Base Integration Guide

Workspace ONE for Android

Android applications can be integrated with the Omnissa Workspace ONE® platform, by using its mobile software development kit. Complete the tasks below as a base for feature integration.

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

Table of Contents

Introduction

The tasks detailed below represent the basic steps in integrating your Android application with the Workspace ONE platform. The tasks you will complete depend on the required integration level of your application.

Integration at the Framework level is necessary if the application will make use of platform features such as authentication, single sign-on, data encryption, or networking.

To integrate at the **Client level**, do the following tasks:

1. Add the Client SDK.
2. Initialize the Client SDK.

To integrate at the **Framework level**, do the following tasks:

1. Add the Client SDK.
2. Add the Framework.
3. Initialize the Framework.

Note that you don't add Client SDK initialization if you are integrating at the Framework level.

Downloads

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

For additional information, please visit the Omnissa Legal Center.

License

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

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

That applies however you obtain or integrate the software.

Integration Guides

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

See other guides in the set for

- an overview of integration levels and the benefits of each.
- details of the integration preparation tasks, which must be done before the tasks in this document.

An overview that includes links to all the guides is available

- in Markdown format, in the repository that also holds the sample code: https://github.com/euc-releases/...IntegrationOverview.md

- in Portable Document Format (PDF), on the Omnissa website: https://developer.omnissa.com/...IntegrationOverview.pdf

Compatibility

Instructions in this document have been tested with the following software versions.

Software	Version
Workspace ONE SDK for Android	25.02.4
Workspace ONE management console	25.06
Android Studio integrated development environment	2025.1.3
Gradle plugin for Android	8.2.2
Kotlin language	2.2.0

Integration Paths Diagram

The following diagram shows the tasks involved in base integration and the order in which they can be completed. Integration Preparation is a prerequisite to base integration. Framework integration is a prerequisite to integrating any of the framework features, which are covered by other guides.

