properties. For an example see the [Appendix: Callback Scheme Sample Code](#).

3. Set the **teamID** to the team identifier used to build the app.

Instructions for locating the required value can be found on the Apple developer website, for example here.

developer.apple.com/help/account/manage-your-team/locate-your-team-id

4. Set the controller **delegate** to the helper instance.
5. Initialize the SDK by calling the controller **start()** function.

The code could look like this.

```
@Published var appMessage: String = "App uninitialised."
var sdkStarted = false
func startSDK()
{
    appMessage = "startSDK() \(sdkStarted)."
    print(appMessage)

    guard !sdkStarted else { return }
    sdkStarted = true

    // Access the AWController singleton.
    let awController = AWController.clientInstance()

    // Set the callback scheme to the custom URL Type.
    do {
        awController.callbackScheme = try schemeForWS1()
    }
    catch {
        appMessage = "schemeForWS1() failed \(error)."
        print(appMessage)
        return
    }

    // Set the team identifier.
    awController.teamID = "YOUR1TEAM2IDENTIFIER2HERE"

    // Set the controller delegate to the helper instance.
    awController.delegate = self

    // Actually initialize the SDK.
    awController.start()
}
```

5. Add a helper function to pass open URL requests to the SDK handler.

The SDK will process open URL requests from Hub during enrollment. This is discussed in the introduction to the [Task: Configure application properties](#).

The code could look like this.

```
func handleOpen(url: URL, fromApplication: String?) -> Bool {
    return AWController.clientInstance()
        .handleOpenURL(url, fromApplication: fromApplication)
}
```

Build the application to ensure that no mistakes have been made. Then continue with the next instructions.

Connect the helper

Continue by connecting the helper class to the app code and user interface.

These instructions assume your application code has a structure like this.

- *Content view* struct that conforms to the SwiftUI **View** protocol.
- *App main* struct that conforms to the SwiftUI **App** protocol and declares the Content view in a WindowGroup.

(If your structure is different you can adapt these instructions.)

Proceed as follows.

1. Connect the helper to your Content view.

Add an **@ObservedObject** variable to the Content view. The variable will be an instance of the helper. The code could look like this.

```
@ObservedObject var sdkHelper: WorkspaceONEHelper
```

2. Add, for example, Text instances to the Content view to show the published variables from the helper.

The code could look like this.

```
var body: some View {
    VStack {
        Text(self.sdkHelper.sdkMessage).font(.title2)
        Text(self.sdkHelper.appMessage).font(.title2)
    }
    .padding()
}
```

At this point the app won't build because declaring the observed object requires a parameter in the Content view instantiation.

3. Update the Content view instantiation in your App main.

Pass the singleton instance of the helper to the Content view. The code could look like this.

```
ContentView(sdkHelper: WorkspaceONEHelper.shared)
```

4. In your App main, connect the helper open URL handler to the Content view.

Add an `onOpenURL` to the Content view instantiation. The `onOpenURL` should first call the helper function, which returns a Boolean value for whether it handled the URL. The code could look like this.

```
ContentView(sdkHelper: WorkspaceONEHelper.shared)
    .onOpenURL { url in
        if WorkspaceONEHelper.shared.handleOpenURL(
            url: url, sourceApplication: nil
        ) { return }

        print("URL not handled by WS1 \(url)")
        // App handling of url goes here.
    }
```

5. Initialize the SDK from your App main.

In your App main, add a *scene phase* listener to the WindowGroup. In the listener, if the scene phase is active then call the singleton helper function to initialize the SDK. The code could look like this.

```
@Environment(\.scenePhase) private var scenePhase

var body: some Scene {
    WindowGroup {
        // Other code was added here in the preceding instructions.
    }
    .onChange(of: scenePhase) {phase in
        if phase == .active {
            WorkspaceONEHelper.shared.startSDK()
        }
    }
}
```

Note. An alternative to a scene phase listener could be to implement `@UIApplicationDelegateAdaptor` and call the SDK initialization helper from its `didFinishLaunchingWithOptions` callback. However, that type of adaptor shouldn't be used according to the reference documentation. See the Apple developer website here for example.

developer.apple.com/documentation/swiftui/uiapplicationdelegateadaptor

This completes SDK initialization from SwiftUI. Now follow the instructions to [Test runtime initialization](#).

Initialize from Storyboard

If the app user interface is implemented in a Storyboard, you can follow these instructions to start the [Task: Initialize the software development kit runtime](#). The instructions assume that all dependencies of that task are completed already.

These instructions are intended to meet general requirements for Workspace ONE integration. They should be easy to adapt to your own specific requirements if needed.

Proceed as follows.

Update the app delegate

Start by extending your app delegate class so that it will start and control the Workspace ONE SDK.

1. **Open the application project in Xcode and then the class that implements the app delegate.**

By default the app delegate class is named **AppDelegate** and will be in the **AppDelegate.swift** file.

The required class will implement the **UIApplicationDelegate** protocol and might be annotated as the main entry point of the app.

2. **Declare the app delegate as implementing the Workspace ONE SDK controller delegate protocol.**

To do that

- add an import statement for **AWSDK**
- add the **AWControllerDelegate** protocol to the app delegate's declaration.

The top of the class file could look like this.

```
import UIKit
import AWSDK

@main
class AppDelegate: UIResponder, UIApplicationDelegate, AWControllerDelegate {
    // Rest of the class here.
}
```

3. Add code to start the SDK when the app is launched.

To start the SDK, do the following.

1. Access the `AWController` singleton through the `clientInstance()` class function.
2. Set the `callbackScheme` to the URL Type added in the [Declare a custom URL scheme](#) instructions.

You can do this by adding code to read the app properties. For an example see the [Appendix: Callback Scheme Sample Code](#).

3. Set the `teamID` to the team identifier used to build the app.

Instructions for locating the required value can be found on the Apple developer website, for example here.

developer.apple.com/help/account/manage-your-team/locate-your-team-id

4. Set the controller `delegate` to the app delegate instance.

5. Initialize the SDK by calling the controller `start()` function.

In case you want to display a status message, you can add a suitable variable that can be observed from another class.

The code could look like this.

```

@objc dynamic var appMessage = "App uninitialised."

func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [
        UIApplication.LaunchOptionsKey: Any
    ]?) -> Bool
{
    // Override point for customization after application launch.

    let awController = AWController.clientInstance()

    // Set the callback scheme to the custom URL Type.
    do {
        awController.callbackScheme = try schemeForWS1()
        appMessage = "schemeForWS1() OK \"\`(awController.callbackScheme)\`\"."
        print(appMessage)
    }
    catch {
        appMessage = "schemeForWS1() failed \"\`(error)\`\"."
        print(appMessage)
        return false
    }

    // Set the team identifier.
    awController.teamID = "YOUR1TEAM2IDENTIFIER2HERE"

    // Set the controller delegate to the helper instance.
    awController.delegate = self

    // Actually initialize the SDK.
    awController.start()

    return true
}

```

4. Make the app delegate conform to the Workspace ONE SDK controller delegate protocol.

To conform, implement the callback `controllerDidFinishInitialCheck`. Best practice would be to do this in your implementation.

- If SDK initialization failed, show an error message and block all or part of the app user interface.
- If SDK initialization succeeded, don't block the app user interface.

In case you want to display a status message, you can add a suitable variable that can be observed from another class.

The code could look like this.

```
@objc dynamic var sdkMessage = "SDK uninitialised."

func controllerDidFinishInitialCheck(error: NSError?) {
    // Good spot for a debug checkpoint.
    if let nsError = error {
        sdkMessage = "controllerDidFinishInitialCheck failed \(nsError)."
    }
    else {
        sdkMessage = "controllerDidFinishInitialCheck OK."
    }
    print(sdkMessage)
}
```

Build the application to ensure that no mistakes have been made. Then continue with the next instructions.

Update the scene delegate

The SDK will process open URL requests from Hub during enrollment. This is discussed in the introduction to the [Task: Configure application properties](#).

URL requests will be received in the scene delegate, which must be modified to convey the required URL requests to the SDK.

Proceed as follows.

1. **Open the application project in Xcode and then the class that implements the scene delegate.**

By default the scene delegate class is named `SceneDelegate` and will be in the `SceneDelegate.swift` file.

The required class will implement the `UIWindowSceneDelegate` protocol.

2. Add an open URL handler that conveys requests to the SDK handler.

To do this, implement the `scene(_:openURLContexts:)` callback. The reference documentation for that callback can be found on the Apple developer website, for example [here](https://developer.apple.com/.../uiscedelegat/).

[developer.apple.com/.../uiscedelegat/...](https://developer.apple.com/.../uiscedelegat/)

To convey open URL requests, do the following.

1. Add an import statement for **AWSDK**.
2. Access the AWController singleton through the `clientInstance()` class function.
3. Invoke the `handleOpenURL()` method for each URL context received.
4. If the method returns true, the SDK has handled the open URL request.
5. Otherwise, the app should handle the request.

