

VMware AirWatch iOS SDK Technical Implementation Guide

Empowering your enterprise applications with MDM capabilities

AirWatch SDK v5.9.3

Have documentation feedback? Submit a Documentation Feedback support ticket using the Support Wizard on support.air-watch.com.

Copyright © 2016 VMware, Inc. All rights reserved. This product is protected by copyright and intellectual property laws in the United States and other countries as well as by international treaties. VMware products are covered by one or more patents listed at <http://www.vmware.com/go/patents>.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

Table of Contents

| | |
|---|-----------|
| Chapter 1: Overview | 5 |
| Introduction to the AirWatch SDK for iOS | 6 |
| Migrate to the Latest SDK Version | 7 |
| Supported Operating Systems and Requirements | 8 |
| Chapter 2: Getting Started | 9 |
| Overview | 10 |
| Xcode Components | 10 |
| Configuring Server Connections, Initial Setup | 13 |
| SDK and Application Profiles | 16 |
| Implementing the Beacon | 16 |
| Implement the DataSampler | 18 |
| Implementing MDM Status | 20 |
| Chapter 3: Advanced Application Management | 23 |
| MAM Functionality With Settings and Policies and the AirWatch SDK | 24 |
| Assign the Default or Custom Profile | 24 |
| Set the AirWatch Agent for Apple iOS | 25 |
| Supported Settings and Policies Options By Component and AirWatch App | 25 |
| Authentication | 28 |
| Offline Access | 35 |
| Compliance or Compromised Protection | 35 |
| Proxy, App Tunneling | 37 |
| Content Filtering | 38 |
| Geofencing | 38 |
| Restrictions, DLP | 38 |
| Branding | 41 |
| Logging | 43 |
| Analytics | 45 |
| Custom Settings | 48 |
| Chapter 4: Content Management | 50 |

| | |
|---|-----------|
| Overview | 51 |
| External Repositories | 51 |
| AirWatch Content | 53 |
| Personal Content | 54 |
| Download Content | 61 |
| Chapter 5: Certificate Provisioning | 63 |
| Overview | 64 |
| Create the Application Profile | 64 |
| Retrieve a Certificate From an Application Profile | 64 |
| Chapter 6: Test Apps With the AirWatch SDK for iOS | 67 |
| Overview | 68 |
| Testing Process | 68 |
| Deliver Profiles to SDK Applications | 68 |
| Use the Persisted Settings in the SDK | 69 |
| Considerations for Testing Certificates | 69 |
| Chapter 7: SDK and Application Payload Classes | 70 |
| Overview | 71 |
| Retrieving Settings | 71 |
| Payload Lists | 71 |
| Chapter 8: Integrate With Swift Applications | 72 |
| Overview | 73 |
| Integration Process | 73 |
| Initialize the SDK | 73 |
| Appendix: Release Notes | 75 |
| AirWatch SDK v5.9.3 January 2017 | 75 |
| AirWatch SDK v5.9.1 August 2016 | 75 |
| AirWatch SDK v5.7.1 April 2016 | 75 |
| AirWatch SDK v5.5 January 2016 | 75 |
| AirWatch SDK v5.4.1 August 2015 | 76 |

| | |
|--|-----------|
| AirWatch SDK v5.3 June 2015 | 76 |
| AirWatch SDK v5.2 March 2015 | 76 |
| AirWatch SDK v5.0 March 2015 | 76 |
| AirWatch SDK v4.3.1 January 2015 | 76 |
| AirWatch SDK v4.3 November 2014 | 77 |
| AirWatch SDK v4.2 – September 2014 | 77 |
| AirWatch SDK v4.1 – June 2014 | 77 |
| AirWatch SDK v4.0 – February 2014 | 77 |
| Accessing Other Documents | 78 |

Chapter 1:

Overview

| | |
|--|---|
| Introduction to the AirWatch SDK for iOS | 6 |
| Migrate to the Latest SDK Version | 7 |
| Supported Operating Systems and Requirements | 8 |

Introduction to the AirWatch SDK for iOS

The AirWatch Software Development Kit (SDK) is a set of tools allowing organizations to incorporate a host of features and functionality into their custom-built iOS applications. The AirWatch SDK enhances the security and functionality of those applications and in turn helps save application development time and money. This document reviews the SDK implementation process, and also covers the options that are available and how they are configured.

Process Overview



Integrating an application with the AirWatch SDK can be broken down into five main steps. A high-level overview of each step is listed below.

Enable the Core SDK Framework within Xcode

These steps detail the core iOS frameworks and the AirWatch frameworks that you add to your project in order for the SDK to function properly. The AirWatch frameworks are made available by running the provided **AirWatch SDK.dmg** file. In order for your custom application to use the AirWatch SDK, you must first complete the following setup procedures in Xcode:

- Add Required Frameworks
- Configure the Server Connections

The following modules enable the device management framework and allow you to configure device management features into your application:

- Implement the Beacon
- Implement the DataSampler

Select and Implement Additional SDK Modules

AirWatch provides a number of pre-configured functions for your app that can be controlled from the AirWatch Console. These modules make up the **Application Management Framework**. You must decide which SDK modules to use within your application.

Developing Modules

Developers have the option, for most modules, to code the expected behavior or to set the behavior in the AirWatch Console. If you want the application to behave a certain way every time, you must code this behavior in to the application.

Coding the Logging Level

The exception is **Logging**. You must code the logging level and you must set this option in the AirWatch Console. This configuration ensures that your network is not burdened with unwanted logging activity.

Using Default Settings for SDK Profiles

Use the **Security Policies** and **Settings** pages to configure settings once and then share them across AirWatch applications using the **iOS Default Settings @ [Organization Group]** profile.

You can also use the **Profiles** page to configure custom settings with specific behaviors.

Implementing each module into your app is a two-step process:

1. Implement the functionality for the desired module within your app (in Xcode).
2. Add the corresponding configuration to the SDK Profile (in the AirWatch Console) that gets assigned to the app.

Using Certificates

The AirWatch SDK allows you to provision and embed certificates into your app upon deployment. The process involves three main steps:

- Configuring the certificate authority (CA) and the CA Template.
- Creating the App Profile in the AirWatch Console.
- Assigning the App Profile to the application in AirWatch prior to app deployment. See [Certificate Provisioning](#).

Set Default Settings within AirWatch

In the AirWatch Console, you must configure default SDK settings to assign it to your app. These settings include configurations specific to each module you plan on utilizing.

Upload the Application to AirWatch

Once your app is completely built, you need to upload the file into AirWatch using the AirWatch Console. During this process, you need to assign the SDK profile you created, making the settings defined in the SDK profile available to your application. The **Mobile Application Management (MAM) Guide** describes how to upload an application.

Deploy the Application

The final step is to deploy your application to managed devices through the AirWatch Console. Users now have access to the application, along with all the SDK enabled features you've implemented.

For more information on deploying applications to managed devices, see the **Mobile Application Management (MAM) Guide**.

Migrate to the Latest SDK Version

In the latest SDK for iOS, we have updated various UI screens presented by the SDK. These pages now incorporate storyboards that require you to import the **new AWKit.bundle** included in the new SDK DMG. This **AWKit.bundle** contains the compiled storyboards required for the app to function.

We have also integrated with the latest Safari View Controller for various process flows on UI screens. Add **SafariServices.Framework** so all AirWatch screens work without error.

Here are the general instructions based on how you use the SDK today:

1. Replace or import the AWKit.bundle.
 - Replace the older version of AWKit.bundle with the newer version provided in the DMG file if you already import AWKit.bundle into your project bundle resources today.
 - Import AWKit.bundle into your project under **Bundle Resources in Xcode Build Phases** if you have never imported the AWKit.bundle into your project.
2. Import the SafariServices.Framework into your project.

Supported Operating Systems and Requirements

The AirWatch SDK for iOS is compatible with the listed operating systems and requires the listed components.

Supported iOS Operating Systems

The AirWatch SDK for iOS supports the use of the Apple operating systems iOS 7.0+. Certain features require a newer operating system and these features are noted.

Requirements

Meet the following requirements before using the AirWatch SDK for iOS:

- To manage organization groups, get access to the AirWatch Console v8.0+ with the appropriate access rights.
- Know application development using Xcode (for more information, see <https://developer.apple.com/xcode/>) and use **Xcode v6.0.1+**.
- To develop the application, get the necessary SDK DMG (AirWatch SDK.dmg) file from AirWatch.

Chapter 2:

Getting Started

- Overview 10
- Xcode Components10
- Configuring Server Connections, Initial Setup13
- SDK and Application Profiles16
- Implementing the Beacon 16
- Implement the DataSampler18
- Implementing MDM Status 20

Overview

Perform the following tasks to prepare to use the AirWatch SDK for iOS.

Xcode Components

Add the listed Xcode components.

Add Required Xcode Frameworks

The SDK depends on the following frameworks to function properly. Follow the steps to add the necessary frameworks to your project.

1. Select the project in the Groups & Files pane in Xcode.
2. Ensure that the proper target is selected on the left side, select the **Build Phases** tab.
3. Expand the **Link Binary With Libraries** section.
4. Select the + button at the bottom of the section to add the required frameworks.
5. Select each one of the following frameworks and select **Add**.
 - **Accelerate.framework**
 - **AssetsLibrary.framework**
 - **AudioToolbox.framework**
 - **AVFoundation.framework**
 - **CFNetwork.framework**
 - **CoreData.framework**
 - **CoreFoundation.framework**
 - **CoreGraphics.framework**
 - **CoreLocation.framework**
 - **CoreMedia.framework**
 - **CoreMotion.framework**
 - **CoreTelephony.framework**
 - **CoreText.framework**
 - **CoreVideo.framework**
 - **Foundation.framework**
 - **ImageIO.framework**
 - **Libc++.tbd**

- `Libsqlite3.tbd`
- `Libstdc++.6.0.9.tbd`
- `Libz.tbd`
- `LocalAuthentication.framework`
- `MediaPlayer.framework`
- `MessageUI.framework`
- `MobileCoreServices.framework`
- `QuartzCore.framework`
- `SafariServices.framework`
- `Security.framework`
- `SystemConfiguration.framework`
- `UIKit.framework`

Add Required Xcode Bundle Resources

The SDK depends on the following bundle resources to function properly, so add the necessary bundles to your project. Find some of these bundles inside the **AWSDK.framework** file structure.

- `SDKLocalization.bundle`
- `AWKit.bundle`

Important: Add the **AWKit.bundle** to your project or the application can crash.

Add the AirWatch SDK Frameworks

The AirWatch frameworks are made available by running the provided **AirWatch SDK.dmg** file.

- Drag **AWSDK.framework** into the Frameworks group in the sidebar of Xcode. Make sure to check **Copy items into destination group's folder**.
- Ensure the application is linking against the **AWSDK.framework**. To verify, select the **Build Phases** tab in the properties of the target. Expand the **Link Binary With Libraries** section to see if **AWSDK.framework** is added. If not, select the + button and select **AWSDK.framework**.
- Import the umbrella header wherever you use the AirWatch SDK. Add `#import <AWSDK/AWSDKCore.h>` to the top of the file.

Add Linker Flags

Since the AirWatch SDK uses Objective-C categories, you must pass linker flags to the linker to properly load them.

- Select the project or workspace in the **Groups & Files** pane.
- Select the target for the application.
- Select the **Build Settings** tab.
- Ensure the **-ObjC** flag is added in the **Other Linker Flags** entry.

Valid Architectures

The SDK currently supports the following architectures:

- ARMv7
- ARMv7S
- ARM64

Do not compile the i386 architecture/simulator because the SDK only supports real physical devices.

Callback Scheme Registration

To receive a callback from the AirWatch Agent, the application exposes a custom scheme in the **info.plist**.

1. Navigate to **Supporting Files** in Xcode.
2. Select the file **<YourAppName>-Info.plist**.
3. Navigate to the **URL Types** section. If it does not exist, add it at the **Information Property List** root node of the PLIST.
4. Expand the **Item 0** entry and add an entry for **URL Schemes**.
5. Set the next **Item 0** under **URL Schemes** to the desired callback scheme.

| Key | Type | Value |
|-----------------------------|------------|--------------|
| ▼ Information Property List | Dictionary | (17 items) |
| ▼ URL types | Array | (1 item) |
| ▼ Item 0 | Dictionary | (1 item) |
| ▼ URL Schemes | Array | (1 item) |
| Item 0 | String | awssobroker2 |

Compile With Xcode 7

If you use Xcode 7 to compile your SDK application, take these steps to ensure that the application functions properly.

1. Add an array key named **LSApplicationQueriesSchemes** to the **info.plist**.
2. Add the bundle identifier of the AirWatch Agent or AWSSOBroker2 application to the array.

