



MobileIron AppConnect 9.2.0 for Android App Developers Guide

March 17, 2021

For complete product documentation see:

[AppConnect for Android Product Documentation Home Page](#)

Copyright © 2012 - 2021 MobileIron, Inc. All Rights Reserved.

Any reproduction or redistribution of part or all of these materials is strictly prohibited. Information in this publication is subject to change without notice. MobileIron, Inc. does not warrant the use of this publication. For some phone images, a third-party database and image library, Copyright © 2007-2009 Aeleeeta's Art and Design Studio, is used. This database and image library cannot be distributed separate from the MobileIron product.

“MobileIron,” the MobileIron logos and other trade names, trademarks or service marks of MobileIron, Inc. appearing in this documentation are the property of MobileIron, Inc. This documentation contains additional trade names, trademarks and service marks of others, which are the property of their respective owners. We do not intend our use or display of other companies' trade names, trademarks or service marks to imply a relationship with, or endorsement or sponsorship of us by, these other companies.



Contents

Contents	3
New features and enhancements	10
AppConnect for Android overview	11
MobileIron components supporting AppConnect apps	12
About wrapping for AppStation	14
Apps that you can wrap	15
Required app development	16
Understanding AppConnect for Android wrapping limitations	16
Using AppTunnel with HTTP/S tunneling	16
Handling app-specific configuration	17
Android devices supporting AppConnect apps	17
Features of AppConnect for Android apps	17
Accessible Apps to preserve the user experience	18
Securing and managing an Android AppConnect app	20
Authorization	21
AppConnect passcode policy	21
AppTunnel with HTTP/S tunneling	22
Supported APIs	22
HTTP/S redirects	23
HelloAppTunnel sample app	23
AppTunnel with TCP tunneling	23
When to use AppTunnel with HTTP/S tunneling versus TCP tunneling	23
SSL between the device and Sentry	24
Certificate authentication with AppTunnel with TCP tunneling	25
App requirements	25
Data loss prevention settings	26



Supported file sizes for streaming media	26
App whitelist	27
Handling app-specific configuration from the MobileIron server	27
Ignoring the auto-lock time	28
MobileIron server configuration	28
App requirements	28
Wrapping technology	29
Handling AppConnect app-specific configuration	31
Overview of configuration handling	31
App-specific configuration data flow	32
Tasks for configuration handling	33
Check at runtime if your app is wrapped	34
Create a callback method to receive configuration updates	35
Implementing the callback method	35
Request the configuration when your app starts	36
Add callback information to AndroidManifest.xml	37
Specify app configuration and policies in .properties files	37
File location of the .properties files	38
Example of the appconnectconfig.properties file	38
Format of the appconnectconfig.properties file	39
Example of the appconnectpolicy.properties file	39
Format of the appconnectpolicy.properties file	39
Changing from the legacy configuration handling to the new mechanism	40
Sample Java app for handling app-specific configuration	40
Best practices for handling app-specific configuration	41
Provide documentation about your app to the MobileIron server administrator	41
Use only a login ID from the MobileIron server if one is expected	42
Testing app-specific configuration handling	42
Requesting a MobileIron Core test instance	42



Downloading Mobile@Work to the device	43
Logging in to the Admin Portal	43
Creating a label for testing your app	43
Enabling AppConnect on MobileIron Core	44
Configuring the AppConnect global policy	44
Uploading the Secure Apps Manager to MobileIron Core	45
Uploading your AppConnect app to MobileIron Core	45
Configuring app-specific settings in MobileIron Core	45
Registering the test device to MobileIron Core	46
Pushing Core configuration changes to the device	46
Sample apps, tester app, and Cordova plugin	47
Summary of sample apps, tester app, and Cordova plugin	47
Location of sample apps, tester app, and Cordova plugin	48
Android API Usage Demo sample app overview	49
Demonstrated APIs	50
Audio Recorder Demo	50
Capture Image Demo	50
Documents Demo	51
Image Decoder Demo	51
Media Meta Data Retriever Demo	52
Media Player Demo	52
Pick Image Demo	52
Share Content Demo	53
Video Recorder Demo	53
HelloReact Demo sample app overview	53
Demonstrated functionality	54
AppConfig Demo	54
Network API Demo	55
Capture Image Demo	55



Pick Image Demo	55
Video View Demo	56
Firebase Cloud Messaging Demo	56
HelloFlutter sample app overview	56
Demonstrated APIs	57
AppConfig Demo	57
Before wrapping an Android app	58
Checking wrapping limitations	59
Determining the wrapping mode	59
The AppConnect wrapping portal	62
Using the AppConnect Wrapping Portal	62
Logging in to help.mobileiron.com	62
Uploading and wrapping an app	63
Downloading the wrapped app	65
The AppConnect for Android wrapping tool	66
Enterprise private key considerations with AppConnect for Android	66
AppConnect for Android Wrapping Tool supported platforms	67
Preparing to use the wrapping tool	67
Using the AppConnect for Android Wrapping Tool in UI mode	68
Launching the wrapping tool	69
Providing developer settings to the wrapping tool	69
Selecting wrapping options in the wrapping tool	70
Wrapping and signing an app with the wrapping tool	71
Signing an app with the wrapping tool	72
Using the AppConnect for Android Wrapping Tool in CLI mode	72
Providing developer settings	73
Setting the keystore	74
Wrapping and signing the app	75
Additional wrapping tasks using CLI	76



Signing an app	76
Using the Generation 1 wrapper	77
Wrapping with a different allowed wrapper version	77
Viewing wrapper arguments for a wrapper version	78
Using the -help command	79
Wrapping tool CLI	79
Troubleshooting the wrapping tool	80
Distributing wrapped apps with an enterprise key (Core)	81
Uploading the apps to the App Catalog	81
Configuring the enterprise public key	81
Applying labels to the new apps	82
Removing labels from the old apps	82
The device user experience when upgrading	83
Behavior when the device does not have the enterprise public certificate	83
After wrapping an Android app	84
Adding the wrapped app's key to the Google API Console	84
Wrapped app's Google API key format	84
Adding the new API key to Google API console	84
Inform the server administrator of your app's requirements	85
Capabilities and limitations of apps you can wrap	86
AppConnect wrapping considerations	86
SQLCipher considerations	87
SQLCipher library version	87
Using both SQLCipher and SQLite is not supported	87
Encryption of the SQLCipher database	87
DownloadManager API considerations	87
Google Cloud Messaging considerations	87
Unsupported GCM features	88
Situations when GCM messages are discarded	88