The code could look like this.

```
// Other imports here, such as UIKit
import AWSDK

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(
        _ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>
    ) {
        for context in URLContexts {
            if AWController.clientInstance().handleOpenURL(
                context.url, fromApplication: context.options.sourceApplication
            ) { continue }

            print("URL not handled by WS1 \(context.url)")
            // App handling of url goes here.
        }
    }
}
```

This completes SDK initialization from Storyboard. Now follow the instructions to [Test runtime initialization](#).

Test runtime initialization

Test initialization of the SDK runtime after completing either the [Initialize from SwiftUI](#) instructions or the [Initialize from Storyboard](#) instructions.

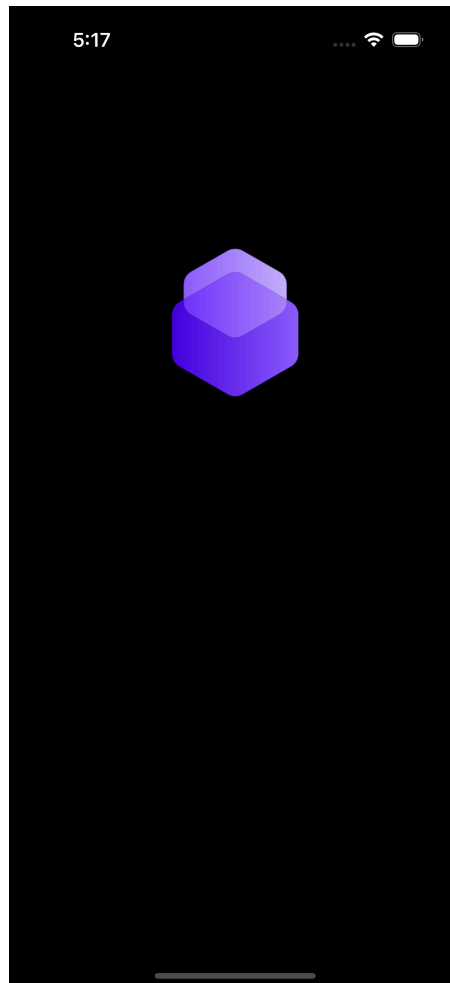
Build and run the application.

First Launch App Enrollment

The first time the app is launched it will have to enroll with the UEM. The SDK runtime will first retrieve some enrollment details from Hub. That retrieval will require the device user interface to flip to Hub and then back to your app. Then the end user must authenticate to UEM in order to complete app enrollment.

These screens and user interfaces should appear in order.

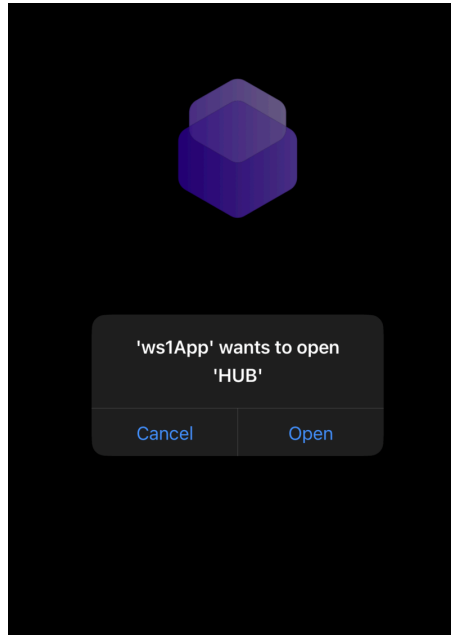
1. SDK splash screen.



Screen Capture: SDK Splash Screen for iOS

The splash screen might appear only briefly.

2. Open Hub confirmation.



Screen Capture: Open Hub confirmation

This is an operating system prompt. Tap Open to continue enrollment by opening Hub.

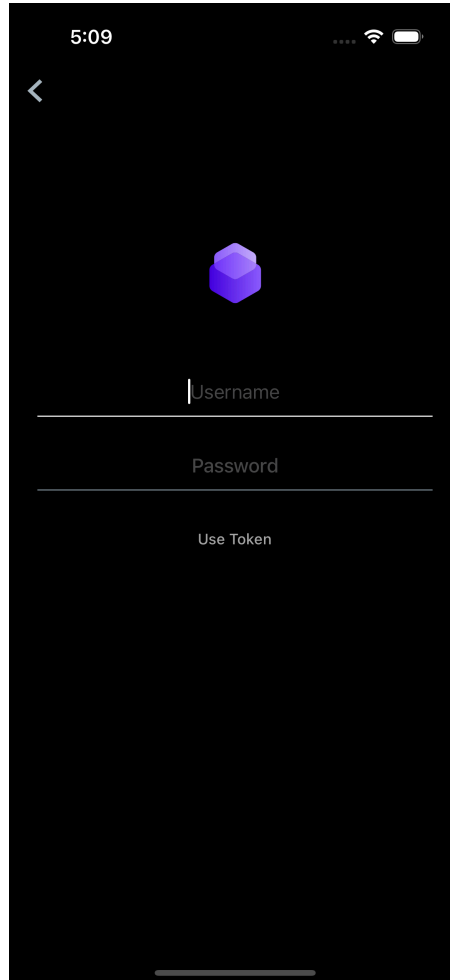
3. The device user interface flips to the Hub app.

Hub might require authentication of the end user, for example by entering an app passcode.

After authentication, if any, the device user interface flips back to your app.

4. Authenticate to enroll.

The appearance of this screen depends on UEM configuration. If basic authentication is configured then the screen could look like this.



Screen Capture: Authenticate to enroll

If Security Assertion Markup Language (SAML) is configured then a web view will open instead. The web view will display an enterprise authentication page.

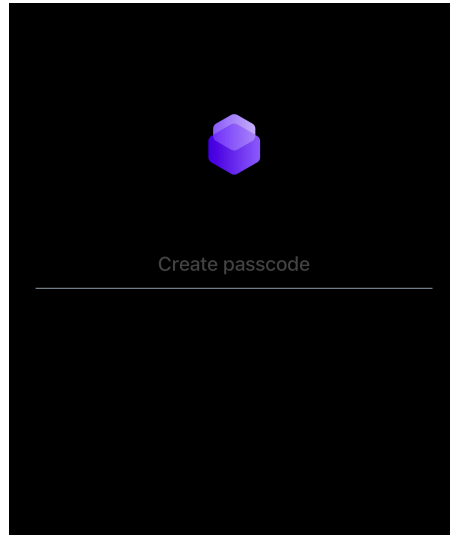
The operating system adds an option to go back to Hub at the top left of the screen. Ignore that option.

Enter the credentials of the user account, which will be the same as used to enroll the Hub app during Preparation.

5. Create app passcode, if required.

Depending on UEM security policy configuration, you may be required to set and confirm an app passcode. This is a new separate passcode to the password that was used to enroll the app and Hub. The app passcode is only used on the device. The device doesn't require an internet connection to check the app passcode.

The first screen of the interaction looks like this.



Screen Capture: Create app passcode

Set and confirm an app passcode to continue.

6. App user interface.

When app enrollment has completed, the app user interface will open. The `controllerDidFinishInitialCheck` callback that you implemented in the previous instructions will be invoked.

The first app launch, and app enrollment, are now complete. If the steps didn't play out as above, or if an error was passed to the initial check callback, check the tips in the [Appendix: Troubleshooting](#) and review the earlier instructions.

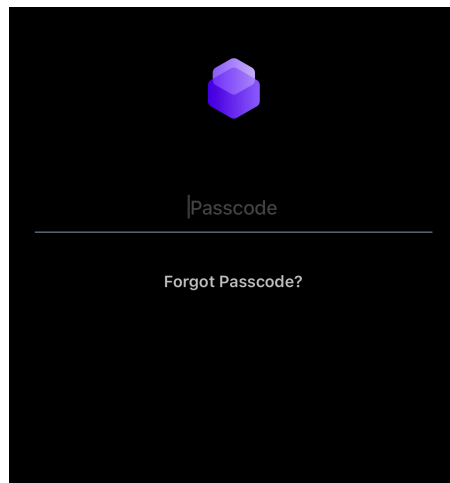
Otherwise, terminate the app and launch it again.

Subsequent Launch

The second and subsequent times the app is launched these screens and user interfaces should appear.

1. **SDK splash screen as in the first launch, perhaps only briefly.**
2. **Enter app passcode, if required.**

If the UEM security policy configuration required that an app passcode was created, then you may be prompted to enter the passcode when the app launches. The screen could look like this.



Screen Capture: Enter app passcode

Passcode entry isn't always required. A full discussion is out of scope of this guide but note the following scenarios.

- App single sign-on (SSO) can be configured in the UEM security policies. Entering the app passcode in a one app then starts an authentication session that other apps join without requiring the app passcode to be entered.
- If the app was recently active and hasn't been terminated then the app passcode won't be required when the app returns to foreground.

3. **App user interface.**

When authentication, if any, has completed, the app user interface will open. The `controllerDidFinishInitialCheck` callback that you implemented in the previous instructions will be invoked.

When you have verified first launch and subsequent launch, you are ready to continue with the next [Task: Declare supported features](#).