The schemes for the Agent and AWSSOBroker2 are as follows:

- airwatch
- AWSSOBroker2

3. Navigate to your Xcode build settings and set **Enable Bitcode** to **No**.

Configuring Server Connections, Initial Setup

To start using the AirWatch SDK, you must set an initial configuration so that the SDK receives the AirWatch Device Services URL. This configuration allows the SDK to communicate with the server for every transaction that requires interaction with the AirWatch Console.

iOS 9+ and the Device Services Server

For iOS 9+, ensure that the Device Services server meets Apple's security requirements. The SDK must communicate with the Device Services server to run. The system might block communication if the server does not comply with requirements. Search the Apple Developer site for current application transport security requirements: https://developer.apple.com/library/prerelease/ios/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html#//apple_ref/doc/uid/TP40016198-SW1.

Initialize the SDK

Before you can use the SDK, you must initialize it. The **AWController** class is the main component responsible for initializing the SDK. In addition, it automatically handles and implements certain core SDK functionalities to improve ease of integration for developers, such as the following functions:

- Passcode mode
- Single sign-on (SSO)
- SSID filtering
- Proxy and tunneling

Calling **start** in **AWController** automatically sets up the proxy to redirect traffic. However, you must wait until the initial **CheckDoneWithError** callback is received before any network traffic redirects. Wait until the SDK finishes setting up before making any network calls through your proxy.

Initialization Example

The example demonstrates how to initialize the SDK.

1. Import the `<AWSDK/AWController.h>` header file. Associate the **AWControllerDelegate** to your app delegate.

```
@interface AppDelegate : UIResponder <UIApplicationDelegate, AWSDKDelegate>
```

2. Inside the app delegate, implement the following code:

```

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Configure the Controller by:
    AWController *controller = [AWController sharedInstance];
    // 1) defining the callback scheme so the app can get called back,
    controller.callbackScheme = @"UrlScheme";
    // 2) set the delegate to know when the initialization has been completed.
    controller.delegate = self;

    return YES;
}

```

3. Start the SDK initialization inside the **applicationDidBecomeActive** delegate method.

```

- (void)applicationDidBecomeActive:(UIApplication *)application {
    [[AWController sharedInstance] start];
}

```

Do not call the start method in **didFinishLaunchingWithOptions** because the SDK may display modal view controllers that rely on a reference view controller. Sometimes, when you use a storyboard, the view controllers have not yet been generated at the time **didFinishLaunchingWithOptions** is called. To avoid any unstable behavior with the app, call `[[AWController sharedInstance] start]` inside **applicationDidBecomeActive** instead.

4. Next, implement the code to handle the callback from the AirWatch Agent or AirWatch Container app.

```

- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url sourceApplication:(NSString *)sourceApplication annotation:(id)annotation {
    return [[AWController sharedInstance] handleOpenURL:url fromApplication:sourceApplication];
}

```

5. Implement the remaining delegate methods:

```

- (void)initialCheckDoneWithError: (NSError *)error

```

This delegate method is invoked when the SDK initializes. This method is **ALWAYS** called after the SDK passes through the initialization flow. If the initialization is successful, then the error object is nil. If the initialization fails, then the error object contains the reason code for why it fails.

```
- (void)receivedProfiles:(NSArray *)profiles
```

This delegate method is invoked when settings of an SDK profile assigned to this application update on the AirWatch Console. It notifies the app that new settings are available. The profiles array contains the list of **AWProfile** objects that contain configuration payloads.

```
- (void)unlock
```

This delegate method is invoked immediately after you initiate a new SSO session by inputting the correct password/passcode.

```
- (void)lock
```

This method is invoked when the SSO session has expired and the SDK passcode input view is displayed. It is intended for use as an indicator of when a user no longer has to access the app. This lock allows the developer to implement the necessary logic to take the proper action for when the app is locked.

```
- (void)wipe
```

This method is invoked when the SDK identifies that the device has been wiped or unenrolled from the AirWatch Console. This method is also invoked when a user reaches the limit of failed passcode attempts defined in the SDK profile.

Note: The AirWatch SDK only invokes this method, and it takes no other actions. The application developer must implement the necessary local app wipe logic.

```
- (void)stopNetworkActivity
```

This method is invoked when the device connects to an SSID that is blacklisted in the SDK profile.

```
- (void) resumeNetworkActivity
```

This method is invoked when the device connections to a valid SSID after network activity is already stopped.

SDK and Application Profiles

The SDK associates with two types of AirWatch profiles. These two types are SDK Profiles and application profiles. You assign both types of profiles to the application from the AirWatch Console. These profiles are different from AirWatch Device Profiles.

- **SDK Profiles** – Used to deliver security policies and settings down to the SDK embedded application. Upon receiving an SDK profile, the SDK automatically stores the most recent profile settings in memory.
- **Application Profiles** – Used to deliver certificates from an upload or a certificate authority down to an application. Consider using the [challenge handler](#) and integrated authentication instead of application profiles.

Polling for Commands and Profile Updates

The SDK checks for new commands from AirWatch when the app is active in the background. Examples of commands are send logs, update SDK profile, and lock application. However, there may be times when your app wants to check for new commands while active in the foreground. You can do so using the **AWCommandManager** class with the **loadCommands** method.

```
// Receive commands.
[[AWCommandManager sharedManager] loadCommands];
```

Implementing the Beacon

You can set up the Beacon to send device information to the AirWatch Console by specifying a time interval. Generic device information such as the device name, OS version, and compromised status is sampled. In addition, the Beacon module is used to start location services by specifying a location mode.

Configuration

To take advantage of the location functionality of the Beacon, the host application registers itself as needing location updates in the background. For information on the location functionality of the Beacon, refer to the **Declaring Your App's Supported Background Tasks** section in the [iOS App Programming Guide](#).

Sample Code

```
// Initialize Beacon.  Modify the values as needed.
AWBeacon *_beacon = [[AWBeacon alloc] initWithAPNSToken:nil
                    transmitInterval:300
                    locationGroup:nil
                    locationMode:AWLocationModeDisabled
                    distance:kCLDistanceFilterNone];

// Starts the beacon to send periodically a ping to the server.
[_beacon start];

// Force to send a ping right now.
[_beacon send];
```

- **initWithAPNSToken** – Determines if your application uses APNS tokens and sends tokens to the AirWatch Console. You can send this value as nil.
- **transmitInterval** – Represents the frequency in which the Beacon checks in with the AirWatch Console (in seconds).
- **locationGroup** – Corresponds to the organization group. If your application uses authentication, it prompts users to log in. You can send this value as nil.
- **locationMode** – Uses location services for the Beacon and includes the coordinates when reporting back to the server. This method also sets up the Beacon to run in the background.
 - **AWLocationModeDisabled** – Specifies no location mode.
 - **AWLocationModeStandard** – Captures data using the GPS (on only GPS-enabled devices), which can consume battery power when enabled.
 - **AWLocationModeSignificant** – Uses the significant location services from iOS and provides updates only when the device location changes at a significant level. Consider using this mode if you want to use location services.
- **distance** – Determines if you are using the standard location servers and sets the threshold, in meters, of when to generate a location service notification.

Starting and Stopping the Beacon

Once you create an instance of the Beacon with the appropriate configuration settings, you can start it or stop it at any time. Start the Beacon after initializing the SDK, and leave it running permanently.

```
[_beacon start]; // Starts sending information to the AW Console
```

```
[_beacon stop]; // Stops sending information to the AW Console
```

Manually Sending a Beacon Message

Instead of waiting for the next interval to send a Beacon to the server, you can explicitly invoke the send command and a packet will be sent to the server.

```
// Force to send a ping right now.
[_beacon send];
```

Implement the DataSampler

The DataSampler module (formerly known as Interrogator) samples detailed device data and reports it back to the AirWatch Console. Device details such as analytics, call logs, GPS location, and network adapters are all sampled with the DataSampler.

The DataSampler samples and transmits on two different time intervals. Device samples remain on to the disk and the system removes them after transmitted. This process allows the developer to sample statistics multiple times before sending them to AirWatch. Samples stored on the disk are useful when a device does not have network connectivity.

AWDataSampler is a singleton object. There can only be one DataSampler for each process.

Configuration

These parameters are required to set up a DataSampler.

- **sampleModules** – Names the bitmask whose flags specify which modules to use.
- **defaultSampleInterval** – Specifies the time in seconds between DataSampler samples for all modules by default.
- **defaultTransmitInterval** – Specifies the time in seconds between DataSampler transmissions for all modules by default.
- **traceLevel** – Determines the error and information logging level of the DataSampler module when it is running.

```
[[AWAnalytics mAnalytics] setEnabled:YES];
// Initialize DataSampler. Modify the values in the initialization
AWDataSamplerConfiguration *config = [[AWDataSamplerConfiguration alloc]

initWithSampleModules:(AWDataSamplerModuleAnalytics | AWDataSamplerModuleGPS)
defaultSampleInterval:3600
```

```

        defaultTransmitInterval:14400
        traceLevel:Error];//This bit mask will enable analytics and GPS sampling

// Configure Data Sampler
[[AWDataSampler mDataSamplerModule] setConfig:config];

// Start the Data Sampler Service.
NSError *error;
[[AWDataSampler mDataSamplerModule] startUp:&error];

```

Modules Available for Sampling

These modules are available for sampling in the DataSampler.

- **AWDataSamplerModuleSystem**
- **AWDataSamplerModuleAnalytics**
- **AWDataSamplerModuleGPS**
- **AWDataSamplerModuleNetworkData**
- **AWDataSamplerModuleNetworkAdapter**
- **AWDataSamplerModuleWLAN2Sample**

Gather Telecom Data

Disable the **AWDataSamplerModuleNetworkData** mask if you gather telecom data using the AirWatch Agent. If you enable this mask for the SDK, then you receive duplicate data from the Agent and from the SDK.

Set Do Not Disturb

You can use the SDK to set the do-not-disturb (DND) status on the AirWatch server. You must enable the DND policy in the AirWatch Console. You can find the policy at **Groups & Settings > All Settings > Devices & Users > General > Privacy > DO NOT DISTURB** section.

The two relevant methods are **fetchDeviceDNDStatus** and **setDeviceDNDStatus** found in the **AWDeviceDNDStatus** object. The example illustrates how to implement a toggle button for DND.

```

[AWDeviceDNDStatus fetchDeviceDNDStatus:^(BOOL responseStatus, BOOL dndStatus, NSDate *dndTime,
NSError *error){
    if(dndStatus){
        [AWDeviceDNDStatus setDeviceDNDStatus:NO completionBlock:^(BOOL responseStatus, BOOL
dndStatus, NSDate *dndTime, NSError *error){

```

```

        NSLog(@"DND Disabled");
    }
}
else{
    [AWDeviceDNDStatus setDeviceDNDStatus:YES completionBlock:^(BOOL responseStatus, BOOL
dndStatus, NSDate *dndTime, NSError *error){
        NSLog(@"DND Enabled");
    }];
}
}
};

```

Implementing MDM Status

The MDM Status module allows an application to check certain properties and the status of some MDM properties for the device that the application lives in.

This information is useful because you cannot obtain it directly from the SDK. You can use the data to improve security and usability at the application level. For example, a developer may want to check that another application is installed on the same device before exposing or hiding certain features in the application.

- **Device Status** – Indicates the following statuses:
 - **Managed Status** – Indicates if the device is Enrolled in the Console or not.
 - **Compliance Status** – Indicates if the device is conforming to all the compliance rules.
- **Requery Method** – Queries the Console to send to the containing device a Query command to collect certain types of device information.
- **Compliance Policies** – Retrieves a list of policies and lists details about each policy.
- **Application List** – Retrieves the list of applications that are available on the same device.