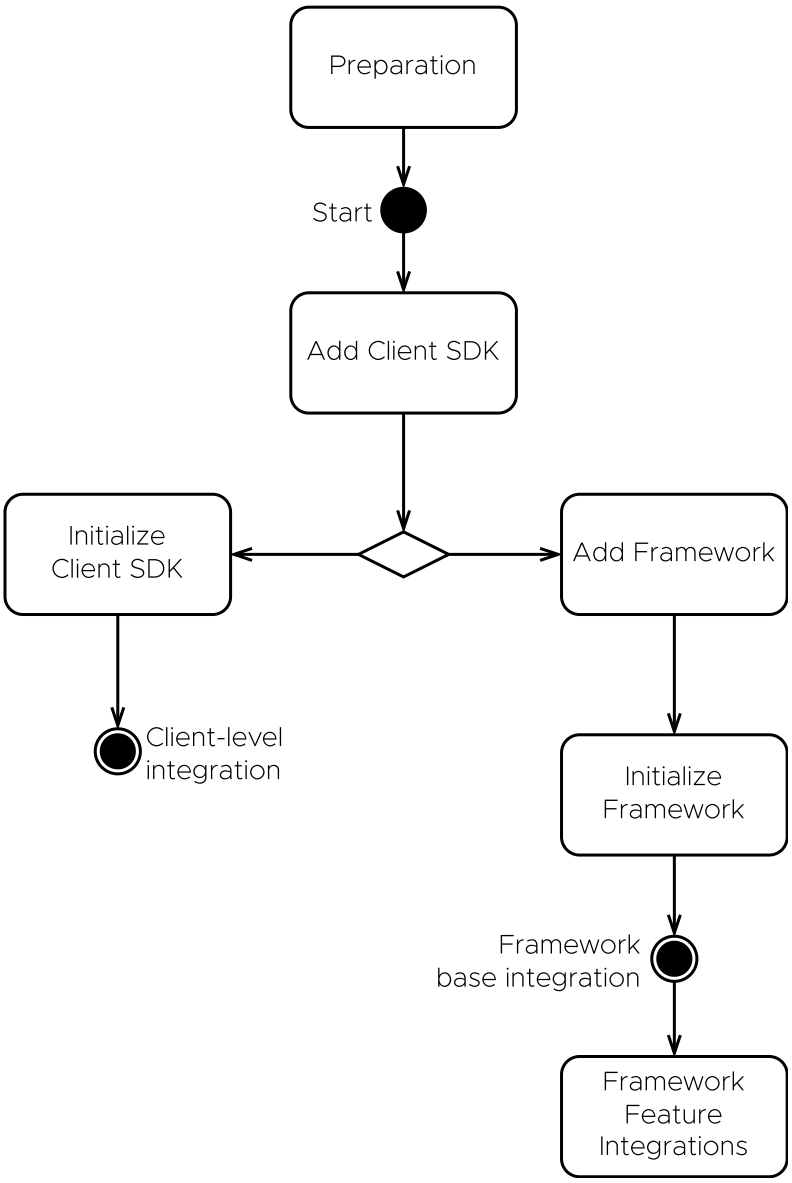


Diagram 1: Base Integration paths

Task: Add Client SDK

Adding the Client SDK is a Workspace ONE platform integration task for Android application developers. It applies to all levels of platform integration.

If you haven't installed your application via Workspace ONE at least once, then do so now. If you don't, the application under development won't work when installed via the Android Debug Bridge (adb). Instructions for installing via Workspace ONE can be found in the Integration Guides document set, in the Integration Preparation guide.

The first step will be to set up the build configuration and files. These instructions assume that your application has a typical project structure, as follows:

- *Project* files in the root directory.
- *Application* module in a sub-directory.
- Separate **`build.gradle`** files for the project and application.

Instructions

Proceed as follows.

Build Configuration and Files

First, update the build configuration and add the required library files.

1. **Update the Gradle Android plugin version, if necessary.**

   In the project build.gradle file, check the Android plugin version. This is typically near the top of the file, inside the `buildscript` block, in the `dependencies` sub-block. The top of the file might look like this:

   ```
   buildscript {
       ...
       repositories {
           ...
       }
       dependencies {
           classpath 'com.android.tools.build:gradle:8.2.2'
           ...
       }
   }
   ```

   In this example, the Gradle Android plugin version is 8.2.2

   Ensure that the Gradle version is at least 8.2

2. **Add the required packaging and compile options.**

   In the application build.gradle file, in the `android` block, add the Java version compatibility declarations shown in the following snippet.

   ```
   ...
   android {
       compileSdk 35

       // Following blocks are added.
        kotlin {
           jvmToolchain 17
       }
       packagingOptions {
          exclude 'META-INF/kotlinx-serialization-runtime.kotlin_module'
       }
       // End of added blocks.

       defaultConfig {
           targetSdk 35
           ...
       }
       buildTypes {
           ...
       }
   }
   ```

3. **Add the required libraries to the build.**

   Still in the application build.gradle file, in the `dependencies` block, add references to the required libraries. For example:

```
repositories {
    maven {
         url = uri("https://maven.pkg.github.com/euc-releases/Android-WorkspaceONE-SDK/")
    credentials {
    /**In gradle.properties file of root project folder, add github.user=GITHUB_USERNAME  & github.token =GITHUB_ACCESS_TOKEN**/
         username = project.findProperty("github.user") ?: System.getenv("USERNAME")
         password = project.findProperty("github.token") ?: System.getenv("TOKEN")
          }
    }
     maven {
         url = uri("https://maven.pkg.github.com/euc-releases/ws1-intelligencesdk-sdk-android/")
     credentials {
     /**In gradle.properties file of root project folder, add github.user=GITHUB_USERNAME  & github.token =GITHUB_ACCESS_TOKEN**/
         username = project.findProperty("github.user") ?: System.getenv("USERNAME")
         password = project.findProperty("github.token") ?: System.getenv("TOKEN")
         }
     }
}

dependencies {
    // Integrate Omnissa Workspace ONE at the Client level.
    //
    // -   Omnissa provides this Software Development Kit (the “Software”) to
    //     you subject to the following terms and conditions. By downloading,
    //     installing, or using the Software, you agree to be bound by the terms
    //     of https://static.omnissa.com/sites/default/files/omnissa-sdk-agreement.pdf
    //     If you disagree with any of the terms, then do not use the Software.

    //    For additional information, please visit the https://www.omnissa.com/legal-center/.
    //

    // -   Review the Omnissa Privacy Notice and the Workspace ONE UEM Privacy
    //     Disclosure for information on applicable privacy policies, and
    //     for additional information, please visit the
    //     https://www.omnissa.com/legal-center/
    implementation "com.airwatch.android:airwatchsdk:25.02.4"
}
```

This completes the required changes to the build configuration. Build the application to confirm that no mistakes have been made. After that, continue with the next step, which is Anchor Event Handler Implementation.