Task: Declare supported features

Declaring supported features is a Workspace ONE platform integration task for mobile application developers. This task is dependent on the [Task: Initialize the software development kit runtime](#). The following instructions assume that the dependent task is complete already.

Declare the Workspace ONE features that your app supports by creating a resource bundle and property list. Proceed as follows.

1. **Open your app project in Xcode.**

2. Create a Settings Bundle named **AWSDKDefaults.bundle**.

(In this guide, the bundle will be created in the application resources. The bundle can instead be in a sub-group or in a package that has been added to the application.)

One way to create the bundle is as follows.

1. In the Xcode navigator select the project itself.

By default, the project itself will be the first item in the navigator.

2. In the Xcode menu, select **File, New, File...**

This opens a template selection interaction.

3. Select the template: **Settings Bundle**.

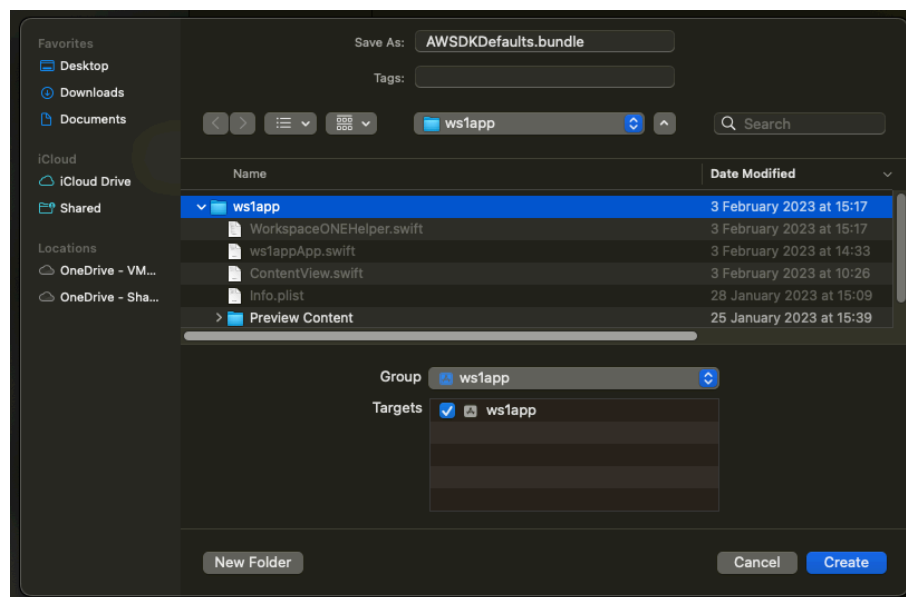
The template can appear in the Resource category. You can also search for it by filtering, for example for “bundle”.

Select the plain Settings Bundle template, not the WatchKit Settings Bundle template.

4. Click **Next**. A file save dialog will open.

5. Enter the name **AWSDKDefaults.bundle** and select to save in the top group of the project.

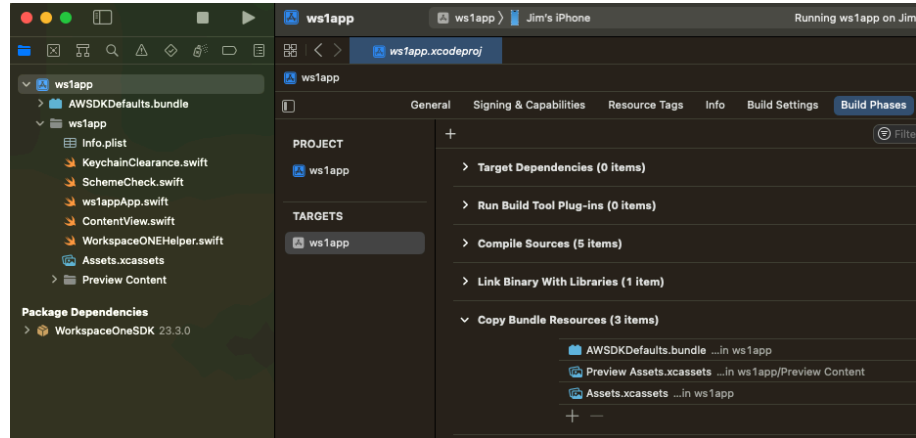
This screen capture shows how the save dialog might look.



Screen Capture: Xcode create bundle

The new bundle will be added to the resources that are copied into the app. You can check this in the target build phases, in the Copy Bundle Resources list. If it doesn't appear there, add it now by clicking the plus.

This screen capture shows the bundle in the Xcode user interface.



Screen Capture: Xcode bundle in build phases

3. Create a property list named **AWSDKDefaultSettings.plist** in the bundle.

One way to do this is as follows.

1. In the Xcode navigator select the project itself.

By default, the project itself will be the first item in the navigator.

2. In the Xcode menu, select File, New, File...

This opens a template selection interaction.

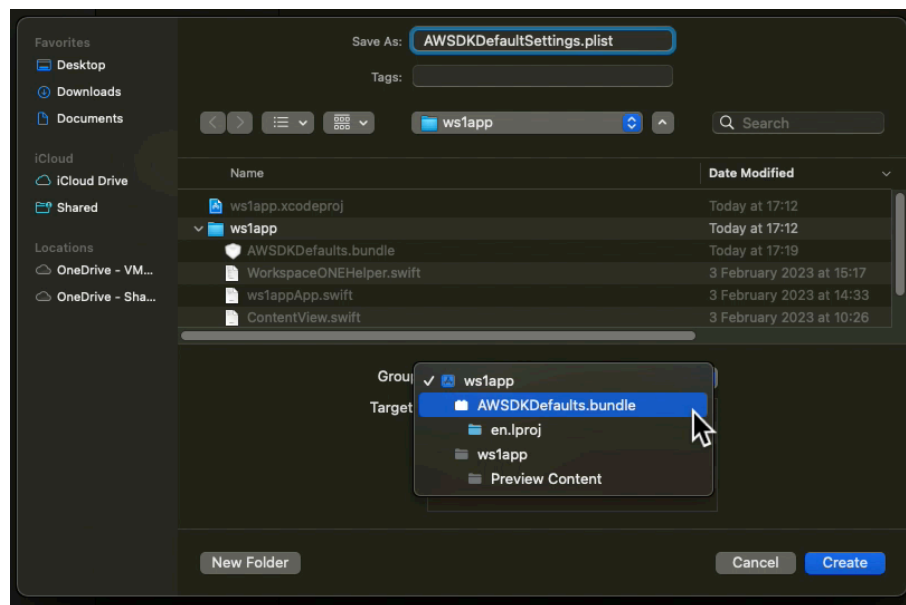
3. Select the template: Property List.

The template can appear in the Resource category. You can also search for it by filtering, for example for “property”.

4. Click Next. A file save dialog will open.

5. Enter the name **AWSDKDefaultSettings.plist** and select to save in the **AWSDKDefaults.bundle** group.

This screen capture shows how the save dialog might look.



Screen Capture: Xcode create property list

The new property list file will be added to the bundle.

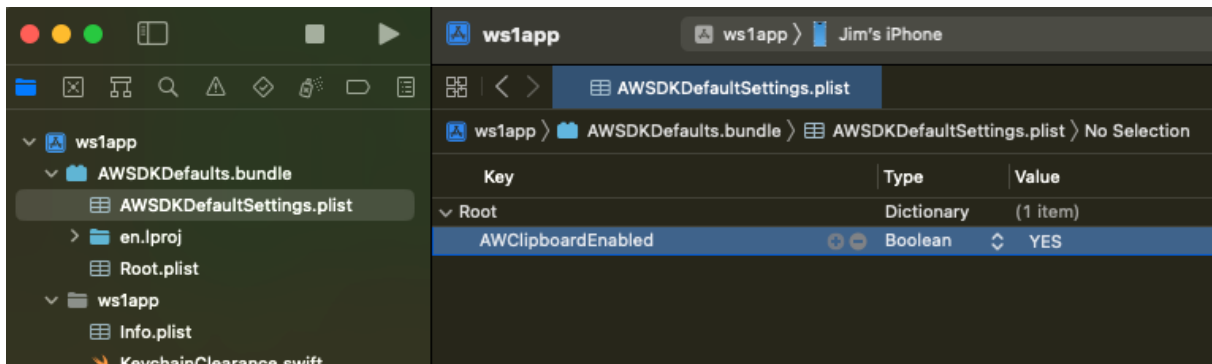
If you save the property list somewhere else by mistake, you can drag and drop it into the bundle to fix it.

4. **Add a first key to the property list, to declare support for the Workspace ONE secure clipboard.**

1. Open the AWSDKDefaultSettings.plist file as a property list.
2. Add to the Root Dictionary a new row as follows.

Key	Type	Value
AWClipboardEnabled	Boolean	YES

This screen capture shows how the property list might look after adding the first key.



Screen Capture: Xcode declare secure pasteboard

That completes the initial declaration of supported features. Build and run the application to confirm that no mistakes have been made. You are now ready to continue with the [Task: Demonstrate Basic Features](#).