Sample Code

The following sample code represents the different methods that can be used to retrieve MDM status of a device.

```

/* This sample code assumes that start in AWController has been called and initialCheckDoneWithError has
returned successfully. */
AWMDMInformationController *amic = [[AWMDMInformationController alloc]
init];
/*
 * Retrieve the user's information.
 */

```

```

[amic fetchUserInfoWithCompletionBlock:^(BOOL success, NSDictionary
*userInfo, NSError *error)

{
    if (error)

    {
        NSLog(@"Retrieve UserInfo Error: %@", error);
        return;
    }
    NSLog(@"=== User Info ===");
    NSLog(@"Success? :%@", success? @"YES":@"NO");
    NSLog(@"Name :%@", [userInfo objectForKey:AWMDMUserNameKey]);
    NSLog(@"LG :%@", [userInfo objectForKey:AWMDMUserLocationGroupKey]);
    NSLog(@"IsActive :%@", [[userInfo objectForKey:AWMDMUserIsActiveKey]
boolValue]? @"YES" : @"NO");
    NSLog(@"AcctType :%@", [userInfo objectForKey:AWMDMUserAccountTypeKey]);
}];

/*
* Retrieve device status.
*/

[amic fetchStatusWithCompletionBlock:^(BOOL isManaged,
    AWErollmentStatus enrollmentStatus,
    AWComplianceStatus complianceStatus,
    NSString *group,
    NSString *groupCode,
    NSError *error){

    if (error)

    {
        NSLog(@"Retrieve Status Error: %@", error);
        return;
    }

    BOOL Managed = isManaged;
    NSString *groupID = groupCode;
    //Use these ENUM values to assess status
    AWErollmentStatus Enrolled = enrollmentStatus;
    AWComplianceStatus Compliant = complianceStatus;

}];

```

```

/*
 * Request that the AirWatch Console requery information from the device.
 */

//Select the type to requery you want to do?
AWRequeryType selectedType = AWRequeryApplist;
    //AWRequeryType selectedType = AWRequeryDeviceInfo;
    //AWRequeryType selectedType = AWRequeryProfileInfo;
    //AWRequeryType selectedType = AWRequerySecurityInfo;

[amic reQuery:selectedType withCompletionBlock:^(BOOL success, NSError
*error) {
    NSLog(@"=== Requery Status ===");
    NSLog(@"Success? :%@", success ? @"YES":@"NO");
}];

```

Chapter 3:

Advanced Application Management

| | |
|---|----|
| MAM Functionality With Settings and Policies and the AirWatch SDK | 24 |
| Assign the Default or Custom Profile | 24 |
| Set the AirWatch Agent for Apple iOS | 25 |
| Supported Settings and Policies Options By Component and AirWatch App | 25 |
| Authentication | 28 |
| Offline Access | 35 |
| Compliance or Compromised Protection | 35 |
| Proxy, App Tunneling | 37 |
| Content Filtering | 38 |
| Geofencing | 38 |
| Restrictions, DLP | 38 |
| Branding | 41 |
| Logging | 43 |
| Analytics | 45 |
| Custom Settings | 48 |

MAM Functionality With Settings and Policies and the AirWatch SDK

The Settings and Policies section of the AirWatch Console contains settings that can control security, behaviors, and the data retrieval of specific applications. The settings are sometimes called SDK settings because they run on the AirWatch SDK framework.

You can apply these SDK features to applications built with the AirWatch SDK, to supported AirWatch applications, and to applications wrapped by the AirWatch app wrapping engine because the AirWatch SDK framework processes the functionality.

Types of Functions

AirWatch has two types of the SDK settings, default and custom. To choose the type of SDK setting, determine the scope of deployment.

- Default settings work well across organization groups, applying to large numbers of devices.
- Custom settings work with individual devices or for small numbers of devices with applications that require special mobile application management (MAM) features.

Default Settings

Find the default settings in **Groups & Settings > All Settings > Apps > Settings And Policies** and then select **Security Policies** or **Settings**. You can apply these options across all the AirWatch applications in an organization group. Shared options easier to manage and configure because they are in a single location.

View the [matrices](#) for information on which default settings apply to specific AirWatch applications or the AirWatch SDK and app wrapping.

Custom Settings

Find the custom settings in **Groups & Settings > All Settings > Apps > Settings And Policies > Profiles**. These options mirror the offerings in the **Devices > Profiles** section. Custom settings for profiles offer granular control for specific applications and the ability to override default settings. However, they also require separate input and maintenance.

Assign the Default or Custom Profile

To apply AirWatch features built with the AirWatch SDK, you must apply the applicable default or custom profile to an application. Apply the profile when you upload or edit the application to the AirWatch Console.

1. Navigate to **Apps & Books > Applications > List View > Internal**.
2. Add or edit an application.
3. Select a profile on the **SDK** tab:
 - **Default Settings Profile**
 - For Android applications, select the **Android Default Settings @ <Organization Group>**.
 - For Apple iOS applications, select the **iOS Default Settings @ <Organization Group>**.

- **Custom Settings Profile** – For Android and Apple iOS applications, select the applicable legacy or custom profile.
4. Make other configurations and then save the application and create assignments for its deployment.

Changes to Default and Custom Profiles

When you make changes to the default or custom profile, default in Settings and Policies or custom in the device profile, AirWatch applies these edits when you select **Save**.

Changes can take a few minutes to push to end-user devices. Users can close and restart AirWatch applications to receive updated settings.

Set the AirWatch Agent for Apple iOS

Configure the AirWatch Agent for Apple iOS to use the correct default profile to apply SDK functionality.

Your configurations in Settings And Policies do not work on devices if you do not set the AirWatch Agent to apply the configurations.

1. Navigate to **Groups & Settings > All Settings > Devices & Users > Apple > Apple iOS > Agent Settings**.
2. Set the **SDK Profile V2** option in the **SDK PROFILE** section to the default profile by selecting **iOS Default Settings @ <Organization Group>**.
3. **Save** your settings.

Supported Settings and Policies Options By Component and AirWatch App

Use the default settings profile to apply an AirWatch SDK feature to an SDK application, an AirWatch application, or a wrapped application by setting the configurations in **Policies and Settings** and then applying the profile. View compatibility information to know what features AirWatch supports for your application.

Scope of Matrices

The data in these tables describes the behaviors and support of the specific application.

Settings and Policies Supported Options for SDK and App Wrapping

| UI Label | SDK |
|--|-----|
| | iOS |
| Passcode: Authentication Timeout | ✓ |
| Passcode: Maximum Number Of Failed Attempts | ✓ |
| Passcode: Passcode Mode Numeric | ✓ |
| Passcode: Passcode Mode Alphanumeric | ✓ |
| Passcode: Allow Simple Value | ✓ |

| UI Label | SDK |
|---|-----|
| | iOS |
| Passcode: Minimum Passcode Length | ✓ |
| Passcode: Minimum Number Complex Characters | ✓ |
| Passcode: Maximum Passcode Age | ✓ |
| Passcode: Passcode History | ✓ |
| Passcode: Biometric Mode | ✓ |
| Username and Password: Authentication Timeout | ✓ |
| Username and Password: Maximum Number of Failed Attempts | ✓ |
| Single Sign On: Enable | ✓ |
| Integrated Authentication: Enable Kerberos | X |
| Integrated Authentication: Use Enrollment Credentials | ✓ |
| Integrated Authentication: Use Certificate | ✓ |
| Integrated Authentication: Use NAPPs Authentication | ✓ |
| Offline Access: Enable | ✓ |
| Compromised Detection: Enable | ✓ |
| AirWatch App Tunnel: Mode | ✓ |
| AirWatch App Tunnel: URLs (Domains) | ✓ |
| Content Filtering: Enable | ✓ |
| Geofencing: Area | ✓ |
| DLP: Copy and Paste | ✓ |
| DLP: Printing | ✓ |
| DLP: Camera | ✓ |
| DLP: Composing Email | ✓ |
| DLP: Data Backup | ✓ |
| DLP: Location Services | ✓ |
| DLP: Bluetooth | ✓ |
| DLP: Screenshot | X |

| UI Label | SDK |
|---|-----|
| | iOS |
| DLP: Watermark | ✓ |
| DLP: Limit Documents to Open Only in Approved Applications | ✓ |
| DLP: Allowed Applications List | ✓ |
| NAC: Cellular Connection | ✓ |
| NAC: Wi-Fi Connection | ✓ |
| NAC: Allowed SSIDs | ✓ |
| Branding: Toolbar Color | ✓ |
| Branding: Toolbar Text Color | ✓ |
| Branding: Primary Color | ✓ |
| Branding: Primary Text Color | ✓ |
| Branding: Secondary Color | ✓ |
| Branding: Secondary Text Color | ✓ |
| Branding: Organization Name | ✓ |
| Branding: Background Image iPhone and iPhone (Retina) | ✓ |
| Branding: Background Image iPhone 5 (Retina) | ✓ |
| Branding: Background Image iPad and iPad (Retina) | ✓ |
| Branding: Background Small, Medium, Large, and XLarge | x |
| Branding: Company Logo Phone | ✓ |
| Branding: Company Logo Phone High Res | ✓ |
| Branding: Company Logo Tablet | ✓ |
| Branding: Company Logo Tablet High Res | ✓ |
| Logging: Logging Level | ✓ |
| Logging: Send Logs Over Wi-Fi | ✓ |
| Analytics: Enable | ✓ |
| Custom Settings: XML Entries | ✓ |

Authentication

The AirWatch SDK provides helper classes to authenticate credentials against AirWatch. An application can limit its access to users by integrating user authentication. Users authenticate to the AirWatch Console, whether it is a basic enrollment user or an Active Directory account. Authentication allows your application to follow enforced corporate security policies. Configure the type of authentication the SDK profile uses to communicate with the application.

```
// Initialize Beacon.  Modify the values as needed.
AWBeacon *_beacon = [[AWBeacon alloc] initWithAPNSToken:nil
                    transmitInterval:300
                    locationGroup:nil
                    locationMode:AWLocationModeDisabled
                    distance:kCLDistanceFilterNone];

// Starts the beacon to send periodically a ping to the server.
[_beacon start];

// Force to send a ping right now.
[_beacon send];
```

Single Sign-On and the SDK

To use single sign-on (SSO), an SDK-enabled app must interact with the AirWatch Agent for handling authentication across multiple apps. After initialization, the app can establish an SSO session with the Agent. It can delegate the handling of user authentication and SSO management to the Agent or AirWatch Container. After a session is established in one app, all the other apps can share the session. Applications do not require authentication or passcodes due to this sharing behavior.

The SSO functionality can also allow the application access to the AirWatch enrollment credentials for that device if necessary. When the SSO session expires, access to any SSO app requires the user to enter a passcode (depending on the authentication security policies set) and reinitialize the SSO session. The default settings or SDK profile also defines the maximum number of failed attempts. If the user exceeds this number, the session expires and the wipe delegate method invokes in the associated applications to signal the developer to remove local app data.

To implement the SDK, you must implement the code to [initialize the SDK](#) using **AWController** and calling **Start** from the **clientInstance**. Also, implement the lock, unlock, and wipe methods with the other delegate methods of **AWSDKDelegate** inside your app delegate.

After you upload the SDK application to the console and assign the a custom or default profile with SSO enabled, the SDK handles communication with the AirWatch Agent to manage the sessions.

Once the app has finished the communication workflow with the AirWatch Agent, the app can use the credentials.

Important: To get credentials, devices must enroll using the AirWatch Agent or AirWatch Container. Otherwise, the properties are nil.



For information on using Touch ID with the SDK, see the following AirWatch Knowledge Base article:
<https://support.air-watch.com/articles/95284457-Using-Touch-ID-with-the-iOS-SDK>.

Implementation in Xcode

```
AWEnrollmentAccount *account = [[AWController sharedInstance] account];
NSString *username = account.username;
NSString *password = account.password;
NSString *groupID = account.identifier;
```

Active Directory Password Changes

If an Active Directory (AD) password changes and becomes out of sync with the object account of the SDK, use an API to update the SDK credentials.

```
- (void)updateUserCredentialsWithCompletion: (void(^)(BOOL success, NSError *error))completionHandler;
```

If the callback works, then find the new credentials in the SDK account object.

Behavior

- SSO disabled – The system displays an authentication prompt within the SDK app for the user to enter in the new credentials.
- SSO enabled – The SDK app flips to the AirWatch Agent or Container application to update the credentials there. The AirWatch Agent v5.1+ for iOS and the Container v2.1+ support this behavior.

Authentication Type

Configure AirWatch applications, applications built using the AirWatch SDK, and app wrapped applications to allow access when users authenticate with a set process. Select an authentication type depending on the credentials desired for access; users can set their own or use their AirWatch credentials.

Select an authentication type that meets the security needs of your network. The passcode gives device users flexibility while username and password offers compatibility with the AirWatch system. If security is not an issue, then you do not have to require an authentication type.

| Setting | Description |
|------------------------------|--|
| Passcode | Designates a local passcode requirement for supported applications. Device users set their passcode on devices at the application level when they first access the application. |
| Username and Password | Requires users to authenticate to supported applications using their AirWatch credentials. Set these credentials when you add users in the Accounts page of the AirWatch Console. |
| Disabled | Requires no authentication to access supported applications. |

Authentication Type and SSO

Authentication Type and SSO can work together or alone.

- **Alone** – If you enable an Authentication Type (passcode or user name/password) without SSO, then users must enter a separate passcode or credentials for each individual application.
- **Together** – If you enable both Authentication Type and SSO, then users enter either their passcode or credentials (whichever you configure as the Authentication Type) once. They do not have to reenter them until the SSO session ends.

Soft Unlock

The soft unlock feature temporarily suspends the authentication UI as it keeps the application and its data or the SDK container encrypted. This temporary hold allows applications to defer handling the view controllers for the passcode, user name, and password lock screens, so that they do not prompt for authentication.