Firebase Cloud Messaging and Crashlytics support	88
MediaPlayer and MediaMetadataRetriever Internet permission requirement	88
Image selection from outside the AppConnect container	89
External storage permissions	89
Support for scoped storage	90
Receiving information from outside the AppConnect container	90
USB OTG support	91
Preference API usage	91
64-bit support	91
Linking native Java methods	92
Wrapping support of commonly used app capabilities	92
Generation 1 and Generation 2 support for commonly used app capabilities	92
Generation 1 wrapper support for commonly used app capabilities	94
Generation 2 Wrapper support for commonly used app capabilities	94
Known wrapper limitations	94
Generation 1 and Generation 2 wrapper limitations	95
Generation 1 wrapper limitations	97
Generation 2 wrapper limitations	97
Legacy mechanism for handling AppConnect app-specific configuration	98
Overview of legacy configuration handling	98
Communicating with the MobileIron client app using intents for legacy configuration handling	100
App-specific configuration legacy data flow	100
Contents of the Intent objects in legacy configuration handling	101
Tasks for legacy configuration handling	102
Check at runtime if your app is wrapped	103
Add a service to AndroidManifest.xml	104
Create a class that extends IntentService	104
Implement onHandleConfig()	104
Reasons for returning an error	105



Request the configuration when your app starts	106
Specify app configuration and policies in .properties files	106
File location of the .properties files	106
Example of the appconnectconfig.properties file	107
Format of the appconnectconfig.properties file	107
Example of the appconnectpolicy.properties file	108
Format of the appconnectpolicy.properties file	108
Sample Java app for legacy app-specific configuration handling	109
App for testing legacy configuration handling	109
Using AppConnectTester	110
Protecting the unwrapped version of your app	111
Best practices for handling app-specific configuration	112
Provide documentation about your app to the MobileIron server administrator	113
Use only a login ID from the MobileIron server if one is expected	113



New features and enhancements

This guide documents the following new features and enhancements:

- **ThumbnailUtils support:** The following ThumbnailUtils deprecated methods are supported: `createAudioThumbnail`, `createImageThumbnail`, `createVideoThumbnail`.
See [Generation 1 and Generation 2 support for commonly used app capabilities](#) .
- **Command line interface (CLI) support with the AppConnect wrapping tool:** Command line interface (CLI) support with the AppConnect wrapping tool: CLI support for wrapping Android app using the AppConnect wrapping tool.
For more information see [Using the AppConnect for Android Wrapping Tool in CLI mode](#)
- **Support for Firebase Crashlytics:** If the `-enableCrashlytics` option is used when wrapping an app, the app crash data is available on the Firebase Crashlytics console. To use the feature, the app must support Firebase Crashlytics and wrapped with the 9.2.0 wrapper. The crash reports appear for the unwrapped app package name on the console. However, the reports include the text label (**wrapped app**), which identifies the report as an AppConnect wrapped app crash report.
For more information, see [Firebase Cloud Messaging and Crashlytics support](#).

For a list of new features introduced in this release see the *MobileIron AppConnect 9.2.0 for Android Release Notes and Upgrade Guide*.



AppConnect for Android overview

MobileIron AppConnect for Android provides a secure and managed container for enterprise applications and data on Android devices. These secure enterprise apps are called *AppConnect apps* or *secure apps*. You create an AppConnect app for Android using the MobileIron AppConnect wrapping technology. This wrapping technology transforms an Android app into a secure app with minimal app development.

The Android devices running AppConnect apps are registered to a MobileIron server: either MobileIron Core, MobileIron Connected Cloud, or MobileIron Cloud. An enterprise uses the MobileIron server to distribute the apps. The apps are called “in-house” apps, regardless of who developed the apps, because the apps are *distributed* by the enterprise. The app developer is either an in-house developer at the enterprise, or a third-party app developer.

To wrap an app, you either:

- Submit it to the MobileIron AppConnect Wrapping Portal. MobileIron signs the wrapped apps with the MobileIron private key. MobileIron also signs the Secure Apps Manager and all secure apps provided by MobileIron with the MobileIron private key.
The AppConnect Wrapping Portal is the simplest and most common way to wrap apps. It can be used for apps that are distributed by MobileIron Core, MobileIron Connected Cloud, or MobileIron Cloud.
- Use the MobileIron AppConnect for Android Wrapping Tool, a desktop app, to wrap and sign your apps. Use the wrapping tool only if you require that your apps are signed with your own enterprise private key. Signing apps with your enterprise private key instead of the MobileIron private key is a security decision that your enterprise makes. The wrapping tool can be used only for apps that are distributed by MobileIron Core or MobileIron Connected Cloud. Apps wrapped with the wrapping tool **cannot** be distributed by MobileIron Cloud.

IMPORTANT:

- You cannot distribute AppConnect apps on Google Play.
- You cannot wrap an app (.apk file) that you get from Google Play.
- Some apps cannot be wrapped, depending on the APIs or features they use. See [Capabilities and limitations of apps you can wrap](#).

NOTE: Legal notices are located [here](#).