Task: Demonstrate Basic Features

The following basic features are now integrated into your application and can be demonstrated. In most cases the policies and settings in the UEM must be configured to activate features in the SDK. Brief instructions for application developers are given here for convenience. Full documentation can be found in the online help and product documentation.

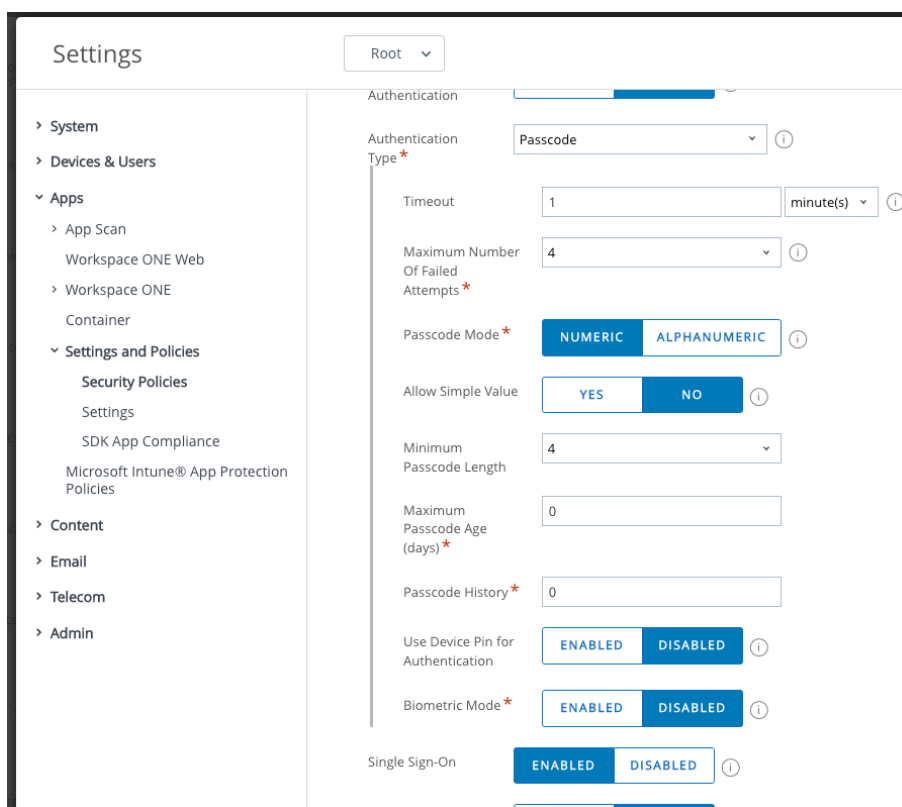
Authentication of the end user

The end user can be forced to authenticate

- when the app opens for the first time after the device is switched on.
- after an inactivity time out has expired.

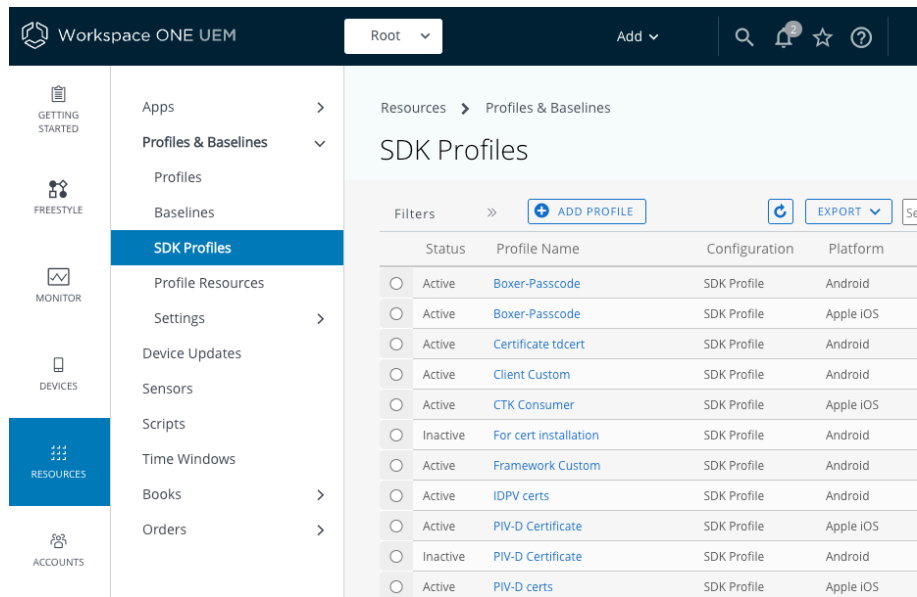
Authentication can be by entry of their domain username and password, or by a separate app passcode that they create on the device.

This feature can be configured in the Organization Group (OG). This screen capture shows the location of the setting in the console user interface.

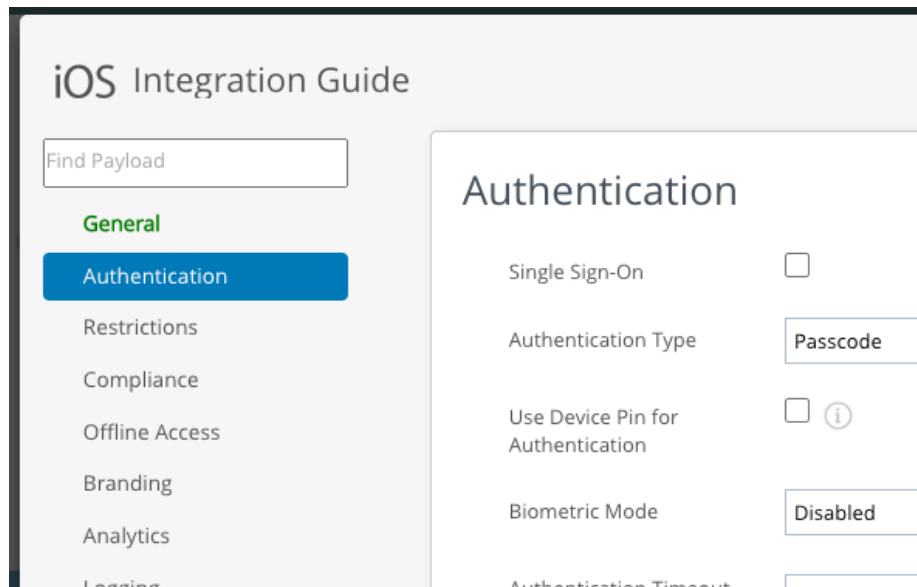


Screen capture: Organization group end user authentication settings in the console

This feature can also be configured in a custom SDK Profile. These screen captures show how to navigate to SDK Profile management in the console user interface, and the location of the setting in the profile editor.



Screen capture: SDK Profile management in the console



Screen capture: Custom profile end user authentication settings in the console

Notes on this feature.

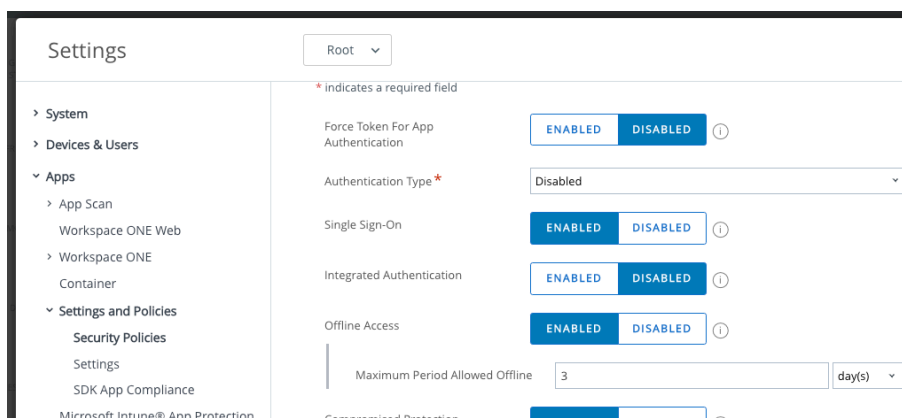
- If Authentication Type is set to Disabled then the feature is deactivated.

- There is an additional option to Use Device Pin for Authentication. If this option is selected then the end user can authenticate by entering the passcode that they use to unlock their device. See the Require Device Passcode Technical Brief, here developer.omnissa.com/docs/17711/RequireDevicePasscode.pdf for details.
- There is an additional option for Biometric Mode authentication. If this option is selected then the end user can authenticate by Touch ID or Face ID.
- There is an additional option for Single Sign-On. If this option is selected then the same passcode will be used by all apps that have access to a shared keychain group. See also the [Add a shared keychain group](#) instructions.

Access to the app offline

The end user can be blocked from access to the app offline. If offline access is blocked then the app user interface will be blocked by an informative error message when the device is offline. That can be tested by engaging flight mode for example.

This feature can be configured in the Organization Group (OG). This screen capture shows the location of the setting in the console user interface.



Screen capture: Organization group offline access setting in the console

This feature can also be configured in a custom SDK Profile. This screen capture shows the location of the setting in the profile editor.