If you haven’t installed your application via Workspace ONE at least once, then the application under development won’t work when installed via the Android Debug Bridge (adb). Instructions for installing via Workspace ONE can be found in the Integration Guides document set, in the Integration Preparation guide.

Anchor Event Handler Implementation

The Workspace ONE Client SDK runtime receives various essential notifications from the management console. An implementation of a specific Android broadcast receiver and action handler must be added to your application to support this. From SDK 23.04 onwards, application need not add implementation for AirWatchSDKBaseIntentService, and must be removed.

Proceed as follows.

1. **Implement a Workspace ONE SDK Event handler class.**
   - Add a new class to your application.
   - Declare the new class and implement `WS1AnchorEvents` interface.
   - While upgrading to SDK 23.04 or above, migrate AirWatchSDKBaseIntentService API implementation to WS1AnchorEvents.

   In Java, the class could look like this:

```
public class AppWS1AnchorEvents implements WS1AnchorEvents {
    @Override
    public void onClearAppDataCommandReceived(Context context, ClearReasonCode reasonCode) {}

    @Override
    public void onApplicationConfigurationChange(Bundle applicationConfiguration, Context context) {}

    @Override
    public void onApplicationProfileReceived(
        Context context,
        String profileId,
        ApplicationProfile awAppProfile) {}

    @Override
    public void onAnchorAppStatusReceived(Context context, AnchorAppStatus awAppStatus) {}

    @Override
    public void onAnchorAppUpgrade(Context context, boolean isUpgrade) {}
}
```

   In Kotlin, the class could look like this:

```
class AppWS1AnchorEvents : WS1AnchorEvents {
    override fun onClearAppDataCommandReceived(context: Context?, reasonCode: ClearReasonCode?) {}

    override fun onApplicationConfigurationChange(
        applicationConfiguration: Bundle?,
        context: Context?,
    ) {}

    override fun onApplicationProfileReceived(
        context: Context?,
        profileId: String?,
        awAppProfile: ApplicationProfile?) {}

    override fun onAnchorAppStatusReceived(context: Context?, awAppStatus: AnchorAppStatus?) {}

    override fun onAnchorAppUpgrade(context: Context?, isUpgrade: Boolean) {}
}
```

2. Declare the permission and interaction filter.

   In the Android manifest file, inside the `manifest` block but outside the `application` block, add declarations like the following.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>

<!-- Following declarations are added -->
<uses-permission android:name="com.airwatch.sdk.BROADCAST" />

<!-- Following tag applies to compileSdkVersion 30 or later. -->
<queries>
    <intent>
        <action android:name="com.airwatch.p2p.intent.action.PULL_DATA" />
    </intent>
</queries>

<!-- End of added declarations.>

<application ...>
...
```

3. Declare the notification receiver. From SDK 23.04 onwards, declaration for AirWatchSDKBaseIntentService must be removed from manifest.

   In the Android manifest file, inside the `application` block, add `receiver` declaration like the following.

```
<application>

    ...

    <receiver
        android:name="com.airwatch.sdk.AirWatchSDKBroadcastReceiver"
        android:permission="com.airwatch.sdk.BROADCAST" >
        <intent-filter>
            <action android:name="${applicationId}.airwatchsdk.BROADCAST" />
        </intent-filter>
        <intent-filter>
            <action
                android:name="com.airwatch.intent.action.APPLICATION_CONFIGURATION_CHANGED"
                />

            <!--
            In the host attribute, replace com.your.package with the package name of your
            application.
            -->
            <data android:scheme="app" android:host="com.your.package" />

        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.PACKAGE_ADDED" />
            <action android:name="android.intent.action.PACKAGE_REMOVED" />
            <action android:name="android.intent.action.PACKAGE_REPLACED" />
            <action android:name="android.intent.action.PACKAGE_CHANGED" />
            <action android:name="android.intent.action.PACKAGE_RESTARTED" />
            <data android:scheme="package" />
        </intent-filter>
    </receiver>

</application>
```

4. **Apps need to implement SDKClientConfig in their Application class and override getEventHandler() and return WS1AnchorEvents Implementation object. From SDK 23.04 onwards, application need to migrate to SDKClientConfig instead of AirWatchSDKBaseIntentService.**