Related topics

For information about AppConnect for Android from the perspective of a MobileIron server administrator:

- MobileIron Core or Connected Cloud: The *MobileIron AppConnect Guide for Core*
- MobileIron Cloud: The *MobileIron Cloud Administrator Guide*



- *AppConnect Secure Apps for Android Release Notes and Upgrade Guide* which includes:
 - summary of new features
 - known and resolved issues
 - limitations
 - support and compatibility

MobileIron components supporting AppConnect apps

AppConnect for Android apps work with the following MobileIron components:

TABLE 1. MOBILEIRON COMPONENTS INVOLVED WITH APPCONNECT FOR ANDROID APPS

MobileIron component	Description
MobileIron Core	The MobileIron on-premise server which provides security and management for an enterprise's devices, and for the apps and data on those devices. An administrator configures the security and management features using a web portal.
MobileIron Connected Cloud	Cloud offering that has the same functionality as MobileIron Core.
MobileIron Cloud	Cloud offering that provides similar functionality as MobileIron Core. However, it does not support all the AppConnect features that MobileIron Core supports.
Standalone Sentry	The MobileIron server which provides secure network traffic tunneling from your app to enterprise servers.
Mobile@Work for Android	The MobileIron client app that runs on an Android device. It interacts with Core or Connected Cloud to apply the appropriate policies and AppConnect apps to the device. It also interacts with the Secure Apps Manager. The device user gets Mobile@Work from Google Play.
MobileIron Go for Android	The MobileIron client app that runs on an Android device. It interacts with MobileIron Cloud to apply the appropriate policies and AppConnect apps to the device. It also interacts with the Secure Apps Manager. The device user gets MobileIron Go from Google Play.
Secure Apps Manager	<p>Another MobileIron app that runs on an Android device. Working with Mobile@Work or MobileIron Go, it handles data encryption and the AppConnect passcode for logging on to AppConnect apps.</p> <p>MobileIron Core and Connected Cloud: The device user receives the Secure Apps Manager from Core or Connected Cloud.</p> <p>MobileIron Cloud: Secure Apps Manager is bundled with MobileIron Go, which the device user gets from Google Play.</p>
MobileIron AppStation for Android	Another MobileIron client app that runs on an Android device. AppStation is used instead of MobileIron Go to interact with MobileIron Cloud when the MobileIron



TABLE 1. MOBILEIRON COMPONENTS INVOLVED WITH APPCONNECT FOR ANDROID APPS (CONT.)

MobileIron component	Description
	Cloud tenant is configured for Mobile Apps Management (MAM) but not Mobile Device Management (MDM). AppStation applies the appropriate policies and AppConnect apps to the device. It also interacts with the Secure Apps Manager for AppStation. The device user gets MobileIron AppStation from Google Play.
Secure Apps Manager for AppStation	<p>Working with the MobileIron AppStation app, Secure Apps Manager for AppStation handles data encryption and the AppConnect passcode for logging on to AppConnect apps. It provides the same feature set as the Secure Apps Manager.</p> <p>Only apps wrapped using the AppStation option work with Secure Apps Manager for AppStation. Such wrapped apps do not work with the regular Secure Apps Manager.</p> <p>Secure Apps Manager for AppStation is bundled with the AppStation app, which the device user gets from Google Play.</p>
The AppConnect app wrapper	<p>Provided by the AppConnect wrapping technology, the app wrapper provides AppConnect capabilities and security to your app.</p> <p>The Generation 1 wrapper supports Java apps and AppTunnel with HTTP/S tunneling.</p> <p>The Generation 2 wrapper supports Java apps, including Java apps with C or C++ code, hybrid web apps, React Native apps, Xamarin apps, AppTunnel with HTTP/S or TCP tunneling, and scoped storage for apps with targetSdkVersion set to 30.</p>
The MobileIron AppConnect Wrapping Portal	<p>Available at help.mobileiron.com in the Developer > Wrapped Apps tab, the AppConnect Wrapping Portal is the simplest and most common way to wrap apps.</p> <p>It can be used for apps that are distributed by MobileIron Core, MobileIron Connected Cloud, or MobileIron Cloud.</p> <p>It can be used to wrap apps that work with one of:</p> <ul style="list-style-type: none"> the Secure Apps Manager the Secure Apps Manager for AppStation.
The MobileIron AppConnect for Android Wrapping Tool	<p>A desktop app, for wrapping and signing your apps.</p> <p>Use the wrapping tool only if you require that your apps are signed with your own enterprise private key. Signing apps with your enterprise private key instead of the MobileIron private key is a security decision that your enterprise makes.</p> <p>The wrapping tool can be used only for apps that are distributed by MobileIron Core or MobileIron Connected Cloud.</p> <p>IMPORTANT: Apps wrapped with the wrapping tool cannot be distributed by MobileIron Cloud.</p>



About wrapping for AppStation

If you are using the AppConnect Wrapping Portal, you have the option to wrap for MobileIron AppStation. AppStation is a MobileIron client app that runs on an Android device. AppStation is used instead of MobileIron Go to interact with MobileIron Cloud when the MobileIron Cloud tenant is configured for Mobile Apps Management (MAM) but not Mobile Device Management (MDM). AppStation applies the appropriate policies and AppConnect apps to the device.

AppStation interacts with the Secure Apps Manager for AppStation instead of the regular Secure Apps Manager. Secure Apps Manager for AppStation:

- provides the same feature set as the regular Secure Apps Manager.
- is bundled with the MobileIron AppStation app, which the device user gets from Google Play

When you wrap an app with the AppConnect Wrapping Portal, you select an option that determines whether the app works with Secure Apps Manager for AppStation or the regular Secure Apps Manager.

IMPORTANT:

- Apps wrapped using the AppStation option work **only** with Secure Apps Manager for AppStation.
- Apps wrapped without the AppStation option work **only** with the regular Secure Apps Manager.

Therefore, if you want your app to work with either Secure Apps Manager for AppStation or the regular Secure Apps Manager, you must wrap it twice: once with the AppStation option and once without the AppStation option.

When you wrap an app **with** the AppStation option, it has the following package name:

- appstation.<your app package name>

When you wrap an app **without** the AppStation option, it has the following package name:

- forgepond.<your app package name>

NOTE: Note that the Secure Apps Manager for AppStation and apps wrapped with the AppStation option have a different shield on their home screen icons than the regular Secure Apps Manager and apps wrapped without the AppStation option.

For information about AppStation for Android and how to deploy AppStation, see [MobileIron AppStation Product Documentation Home Page](#).



Apps that you can wrap

MobileIron supports wrapping the following types of apps:

- apps written in Java
- apps written in Java that include C or C++ code
C and C++ are native code languages on Android devices. Java apps that include C or C++ code are built with the Android Native Development Kit (NDK).
- apps built with the Xamarin development platform that use ModernHttpClient and OkHttp. HTTP tunneling along with Kerberos Constrained Delegation is also supported when using these libraries.