Use this feature to continue critical business tasks without interruption from authentication prompts. For example, a VOIP application receiving an incoming call allows the user to take the call without entering the SDK pin. It defers prompts until after the call ends. During this soft unlock period, the SDK container remains encrypted and protected. The effect is cosmetic to allow the controlled handling of the view controller hierarchy.

Implement Soft Unlock

The interface for soft unlock consists of two methods and no configurations in the AirWatch Console.

1. removeLockAnimatedWithCompletionBlock

The method pauses incoming lock screens or defers a lock screen. The app can display its own UI for incoming VOIP calls or other events.

```
-(void) removeLockAnimated:(BOOL) animated withCompletionBlock:
(AWControllerSoftLockCompletionHandler) completionHandler
```

2. resumeLock

The method calls resume to allow the SDK to display of the authentication UI after the app completes its temporary transaction.

```
-(void) resumeLock
```

Balance Calls

You must balance every call to removeLockAnimatedWithCompletionBlock with a resumeLock call. If a removeLock call is not balanced with a resumeLock call, then the SDK does not prompt the user for authentication, whether it is running in the foreground or background. The application user must close and restart the application to enable the authentication prompt.

SSO Session and the AirWatch Agent

Once an end user authenticates with an application participating in SSO, a session establishes. The session is active until the **Authentication Timeout** defined in the SDK profile is reached or if the user manually locks the application.

When using the Agent as a "broker application" for features such as the single sign-on option, configure the AirWatch Agent with the applicable SDK profile. If you are using the default SDK profile, ensure that the Agent is configured to use this profile. If you do not set the Agent to use the default SDK profile, then the system does not apply your configurations you configure in the Settings and Policies section.

Challenge Handler and Integrated Authentication

On the **Security Policies** page, you can set **Integrated Authentication** to **Enabled** to allow the SSO credentials or a certificate to be passed on and used for authenticating into Web sites, such as content repositories (SharePoint) or wikis. The AirWatch SDK does not support the use of SCEP for handling certificates. Do not select SCEP options for certificate authorities for SDK implementations.

Once enabled, you must define a list of allowed sites, which are the only sites supported with Integrated Authentication.

On the application side, use the challenge handler component in the **AWController** class of the AirWatch SDK. Inside the **AWController**, use certain methods to handle an incoming authentication challenge for connections made with **NSURLConnection** and **NSURLSession**. Find the available methods in the list.

| Method | Description |
|---|---|
| -(BOOL)canHandleProtectionSpace: (NSURLProtectionSpace*)protectionSpace withError: (NSError**)error | <p>Checks that the AirWatch SDK has the means to handle this type of authentication challenge. The SDK makes the several checks to determine that it can handle challenges.</p> <ol style="list-style-type: none"> 1. Is the Web site challenging for authentication on the list of allowed sites in the SDK profile? 2. Is the challenge one of the supported types: <ul style="list-style-type: none"> • Basic • NTLM • Client certificate 3. Does the SDK have a set of credentials to respond with: <ul style="list-style-type: none"> • Certificate • User name and password <p>If all three of the criteria are met, then this method returns YES.</p> <p>The SDK does not handle server trust, so your application must handle NSURLAuthenticationMethodServerTrust.</p> |

| Method | Description |
|--|--|
| -(BOOL)handleChallenge: (NSURLAuthenticationChallenge*)challenge | <p>Responds to the actual authentication challenge from a network call made using NSURLConnection.</p> <p>It returns YES or NO depending on if it can respond to the authentication challenge.</p> <p>The system calls the canHandleProtectionSpace method in AWController first to validate that the system can process the challenge.</p> |
| -(BOOL)handleChallengeForURLSessionChallenge: (NSURLAuthenticationChallenge *)challenge completionHandler:(void (^)((NSURLSessionAuthChallengeDisposition disposition, NSURLError *error)))completionHandler; | <p>Responds to the actual authentication challenge from a network call made using NSURLSession.</p> <p>This method is the same as the handleChallenge method, except the system uses this method with calls made with NSURLSession. This call involves using a completion block to handle authentication challenges.</p> |
| -(void)fetchNewCertificatesWithError:(NSError**)error | <p>Forces the SDK to fetch a new certificate.</p> <p>The SDK automatically handles retrieving certificates initially during setup, after you call start in AWController. However, in the event you must force the SDK to fetch a new certificate, use this method.</p> <p>Ensure that a certificate is properly configured in the authentication and credentials payload of the SDK profile.</p> <p>This method resolves issues with revoked and corrupt certificates.</p> |

Integrated authentication requires several configurations to work.

- The URL of the requested Web site must match an entry in your list of **Allowed Sites**.
- The system must make the network call so that the process provides an **NSURLAuthenticationChallenge** object.
- The Web site must return a 401 status code requesting authentication with one of the listed authentication methods.
 - **NSURLAuthenticationMethodBasic**
 - **NSURLAuthenticationMethodNTLM**
 - **NSURLAuthenticationMethodClientCertificate**
- The challenge handler can only use the enrollment credentials of the user when attempting to authenticate with a Web site. If a Web site requires a domain to log in, for example **ACME\jdoe**, and users enrolled with a basic user name, like **jdoe**, then the authentication fails.

Content Repository Behavior

Content repositories use the saved enrollment credentials (which are encrypted and shared with all SSO apps). If the content repository requires a different password, the connecting app prompts the user for the password at the time of accessing the repository.

Sample Code

This example illustrates how to handle a challenge from a network call made through **NSURLConnection**.

Note: This example is generic, so expand upon it in your application to handle errors and fallback scenarios.

```
- (void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:
(NSURLAuthenticationChallenge *)challenge{
    NSError*error;

    if([[AWController sharedInstance]
    canHandleProtectionSpace:challenge.protectionSpace
    withError:&error]){

        if([[AWController sharedInstance]
        handleChallenge:challenge]){

            NSLog(@"Challenge handled successfully");

        }else{

            NSLog(@"Challenge could not be handled");

        }

    }else{

        //SDK does not have the means to handle this
        authentication. Add your own fallback and SSL logic
        here.

    }

}
```

RSA Adaptive Authentication

AirWatch integrates with RSA's Adaptive Authentication (RSA AA) to further secure communication through to the AirWatch Tunnel.

This topic outlines how to implement the logic for app tunnel traffic through an AirWatch Tunnel that is protected by RSA AA.

Refer to the **VMware AirWatch Tunnel Admin Guide**, available on the Resources Portal (<https://resources.airwatch.com>) for more information on setting up this integration in your infrastructure.

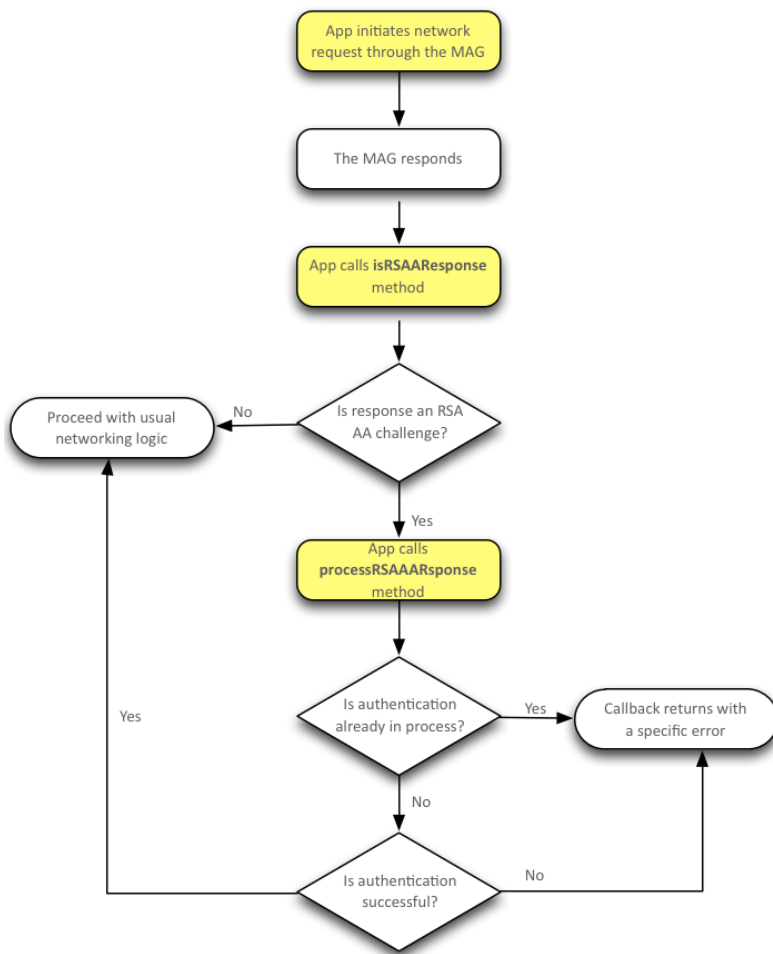
Implementation

The relevant methods for RSA AA are located in an AWController category. Start by importing the AWController_RSAAA.h file and use the two listed methods in your networking logic to authenticate with RSA AA through the AirWatch Tunnel.

| Method | Description |
|----------------------|---|
| isRSAAAResponse | <p>The AirWatch Tunnel checks that devices authenticated with RSA AA before it lets your application traffic tunnel through the AirWath Tunnel. the authentication of devices with RSA AA. If</p> <p>If a device is not authenticated, then your app receives a special response when making your network call.</p> <p>When you receive your networking response, pass the response into the isRSAAAResponse method provided by the AirWatch SDK to determine if the error is an RSA AA challenge.</p> |
| processRSAAAResponse | <p>If the isRSAAAResponse method returns YES, pass that response into the processRSAAAResponse method provided by the AirWatch SDK.</p> <p>Once the system calls the process method, the SDK presents a view controller that prompts the user to answer a question.</p> <p>The system calls the completion block for the processRSAAAResponse method when the listed items are true.</p> <ul style="list-style-type: none"> • The user answers the question successfully. • The user cancels the authentication process. • Another request invoked the authentication process already and is in progress. |

RSA Adaptive Authentication – Developer Logic Flow

The flow chart outlines how the methods process the networking logic to authenticate RSA AA through the AirWatch Tunnel.



Offline Access

The AirWatch SDK provides a way to allow access to the application when the device is offline and not communicating with the mobile network. It also allows access to AirWatch applications that use the SSO feature while the device is offline.

Offline Behavior

The SDK automatically parses the SDK profile and honors the offline access policy once **AWController** is started. If you enable offline access and an end-user exceeds the time allowed offline, then the SDK automatically presents a blocker view to prevent access into the application. The system calls the **AWSDKDelegate's** `lock` method so your application can act locally.

Compliance or Compromised Protection

The AirWatch SDK provides helper class **AWCompliance** to disallow the application on compromised devices. The **AWCompliance** is a service that runs in the background to check if the device is compromised. If it identifies the

application is running on a jailbroken device, it sends the notifications configured in the SDK Profile under the **Compliance** settings.

Implementation in Xcode

The following code shows how to initialize the Compliance service to start monitoring the device for a jailbroken status.

Retrieve Compliance Settings From Cached Settings

```
// Start the AWCompliance Service and register for notifications
NSError *error = nil;
if (![AWCompliance startService:&error])
    NSLog(@"An error occurred: %@", error);
else {
    NSLog(@"Successfully started AWCompliance Service");
    // Add a notification for the DisplayMessage SDK Action
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector
(handleDisplayMessageNotif:)
    name:AWComplianceDisplayMessageNotificationobject:nil];
    // Add other notification handlers here if desired.
}
- (void)handleDisplayMessageNotif:(NSNotification*)notif
{
    id action = [[notif userInfo] objectForKey:AWComplianceUserInfoObjectKey];
    if ([action isKindOfClass:[AWAction class]])
    {
        NSLog(@"%@", [(AWAction*)action value]);
    } else {
        NSLog(@"There is no action stored in the userinfo object");
    }
}
```

Manually Checking for the Compromised Status of the Device

You can check the Compromised Status of the device directly in your application, whether the device is online or offline. Your application can use only this function if the device has run a Beacon call successfully at least once in the past.

Run the Beacon

Run the Beacon in your application to make sure you can use the Compromised Status check functionality. If this function is invoked without having run a Beacon in the past, it returns a 'not available' status code.

Checking Device Status

The sample code shows how to check the status of the device.

```
if ([[AWCompliance sharedInstance] jailBrokenStatus] == AWDeviceJailBroken) {
    NSLog(@"Device is jailbroken. Take necessary actions");
}
```

There are three possible statuses that are returned from the method.

- **AWDeviceJailBroken:** The device is identified as compromised.
- **AWDeviceNotJailBroken:** The device is not identified as compromised.
- **AWJailBrokenStatusNotAvailable:** The status is not available because the system has not run a Beacon call successfully on the device since last enrollment.