```
public class AppApplication extends Application implements SDKClientConfig {
   @NonNull
   @Override
   public WS1AnchorEvents getEventHandler() {
      return new AppWS1AnchorEvents();
   }
}`
```

This completes the required anchor event handler implementation. Build the application to confirm that no mistakes have been made.

If you haven't installed your application via Workspace ONE at least once, then the application under development won't work when installed via the Android Debug Bridge (adb). Instructions for installing via Workspace ONE can be found in the Integration Guides document set, in the Integration Preparation guide.

Next Steps

After completing the above, continue with the next task, which could be either of the following.

- Initialize the Client SDK, if your application will use only Client-level integration.
- Add the Framework, otherwise.

Task: Initialize Client SDK

Client SDK initialization is a Workspace ONE platform integration task for Android application developers. It applies only to Client-level integration, not to Framework integration.

The Client SDK initialization task is dependent on the Add the Client SDK task. The following instructions assume that the dependent task is complete already.

SDKManager

The main class of the Client SDK is SDKManager. It must be initialized before use. Initialize it by calling the **`init`** class method. The call must be on a background thread. An Android Context object is required, which could be an Activity instance for example.

In Java, code for an Activity that initializes the SDKManager could look like this:

```
public class MainActivity extends Activity {
    SDKManager sdkManager = null;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        startSDK();
    }

    private void startSDK() { new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                final SDKManager initSDKManager = SDKManager.init(MainActivity.this);
                sdkManager = initSDKManager;
                toastHere(
                    "Workspace ONE console version:" + initSDKManager.getConsoleVersion());
            }
            catch (Exception exception) {
                sdkManager = null;
                toastHere(
                    "Workspace ONE failed " + exception + ".");
            }
        }
    }).start(); }

    private void toastHere(final String message) {runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(MainActivity.this, message, Toast.LENGTH_LONG).show();
        }
    });}

    ...
}
```

In Kotlin, code for an Activity that initializes the SDKManager could look like this:

```
class MainActivity : Activity() {

    private var sdkManager: SDKManager? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ...
        startSDK()
    }

    private fun startSDK() { thread {
        try {
            val initSDKManager = SDKManager.init(this)
            sdkManager = initSDKManager
            toastHere("Workspace ONE console version:${initSDKManager.consoleVersion}")
        }
        catch (exception: Exception) {
            sdkManager = null
            toastHere("Workspace ONE failed $exception.")
        }
    } }

    private fun toastHere(message: String) { runOnUiThread {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show()
    } }

    ...
}
```

Calling the `init` method completes SDK Manager initialization. Build and run the application to verify that no mistakes have been made.

Next Steps

After the SDKManager instance has been received from the init call, its other methods can be called. Check the reference documentation for details of the programming interface.

This completes Client-level integration.

Task: Add Framework

Adding the Framework is a Workspace ONE platform integration task for Android application developers. Adding the Framework is necessary if the application will make use of platform features such as authentication, single sign-on, data encryption, or networking.

This task is dependent on the Add the Client SDK task. The following instructions assume that the dependent task is complete already.

Build Configuration and Files

This task involves changing your application project's build configuration and files. These instructions assume that your application has a typical project structure, same as the Add Client SDK task.

Proceed as follows.

1. **Add the required libraries to the build.**

   In the application build.gradle file, in the `dependencies` block, add references to the required libraries. For example:

```
repositories {
    maven {
        url = uri("https://maven.pkg.github.com/euc-releases/Android-WorkspaceONE-SDK/")
    credentials {
    /**In gradle.properties file of root project folder, add github.user=GITHUB_USERNAME  & github.token =GITHUB_ACCESS_TOKEN**/
        username = project.findProperty("github.user") ?: System.getenv("USERNAME")
        password = project.findProperty("github.token") ?: System.getenv("TOKEN")
        }
    }
    maven {
        url = uri("https://maven.pkg.github.com/euc-releases/ws1-intelligencesdk-sdk-android/")
    credentials {
    /**In gradle.properties file of root project folder, add github.user=GITHUB_USERNAME  & github.token =GITHUB_ACCESS_TOKEN**/
        username = project.findProperty("github.user") ?: System.getenv("USERNAME")
        password = project.findProperty("github.token") ?: System.getenv("TOKEN")
        }
    }
}

dependencies {
    // Integrate Omnissa Workspace ONE at the Client level.
    //
    // -   Omnissa provides this Software Development Kit (the “Software”) to
    //     you subject to the following terms and conditions. By downloading,
    //     installing, or using the Software, you agree to be bound by the terms
    //     of https://static.omnissa.com/sites/default/files/omnissa-sdk-agreement.pdf
    //     If you disagree with any of the terms, then do not use the Software.