- hybrid web apps

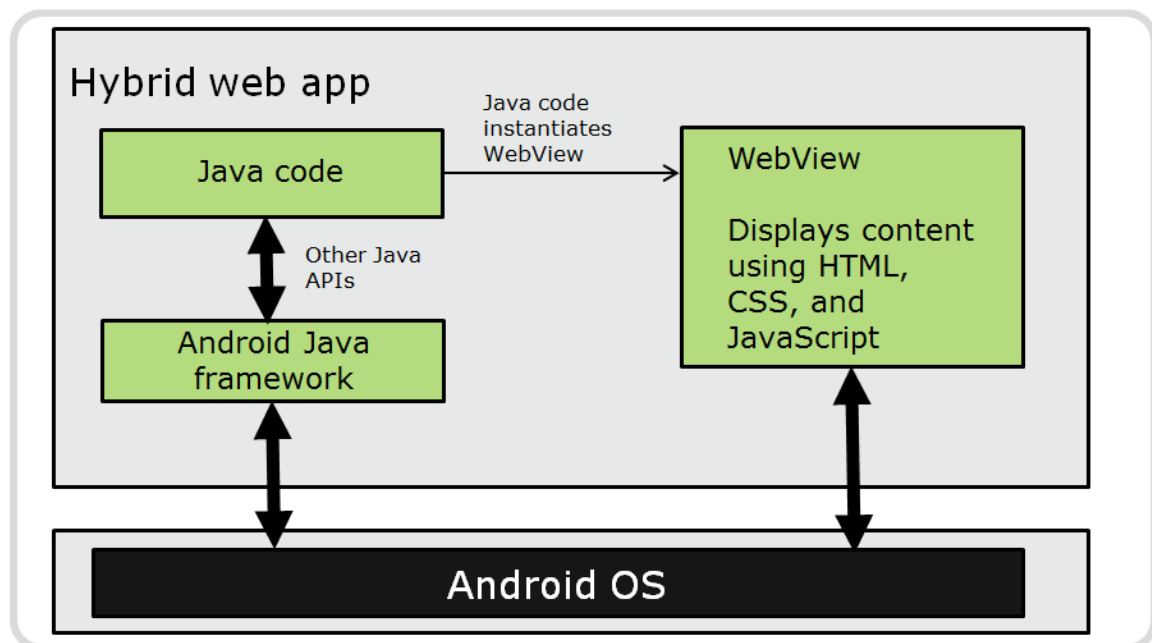
A hybrid web app is an Android app (APK file) that the device user installs on the device, unlike a pure web app that the user accesses through a web browser. A hybrid web app includes at least one activity (screen) that displays a web page.

Business logic and content presentation occurs using Android WebView and WebKit technologies, specifically within an object of the Java class `android.Webkit.WebView`. The WebView object locally renders content using web technologies such as HTML, CSS, and JavaScript. The WebView object can access the web content from a network resource or from embedded web content.

Like other wrapped app data, data related to the `android.webkit.WebView` class is encrypted. This web-related data can include cookies, the web cache, and web databases.

The following diagram illustrates a hybrid web app on an Android device.

FIGURE 1. HYBRID WEB APP ON AN ANDROID DEVICE



- PhoneGap apps

You can wrap an APK file that was created using the PhoneGap mobile development framework. The wrapped PhoneGap app is a type of wrapped hybrid web app.

See phonegap.com for information about creating PhoneGap apps.

- React Native apps

You can wrap an APK file that was created using the React Native mobile development framework.

IMPORTANT: Wrapping does not support all Java APIs and features or all NDK features. Details are listed in [Capabilities and limitations of apps you can wrap](#).

Required app development

- [Understanding AppConnect for Android wrapping limitations](#)
- [Using AppTunnel with HTTP/S tunneling](#)
- [Handling app-specific configuration](#)

Understanding AppConnect for Android wrapping limitations

AppConnect for Android wrapping technology does not support all Android coding capabilities. Details are listed in [Capabilities and limitations of apps you can wrap](#).

Before submitting an app for wrapping, review the supported app capabilities and the limitations. Make necessary changes to the app, if any, and verify the app's behavior.

Using AppTunnel with HTTP/S tunneling

Using MobileIron's AppTunnel feature, a secure enterprise app can securely tunnel HTTP and HTTPS network connections from the app to servers behind a company's firewall. If the app already uses specific, commonly used APIs to access enterprise servers with HTTP/HTTPS connections, no additional development is necessary to use standard AppTunnel. Otherwise, additional development is necessary to change to the required APIs.

Inform the MobileIron server administrator that your app requires AppTunnel with HTTP/S tunneling. The administrator requires this information to correctly configure AppTunnel on the server for your app.

For more information, see [AppTunnel with HTTP/S tunneling](#).

NOTE: All AppConnect apps can use AppTunnel with TCP tunneling to secure data-in-motion between the app and corporate resources with no additional development. See [AppTunnel with TCP tunneling](#).



Handling app-specific configuration

With some straightforward additional development, an app can receive app-specific configuration from the MobileIron server.

Inform the MobileIron server administrator the details about your app-specific configuration. The administrator requires this information to provide correct configuration values to your app from the MobileIron server.

For more information, see [Legacy mechanism for handling AppConnect app-specific configuration](#).

Android devices supporting AppConnect apps

AppConnect for Android apps are supported on the following Android devices:

- devices with 32-bit ARM processors
- devices with 64-bit ARM processors
- devices running Android 5.0 through the most recently released version as supported by MobileIron

Note The Following:

- Android version support for each app can vary.
- Support for specific *features* of AppConnect for Android depend on the version of various MobileIron components and the Android OS version. See the *AppConnect and AppTunnel Guide* for MobileIron Core and Connected Cloud deployments, and the *MobileIron Cloud Guide* for MobileIron Cloud deployments.

Features of AppConnect for Android apps

An Android device user can use an AppConnect app only if:

- the device user has been authenticated through the MobileIron server.
The user must use the Mobile@Work, MobileIron Go, or MobileIron AppStation for Android app to register the device with the MobileIron server. Registration authenticates the device user. Only registered devices can use an AppConnect app.
- the server administrator has authorized the device user to use the AppConnect app.
- the device user has entered the passcode for using AppConnect apps, if required by the server administrator.