Proxy, App Tunneling

The purpose of app tunneling is to redirect the traffic in your application through a specific gateway. To access internal resources in your organization, use the AirWatch Tunnel as the proxy. You can also redirect all traffic using an F5 or a standard proxy to add more controls to your application.

Known Limitations and Other Considerations

Due to platform and other technical limitations, only network traffic made from certain network classes can tunnel. Consider the purpose of the listed classes and review their known limitations.

- **NSURLConnection** – Calls made with **NSURLConnection** tunnel. There is one exception to this behavior. If calls are made synchronously on the main thread, they do not tunnel.
- **NSURLSession** – Calls made using **NSURLSession** tunnel only on iOS 8+ devices and depending on the configuration used. Default and ephemeral configuration types tunnel. However, background configuration types do not tunnel.
- **CFNetwork** – Most calls made using **CFNetwork** tunnel. However, **CFSocketStream** do not tunnel.
- **URLs that contain .local** – Requests with URLs containing .local do not tunnel. Various Apple services on the device use this .local string pattern. The SDK does not tunnel these requests through the AirWatch Tunnel to avoid interfering with these services.

Implementation in Xcode

You do not need extra code to use the app tunnel. You only need the SDK initialization code in the section titled [Initialize the SDK](#).

To activate app tunneling, make sure that this app has an SDK profile assigned to it in the AirWatch Console and that the profile has **App Tunneling** enabled with a proper proxy configuration. When you call start in **AWController**, it reads the SDK profile assigned to your application. If needed, it also starts the traffic redirection service for the application.

After you receive the **initialCheckDoneWithError** callback from the **AWSDKDelegate**, check to see if the error object is nil or not.

Content Filtering

The Forcepoint content filtering component in the SDK is used for infrastructures that use a Forcepoint proxy or content filter. No code is needed for implementing this functionality other than calling start in **AWController** with a proper content filtering configuration defined in the SDK profile.

Geofencing

Geofence settings are configured within an SDK profile. To do so, create an SDK profile or edit an existing one. A tab labeled **Geofencing** is visible on the left side of the profile editor. After you select the **Enabled** check box, enter settings to customize a geofence. A profile containing the geofence settings are obtained through an install profile command.

Running Location Services

Geofencing only works on devices that have Location Services running. To turn on the location services, the device must be connected to either a cellular network or a Wi-Fi hotspot*, or the device must have integrated GPS capabilities†. If a device switched to "airplane mode" the location services are deactivated, and the Geofencing feature stops working.

Device Connectivity Options

| Device | Wi-Fi* | Cellular Network | Built-In GPS† |
|--|--------|------------------|---------------|
| iPhone | ✓ | ✓ | ✓ |
| iPad Wi-Fi + 3G/4G | ✓ | ✓ | ✓ |
| iPad Wi-Fi | ✓ | | |
| iPod Touch | ✓ | | |
| * Wi-Fi hotspot must be in a location server database in order for the location services to retrieve a location | | | |
| † A device with integrated GPS uses "Assisted GPS" to register its location. Assisted GPS is a combination of GPS data collected from satellites, plus additional information obtained from servers and networks. It may take up to 40 seconds for a device's GPS to register a location; during this time, the device relies solely on data from its cellular network or Wi-Fi hotspot. | | | |

Restrictions, DLP

In this section of the SDK profile, you can identify what type of restriction rules you implement in your application.

Initialize the AWRestrictions Class

To monitor the MDM enrollment and allow offline mode restrictions, the application must initialize the **AWRestrictions** class.

The example code starts the monitoring of the defined SDK restrictions. The system can initialize it multiple times with no adverse effect.

```

NSError *error = nil;
[AWRestrictions startService:&error];

if (error)
{
    NSLog(@"AWRestrictions startService error: %@", error);
}

```

Check Device Enrollment

Use the SDK to retrieve the enrollment status of the device. The snippet of code shows how to perform that check.

```

AWDeviceStatusConfiguration *configuration = [[AWDeviceStatusConfiguration
alloc] initWithHostName:nil endpointPath:nil deviceStatusAction:nil];

// Create the device status controller.
AWDeviceStatusController *statusController = [[AWDeviceStatusController
alloc] initWithConfiguration:configuration];

// Query AirWatch to determine if the device is enrolled.
[statusController queryDeviceEnrollmentStatus:^(BOOL enrolled, NSError
*error)
{
    // Log the result of the enrollment check.
    NSLog(@"This device %@ enrolled.", (enrolled == YES) ? @"is" : @"is not");
    // Clean up.
    [configuration release];
    [statusController release];
}];

```

Preventing Cut and Copy Actions

The AirWatch SDK profile allows you to prevent the application from allowing users to copy and cut application data.

Use a custom class that inherits the controls to prevent users from copying or cutting your content. Override the `canPerformAction` method. Returning a **NO** prevents the action and returning a **YES** allows it.

```

- (BOOL)canPerformAction:(SEL)action withSender:(id)sender
{
    if (action == @selector(copy:) || action == @selector(paste:) || action ==
@selector(cut:))
    {
        return !AWRestrictionPayload.preventCopyAndPaste;
    }
    return [super canPerformAction:action withSender:sender];
}

```

Use DLP, VMware Browser, AirWatch Boxer, or AirWatch Inbox in SDK-Built Apps

Configure your applications built with the AirWatch SDK to open in the VMware Browser and to compose emails in AirWatch Boxer or AirWatch Inbox. This feature enables end users to use alternative systems other than Safari and the Mail app. To develop this feature, create a bundle in your iOS application and configure AirWatch to enforce the behaviors in the bundle.

Configure both systems, the browser and email systems, for this feature to work. Perform the procedures in the listed order.

1. Initial Set Up of the Bundle and PLIST
2. Enable Links for Browser
3. Enable Links for Inbox
4. Contain Data to Browser and Inbox

Initial Set Up of the Bundle and PLIST

Perform these steps before you enable any links. Use this bundle and PLIST for both HTTP/HTTPS links and MAILTO links.

1. Create a bundle named `AWSDKDefaults`.
2. Create a PLIST named `AWSDKDefaultSettings.plist` and put it in the `AWSDKDefaults` bundle.

Enable Links for Browser

To enable the application to open HTTP / HTTPS links in the VMware Browser, enable a few dictionary and PLIST flags.

1. Work in the `AWSDKDefaults` bundle.
2. Create a dictionary named `AWURLSchemeConfiguration` and put it in the `AWSDKDefaultSettings.plist`.
3. Inside the `AWURLSchemeConfiguration` dictionary, create a new Boolean entry with the key name **enabled** and set the Boolean value to **Yes**.

If you set the Boolean value to **No**, then the HTTP and HTTPS links open in Safari. If set to **Yes**, then your SDK app looks to see if you enabled data loss prevention in the SDK profile.

- DLP Enabled – The app opens in VMware Browser.
- DLP Disabled – The app opens in Safari.

Enable Links for Boxer or Inbox

To enable the application to open MAILTO links in AirWatch Boxer or AirWatch Inbox, enable a few dictionary and PLIST flags.

1. Work in the `AWSDKDefaults` bundle.
2. Create a dictionary named `AWMailtoSchemeConfiguration` and put it in the `AWSDKDefaultSettings.plist`.
3. Configure the `AWMailtoSchemeConfiguration` dictionary, create a new Boolean entry with the key name as **enabled** and set the Boolean value to **Yes**.

If you set the Boolean value as **No**, then MAILTO links open in the native mail. If set to **Yes**, then your SDK app looks to see if you enabled data loss prevention in the SDK profile.

- DLP Enabled – The app opens in AirWatch Boxer or AirWatch Inbox.
- DLP Disabled – The app opens in the iOS Mail app.

Contain Data to Browser and Inbox

Use the data loss prevention, DLP, settings in the AirWatch default SDK profile to enforce the application to use VMware Browser and AirWatch Boxer or AirWatch Inbox.

If you do not enable data loss prevention in the SDK policy, the application opens links in Safari and composes email in the iOS Mail app.

1. Navigate to **Groups & Settings > All Settings > Apps > Settings and Policies > Security Policies**.
2. Select **Enabled** for **Data Loss Prevention**.
3. Disable the **Enable Composing Email** check box for the MAILTO links. If you do not disable this option, the application opens from the Mail app and not from Inbox.

Limitation With MFMailComposeViewController

If you use the `MFMailComposeViewController` scheme in your MessageUI framework, this functionality is not supported. The system cannot specify how end users access your application when it is an attachment in an email. End-users access the application with the Mail app and not Inbox.

Branding

Branding can change the look of the application with minimal development, and it can delegate several user interface properties to the configuration settings in the SDK profiles.

Set up different branding configurations based on the AirWatch organization groups. Customize logos and background images within your application remotely. Many organizations find branding by organization groups useful for updating the branding elements inline for time-sensitive events or marketing initiatives.

Implementation in Xcode

The sample code shows how to retrieve branding settings from the locally cached settings.

Retrieve Branding Settings From the Local Cached Settings

```
// Get an instance of the Command Manager
AWCommandManager *commandManager = [AWCommandManager sharedManager];
// Get a pointer to the SDK Profile stored locally.
AWProfile *profile = [commandManager sdkProfile];
// profile may be nil if an AWProfile has not been received from the Console
if (profile == nil)
{
    NSLog(@"There is no SDK Profile currently installed");
}
```

```

} else
{
    AWBrandingPayload *brandingPayload = [profile brandingPayload];
    if (brandingPayload != nil)
    {
        // Get the navigation bar color.
        UIColor *navigationBarColor = [brandingPayload toolbarColor];

        // Get the primary text color.
        UIColor *primaryTextColor = [brandingPayload primaryTextColor];

        // Store the navigation bar color in the defaults.
        [[NSUserDefaults standardUserDefaults] setObject:navigationBarColor forKey:kAWNavigationBarColor];

        // Store the text label color in the defaults.
        [[NSUserDefaults standardUserDefaults] setObject:primaryTextColor forKey:kAWPrimaryTextColor];

        // Synchronize the defaults.
        [[NSUserDefaults standardUserDefaults] synchronize];

        // Coordinate the branding settings with UIAppearance.
        [[UINavigationBar appearance] setTintColor:navigationBarColor];
        [[UILabel appearance] setColor:primaryTextColor];
    }
}

```

Dimensions for Images on iOS Devices

It is difficult to find a single image that displays perfectly on every mobile device. However, certain ratios and dimensions for the images displayed on iOS devices can work for most displays.

Find out the ratios that often work best for branding and icons when you upload images for iOS devices.

Max Constraints

- iPhone – Not exceeding a ratio of 2.88 width over height
- iPad – Not exceeding a ratio of 4.39 width over height

Logo Ratios

- iPhone – 1.35 width over height
- iPad – 1.26 width over height

Other Considerations

If the image exceeds a height of 111 points (iPhone) or 175 points (iPad), then the image scales down while maintaining the aspect ratio. Points, which are specific to Apple iOS, differ from pixels. The conversion from points to pixel depends specifically on the device. Examples include the following ratios:

- iPhone 4 – 1 point = 1 pixel
- Retina iPads – 1 point = 2 pixels
- iPhone 6 Plus – 1 point = 3 pixels

Logging

The Logging module of the AirWatch SDK allows developers to instrument their applications to discover bugs or any issues when the application is deployed to users.

Code the Level

You must code the logging level and you must set this option in the AirWatch Console. This configuration ensures that your network is not burdened with unwanted logging activity.

Log Types

AirWatch SDK Logging provides the following types of logs:

| Setting | Description |
|-------------------------|--|
| Crash Logs | This type of log captures data from an application the next time the application runs after it crashes. These logs are automatically collected and uploaded to the AirWatch Console without the need for extra code in the SDK application. |
| Application Logs | This type of log captures information about an application. You set the log level in the default SDK profiles section, Groups & Settings > All Settings > Apps > Settings and Policies > Settings > Logging . You must add code into the application to upload these logs to the AirWatch Console. |
| Device Logs | This type of log captures information on devices using NSLog statements from all the applications running on the device. They provide some context of what else is happening on the device when an issue arises. You must add code into the application to upload them to the AirWatch Console. |

Implementation in Xcode for Application Logs

Code logic for **sendApplicationLogsWithCompletion** inside a method that the application-user calls manually. For example, create a button-click method with the logic for **sendApplicationLogsWithCompletion**.

```
AWLogVerbose(@"Send the application logs.");

[[AWLog sharedInstance] sendApplicationLogsWithCompletion:^(BOOL success, NSError *error) {
    AWLogInfo(@"Send application logs %@", success ? @"succeeded" : @"failed");
}
```

```
if(false == success){
  NSLog(@"error: %@", error.localizedDescription);
}
}];
```

There are four macros that follow the same format as the well-known `NSLOG(...)` macro in iOS. Each one of them records an event for the four logging levels.