    //    For additional information, please visit the https://www.omnissa.com/legal-center/.
    //
    // -   Review the Omnissa Privacy Notice and the Workspace ONE UEM Privacy
    //     Disclosure for information on applicable privacy policies, and
    //     for additional information, please visit the
    //     https://www.omnissa.com/legal-center/
    implementation "com.airwatch.android:awframework:25.02.4"
}
```

   Your application might already require different versions of some of the same libraries required by the SDK. Warning messages will be generated in the build output in that case, for example stating that there are more then one version available in the classpath.

   You can resolve this by selecting one or other version, either the SDK requirement or your app's original requirement.

In principle, the SDK isn't supported with versions other than those which gets imported via maven. In practice however, problems are unlikely to be encountered with later versions.

2. **Add annotation processor support.**

   In the application build.gradle file, add the `kotlin-kapt` plugin. The plugin can be added in the plugins block at the start of the file, for example as shown in the following snippet.

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-android-extensions'

    // Following line adds the required plugin.
    id 'kotlin-kapt'
}
...
```

3. **Add the required packaging and compile options.**

   Still in the application build.gradle file, in the `android` block, add the packaging option shown in the following snippet.

```
...
android {
    compileSdk 35

    // Following block is added.
    packagingOptions {
        pickFirst '**/*.so'
        exclude 'META-INF/LICENSE.txt'
        exclude 'META-INF/NOTICE.txt'
        jniLibs {
         useLegacyPackaging true
        }
    }
    // End of added block.

    defaultConfig {
        targetSdk 35
        ...
    }
    buildTypes {
        ...
    }
}
...
```

   The above assumes that support for earlier Android operating system versions and processor architectures isn't required in the application.

4. **App targeting API level 31 or above, override getEventHandler() in App's Application class to return WS1AnchorEvents object.**

```
    public class AppApplication extends AWApplication {
       @NonNull
       @Override
       public WS1AnchorEvents getEventHandler() {
          return new AppWS1AnchorEvents();
       }
    }
```

This completes the required changes to the build configuration. Build the application to confirm that no mistakes have been made. After that, continue with the next task, which is to Initialize the Framework.

Task: Initialize Framework

Framework initialization is a Workspace ONE platform integration task for Android application developers. It applies to Framework-level integration, not to Client-level integration.

The Framework initialization task is dependent on the Add the Framework task. The following instructions assume that the dependent task is complete already.

Select initialization class

Framework initialization can start from either an Android Application subclass, referred to as initialization by *delegation*, or from a Workspace ONE SDK AWApplication subclass, referred to as initialization by *extension*. Choose the better option for your application, as follows.

- If your application has an Android Application subclass, then choose it as the Framework initialization class. Proceed to these instructions:
  Initialize by delegation from an Android application subclass.
- Otherwise, create a Workspace ONE SDK AWApplication subclass and it will be the Framework initialization class. Proceed to these instructions:
  Create an initialization subclass by extension.

Initialize by delegation from an Android Application subclass

Follow these instructions to initialize from an Android Application subclass. This is an alternative to creating an AWApplication subclass. See Select initialization class for a discussion of the alternatives.

Update your Android Application subclass as follows.

- Declare that the class implements the AWSDKApplication interface.
- Add an AWSDKApplicationDelegate instance as a property.
- Move the code from the body of your onCreate method, if any, to an override of the AWSDKApplication onPostCreate method.
- Override the AWSDKApplication getMainActivityIntent() method to return an Intent for the application's main Activity.
- Override the following Android Application methods:
  - onCreate
  - getSystemService
  - attachBaseContext

  The required overrides are shown in the code snippets below, in Kotlin and in Java.
- Implement all the other AWSDKApplication methods by calling the same method in the AWSDKApplicationDelegate instance.

  Kotlin delegation-by can be used for the implementation. This is illustrated in the Initialization by delegation in Kotlin code snippet below.