With the AppConnect passcode, the device user can access all the AppConnect apps. When presented to device users, it is called the secure apps passcode. On the server Admin Portal, the administrator configures the rules for this AppConnect passcode. Access to AppConnect apps times out after a period of inactivity, after which the device user must reenter the AppConnect passcode.

The AppConnect passcode is not the same passcode as the device password, if a device password exists. The device user can choose the same values for both the AppConnect apps passcode and the device password, or choose a different value for each of them.



AppConnect apps:

- encrypt their application data.
Application data on the device is encrypted using AES-256 encryption. The encryption key is not stored on the device. It is programmatically derived, in part from the device user's AppConnect passcode. Therefore, the application data is secure even on a device that becomes compromised. For hybrid web apps, data related to the android.webkit package's WebView class is encrypted. This web-related data can include cookies, the web cache, and web databases.
NOTE: File names are not encrypted.
- use only containerized data.
AppConnect apps can share data only with other AppConnect apps. Unsecured apps cannot access the data. Data in the secure container stays in the secure container.
Exceptions are described in [Accessible Apps to preserve the user experience](#).
- enforce data loss prevention.
The server administrator determines the data loss prevention policies for an app. For example, these policies include whether an app allows screen capture, copy/paste interaction with other apps, and access to the camera, gallery, or media player. The AppConnect app's wrapper enforces the policies.
- can tunnel network connections to servers behind an enterprise's firewall.
This capability means that device users do not have to separately set up VPN access on their devices to use the app.
- can send a certificate to identify and authenticate the app user to an enterprise server.
Depending on the enterprise server implementation, this authentication occurs without interaction from the device user beyond entering the AppConnect passcode. That is, the device user does not need to enter a user name and password to log into enterprise services. Therefore, this feature provides a higher level of security and an improved user experience.
This feature is not available with MobileIron Cloud.
- can receive app-specific configuration information from the MobileIron server.
This capability requires some additional app development. It means that device users do not have to manually enter configuration details that the app requires. Furthermore, for security reasons, some apps do not want to allow the device users to provide certain configuration settings at all. By automating the configuration process for the device users, each user has a better experience when installing and setting up apps. Also, the enterprise has fewer support calls, and the app is secured from misuse due to configuration.
- provide anti phishing protection.
If anti-phishing is enabled in the UEM using Mobile Threat defense and users have enabled anti-phishing on their device, when users tap on a URL in their AppConnect app, anti-phishing protection is triggered. However, entering a URL directly into a browser or tapping a web link in a browser does not trigger anti-phishing support. For information about Mobile Threat Defense, see the *MobileIron Threat Defense Solution Guide* for your UEM.

Accessible Apps to preserve the user experience

AppConnect apps can share data only with other AppConnect apps.

However, some exceptions exist to this rule to:



- preserve the device user experience.
- enable the use of system services, such as making voice calls.

The exceptions are:

- maps
Tapping a meeting location in an AppConnect email app launches a maps app.
- phone calls
Tapping a phone number in any AppConnect app will make a phone call.
- SMS
An AppConnect app can allow the device user to send an SMS to a corporate contact.
- browsers
Tapping a link in an AppConnect app launches a browser. However, the MobileIron server administrator can limit the behavior to opening the link in a secure browser by using a data loss prevention policy.



Securing and managing an Android AppConnect app

A MobileIron server administrator configures how mobile device users can use secure enterprise applications. The administrator sets the following app-related settings that impact your AppConnect app's behavior:

- [Authorization](#)
- [AppConnect passcode policy](#)
- [AppTunnel with HTTP/S tunneling](#)

This feature sometimes requires additional development because an AppConnect app can use HTTP/HTTPS tunneling *only if* the app accesses the enterprise server using certain APIs.

- [AppTunnel with TCP tunneling](#)
- [Certificate authentication with AppTunnel with TCP tunneling](#)
- [Data loss prevention settings](#)
- [Handling app-specific configuration from the MobileIron server](#)

This feature requires additional app development.

- [Ignoring the auto-lock time](#)

The following steps show the flow of information from the MobileIron server to an AppConnect app:

1. The server administrator decides which app-related settings to apply to a device or set of devices.
2. The server sends the information to the MobileIron client app (Mobile@Work, MobileIron Go, or MobileIron AppStation).
3. The MobileIron client app passes the information to the AppConnect app. The MobileIron client app, the Secure Apps Manager (or Secure Apps Manager for AppStation), and the AppConnect app wrapper enforce the app-related settings.

Related topics

For information about AppConnect for Android from the perspective of a MobileIron server administrator:

- MobileIron Core or Connected Cloud: The *MobileIron AppConnect Guide for Core*
- MobileIron Cloud: The *MobileIron Cloud Administrator Guide*



Authorization

The server administrator determines which AppConnect apps can:

- be installed on each device.
- can run on each device.

If an AppConnect app is allowed to run, it is an authorized app. When a device user runs an unauthorized app, the Secure Apps Manager displays a message to the user, and the app exits.

An administrator also determines how many days a device can be out of contact with the server (not available on MobileIron Cloud). When the number of days is exceeded, the Secure Apps Manager removes the data for all the AppConnect apps. Finally, an administrator determines when a device is no longer registered with the server, such as when an employee leaves the company. At that time, all AppConnect apps on the device become unauthorized, and the Secure Apps Manager removes the data for all AppConnect apps.

The AppConnect wrapper, along with the Secure Apps Manager and the MobileIron client app, handles app authorization. **No additional app development is necessary.**

AppConnect passcode policy

One AppConnect passcode controls all AppConnect (secure) apps on the device.

The server administrator determines whether to require an AppConnect passcode. If it is required the administrator also determines:

- the complexity of the AppConnect passcode (such as length, whether it must be numeric or alphanumeric, and how many complex characters are required)
- the auto-lock time for the AppConnect passcode, after which the device user must re-enter the passcode
- passcode history rules
- passcode age rules
- whether to block or retire the AppConnect app when the maximum number of failed attempts exceeds the value set in the AppConnect policy (The options to choose to block or retire are not available on MobileIron Cloud)
- whether the device user can reset the passcode
- passcode strength rules (not available on MobileIron Cloud)
- whether the device user can also use a fingerprint as a convenience to access secure apps

The AppConnect wrapper, along with the Secure Apps Manager and the MobileIron client app, manages the AppConnect passcode. **No additional app development is necessary.**

Although not common, some apps require that certain screens are not interrupted by the auto-lock timeout. See [Ignoring the auto-lock time](#).