- **Verbose** – `AWLogVerbose`
- **Info** – `AWLogInfo`
- **Warning** – `AWLogWarning`
- **Error** – `AWLogError`

Implementation in Xcode for Crash Log Reporting

```
/*
*****IMPORTANT*****

To anyone testing this in the xcode debugger. The debugger catches the signal before AWCrashTracker can. This
means the tracker can't actually write the crash log to disk.

Stop the debugger run the sandbox app from the device, crash the app, and then run the debugger again. You
will now be able to step through the crash reporting code.

*****

*/
-(void)initializeCrashLogReporting
{
  AWCrashLogReporter *crashLogReporter = [[AWCrashLogReporter alloc] init];
  // Report previous crash log to console
  if ([crashLogReporter hasPendingCrashReport]) {
    NSLog(@"Application crashed previously !!");

    NSLog(@"Reporting crash to console started ...");

    [crashLogReporter reportCrashLogToConsole];
  }
  NSLog(@"Application crashed previously !!");
  NSError *error = nil;
```

```
// Instantiate the crash log reporter for this session.
if(![crashLogReporter startUp:&error])

    {NSLog(@"Crash tracking instantiation failed !!!");

    NSLog(@"Error startingup crash log reporter : %@", error.localizedDescription);
    }
}
```

Access Log Files for Apps That Use the SDK Framework

Use the App Logs and the View Logs pages to access log information for applications that are built with the AirWatch SDK or use the SDK framework for functionality. These applications have default configurations in the Settings and Policies section of the console or have custom profiles that overwrite default configurations.

Access SDK and Wrapped App Logs by Log File

Access SDK application logs from the App Logs page.

1. Navigate to **Apps & Books > Applications > Analytics > App Logs**.
2. Download or delete logs using the actions menu.

Access Logs by the View Logs Page

Use the View Logs feature from the actions menu to quickly access available log files pertaining to applications that use SDK functionality.

1. Navigate to **Apps & Books > Applications > List View** and select the **Internal** tab.
2. Select the application.
3. Select the **View Logs** option from the actions menu.

Analytics

Analytics track the important events that occur within your application. The system uses these metrics to analyze use patterns and to account for how people use your app.

Analytic Types

The AirWatch SDK for iOS offers the several types of analytics.

- Event Analytics – Records and reports information about events specific to your organization that you code into the application.
- Data Usage Analytics – Records and reports information about network traffic to track telecom statistics.

Developing Event Analytics

Developing analytics for your AirWatch SDK applications requires several steps.

1. Code the events into your SDK application.
2. Set up the DataSampler module and configure it to report analytics.
3. Enable Analytics in the AirWatch Console. Use this option to turn Analytics on and off so that you do not use excessive bandwidth when you do not need data collected.

You must perform both of these tasks or the application does not collect data and it does not transmit the data to the AirWatch Console.

Implementation of Event Analytics in Xcode

The snippet of code shows you how to retrieve the configuration for the analytics module from the AirWatch Console. See how to call to register an event when analytics are enabled. In your implementation, check the cached value of this setting and register an event.

```
/**
Configure the Analytics Module based on the SDK Settings configured on the AirWatch Console.

Invoke this method when starting the application and every time a new command is received.
Make sure the DataSampler has been initialized.
*/

-(void)configureAnalyticsBasedOnSDKSettings
{

// Get an instance of the Command Manager
AWCommandManager *commandManager = [AWCommandManager sharedManager];

// Get the Analytics Payload, in case there is an existing SDK profile available and it contains
analytics settings.
AWAnalyticsPayload *analyticsPayload = [[commandManager sdkProfile] analyticsPayload];

if (analyticsPayload) {
// Set the analytics payload to the value from the profile.
[[AWAnalytics mAnalytics] setEnabled:analyticsPayload.enabled];
} else {
// Set the default status for analytics (Whether enabled or not).
[[AWAnalytics mAnalytics] setEnabled:NO];
}

}
```

```
/**
Configure the analytics mode to enable to start capturing events.
Make sure the DataSampler has been initialized.
*/
-(void)enableAnalytics
{
// Set the default status for analytics (Whether enabled or not).
[[AWAnalytics mAnalytics] setEnabled:NO];
}
```

To invoke an event, use the **recordEvent** command of the **AWAnalytics** module. For example, to record that the user tapped a button for feature ABC, you can use the sample code.

```
[AWAnalytics mAnalytics] recordEvent:AWAnalyticsCustomEvent
eventName:@"Tapped Button"
eventValue:@"Feature ABC"
valueType:AWAnalyticsValueString];
```

Developing Data Usage Analytics

The AirWatch SDK for iOS allows your application to monitor the amount of network traffic used specifically by your application. It also reports network traffic used to the AirWatch Console for telecom tracking purposes. The SDK hooks into the common iOS networking classes to monitor the amount of data transferring in an application.

How to Enable Data Tracking in Your SDK Application

Enable the data tracking module by embedding a PLIST into your application project.

1. Create a PLIST named **AWSDKDefaultSettings.plist**, add it to your application project, and copy it to your bundle resources.
2. Define the **AWDataUsageEnabled** boolean flag in the PLIST.
3. Define the **AWDataUsageConfiguration** dictionary with **SyncInterval** and **Network** in the PLIST.

| Key | Type | Value |
|----------------------------|------------|---------------------|
| ▼ Root | Dictionary | (2 items) |
| ▼ AWDataUsageConfiguration | Dictionary | (2 items) |
| SyncInterval | String | kSyncPerDayBasis |
| Network | String | kNetworkMonitorWWAN |
| AWDataUsageEnabled | Boolean | YES |

AWSDKDefaultSettings.plist

The following keys are in the PLIST.

AWDataUsageConfiguration

- **SyncInterval** – Interval which defines how often the samples of data transmit to the AirWatch server.
 - **kSyncOnResume**
 - **kSyncPerDayBasis**
If data use tracking is enabled but SyncInterval is not defined, this key is the default value.
 - **kSyncEveryOneHour**
 - **kSyncEveryTwoHours**
 - **kSyncEveryFourHours**
 - **kSyncEveryEightHours**
- **Network** – Type of data to collect.
 - **kNetworkMonitorWWAN** – Track only cellular data.
Default if network is not defined and data tracking is enabled.
 - **kNetworkMonitorWIFI** – Track only WIFI data.
 - **kNetworkMonitorBoth** – Track both WIFI and cellular data.
- **AWDataUsageEnabled** – Enable tracking.

Transmit to the AirWatch Server

No additional code is required for transmitting data usage analytics to the AirWatch server. The SDK checks on every app launch if the interval for transmitting to AirWatch has been reached and handles the transmission accordingly.

However, event analytics requires implementation of the `DataSampler` class to transmit, as mentioned in the [Developing Event Analytics](#) section.

Supported Networking Classes

SDK data usage tracking is supported for the listed iOS network classes.

- **NSURLSession**
With the exception that traffic made using `dataTaskWithRequest` and `dataTaskWithURL` is not monitored on iOS 7 devices.
- **NSURLConnection**
- **AVPlayer**

Custom Settings

The AirWatch SDK allows you to define your own custom settings for your application using an SDK Profile. You can paste raw text in this section, and the SDK makes this content available inside the application using the **AWCustomPayload** object.

You can define an XML, JSON, or key-value pairs for your settings and parse the raw text in the application once it is received. However, you can use other text formats like csv or plain text.

Implementation in Xcode

The sample shows how to retrieve custom settings from the local cached settings.

```
// Get an instance of the Command Manager
AWCommandManager *commandManager = [AWCommandManager sharedManager];
// Get a pointer to the SDK Profile stored locally.
AWProfile *profile = [commandManager sdkProfile];
// profile may be nil if an AWProfile has not been received from the console
if (profile == nil)
{
    NSLog(@"There is no SDK Profile currently installed");
} else
{
    AWCustomePayload *customPayload = [profile customPayload];
    if (customPayload != nil)
    {
        NSString *customSettings = [customPayload settings];
        // Do custom processing on your settings.
        NSLog(@"%@", customSettings);
    }
}
```

Chapter 4:

Content Management

- Overview 51
- External Repositories 51
- AirWatch Content 53
- Personal Content 54
- Download Content 61

Overview

The Content Management framework of the SDK allows applications to integrate with the same engine that provides the back-end functionality for the VMware Content Locker. AirWatch supports two ways to manage content. One method stores the content in the AirWatch database. The other other stores content on external repositories like SharePoint or a file system.

To understand how the content is managed for the AirWatch native content, or for repositories, we define the listed terms.

| Term | Description |
|------------------|---|
| Category | A category is a way to grouping content in an AirWatch native content storage. A category can have subcategories and each category can have content associated to it. A category behaves logically like a folder, but only applies to the AirWatch storage. |
| Repository | A repository refers to an external system like SharePoint or a File System that has been associated to the Content Framework in the AirWatch Console. Repositories represent external sources of content. |
| Folder | Folders only apply to Repositories and correspond to the hierarchy defined on each external repository. |
| File | The actual pieces of content that you can find in a Folder (for an external repository) or in a Category (for AirWatch content). |
| Personal Content | Content that users upload and the system stores in the AirWatch Console. It is only visible and available to a particular user. |

External Repositories

The following are samples of the different functions available in the iOS SDK for Content Management. These next functions refer to external repositories like SharePoint or a File System that have been configured in the AirWatch Console to be the source of content.

List All Repositories

To list all the available repositories available in the AirWatch Console, use the `fetchRepositoriesWithCompletion` method as follows

```
[contentController fetchRepositoriesWithCompletion:^(BOOL success, NSError
*error, NSArray *repositories)
{
    NSLog(@"Fetch Repositories Success: %@", success ? @"YES" : @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
        return;
    }
    for (AWContentRepository *repo in repositories) {
        NSLog(@"%@", [repo name]);
    }
}
```

```
}};
```

List All Folders in a Repository

Once you have identified all the repositories, you can list all the Folders in a repository.

```
NSInteger repositoryID = -1; // Repository ID should be obtained from
fetchRepositoriesWithCompletion
[contentController fetchFoldersForRepository:repositoryID forceSync:YES
withCompletion:^(BOOL success, NSError *error, NSArray *folders)
{
    NSLog(@"Fetch Folders Success: %@", success ? @"YES" : @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
    } else
    {
        for (AWContentFolder *folder in folders) {
            NSLog(@"%@", [folder name]);
        }
    }
}
}};
```

List All Content in a Folder

The logical next step in this sequence is to list all the contents of a folder. The following sample code explains how to achieve that:

```
NSInteger folderID = -1; // Folder ID can be obtained from
fetchFoldersForRepository:forceSync:withCompletion
NSInteger repositoryID = -1; // This should be the same ID used to fetch the
folders
[contentController fetchAvailableContentForFolder:folderID
inRepository:repositoryID
withCompletion:^(BOOL success,
    NSError *error,
    NSArray *folderEntities,
    NSArray *contentEntities)
{
    NSLog(@"Fetch Content for Folder Success: %@", success ? @"YES" : @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
    } else
    {
        NSMutableArray *temp = [NSMutableArray arrayWithArray:folderEntities];
    }
}
```

```

        [temp addObjectFromArray:contentEntities];
        for (AWContentEntity *content in temp) {
            NSLog(@"%@", [content name]);
        }
    }
}];

```

AirWatch Content

A Category is like a folder that is used to describe content stored in the AirWatch database. The following functions are used to query the content in AirWatch.

Use an initialized AWErollmentAccount object as the variable 'account.' Use an initialized AWContentController object as the variable 'contentController.'

List All Categories

The sample code shows how to list the categories in AirWatch content storage. List the sub categories by setting the parentCategoryIdentifier.

```

// CategoryIdentifier 0 is the root category.
// Using the root category you will be able to obtain
all categories
[contentController fetchCategories:account
parentCategoryIdentifier:0
all:YES
withCompletion:^(BOOL success,
NSError *error,
NSArray *receivedCategories)
{
    NSLog(@"Fetch Folders Success: %@", success ? @"YES"
: @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
    } else
    {
        for (AWContentCategory *category in
receivedCategories)
        {
            NSLog(@"%@", [category name]);
        }
    }
}

```

```
    }
  }];
}
```

List all Content in a Category

Given a category, you can list all the content that is assigned to that category.

```
NSInteger categoryID = -1;
// categoryID can be obtained by using
the fetchCategories:parentCategoryId:all:withCompletion
method.
AWEnrollmentAccount *account = [(AppDelegate*)
[[UIApplication sharedApplication] delegate] account];
[contentController fetchAllAvailableContent:account
parentCategoryId:categoryID
withCompletion:^(BOOL success, NSError *error, NSArray
*availableContent)
{
    NSLog(@"Fetch Content for Category Success: %@",
success ? @"YES" : @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
    } else
    {
        for (AWContentEntity *content in availableContent)
        {
            NSLog(@"%@", [content name]);
        }
    }
}];
```

Personal Content

An application using the Content Management Framework of the SDK can allow a user to upload files to the Personal Content area in the AirWatch Console. This content is only for the particular user and it is not visible to anyone else.