Initialization by delegation in Java

In Java, the class could look like this:

```
public class Application extends android.app.Application implements AWSDKApplication {
    // SDK Delegate.
    private final AWSDKApplicationDelegate awDelegate = new AWSDKApplicationDelegate();
    @NotNull
    @Override
    public AWSDKApplication getDelegate() { return awDelegate; }

    // Android Application overrides for integration.
    @Override
    public void onCreate() {
        super.onCreate();
        this.onCreate(this);
    }

    @Override
    public Object getSystemService(String name) {
        return this.getAWSystemService(name, super.getSystemService(name));
    }

    @Override
    public void attachBaseContext(@NotNull Context base) {
        super.attachBaseContext(base);
        attachBaseContext(this);
    }

    // Application-specific overrides.
    @Override
    public void onPostCreate() {
        // Code from the application's original onCreate() would go here.
    }

    @NonNull
    @Override
    public Intent getMainActivityIntent() {
        // Replace MainActivity with application's original main activity.
        return new Intent(getApplicationContext(), MainActivity.class);
    }

    // Mechanistic AWSDKApplication abstract method overrides.

    // Methods that return a value could follow this as a template:
    @Nullable
    @Override
    public Object getAWSystemService(@NotNull String name, @Nullable Object systemService) {
        return awDelegate.getAWSystemService(name, systemService);
    }

    // Methods that return void could follow this as a template:
    @Override
    public void attachBaseContext(@NotNull android.app.Application application) {
        awDelegate.attachBaseContext(application);
    }

    @Override
    public Intent getMainLauncherIntent() {
        return awDelegate.getMainLauncherIntent();
    }


    // ... Many more overrides here.
}
```

Initialization by delegation in Kotlin

In Kotlin, the class could look like this:

```
// This class uses Kotlin delegation to implement the AWSDKApplication
// interface.
// A new AWSDKApplicationDelegate instance is allocated on the fly as the
// delegate. For background on Kotlin delegation, see:
// https://kotlinlang.org/docs/reference/delegation.html
open class Application:
    android.app.Application(),
    AWSDKApplication by AWSDKApplicationDelegate()
{
    // Android Application overrides for integration.
    override fun onCreate() {
        super.onCreate()
        onCreate(this)
    }

    override fun getSystemService(name: String): Any? {
        return getAWSystemService(name, super.getSystemService(name))
    }

    override fun attachBaseContext(base: Context?) {
        super.attachBaseContext(base)
        attachBaseContext(this)
    }

    // Application-specific overrides.
    override fun onPostCreate() {
        // Code from the application's original onCreate() would go here.
    }

    override fun getMainActivityIntent(): Intent {
        // Replace MainActivity with application's original main activity.
        return Intent(applicationContext, MainActivity::class.java)
    }

    override fun getMainLauncherIntent(): Intent {
        return super.getMainLauncherIntent();
    }
}
```

Next

This completes initialization from an Android Application subclass. Now continue with the next step, which is to configure the initialization class in the manifest.

Create an initialization subclass by extension

Follow these instructions to create a Framework initialization AWApplication subclass. This is an alternative to initialising from an Android Application subclass. See Select initialization class for a discussion of the alternatives.

Add to your application code a new class that:

- Is declared as an AWApplication subclass.
- Overrides the getMainActivityIntent() method to return an Intent for the application's main Activity.
- Implements the other required methods with dummies.
- Overrides the getMainLauncherIntent() method so that the SDK knows which Activity to give control to during initialization.

In Java, the class could look like this:

```
// Note the fully qualified class name in the extends declaration.
public class AWApplication extends com.airwatch.app.AWApplication {
    @NotNull
    @Override
    public Intent getMainActivityIntent() {
        return new Intent(getApplicationContext(), MainActivity.class);
    }

    @Override
    public void onSSLPinningRequestFailure(
            @NotNull String host, X509Certificate x509Certificate
    ) {
    }

    @Override
    public void onSSLPinningValidationFailure(
            @NotNull String host, X509Certificate x509Certificate
    ) {
    }

    @Override
    public Intent getMainLauncherIntent() {
        return new Intent(getApplicationContext(), SDKSplashActivity.class);
    }
}
```

In Kotlin, the class could look like this:

```
// Note the fully qualified base class name.
open class AWApplication: com.airwatch.app.AWApplication() {
    override fun getMainActivityIntent(): Intent {
        return Intent(applicationContext, MainActivity::class.java)
    }

    override fun onSSLPinningRequestFailure(
        host: String,
        serverCACert: X509Certificate?
    ) {
    }

    override fun onSSLPinningValidationFailure(
        host: String,
        serverCACert: X509Certificate?
    ) {
    }

    override fun getMainLauncherIntent(): Intent {
        return Intent(applicationContext, SDKSplashActivity::class.java)
    }

}
```

This completes the creation of an initialization subclass. Now continue with the next step, which is to configure the initialization class in the manifest.