AppTunnel with HTTP/S tunneling

Using MobileIron's AppTunnel feature, a secure enterprise app can securely tunnel HTTP and HTTPS network connections from the app to servers behind a company's firewall. A Standalone Sentry is necessary to support AppTunnel with HTTP/S tunneling. The server administrator handles all HTTP/S tunneling configuration on the server Admin Portal. Once configured, the AppConnect app wrapper, the MobileIron client app, the Secure Apps Manager, and a Standalone Sentry handle tunneling for the app.

Supported APIs

An AppConnect app can use HTTP/HTTPS tunneling *only if* the app accesses the enterprise server using one of the following APIs:

- `java.net.HttpURLConnection`
- `java.net.ssl.HttpsURLConnection`
- `Android HttpClient`
- `DefaultHttpClient`, using the standard Apache `HttpClient` library with the `org.apache.http` package name

HTTP/S tunneling is not supported with non-standard libraries such as the Apache `HttpClient` library repackaged under the `ch.boye.httpclientandroidlib` package.

- `OkHttpClient` version 2.5 or less when using Generation 1 wrapping
 - Generation 1 wrapping replaces all `OkHttp` classes with wrapped `OKHttp` version 2.5 classes. Therefore, the app can have issues if it uses any `OkHttp` classes or methods in versions newer than 2.5.
 - Generation 2 wrapping does not support HTTP/HTTPS tunneling with any version of `OKHttp`. However, you can use AppTunnel with TCP tunneling when using Generation 2 wrapping.
- `ModernHttpClient` in apps built with the Xamarin development platform

Use these APIs as you normally would. Whether the server administrator has configured tunneling for the app does not impact how you use these APIs.

NOTE: AppTunnel with HTTP/S tunneling is not supported for Phonegap or React Native apps, because these apps do not use the supported networking APIs.

Inform the server administrator that your app requires AppTunnel with HTTP/S tunneling, including information about the enterprise server that it accesses. The administrator requires this information to correctly configure AppTunnel on the server for your app.

Related topics

[AppTunnel with TCP tunneling](#)



HTTP/S redirects

If a server redirects an HTTP/S request (tunneled or not) to another URL, if the URL matches an AppTunnel rule, the request is tunneled only if the wrapped app uses the class `DefaultHttpClient`.

If the app uses other APIs that support HTTP/S tunneling, redirected requests are **not** tunneled.

HelloAppTunnel sample app

A sample app called HelloAppTunnel demonstrates using each of the APIs in [Supported APIs](#).

Related topics

[Sample apps](#), [tester app](#), and [Cordova plugin](#)

AppTunnel with TCP tunneling

AppTunnel can tunnel TCP traffic between an app and a server behind the company's firewall, securing the data-in-motion. A Standalone Sentry is necessary to support AppTunnel with TCP tunneling. Also, support for AppTunnel with TCP tunneling requires wrapping the app with the Generation 2 wrapper.

NOTE: UDP tunneling is not supported.

Inform the server administrator that your app requires AppTunnel with TCP tunneling, including information about the enterprise server that it accesses. The administrator requires this information to correctly configure AppTunnel with TCP tunneling for your app on the MobileIron server. Once configured, the AppConnect wrapper, the Secure Apps Manager, and the MobileIron client app, manage TCP tunneling. **No additional app development is necessary.**

When to use AppTunnel with HTTP/S tunneling versus TCP tunneling

AppTunnel with TCP tunneling, rather than AppTunnel with HTTP/S tunneling, is required to secure data-in-motion for:

- Java apps that use C or C++ code to access an enterprise server
- Java apps that use APIs outside of the specific set of HTTP/S APIs that AppTunnel with TCP tunneling supports.

You can also use AppTunnel with TCP tunneling with Java apps that *do* use the HTTP/S APIs that AppTunnel with HTTP/S tunneling supports. However, AppTunnel with TCP tunneling is not necessary for such apps, since AppTunnel with HTTP/S tunneling is supported.

- Xamarin apps that use APIs other than `ModernHttpClient`.
- Hybrid web apps, including PhoneGap apps



These apps use Android WebView and WebKit technologies to access and display web content. Because WebView does not use one of the HTTP/S APIs that AppTunnel with HTTP/S tunneling supports, AppTunnel with TCP tunneling is required.

- React Native apps

Because React Native apps do not use one of the HTTP/S APIs that AppTunnel with HTTP/S tunneling supports, AppTunnel with TCP tunneling is required.

NOTE: AppTunnel with TCP tunneling does not support Kerberos authentication to the enterprise server. It supports only pass through authentication. With pass through authentication, the Standalone Sentry passes the authentication credentials, such as the user ID and password (basic authentication) or NTLM, to the enterprise server. Therefore, apps that must use AppTunnel with TCP tunneling, such as hybrid apps, cannot use Kerberos authentication to the enterprise server. However, these apps can use [Certificate authentication with AppTunnel with TCP tunneling](#).

The following table shows whether to use AppTunnel with HTTP/S tunneling or TCP tunneling with an Android secure app depending on the code that is making the network connection. It also shows which generation of the wrapper you can use.