The screenshot shows the 'iOS Integration Guide' interface. On the left is a sidebar with a search bar labeled 'Find Payload' and a list of menu items: 'General' (highlighted in green), 'Authentication', 'Restrictions', 'Compliance', 'Offline Access' (highlighted in blue), and 'Branding'. The main content area is titled 'Offline Access' and contains two settings: 'Offline Access' with a checked checkbox, and 'Maximum period of time allowed to be offline' with a numeric input field set to '0' and a dropdown menu set to 'day(s)'.

iOS Integration Guide

Find Payload

General

Authentication

Restrictions

Compliance

Offline Access

Branding

Offline Access

Offline Access ☒

Maximum period of time allowed to be offline day(s) ▼

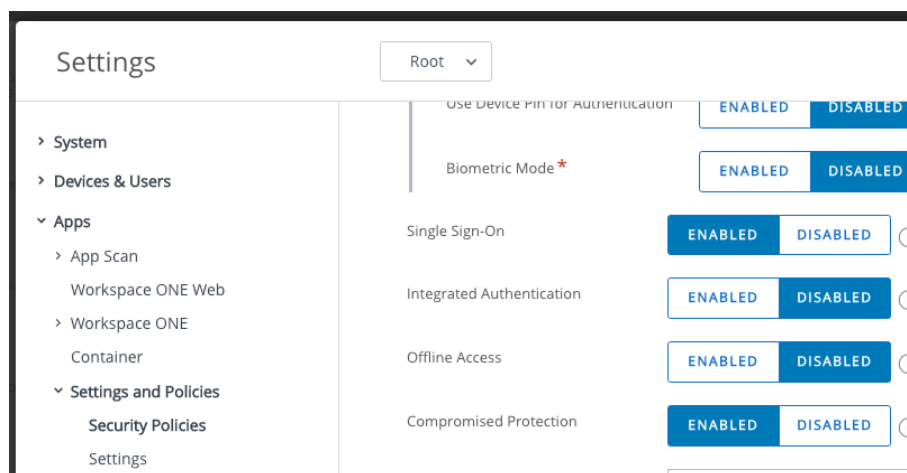
Screen capture: Custom profile offline access setting in the console

Device compromise protection

Device compromise is the deactivation of the built-in security features of a mobile device operating system. It is commonly referred to as *jailbreaking*, if applied to iOS and iPadOS devices. Device compromise increases the vulnerability of enterprise data on the device to unauthorized access, either by accidental leakage or by deliberate attack.

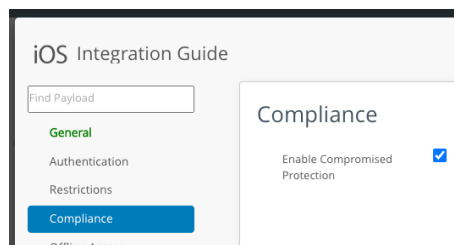
The SDK can protect enterprise data by wiping its credentials and encryption keys when device compromise is detected. The SDK will also report device compromise to the UEM.

This feature can be configured in the Organization Group (OG). This screen capture shows the location of the setting, Compromised Protection, in the console user interface.



Screen capture: Organization group device compromise protection setting in the console

This feature can also be configured in a custom SDK Profile. This screen capture shows the location of the setting in the profile editor.



Screen capture: Custom profile device compromise protection setting in the console

To demonstrate this feature on iOS or iPadOS you would have to jailbreak the device. Instructions aren't provided.

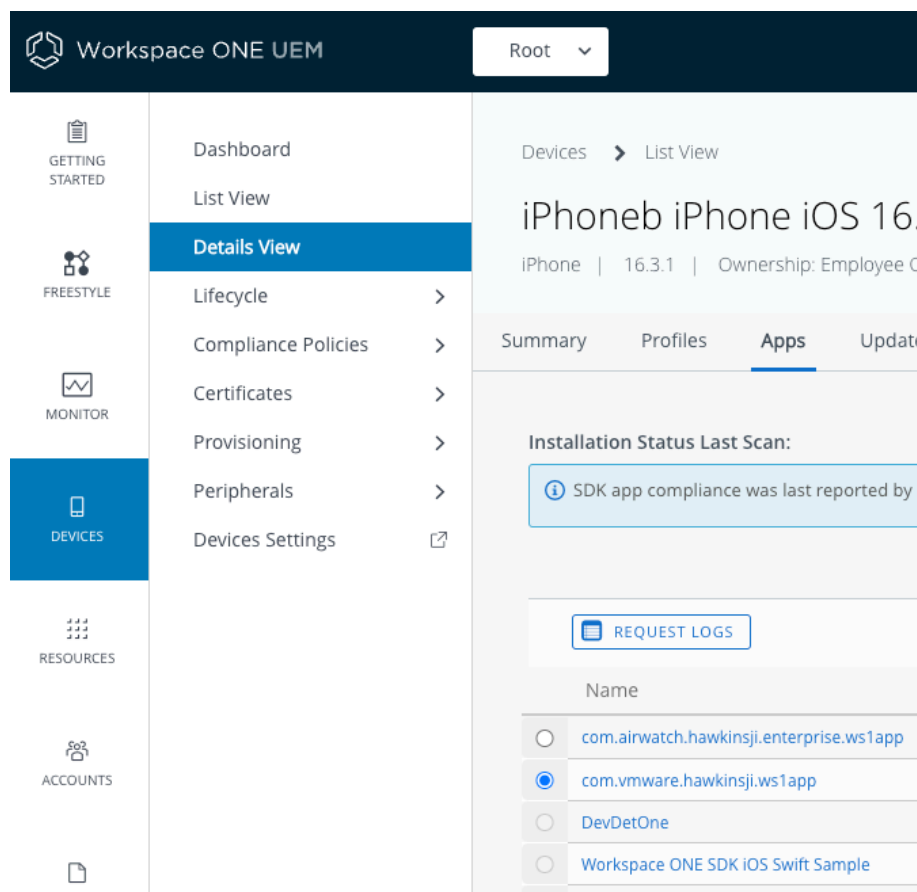
Logging

Log files written by the SDK can be uploaded to the UEM console to aid in diagnosis of problems.

To request logs proceed as follows.

1. Log in to the UEM and select the Organization Group (OG) of the end user that you are using for development.
2. Navigate to Devices, List View.
3. Locate your developer device in the list and click to open the Details View.
4. In the details view, select the Apps tab.
5. Locate your app in the list of apps that appears and select its row.

This screen capture shows how the screen looks after selection of the app.



Screen capture: App on device selected in the console

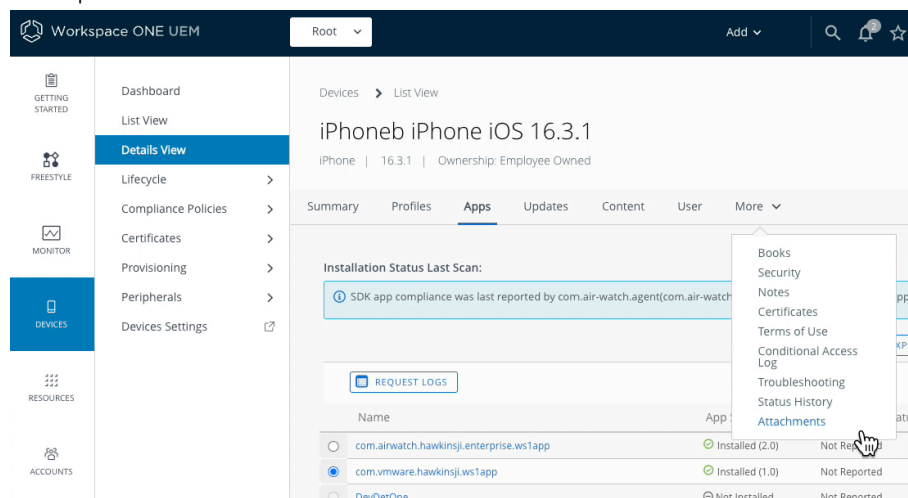
6. Click the Request Logs button that appears.

On your device, open the app. If the app was already running then terminate it using the device task manager first. That ensures the app will contact the UEM and receive the log request now.

To check the upload proceed as follows.

1. **Navigate to the Details View of the device, same as in the above instructions.**
2. **Open the Attachments tab, which could be behind a More drop-down.**

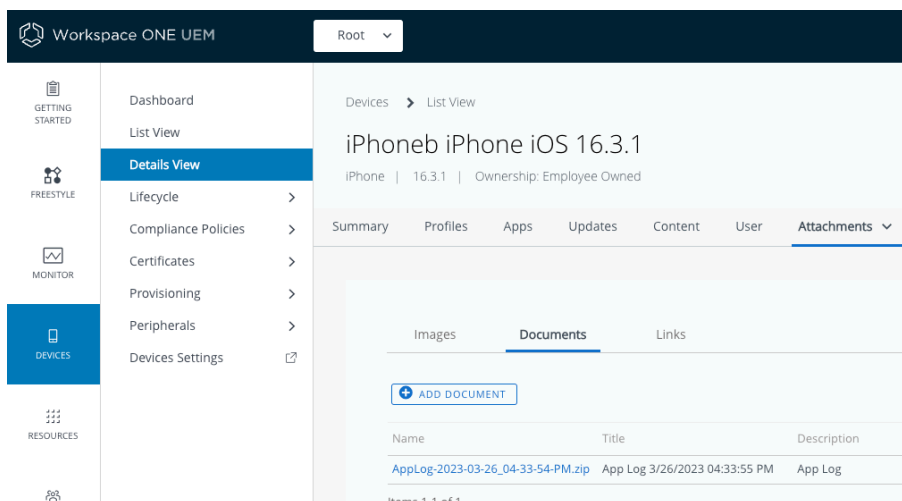
This screen capture shows the location in the console user interface.