Configure the initialization class in the manifest

Follow these instructions to configure your selected initialization class in the Android manifest. The initialization class will be either the existing Android Application subclass, or a new AWApplication subclass that was just created. See Select initialization class for a discussion of the alternatives.

Proceed as follows.

1. **Add the Android schema tools.**

   The tools can be added at the top of the file, in the manifest tag, for example like this:

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

2. **Update the application declaration.**

   The application declaration must be updated to:

   - Declare an application class name, if it wasn't already declared.
   - Replace the label.
   - Override the allowBackup flag with the setting from the SDK manifest.

   These updates can be made in the application tag, for example like this:

```
<application
    android:name=".YourApplicationOrAWApplicationSubClass"
    android:label="@string/app_name"
    ...
    tools:replace="android:label, android:allowBackup, android:networkSecurityConfig"
    >
```

3. **Set the launcher and main Activity to be from the Framework.**

   If the application had a previous declaration for launcher and main Activity, remove it. Instead, declare the Framework SDKSplashActivity as launcher and main.

   New declarations could look like this, for example:

```
<activity
    android:name=".MainActivity"
    >
    <!-- Original launcher and main declarations removed. -->
</activity>
<activity
    android:name="com.airwatch.login.ui.activity.SDKSplashActivity"
    android:label="@string/app_name"
    >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

4. **From SDK 23.04 onwards, application need not add implementation for AirWatchSDKBaseIntentService, and must be removed while upgrading SDK.**

5. **Declare the required permission.**

If your app targets Android 13 or higher, then in order to see notifications declare the below permission in your app's manifest file if not present already. developer.android.com/...13/...#notification-permission

```
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
```

This completes the initialization class configuration.

Request the required Permissions

If your app targets Android 13 or higher, request the new notification permission from your app's MainActivity if not requested already. developer.android.com/...13/...#notification-permission

Below is the code snippet, for example:

```
private void setupPermissions() {
    if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        int permission = ContextCompat.checkSelfPermission(
            this,
            Manifest.permission.POST_NOTIFICATIONS
        );

        if (permission != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(
                this,
                new String[]{Manifest.permission.POST_NOTIFICATIONS},
                NOTIFICATION_REQ_CODE
            );
        }
    }
}