If personal content is enabled, the system creates a personal content repository for each individual user the first time when users access content with the listed methods.

- Self Service Portal
- Content Locker

- SDK calls

The system creates personal content repositories at the customer organization group (OG), which is the root OG for each customer. If an organization has defined a structure of OGs with users associated at different levels, you can move a user from one OG to another. This action does not affect the personal content repository for that particular user.

Unenrolling a device does not affect or remove the personal content repository because a user can access it from other devices or the Self-Service Portal. However, if you delete users from the console, the system removes their personal content repositories.

The sample code shows several functions that you can use to manage personal content.

Identifying the Personal Content Repository ID

The personal content repository for the current user is included in the list of all available repositories. The sample code shows how to identify what is the repository ID for the given user.

```
[contentController fetchRepositoriesWithCompletion:^(BOOL success, NSError
*error, NSArray *repositories)
{
    NSLog(@"Fetch Repositories Success: %@", success ? @"YES" : @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
        return;
    }

    // Use the isPersonal property to determine if the repository is
    // personal.
    // There will only be one, the current user's, personal repository.
    AWContentRepository *personalRepo = nil;
    for (AWContentRepository *repo in repositories)
    {
        if ([repo isPersonal])
        {
            personalRepo = repo;
        }
    }

    NSInteger personalRepoID = [personalRepo identifier];
}];
```

Create Personal Folder

The code shows how an app creates a folder in the personal content repository on behalf of a user. This action is how regular folders behave. You can also create a tree structure of folders and subfolders.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSString *folderName = @"TestFolder";
[contentController createPersonalFolder:folderName
inRepository:repoID
parentFolder:folderID
withCompletion:^(BOOL success, NSError *error, AWContentFolder *createdFolder)
{
    if (error)
    {
        NSLog(@"Failed to create folder, %@",error);
    }
    else {
        NSLog(@"Created Folder :: %@", [createdFolder name]);
    }
}];

```

Update Personal Folder

Use this function to rename a folder in the personal content repository.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSString *folderName = @"RenamedFolder";
[contentController renamePersonalFolder:folderID
inRepository:repoID
withName:folderName
withCompletion:^(BOOL success, NSError *error, AWContentFolder *updatedFolder)
{
    if (success)
    {
        NSLog(@"Updated Name %@", [updatedFolder name]);
    } else
    {
        NSLog(@"Failed to updat folder name: %@",error);
    }
}];

```

Move Personal Folder

This function shows how to move a sub folder under another subfolder.


```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
// A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSInteger folderID = -1;
NSInteger parentFolderID = 9999;

[contentController movePersonalFolder:folderID
inRepository:repoID
toFolder:parentFolderID
withCompletion:^(BOOL success, NSError *error, AWContentFolder *folder)
{
    if (success)
    {
        NSLog(@"Folder moved.");
        NSLog(@"ParentID after: %d",[folder parent]);
    } else
    {
        NSLog(@"Failed to move folder :: %@", error);
    }
}
];

```

Delete Personal Folder

You can remove a folder in the personal content repository with the example function. The system raises an error with the details of why the action failed.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
__block NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
[contentController deletePersonalFolder:folderID
inRepository:repoID
withCompletion:^(BOOL success, NSError *error)
{
    if (success)
    {
        NSLog(@"Deleted folder with id: %d", folderID);
    }
    else
    {
        NSLog(@"Failed to delete folder with id:%d :: %@", folderID, error);
    }
}
];

```

Upload Personal Content

After you create a folder for a user in the personal content repository, you can upload content. The example show how to upload files to a folder.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSString *filename = @"NewFile.txt";
// Pull data from the file you're uploading, this only used as an example.
NSData *contentData = [@"This is text" dataUsingEncoding:NSUTF8StringEncoding];
[contentController uploadPersonalContent:filename
contentData:contentData
toFolder:folderID
inRepository:repoID
withCompletion:^(BOOL success, NSError *error, AWContentEntity *content)
{
    if (success)
    {
        NSLog(@"Uploaded Content - %@", [content name]);
    } else
    {
        NSLog(@"Failed to upload content :: %@", error);
    }
}
];

```

Delete Personal Content

This example shows how to delete specific files or content from a personal content folder.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSInteger contentID = -1; // Content id can be obtained by using
fetchAvailableContentForFolder:inRepository:withCompletion:
[contentController deletePersonalContent:contentID
fromFolder:folderID
inRepository:repoID
withCompletion:^(BOOL success, NSError *error)
{
    if (success) {
        NSLog(@"Content was deleted successfully.");
    } else
    {
        NSLog(@"Failed to delete content. %@", error);
    }
}
];

```

Move Personal Content

Move files in personal content from one folder to another.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger parentID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSInteger contentID = -1; // Content id can be obtained by using
fetchAvailableContentForFolder:inRepository:withCompletion:
[contentController movePersonalContent:contentID
inRepository:repoID
toFolder:parentID
withCompletion:^(BOOL success, NSError *error, AWContentEntity *content)
{
if (success)
{
    NSLog(@"Moved Content Successfully. New Folder ID: %d", [content
folderIdentifier]);
} else
{
    NSLog(@"Failed to Move Content %@", error);
}
}
}];

```

Rename Personal Content

Use the listed function to rename an existing file in the personal content repository.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger contentID = -1; // Content id can be obtained by using
fetchAvailableContentForFolder:inRepository:withCompletion:
NSString *newName = @"NewName.txt";
[contentController renamePersonalContent:contentID
inRepository:repoID
name:newName
withCompletion:^(BOOL success, NSError *error, AWContentEntity *content)
{
if (success)
{
    NSLog(@"ReNamed Content Successfully. New name: %@", [content name]);
} else
{
    NSLog(@"Failed to Move Content %@", error);
}
}
}];

```

Upload an Updated Version of Personal Content

If a user wants to upload a new version of an existing file that already resides in the personal content repository, use the example function.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSString *filename = @"NewFile.txt";
// Pull data from the file you're uploading, this only used as an example.
NSInteger contentID = -1; // Content id can be obtained by using
fetchAvailableContentForFolder:inRepository:withCompletion:
NSData *contentData = [@"This is text" dataUsingEncoding:NSUTF8StringEncoding];
[_contentController updatePersonalContent:contentID
contentData:contentData
toFolder:folderID
inRepository:repoID
withCompletion:^(BOOL success, NSError *error, AWContentEntity *content)
{
    if (success)
    {
        NSLog(@"Uploaded Content Ver. Successfully.");
    } else
    {
        NSLog(@"Failed to Upload Content Ver. %@", error);
    }
}];

```

Get Permission of Personal Content

Use the function to list all the permissions that are currently assigned to a file in the personal content repository.

```

NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
[_contentController fetchAvailableContentForFolder:folderID
inRepository:repoID
withCompletion:^(BOOL success,
NSError *error,
NSArray *folderEntities,
NSArray *contentEntities)
{
    NSLog(@"Printing permissions of content");
    for (AWContentEntity *e in contentEntities)
    {
        NSLog(@"- %@", [e name]);
        NSLog(@" R: %@", [e hasReadPermission]? @"YES" : @"NO");
        NSLog(@" W: %@", [e hasWritePermission]? @"YES" : @"NO");
        NSLog(@" D: %@", [e hasDeletePermission]? @"YES" : @"NO");
        NSLog(@"SR: %@", [e hasShareReadPermission]? @"YES" : @"NO");
        NSLog(@"SW: %@", [e hasShareWritePermission]? @"YES" : @"NO");
    }
}

```

```
}};
```

Get a Revision List for Personal Content

The example depicts how to list all the revisions that users uploaded for a given piece of content in the personal content repository.

```
NSInteger repoID = -1; // A repository id can be obtained by using
fetchRepositoriesWithCompletion:
NSInteger folderID = -1; // A folder id can be obtained by using
fetchFoldersForRepository:forceSync:withCompletion:
NSInteger contentID = -1; // Content id can be obtained by using
fetchAvailableContentForFolder:inRepository:withCompletion:
[contentController revisionListForPersonalContent:contentID
inFolder:folderID
inRepository:repoID
withCompletion:^(BOOL success,
NSError *error,
NSArray *contentVersions)
{
    if (!success)
    {
        NSLog(@"Failed to get revision list. %@", error);
        return;
    }
    for (AWContentVersion *version in contentVersions)
    {
        NSLog(@"ID: %d -- Ver: %@", [version identifier], [version friendlyVersion]);
    }
}
}];
```

Download Content

To download content from a storage type, obtain an `AWContentEntity` instance, set a delegate, and invoke the asynchronous method. There are several delegates that are invoked when the system terminates the command, due to a successful completion of the task, or when an error occurs.

The sample code shows how to retrieve the content from a Folder in a repository. You can modify it to retrieve the content from a Category from AirWatch Content or from a Personal Content Repository.

```
// Delegate methods that should be implemented in order to download content.
- (void)contentController:(AWContentController *)contentController
failedToDownloadContent:(AWContentVersion *)contentVersion withError:(NSError
*)error;
```

```

- (void)contentController:(AWContentController *)contentController
didReceiveResponse:(NSURLResponse *)response forContentDownload:
(AWContentVersion*)contentVersion;
- (void)contentController:(AWContentController *)contentController
downloadedData:(NSData*)data forContent:(AWContentVersion *)contentVersion;
- (void)contentController:(AWContentController *)contentController
didFinishDownloading:(AWContentVersion*)contentVersion;

// List all Content in a Folder. This could be modified to list all the content
in a Category from the AirWatch Content, or could be from a Personal Content
Repository.
NSInteger folderID = -1; // Folder ID can be obtained from
fetchFoldersForRepository:forceSync:withCompletion
NSInteger repositoryID = -1; // This should be the same ID used to fetch the
folders
[contentController fetchAvailableContentForFolder:folderID
inRepository:repositoryID
withCompletion:^(BOOL success,
NSError *error,
NSArray *folderEntities,
NSArray *contentEntities)
{
    NSLog(@"Fetch Content for Folder Success: %@", success ? @"YES" : @"NO");
    if (error)
    {
        NSLog(@"Error: %@", error);
    } else
    {
        if ([contentEntities count])
        {
            // The following code shows where the content download is initiated.
            AWContentEntity *contentEntity = contentEntities[0];
            [_contentController setDelegate:self];
            [_contentController downloadContentVersion:[contentEntity version]];
        }
    }
}];

```

Chapter 5:


Certificate Provisioning

- Overview 64
- Create the Application Profile64
- Retrieve a Certificate From an Application Profile 64

Overview

Provisioning certificates to your app involves three steps.

- Configure the certificate authority (CA) and the CA template. AirWatch has numerous guides outlining how to configure various certificates. See the applicable guide for information on configuring CAs and CA templates in the AirWatch Console.
- Create the app profile in the AirWatch Console.
- Assign the app profile to the application in AirWatch prior to app deployment.

 For more information on provisioning certificates for the AirWatch SDK, see the following AirWatch Knowledge Base article: <https://support.air-watch.com/articles/93877157-Application-Certificate-Management-with-the-AirWatch-SDK>.

Create the Application Profile

You can create the necessary application profile after you configure the CA and certificate template settings in the AirWatch Console. This profile is deployed to the app just like an SDK profile.

1. Navigate to **Groups & Settings > All Settings > Apps > Settings And Policies > Profiles**.
2. Select **Add Profile** and then choose the **Application Profile** for iOS.
3. Fill out the General information, make sure to give the profile a name, and then choose **Credentials**.
4. Select **Defined Certificate Authority** and then choose the correct **Certificate Authority** and **Certificate Template** from the choices provided.
5. Select **Save**.

Assign the application profile to the application during the upload process outlined in the **Mobile Application Management (MAM) Guide**.

Retrieve a Certificate From an Application Profile

Retrieving certificates requires extra considerations because it involves more than handling a profile. The system stores most profile payloads locally. However, it does not store `AWCertificatePayload` locally. To retrieve the certificate, you must wait for notification that the system downloaded the certificate.

Important: For the SDK to check the server for a certificate, you must call the `loadCommands` method or the notification does not trigger.

The code shows how to retrieve a certificate payload so that your application can consume and use the certificate.