TABLE 2. APPTUNNEL WITH HTTP/S OR TCP TUNNELING USE DEPENDING ON CODE TYPE

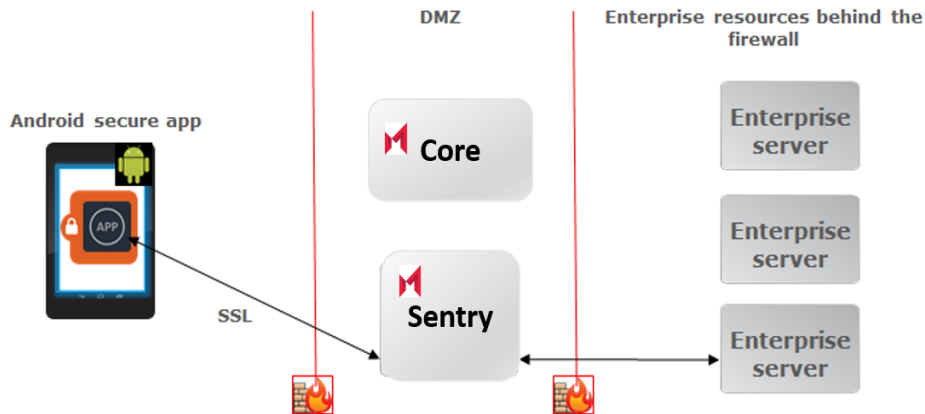
Code type	AppTunnel with HTTP/S tunneling	AppTunnel with TCP tunneling
Java code using supported HTTP/S APIs	Supported with: <ul style="list-style-type: none"> • Generation 1 wrapper • Generation 2 wrapper 	Supported with Generation 2 wrapper
Java code using unsupported HTTP/S APIs	Not supported	Supported with Generation 2 wrapper
Xamarin apps using supported HTTP/S APIs	Supported with Generation 2 wrapper	Supported with Generation 2 wrapper
Xamarin apps using unsupported HTTP/S APIs	Not supported	Supported with Generation 2 wrapper
C or C++ code	Not supported	Supported with Generation 2 wrapper
Hybrid web app, including Phonegap	Not supported	Supported with Generation 2 wrapper
React Native app	Not supported	Supported with Generation 2 wrapper

SSL between the device and Sentry

When an app uses AppTunnel with TCP tunneling, the traffic between the device and the Standalone Sentry is secured using an Secure Sockets Layer (SSL) session, as shown in the following diagram.



FIGURE 2. APPTUNNEL WITH TCP TUNNELING



Certificate authentication with AppTunnel with TCP tunneling

Secure Apps for Android supports certificate authentication with AppTunnel with TCP tunneling. An app that uses AppTunnel with TCP tunneling can send a certificate to identify and authenticate the app user to an enterprise server. Depending on the server implementation, this authentication occurs without interaction from the device user beyond entering the AppConnect passcode, if one is required. That is, the device user does not need to enter a user name and password to log into enterprise services. Therefore, this feature provides a higher level of security and an improved user experience.

This feature is supported only with AppTunnel with TCP tunneling, not with AppTunnel with HTTP/S tunneling.

This feature is not available with MobileIron Cloud.

Inform the MobileIron server administrator that your app requires certificate authentication with AppTunnel with TCP tunneling. The administrator requires this information to correctly configure the feature for your app on the server.

To use this feature, the enterprise server must use client certificate authentication with Secure Sockets Layer (SSL).

App requirements

No additional app development is necessary.

However, the feature is supported only if the app:

- is wrapped with the Generation 2 wrapper.
- initiates a connection that does not use Secure Socket Layer (SSL) to the enterprise server. For example, the app can initiate the connection with a HTTP request, but not with an HTTPS request.

IMPORTANT: The connection that this feature makes to the enterprise server is secure; it uses SSL.

Data loss prevention settings

A MobileIron server administrator specifies on the server Admin Portal the data loss prevention settings for AppConnect apps. Data loss prevention settings specify whether the following features are allowed:

- screen capture
- copy/paste
- camera access
- gallery access
- media player access
- unsecured browser access
- access Web@Work from links in apps that are not AppConnect-enabled

The administrator applies the appropriate settings to a set of devices. Different sets of devices can have different data loss prevention settings.

The AppConnect app wrapper enforces the data loss prevention settings in the app. That is, depending on the server configuration, the app wrapper disables or enables the features. **No app development is necessary.**

Supported file sizes for streaming media

The data loss prevention setting for media player access enables or disables the app's media streaming capability. If the setting is enabled, the app can stream MP3 audio files, WAV audio files, and MP4 video files to media players.

However, when streaming media files from the secure app to a media player, the supported file size depends on:

- the device specifications
- the Android version on the device
- the apps running concurrently on the device

IMPORTANT: MobileIron recommends that you perform tests to profile app performance based on the device, Android version, concurrently running apps, and media file size.



App whitelist

Administrators can configure a whitelist of non-AppConnect apps to open from links in AppConnect apps. To enable the feature, configure a key-value pair in the Secure Apps Manager app configuration.

Configuring the whitelist allows users to choose a non-AppConnect app, such as WebEx, GotoMeeting, to open from an AppConnect app such as Email+. Device users can choose from compatible AppConnect apps and non-AppConnect apps configured in the whitelist that are installed on the device.

No app development is necessary.

For information on configuring the whitelist, see "App whitelist" in the *MobileIron AppConnect Guide for Core*

Handling app-specific configuration from the MobileIron server

Handling app-specific configuration from the MobileIron server requires some application development before wrapping the app. If you do not use this feature, develop your app to set up its configuration as you typically would. For example, set up the app to prompt the device user for configuration settings.

You determine the app-specific configuration that your app requires from the MobileIron server. Examples are:

- the address of a server that the app interacts with
- whether particular features of the app are enabled for the user
- user-related information from LDAP, such as the user's ID and password
- certificates for authenticating the user to the server that the app interacts with

Each configurable item is a key-value pair. Each key and value is a string. A server administrator specifies the key-value pairs on the server for each app. The administrator applies the appropriate set of key-value pairs to a set of devices. Sometimes more than one set of key-value pairs exists on the server for an app if different users require different configurations. For example, the administrator can assign a different server address to users in Europe than to users in the United States.

NOTE: When the value is a certificate, the value contains the base64-encoded contents of the certificate, which is a SCEP or PKCS-12 certificate. If the certificate is password encoded, the server automatically sends another key-value pair. The key's name is the string `<name of key for certificate>_MI_CERT_PW`. The value is the certificate's password.

For more information on implementing this feature in your app, see [Handling AppConnect app-specific configuration](#).



Ignoring the auto-lock time

A MobileIron server administrator can specify that a particular secure app is allowed to ignore the auto-lock time. Ignoring the auto-lock time is important for apps in which staying on a screen is critical. The auto-lock time specifies the length of a period of inactivity. After this period of inactivity, the device user is prompted to reenter his secure apps passcode to continue accessing secure apps. This interruption to the app is sometimes not acceptable.

For example, in a navigation app, the device user taps the screen only infrequently, but the screen must continue displaying. Therefore, the app is designed to ignore the Android screen timeout setting, which turns off the screen after a period of time.