@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    if (requestCode == NOTIFICATION_REQ_CODE){
        if (grantResults.length == 0 || grantResults[0] != PackageManager.PERMISSION_GRANTED) {
        toastHere("Notification Permission has been denied by user");
        } else {
        toastHere("Notification Permission has been granted by user");
        }
    }
}
```

This completes requesting the required permissions.

Next Steps

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

The Workspace ONE splash screen should be shown at launch, Other SDK screens might also be shown depending on the configuration in the management console. See the Appendix: User Interface Screen Capture Images.

After completing the above, you can proceed to:

- Networking integration.
- Branding integration.
- Integration of other framework features.

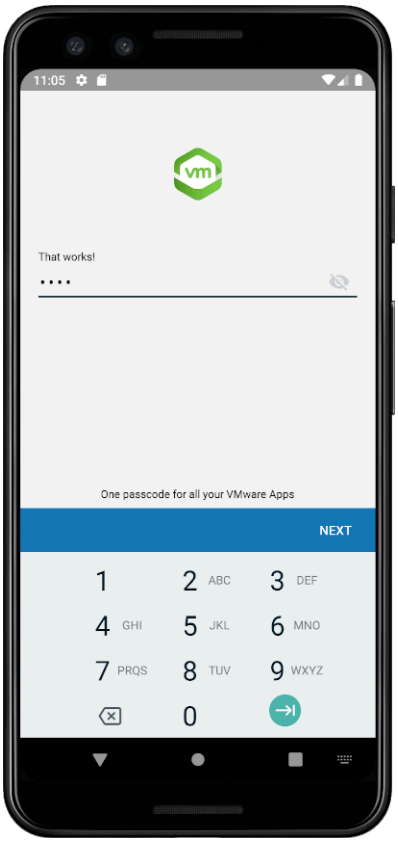
See the respective documents in the Workspace ONE Integration Guide for Android set. An overview that includes links to all the guides in the set is available

- in Markdown format, in the repository that also holds the sample code: https://github.com/euc-releases/...IntegrationOverview.md
- in Portable Document Format (PDF), on the Omnissa website: https://developer.omnissa.com/...IntegrationOverview.pdf

Appendix: User Interface Screen Capture Images

The following images show screens that are part of the Workspace ONE SDK user interface.

Screen capture 1: Splash screen



Screen capture 2: Login screen

The splash screen should be shown during every launch of an application that is integrated to the Framework level. The login screen might be shown afterwards, depending on the application state and the configuration in the management console.

Appendix: Troubleshooting

Kotlin Compatibility

Occasionally, one may encounter an exception containing the message “Class ‘kotlin.Unit’ was compiled with an incompatible version of Kotlin. The binary version of its metadata is 1.. version, expected version is 1.8.21.” during compilation. This exception is due to incompatible versions of your app with the Workspace One SDK. As of Release 23.09, all apps consuming WS1 will be required to use Kotlin v1.8.21 or higher.

Empty Response from AirWatch MDM Service

Occasionally, one may encounter the message “Empty Response from Airwatch MDM Service” in the adb log during app integration into Workspace ONE. This error message is triggered when the app was not installed via Intelligent Hub.

To resolve this error, it is recommended to upload the APK to the UEM once, then install the app through Intelligent Hub.

For detailed instructions please refer to the Integration Preparation Guide, specifically

Appendix: How to upload an Android application to the management console

- as Markdown: Preperation Guide - Appendix: How to upload an Android application to the management console
- as PDF: Preperation Guide - Appendix: How to upload an Android application to the management console

and

Task: Install application via Workspace ONE

- as Markdown: Preparation Guide - Task: Install application via Workspace ONE
- as PDF: Preparation Guide - Task: Install application via Workspace ONE

Once the APK has been uploaded to the UEM and installed via Workspace ONE, the app can then be subsequently side-loaded by the ADB provided the side load is signed by the same developer key as the original upload. To ensure your APK is signed on every build please refer to the Preparation Guide, specifically

Appendix: How to generate a signed Android package every build

- as Markdown Preparation Guide - How to generate a signed Android package every build
- as PDF: Preparation Guide - How to generate a signed Android package every build

Document Information

Published Locations

This document is available

- in Markdown format, in the repository that also holds the sample code: https://github.com/euc-releases/...BaseIntegration.md
- in Portable Document Format (PDF), on the Omnissa website: https://developer.omnissa.com/...BaseIntegration.pdf

Revision History

The following table shows the revision history of this document.

Date	Revision
03jul2020	First publication, for 20.4 SDK for Android.
31jul2020 to 09dec2021	Updated for 20.7 to 21.11 SDK for Android releases.
26Jan2022	Update for 22.1 SDK for Android.
28Feb2022	Update for 22.2 SDK for Android.
04Apr2022	Updated for 22.3 SDK for Android.
29Apr2022	Updated for 22.4 SDK for Android.
06Jun2022	Updated for 22.5 SDK for Android.
05Jul2022	Updated for 22.6 SDK for Android.
23Aug2022	Updated for 22.8 SDK for Android.
04Nov2022	Updated for 22.10 SDK for Android.
13Dec2022	Updated for 22.11 SDK for Android.
25Jan2023	Updated for 23.01 SDK for Android.
15Mar2023	Updated for 23.03 SDK for Android.
27Apr2023	Updated for 23.04 SDK for Android.
06Jun2023	Updated for 23.06 SDK for Android.
24Jul2023	Updated for 23.07 SDK for Android.
07Sep2023	Updated for 23.09 SDK for Android.
25Oct2023	Updated for 23.10 SDK for Android.
18Dec2023	Updated for 23.12 SDK for Android.
25Jan2024	Updated for 24.01 SDK for Android.
15May2024	Updated for 24.04 SDK for Android.
05Jul2024	Updated for 24.06 SDK for Android.
28Aug2024	Updated for 24.07 SDK for Android.
29Oct2024	Updated for 24.10 SDK for Android.
10Dec2024	Updated for 24.11 SDK for Android.
04mar2025	Documentation update for Android.
12Mar2025	Updated for Android SDK 25.02.
26May2025	Updated for Android SDK 25.02.1.
04Aug2025	Updated for Android SDK 25.02.3.
24Sep2025	Updated for Android SDK 25.02.4.