Screen capture: Selecting device attachments

3. **On the Attachments tab, select the Documents sub-tab.**

The files appear with description App Log. This screen capture shows how the screen appears.



Screen capture: Device attachments documents tab in the console

Download a log file zip archive to view the contents.

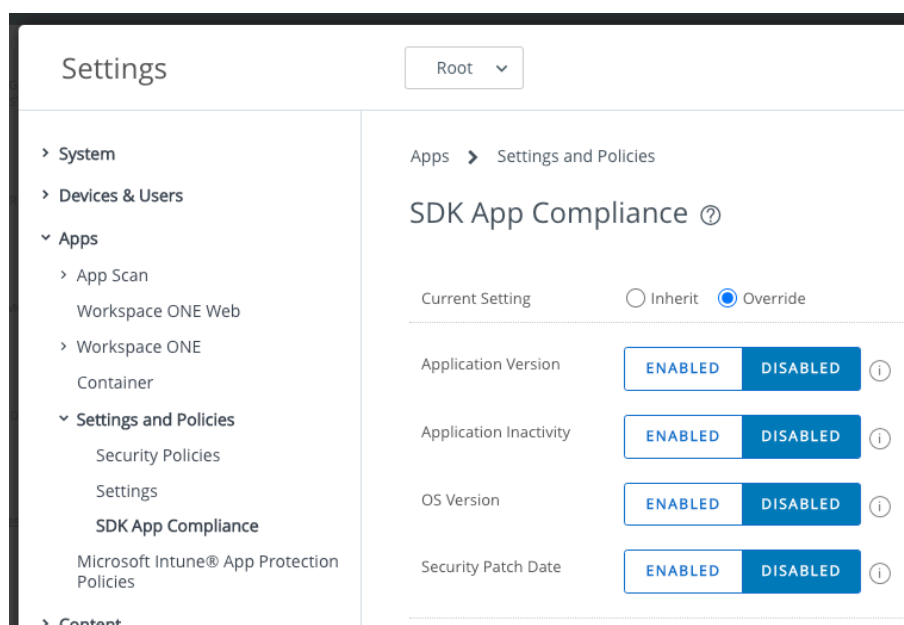
On-device app compliance enforcement

The SDK has an app compliance feature that can enforce conditions like the following.

- Minimum application version.
- Application cannot be inactive beyond a number of days.
- Minimum operating system version.

This is a separate feature to the UEM Compliance Policies engine, which is much broader in scope.

The SDK App Compliance feature can be configured in the Organization Group (OG). This screen capture shows the location of the setting in the console user interface.



Screen capture: Organization group SDK app compliance in the console

(Ignore the Security Patch Date, which only applies to Android.)

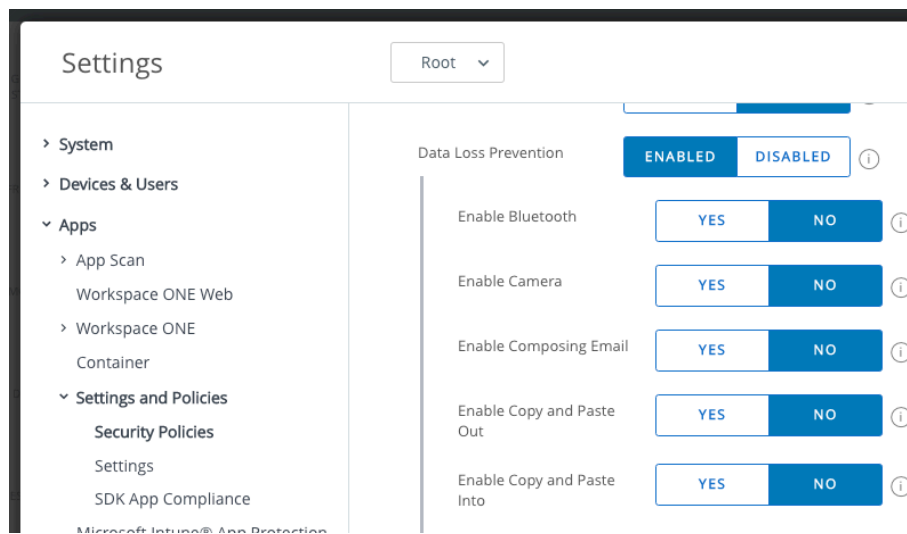
SDK App Compliance will be enforced on managed and unmanaged devices, and doesn't require the device to be online at the point of enforcement.

Protected pasteboard

The protected pasteboard is a Workspace ONE data loss prevention feature implemented by the SDK. The feature has the following parts.

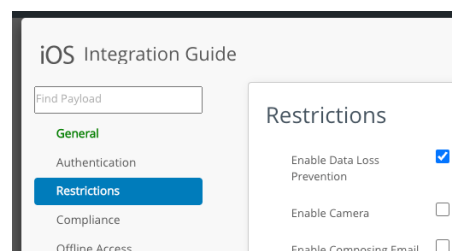
- Outbound protection, in which data copied from an SDK app cannot be pasted into a non-SDK app.
- Inbound protection, in which data copied from a non-SDK app cannot be pasted into an SDK app.

The feature can be configured in the Organization Group (OG). This screen capture shows the location of the setting in the console user interface.



Screen capture: Organization group data loss prevention settings in the console

This feature can also be configured in a custom SDK Profile. This screen capture shows the location of the setting in the profile editor.



Screen capture: Custom profile data loss prevention settings in the console

Support for the protected pasteboard requires build-time configuration in the mobile app. For instructions see the [Task: Declare supported features](#).

Further integrations

Those are some basic feature that can be demonstrated already. For further integrations, see the [Next Steps](#).

Next Steps

When you have completed all the above tasks, base integration is complete. You can now integrate further Workspace ONE features, such as these.

- App data at rest encryption.
- Data loss prevention measures beyond the pasteboard.
 - Screen capture detection.
 - Third party keyboard blocking.
 - Camera access blocking.
 - User interface watermarking.
 - Printing blocking.
 - Data backup blocking.
 - Writing tools blocking.
- Networking.
- Branding.

Some of those features will require only build-time configuration; others will require code changes.

See the home page of the Workspace ONE SDK for iOS and iPadOS for links to other developer guides and resources.

<https://developer.omnissa.com/ws1-uem-sdk-for-ios/>

Appendix: Callback Scheme Sample Code

This code can be used to read the declared URL Type list to discover or check the callback scheme for Workspace ONE integration. For sample usage of this code, see either of these sets of instructions.

- [Initialize from SwiftUI.](#)
- [Initialize from Storyboard.](#)

The sample code is also published here.