Register the notification observer at start up.

```
[[NSNotificationCenter defaultCenter]
 addObserver:self

        selector:@selector(handleUpdatedProfile:)

        name:AWNotificationCommandManagerInstalledNewProfile
        object:nil];
```

Implement the **handleUpdatedProfile** that will receive the notifications when a command is processed and can extract the certificate information.

```
- (void)handleUpdatedProfile:(NSNotification
*)notification
{
    // Get the profile that just got received in this call;
    it could be an Application Profile, or an SDK Profile.

    AWProfile *profile = (AWProfile *)notification.object;

    // IMPORTANT: If expecting an application profile with a
    certificate, you can ONLY obtain the certificate values
    from the notification object.

    // For security reasons the certificate does not get
    stored locally, like all the other settings in an SDK
    profile

    if (profile.certificatePayload){

        AWCertificatePayload *certificatePayload =
        profile.certificatePayload;

        if ([[certificatePayload certificateData] length] > 0 & &
        [[certificatePayload certificatePassword] length] > 0)
        {

            NSString *certificateName = [certificatePayload
            certificateName];

            NSData *certificateData = [certificatePayload
```

```
certificateData];

NSString *certificatePassword = [certificatePayload
certificatePassword];

// TO DO: Use the certificate here... If you don't
consume it here, it will not be available later in
the AWCommandManager.

}

}
```

Chapter 6:

Test Apps With the AirWatch SDK for iOS

| | |
|---|----|
| Overview | 68 |
| Testing Process | 68 |
| Deliver Profiles to SDK Applications | 68 |
| Use the Persisted Settings in the SDK | 69 |
| Considerations for Testing Certificates | 69 |

Overview

The AirWatch SDK allows you to configure apps dynamically using default settings and application profiles that you can define in the AirWatch Console and associate to your applications.

- **Default Settings for SDK Profiles** – Provides settings that you can potentially use across multiple AirWatch applications, for example branding schemes, or the type of authentication you want to use on all your applications.
- **Application Profiles** – Configures an individual application. For example, when you deploy a certificate for use by a particular application.

You can design your application to use as many or as few predefined settings in profiles, to create a flexible application based on your goals.

Test the over-the-air configuration of your app using application and SDK profiles outlined in the following sections.

Testing Process

It is important to test the integration of your application with the AirWatch SDK, including the delivery of profiles from the AirWatch Console to your application.

Initialize the SDK in your application to set communication with the AirWatch server. Then test the application with the listed process.

1. **Enroll your test device** – Enroll devices to the AirWatch Console to enable communication between them.
Due to Apple restrictions, you must test your application in a device and not in the emulator.
2. **Create the application record in the AirWatch Admin Console** – Create an empty application with the bundle ID of the testing-application to identify the application. Upload the empty application to the console. This step enables the console to send commands to the application with the record.
Enable the option **Application Uses AirWatch SDK** in the application record when you upload the application. If you do not enable this option, the SDK does not initialize correctly. However, you do not have to apply an SDK profile.
3. **Push the application to test devices** – Select **Save & Publish** to push the application to test devices with the app catalog. Use devices for testing that are AirWatch managed devices.
4. **Run your application in Xcode** – Run your application in Xcode. The console pushes the initialization data to the application when the application installs on test devices. After the application initializes, you can run the application as many times as you want to debug it.
5. **Force the delivery of profiles to your application** – If your application uses SDK profiles or application profiles, retrieve the profiles from the AirWatch server. See [Deliver Profiles to SDK Applications](#).

Deliver Profiles to SDK Applications

The AirWatch SDK allows developers to configure their applications dynamically using SDK and application profiles.

- **SDK Profiles** provide common settings for multiple applications. For example branding schemes, or the type of authentication you want to use on all your applications.
- **Application Profiles** deploy certificates for your SDK applications.

The AirWatch SDK allows your application to integrate with the AirWatch Console to retrieve the most current settings that apply to the application. You can update settings periodically when required without a single change to the source code of the application.

Every time you save an application profile or an SDK profile, a new command to install the profiles goes to the Command Queue. The install command works on all applications and devices that are associated with the saved profiles. From the testing perspective, change what you want to test in the profile. This change generates a new command that you can retrieve from the test code. Use the load commands method entitled `AWCommandManager`.

Use the Persisted Settings in the SDK

You can access all settings that the AirWatch Console retrieves from an SDK Profile or an application profile, except certificates, offline. You do not have to persist the profile settings in your application because the SDK persists them. Instead of publishing the application multiple times to generate new commands, you can test with the most recent settings retrieved from the console.

Considerations for Testing Certificates

Consider the relationship between application profiles and certificates, so that your testing is successful.

- Make sure that you have created an application profile and you have associated the certificate authority and the CA Template to the application profile.
- Every time an app retrieves a new application profile, the system creates a new certificate for your application. You can use a test certificate authority to handle the volume.
- The SDK persists and manages all profile settings. The certificates defined in the application profile are the exception. To consume a certificate, retrieve the certificate and store it in a safe place like the keychain.

Chapter 7:

SDK and Application Payload Classes

- Overview 71
- Retrieving Settings 71
- Payload Lists 71

Overview

This section lists all classes used to represent configuration settings from the SDK or application profiles. You can use them to access a specific setting that determines the behavior of your application.

Retrieving Settings

Use the example code to retrieve the settings for the different configuration payloads.

```
// Get instance of AWCommandManager
AWCommandManager *manager = [AWCommandManager sharedManager];
// Get the cached profile
AWProfile *sdkProfile = [manager sdkProfile];
if (sdkProfile)
{
    // Get payloads from the profile, for example the geofencePayload.
    AWGeofencePayload *geofencePayload = [sdkProfile geofencePayload];
}
```

You can also obtain payloads from the AW SDK framework documentation included in the SDK distribution package.

Payload Lists

The following lists outline the payloads supported by the AirWatch SDK for iOS.

SDK Profile Payloads

- **AWAnalyticsPayload** – Represents the analytics group of an SDK profile.
- **AWAuthenticationPayload** – Represents the authentication group of an SDK profile.
- **AWBrandingPayload** – Represents the branding group of an SDK profile.
- **AWCompliancePayload** – Represents the compliance group of an SDK profile.
- **AWCustomPayload** – Represents the custom group of an SDK profile.
- **AWGeofencePayload** – Represents the geofence group of an SDK profile.
- **AWLoggingPayload** – Represents the logging group of an SDK profile.
- **AWRestrictionsPayload** – Represents the restrictions group of an SDK profile (this group was formerly known as **Access Control** in the SDK Profile).

Application Profile Payloads

- **AWCertificatePayload** – Represents the credentials group of an application profile.

Chapter 8:

Integrate With Swift Applications

- Overview 73
- Integration Process 73
- Initialize the SDK 73

Overview

Swift is a programming language developed by Apple as an alternative to C-based languages. The language strives to make writing code faster and safer with many features including supporting inferred types, automatically managing memory, and the required initialization of variables before use.

Integration Process

The high-level steps for integration include the following processes:

1. Initialize the AirWatch SDK for iOS so that it can integrate with Swift.
2. Add source code for the AirWatch SDK features you want in the app.
See topics in this guide for available advanced app management features to code and add to the app.
3. Upload and push the app from the AirWatch Console.
See the **AirWatch Mobile Application Management (MAM) Guide** for instructions on deploying apps to devices using the AirWatch Console.

Initialize the SDK

To initialize the SDK, import the AirWatch SDK to your Xcode project and implement the required methods.

Note: Bridging Headers are not required with SDK 5.9.1.X+.

Import the AirWatch SDK

Review the frameworks in your Xcode project and import the AirWatch SDK.

1. Review the following libraries, resources, and entries. See [Xcode Components on page 10](#).
 - Check that the required libraries and bundle resources are added to your project.
 - Check that the Info.plist contains the required entries to register the callback scheme.
2. Set the **Enable Bitcode** option to **No** in your Xcode build settings.
3. Add the -ObjC flag in **Other Linker Flags** in your Xcode build settings.
4. Enter `import AWSDK` in your AppDelegate.swift to import the AirWatch SDK.

Implement Required Methods

Implement the methods from the AWSDKDelegate class into the AppDelegate.swift in Xcode.

1. Conform to the AWSDKDelegate.
2. Set the callback scheme and the delegate.

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[NSObject: AnyObject]?) -> Bool {
    AWController.clientInstance().callbackScheme="URLScheme";
    AWController.clientInstance().delegate = self;
    return true
}
```

3. Start the AWController.

```
func applicationDidBecomeActive(application: UIApplication) {
    AWController.clientInstance().start();
}
```

4. Implement remaining delegate methods.

```
func application(application: UIApplication, openURL url: NSURL, sourceApplication: String?,
annotation: AnyObject) -> Bool {
    return AWController.clientInstance().handleOpenURL(url, fromApplication:
    sourceApplication);
}
func initialCheckDoneWithError(error: NSError!){ }
func receivedProfiles(profiles: [AnyObject]!){ }
func unlock() {
}
func lock() {
}
func wipe()
}
func stopNetworkActivity() {
}
func stopNetworkActivity(networkActivityStatus: AWNetworkActivityStatus) {
}
func resumeNetworkActivity() {
}
```

Appendix:

Release Notes

This topic lists enhancements implemented for the SDK for iOS.

AirWatch SDK v5.9.3 January 2017

- Fixed an issue where the updateUserCredentials API was not properly updating credentials for SAML enrolled devices.
- Fixed an issue where tunneling intermittently failed in check-in / check-out scenarios.

AirWatch SDK v5.9.1 August 2016

- Added support for iOS 10.
- Added a patch for Tunnel Proxy Reachability Security.



For more information on the security patch for the tunnel proxy, see the following AirWatch Knowledge Base article: [AirWatch Tunnel Proxy \(Legacy Mobile Access Gateway Proxy\) reachability vulnerability](#).

- Fixed an issue where the proxy certificate was not being retrieved in certain scenarios.

AirWatch SDK v5.7.1 April 2016

- Fixed an issue with the SDK unlock delegate not being called after re-enrollment.
- Resolved inconsistent behavior with offline access.
- Fixed a race condition with the updateUserCredentialsWithCompletion method where the method fails when called inside the initialCheckDoneWithError delegate.
- Fixed a crash caused by a category conflict with UIColor+HexString.
- Removed the repeated printing of obsolete “AWSSOBroker” logs.
- Addressed other minor bug fixes.

AirWatch SDK v5.5 January 2016

- Added support for RSA Adaptive Authentication.
- Initialize the AirWatch SDK for iOS to integrate with applications built using Swift.

AirWatch SDK v5.4.1 August 2015

- Added support for data usage analytics to track telecom usage in SDK-developed applications.
- Added "Soft Unlock" capability for temporarily dismissing the SDK authentication UI while leaving the SDK container encrypted.
- Added support for developing apps for latest Apple iOS release.
- Added an API to update the SDK credentials when an AD password changes.

AirWatch SDK v5.3 June 2015

Internal release to test features and performance.

AirWatch SDK v5.2 March 2015

- Added support for 64-bit applications (arm64).
- Added functionality for client certificate authentication for network calls made with `NSURLConnection` and `NSURLSession` methods.
- Added a challenge handler utility to delegate the handling of Web authentication challenges to the SDK.
- Added the ability to integrate network connections made in SDK-developed applications with a Forcepoint content filter or proxy.

AirWatch SDK v5.0 March 2015

Enhancements

- Updated the user interface for passcode and authentication options.
- Added support for Single Sign On (SSO) when authenticating with TouchID and EyeVerify.
- Improved performance for background modes.

AirWatch SDK v4.3.1 January 2015

Enhancements

Used certificate caching for the AirWatch Tunnel certificate retrieval process. This enhancement improves app tunneling performance for apps built using the AirWatch SDK for iOS.

AirWatch SDK v4.3 November 2014

Enhancements

- Updated initialization by adding the caching of data for quicker subsequent starts.
- Updated App Tunneling functionality by improving data interception to help stabilize performance.
- Added device user interface features for authentication. Enhancements include automatically displaying the keyboard on the Passcode page and offering the option to remember user name and password values on the Authentication page.

AirWatch SDK v4.2 – September 2014

Enhancements

- Added support for SSO with Username/Password authentication method in v7.3+ environments.
- Added biometric authentication modes with EyeVerify and iOS 8 TouchID.

AirWatch SDK v4.1 – June 2014

Enhancements

- Refactored SDK to use ARC to prevent memory leakage
- Added support to **AWController** for authenticating with user name and password and a custom SDK profile
- Updated compromised detection functionality
- Added general bug fixes

AirWatch SDK v4.0 – February 2014

Enhancements

- Introduction of **AWController** class for consolidation of initialization methods and improve usability.
- Single Sign On (SSO) for applications using the AirWatch Agent or AirWatch Container as an anchor.
- Support for SSID filtering and controlling network access.

Accessing Other Documents

While reading this documentation you may encounter references to documents that are not included here.

The quickest and easiest way to find a particular document is to navigate to <https://support.air-watch.com/articles/103013947> and search for the document you need. Each release-specific document has a link to its PDF copy on AirWatch Resources.

Alternatively, you can navigate to AirWatch Resources (<https://resources.air-watch.com>) on myAirWatch and search. When searching for documentation on Resources, be sure to select your AirWatch version from the drop-down menu and to select the Documentation category on the left.