github.com/euc-releases/.../SchemeCheck.swift

```
// Copyright 2023 Omnisia, Inc.
// SPDX-License-Identifier: BSD-2-Clause

import Foundation

private let CFBundleURLTypes = "CFBundleURLTypes"
private let CFBundleURLSchemes = "CFBundleURLSchemes"
private let path = "://dummpath"

/// Get the scheme of the first declared URL Type.
///
/// This function returns the declared scheme on success, or throws an
/// error for any failure.
///
/// - Throws: If the URL Type list is empty, or any properties are missing,
/// or the value in the declaration can't be a URL scheme.
/// - Returns: String containing the scheme of the first URL Type.
func schemeForWS1() throws -> String {
    return try
        Bundle.main.urlTypes().first!.schemes().first!.validateAsURLScheme()
}

/// Checks if a specified scheme is in any declared URL Type.
///
/// This function returns the specified scheme on success, or throws an
/// error for any failure.
///
/// - Parameter expectedScheme: String containing the scheme.
/// - Throws: If the URL Type list is empty, or any properties are missing,
/// or if the specified value can't be found, or if the specified value
/// can't be a URL scheme.
/// - Returns: The expectedScheme parameter.
func schemeForWS1(_ expectedScheme:String) throws -> String {
    var discoveredSchemes:[String] = []
    for urlType in try Bundle.main.urlTypes() {
        for scheme in try urlType.schemes() {
            if scheme == expectedScheme {
                return try expectedScheme.validateAsURLScheme()
            }
            discoveredSchemes.append(scheme)
        }
    }

    throw NSError(domain: "", code: 9, userInfo: [
        NSLocalizedDescriptionKey:"Expected scheme \"\(expectedScheme)\"
        + " not found in main bundle URL Types."
        + " Schemes found \(discoveredSchemes.count): "
        + discoveredSchemes.joined(separator: " ")
        + "."
    ])
}
```

```

    })
}

private extension Bundle {
    func urlTypes() throws -> [[String:Any]] {
        guard let infoDictionary = self.infoDictionary else {
            throw NSError(domain: "", code: 1, userInfo: [
                NSLocalizedDescriptionKey:"Main bundle has no infoDictionary"
            ])
        }
        guard let typesAny = infoDictionary[CFBundleURLTypes] else {
            throw NSError(domain: "", code: 2, userInfo: [
                NSLocalizedDescriptionKey:"No \(CFBundleURLTypes) in main bundle"
                + ". Keys \(infoDictionary.keys.count): "
                + infoDictionary.keys.joined(separator: " ")
            ])
        }
        guard let typesArray = typesAny as? [[String:Any]] else {
            throw NSError(domain: "", code: 3, userInfo: [
                NSLocalizedDescriptionKey:
                    "Couldn't cast \(CFBundleURLTypes) to [[String:Any]]."
                + " " + String(describing: typesAny)
            ])
        }
        guard let _ = typesArray.first else {
            throw NSError(domain: "", code: 4, userInfo: [
                NSLocalizedDescriptionKey:
                    "\[ \(CFBundleURLTypes) ] is an empty array."
            ])
        }
        return typesArray
    }
}

private extension Dictionary where Key == String {
    func schemes() throws -> [String] {
        guard let schemesAny = self[CFBundleURLSchemes] else {
            throw NSError(domain: "", code: 5, userInfo: [
                NSLocalizedDescriptionKey:
                    "The " + CFBundleURLTypes + " dictionary has no "
                    + CFBundleURLSchemes + ". Keys \(self.keys.count): "
                    + self.keys.joined(separator: " ")
            ])
        }
        guard let schemes = schemesAny as? [String] else {
            throw NSError(domain: "", code: 6, userInfo: [
                NSLocalizedDescriptionKey:
                    "Couldn't cast " + CFBundleURLSchemes + " to [String]. "
                    + String(describing: schemesAny)
            ])
        }
        guard let _ = schemes.first else {
            throw NSError(domain: "", code: 7, userInfo: [
                NSLocalizedDescriptionKey:
                    "First " + CFBundleURLSchemes + " is an empty array."
            ])
        }
        return schemes
    }
}

private extension String {
    func validateAsURLScheme() throws -> String {
        // Create a URL from the unchecked scheme to validate it. Note that URL
        // creation itself will succeed if the scheme is invalid but the .scheme
        // property will be nil.
        guard let _ = URL(string:self.appending(path))?.scheme else {
            let message = self.contains("_")
            ? " Scheme contains underscore _, which isn't allowed."
            : ""
            throw NSError(domain: "", code: 8, userInfo: [
                NSLocalizedDescriptionKey:
                    "Scheme \(self) + self

```

```
        + "\" can't be used to construct a URL." + message
    })
    }
    return self
}
}
```

Appendix: Troubleshooting

In case of difficulty when following the instructions to [Test runtime initialization](#), check these troubleshooting tips first.

Tips that mention flipping to Hub only apply to initial enrollment of the app. Subsequent launches after enrollment has finished don't require a flip to Hub.

OpenURLRequestFailed error and no flip to Hub

It could happen that the device **doesn't flip to Hub** and the initial check callback receives an **OpenURLRequestFailed** error.

The full error could appear like one of these.

```
controllerDidFinishInitialCheck failed (extension in AWSDK):
AWError.AWError.SDK.OpenURLRequestFailed.airWatchApplicationNotInstalled.

controllerDidFinishInitialCheck failed (extension in AWSDK):
AWError.AWError.SDK.OpenURLRequestFailed.airWatchApplicationSchemeNotInAllowedLists.
```

Check the app property **LSApplicationQueriesSchemes** (Queried URL Schemes) is present and has all the required items.

The [Add Queried URL Schemes and other required properties](#) instructions list the required items and how to configure them.

No flip back from Hub

It could happen that the device flips to Hub but **doesn't flip back** to your app and the initial check callback isn't invoked.

- Check that the custom URL scheme is registered correctly.

Instructions for registration are in the [Task: Configure application properties](#) in the [Declare a custom URL scheme](#) instructions. The last instruction includes a way to confirm registration.

- Check that the correct custom URL scheme is specified to the SDK.

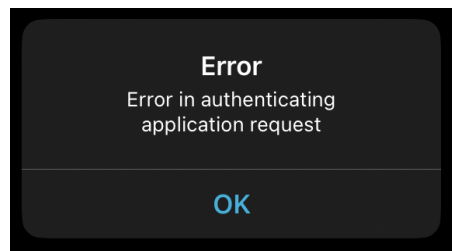
The scheme is specified by the app code accessing the AWController singleton and setting its **callbackScheme** property before starting the SDK.

See the instructions to [Initialize from SwiftUI](#) or [Initialize from Storyboard](#) for detailed steps.

OpenURLRequestFailed error after flipping to Hub and back

It could happen that the device flips to Hub and back but the initial check callback receives an `OpenURLRequestFailed` error.

The Hub app could be showing an error like this.



Screen Capture: Hub error authenticating application

The error returned to the callback could appear like this.

```
controllerDidFinishInitialCheck failed (extension in AWSDK):  
AWError.AWError.SDK.OpenURLRequestFailed.failedToFetchEnvironmentDetailsFromAnchor.
```

The problem could be that the app isn't recognized by Hub or UEM as one that the enrolled end user should be running.

- If the app hasn't been installed via Workspace ONE at least once then do so now. Instructions can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.

- Check the bundle identifier is the same in the UEM as in your app project.

You can check what bundle identifier the UEM has by following these steps.

1. **Log in to the UEM and select the Organization Group (OG) of the end user that you are using for development.**
2. **Navigate to: Resources, Apps, Native.**

Depending on your console version and customization the navigation could be different.

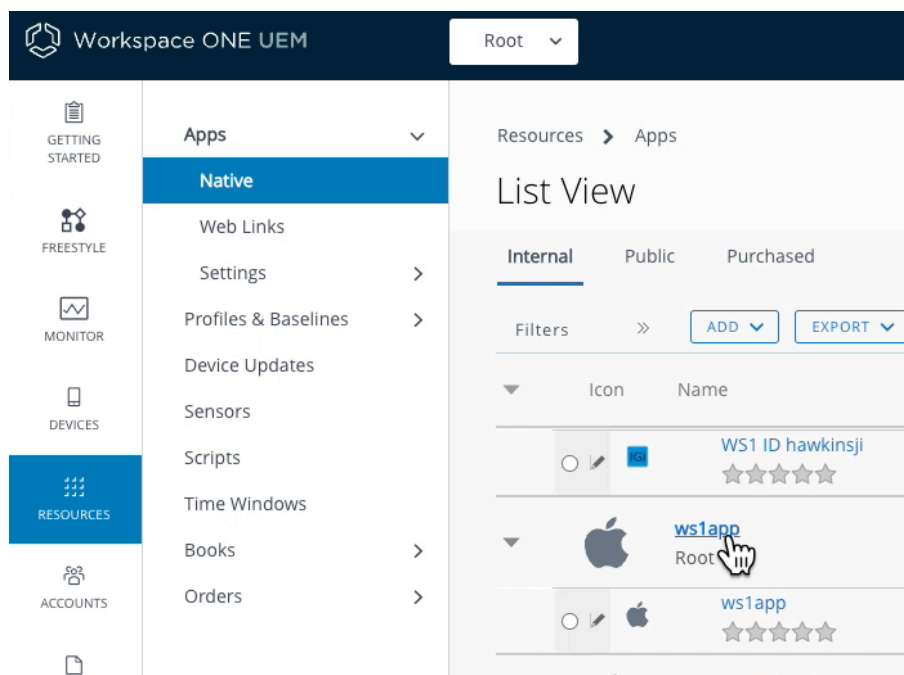
- Resources might be labelled Apps & Books instead.
- Apps might be labelled Applications instead.

However you navigate, a list of applications will open.

3. **Select the Internal tab if it isn't selected by default.**
4. **Select your app from the list.**

Select the upper row for the app, not the version row.

This screen capture shows the location in the console user interface.

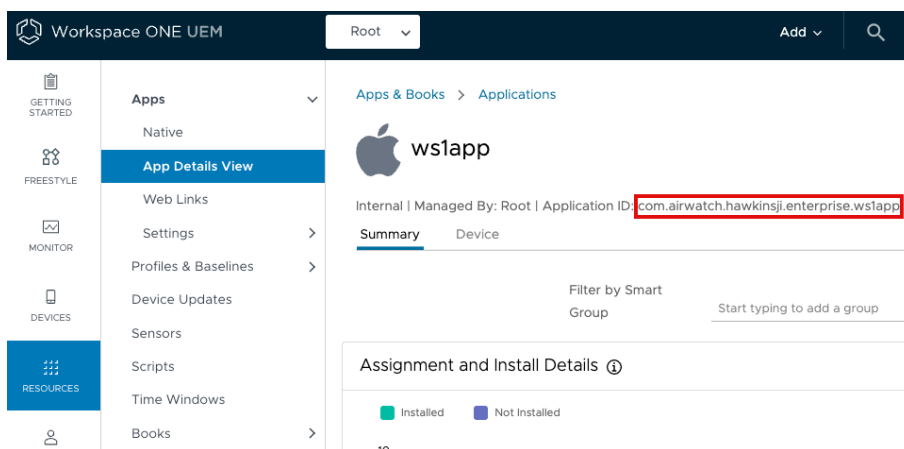


Screen capture: Select app in console

A detail screen will open for the app.

The Application ID on the detail screen will be the bundle identifier.

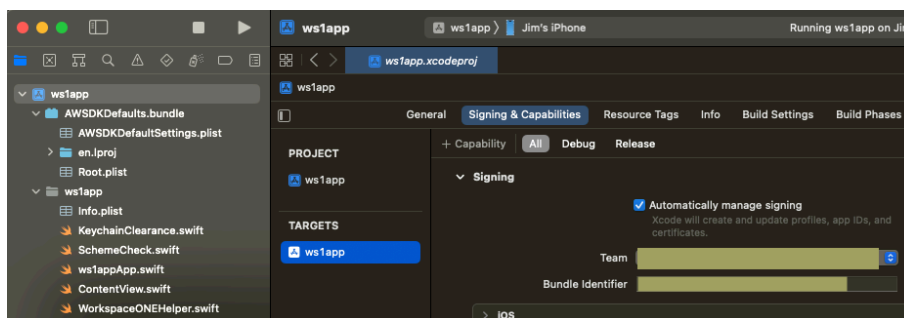
The required value is below the app icon and name, and above the Summary and Device tab selectors. This screen capture shows the location in the console user interface.



Screen capture: Application ID in console

In that screen capture the bundle identifier and Application ID is **com.airwatch.hawkinsji.enterprise.ws1app**

Compare that value with the Bundle Identifier in the Xcode project. You can see it in the target editor. The location in the Xcode user interface is shown in this screen capture.



Screen capture: Bundle identifier in Xcode

In that screen capture the bundle identifier is **com.airwatch.hawkinsji.enterprise.ws1app**

If the bundle identifiers are different, then do one of the following.

- Change the bundle identifier in Xcode to be the same as the Application ID in UEM.
- Generate a new app installer file for your app and upload it to the UEM console. Instructions can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.

Otherwise, if the bundle identifiers are the same, then check the following.

- Ensure that the app is assigned to the end user.

Setting end user assignment should have been done as part of following the Integration Preparation Guide discussed in the [Welcome](#) section.

The easiest way to check could be to step through the instructions to Configure and publish the application, which is where the app assignment is set.

- Ensure that the app is being signed by the same developer team now as when the app installer was uploaded to UEM. There's no way to check the signing team in the UEM user interface. If you aren't sure then you have to generate a new app installer file for your app and upload it to the UEM console. Instructions can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.
- Ensure that the correct team identifier has been passed to the SDK.

Check for the error code 9109. It could appear in the Xcode console in a message like this.

```
[E] ErrorResponse received from anchor app:
Optional(AWOpenURLClient.OpenURLError.ErrorResponse("9109"))
[com.air-watch.sdk.main AWController+OpenURLRequestResponse:228]
```

It could also appear in the open URL received by your app. For example, if the custom URL scheme is "ws1app" then the received URL could be like this.

ws1app://appRegisterResponse?erc=9109

That error code could mean that the **teamID** property is set to a different value to the identifier of the team that signed the app. That causes Hub verification of the app to fail. Rectify it by setting the property to the correct value.

See the [Task: Initialize the software development kit runtime](#) for details of how to obtain and set the required value.

Repeat enrollment or reset app state

In case you want to repeat the enrollment process, or want to completely reset the state of the app on the device, note the following.

- Removing an app from an iOS or iPadOS device doesn't clear the app keychain.

The SDK will store some identifiers and other management data in the app keychain. The operating system doesn't provide any user interface to clear the app keychain, other than by resetting the whole device to factory defaults.

You can add code to your app to delete its keychain. See the [Appendix: Keychain Clearance Sample Code](#) for some Open Source sample code.

Clearing the keychain won't delete any other data such as files written by the SDK.

- Removing the app from a device will remove any files written by the app or SDK.

So a reset sequence could be like this.

1. Clear the keychain from within the app.
2. Uninstall the app.
3. Install the app via Workspace ONE. Instructions can be found in the Integration Preparation Guide discussed in the [Welcome](#) section.

Appendix: Keychain Clearance Sample Code

If you want to add a keychain clearance option to your application, you can use the code here. The use for keychain clearance is discussed in the [Appendix: Troubleshooting](#).

The sample code is also published here.

github.com/euc-releases/.../KeychainClearance.swift

```
// Copyright 2023 Omnisia, Inc.
// SPDX-License-Identifier: BSD-2-Clause

import Foundation
import UIKit
import SwiftUI

// Handy extension to get an error message from an OSStatus.
extension OSStatus {
    var secErrorMessage: String {
        return (SecCopyErrorMessageString(self, nil) as String?) ?? "\(self)"
    }
}

func clearAppKeyChain() {
    [
        // List here is in the same order as in the reference documentation.
        kSecClassGenericPassword,
        kSecClassInternetPassword,
        kSecClassCertificate,
        kSecClassKey,
        kSecClassIdentity
    ].forEach {secClass in
        // Query to find all items of this security class.
        let query: [CFString: Any] = [kSecClass: secClass]
        let status = SecItemDelete(query as CFDictionary)
        if status == errSecSuccess || status == errSecItemNotFound {
            print("Deleted \"\((secClass)\" from keychain.")
        }
        else {
            print("Failed to delete \"\((secClass)\" from keychain:"
                , " \((status.secErrorMessage)")
        }
    }
}

private let alertTitle = "Delete the app keychain?"
private let alertMessage =
    "Everything will be deleted and you will have to enrol again."
private let alertOKLabel = "Delete"
private let alertCancelLabel = "Cancel"

extension View {
    func alertClearAppKeyChain(isPresented: Binding<Bool>) -> some View {
        // TOTH https://flaviocopes.com/swiftui-alert/
        // Uses the deprecated Alert but still useful to get started.
        return self.alert(alertTitle, isPresented:isPresented) {
            Button(alertCancelLabel, role: .cancel) {}
            Button(alertOKLabel, role: .destructive) { clearAppKeyChain() }
        } message: {
            Text(alertMessage)
        }
    }
}
```

```
func alertClearAppKeyChain(_ viewController:UIViewController) {
    let confirm = UIAlertAction(title: alertOKLabel, style: .destructive) {_ in
        clearAppKeyChain()
    }
    let cancel = UIAlertAction(title: alertCancelLabel, style: .cancel) {_ in}
    let alert = UIAlertController(
        title: alertTitle,
        message: alertMessage,
        preferredStyle: .alert
    )
    alert.addAction(confirm)
    alert.addAction(cancel)
    viewController.present(alert, animated: true)
}
```

Acknowledgements

- Code for the extension to get an error message from an OSStatus.
github.com/omnissa-archive/captive-web-view/.../StoredKey.swift#L37
- Reference documentation for **kSecClass** constants.
[developer.apple.com/.../keychain_items/...](https://developer.apple.com/.../keychain_items/)
- Query to find all items in a security class.
github.com/omnissa-archive/captive-web-view/.../StoredKey.swift#L184

Document Information

Published Locations

This document is available

- in Markdown format, in the repository that also holds the sample code:
github.com/euc-releases/.../IntegrationGuideForiOS/...BaseIntegration/
- in Portable Document Format (PDF), on the Omnisia website:
developer.omnisia.com/.../WorkspaceONE_iOS_BaseIntegration.pdf

Revision History

19may2023 First correct publication, for 23.04 SDK for iOS.
 08jun2023 Publication for 23.06 SDK for iOS.
 19jul2023 Publication for 23.07 SDK for iOS.
 11sep2023 Publication for 23.09 SDK for iOS.
 31oct2023 Publication for 23.10 SDK for iOS.
 14Dec2023 Publication for 23.12 SDK for iOS.
 29Jan2024 Publication for 24.01 SDK for iOS.
 15May2024 Publication for 24.04 SDK for iOS.
 01Jul2024 Publication for 24.06 SDK for iOS.
 06Aug2024 Publication for 24.07 SDK for iOS.
 30Sep2024 Publication for 24.09 SDK for iOS.
 28Nov2024 Publication for 24.11 SDK for iOS.
 11Feb2025 Publication for 25.02 SDK for iOS.
 10Apr2025 Publication for 25.04 SDK for iOS.
 30May2025 Publication for 25.04.1 SDK for iOS.
 18Jun2025 Publication for 25.06 SDK for iOS.
 29Aug2025 Publication for 25.08 SDK for iOS.
 10Nov2025 Publication for 25.11 SDK for iOS.
 20Jan2026 Publication for 26.01 SDK for iOS.

License

This software is licensed under the [Omnissia Software Development Kit \(SDK\) License Agreement](#); you may not use this software except in compliance with the License.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.