Such an app also requires that when the auto-lock time expires, the app's screen continues displaying. The normal behavior of having the Secure Apps Manager prompt for the secure apps passcode is not compatible with the app's functionality.

MobileIron server configuration

The MobileIron server administrator configures whether an app is allowed to ignore the auto-lock time in the app's app-specific configuration on the server's Admin Portal. The administrator creates a special key-value pair that turns on the feature. The key is `AC_IGNORE_AUTO_LOCK_ALLOWED` with the value to `true`.

NOTE: The AppConnect wrapper around your app handles this key-value pair. Your app does not receive the key-value pair.

App requirements

If the administrator allows your app to ignore the auto-lock time, a screen continues to display uninterrupted only if one of the following are true:

- The app has set the `KEEP_SCREEN_ON` flag to `true` for the `android.view` object.
- The `android:keep_screen_on` element is set to `true` in the app's layout XML file.
- The app has set the `FLAG_KEEP_SCREEN_ON` flag to `true` in the `android.view.WindowManager.LayoutParams` class.

IMPORTANT: Your app's documentation must indicate that it requires the MobileIron server administrator to allow your app to ignore the auto-lock time. Explain to the administrator the compelling reasons for ignoring the auto-lock time.



Wrapping technology

AppConnect apps are built using MobileIron's AppConnect wrapping technology. This technology secures the app from leaking data outside the secure container.

You can create a secure app with minimal application development in many cases. Some development is sometimes necessary to use AppTunnel with HTTP/S tunneling, depending on the APIs the app uses to access enterprise servers. With some APIs, no development is necessary. Also, with some straightforward additional development, an app can receive app-specific configuration from the MobileIron server.

IMPORTANT: Wrapping does not support all Android coding capabilities. Before submitting an app for wrapping, see [Capabilities and limitations of apps you can wrap](#)

AppConnect wrapping does the following:

1. Examines an app's APK file for operating system calls that impact security.
2. Replaces these calls with secure AppConnect calls.
3. Generates a replacement APK file.

The resulting AppConnect app:

- can run only if the MobileIron server administrator has authorized the app to run on the device.
- ensures that a user logs in with his AppConnect passcode before using the AppConnect app, if the server administrator requires an AppConnect passcode.
- overlays the app's icon with a small badge.
Device users can have both AppConnect apps and regular, unsecured apps on their devices. This small badge indicates to the user that the app is a secure app.
The badge for wrapped apps for use with the Secure Apps Manager is different than the badge for wrapped apps for use with the Secure Apps Manager for AppStation.
- shares data with only other AppConnect apps.
Exceptions are described in [Accessible Apps to preserve the user experience](#).
- enforces data loss prevention settings, depending on the MobileIron server policy.
- supports receiving app configuration from the MobileIron server.
- supports AppTunnel with HTTP/S tunneling.
- supports AppTunnel with TCP tunneling when using the Generation 2 mode of the wrapper
- supports certificate authentication to the enterprise server when using the Generation 2 mode of the wrapper (not available with MobileIron Cloud)



- encrypts and decrypts data at runtime.

NOTE: File names are not encrypted.

- remembers the encryption key when running in the background, even when the device user is not logged in to AppConnect apps.

Email apps, for example, run in the background to synchronize data with the email server. To successfully access their data, these apps require the encryption key. AppConnect wrapping ensures the key is available in the app's memory.

NOTE: Device users must still login with their AppConnect passcodes to access the app, if the MobileIron server administrator requires an AppConnect passcode.

- supports scoped storage when using the Generation 2 mode of the wrapper.



Handling AppConnect app-specific configuration

IMPORTANT:

- This mechanism for handling AppConnect app-specific configuration was added with AppConnect 8.6.0.0 for Android.
- The previous (legacy) mechanism has been deprecated, but is still supported.
- If your app uses the legacy mechanism, modify your app to use this mechanism as soon as possible.
- If you are adding app-specific configuration handling to your app for the first time, use this mechanism.

For information about AppConnect app-specific configuration, see:

- [Overview of configuration handling](#)
- [App-specific configuration data flow](#)
- [Tasks for configuration handling](#)
- [Sample Java app for handling app-specific configuration](#)
- [Best practices for handling app-specific configuration](#)
- [Testing app-specific configuration handling](#)

Overview of configuration handling

The MobileIron server administrator can set up app-specific configuration on the server for AppConnect for Android apps. This configuration is in the form of key-value pairs. Your app can receive these key-value pairs. Specifically, when you implement configuration handling in your app, your app:

- requests the current configuration when it first runs.
Your app then receives an asynchronous response containing the key-value pairs.
- receives updates to the configuration.

Java developers

MobileIron provides a sample AppConnect app called HelloAppConnect-newAPI that implements configuration handling. You can use this sample app's code as a starting point for your own.



Phonegap developers

You can implement app-specific configuration in a Phonegap app by using a MobileIron-provided Cordova plugin. This plugin provides the necessary APIs to receive the app-specific configuration from the MobileIron server. MobileIron provides the following:

- AppConnectCordovaConfigPlugin-w.x.y.z.zip, the Cordova plugin. (w.x.y.z corresponds to the AppConnect version for Android)
See README.md in the ZIP file for information on using the plugin.
- A sample Phonegap app that uses the plugin, available as a starting point for your own app.

React Native developers

You can implement app-specific configuration in a React Native app by using MobileIron-provided files that make up a React Native package called ConfigServicePackage. The files provide the necessary APIs to receive the app-specific configuration from the MobileIron server. MobileIron provides a sample React Native app called HelloReact that includes:

- all files relating to getting app-specific configuration
- a README.txt with instructions for using the files
- sample code for using the files

Related topics

- [Sample Java app for handling app-specific configuration.](#)
- [Sample apps, tester app, and Cordova plugin](#)
- [HelloReact Demo sample app overview](#)

App-specific configuration data flow

The MobileIron server passes the app-specific configuration to the MobileIron client app (Mobile@Work when using MobileIron Core or Connected Cloud, and MobileIron Go when using MobileIron Cloud). The MobileIron client app in turn passes the configuration to the AppConnect wrapper around your app, which passes it to your app.

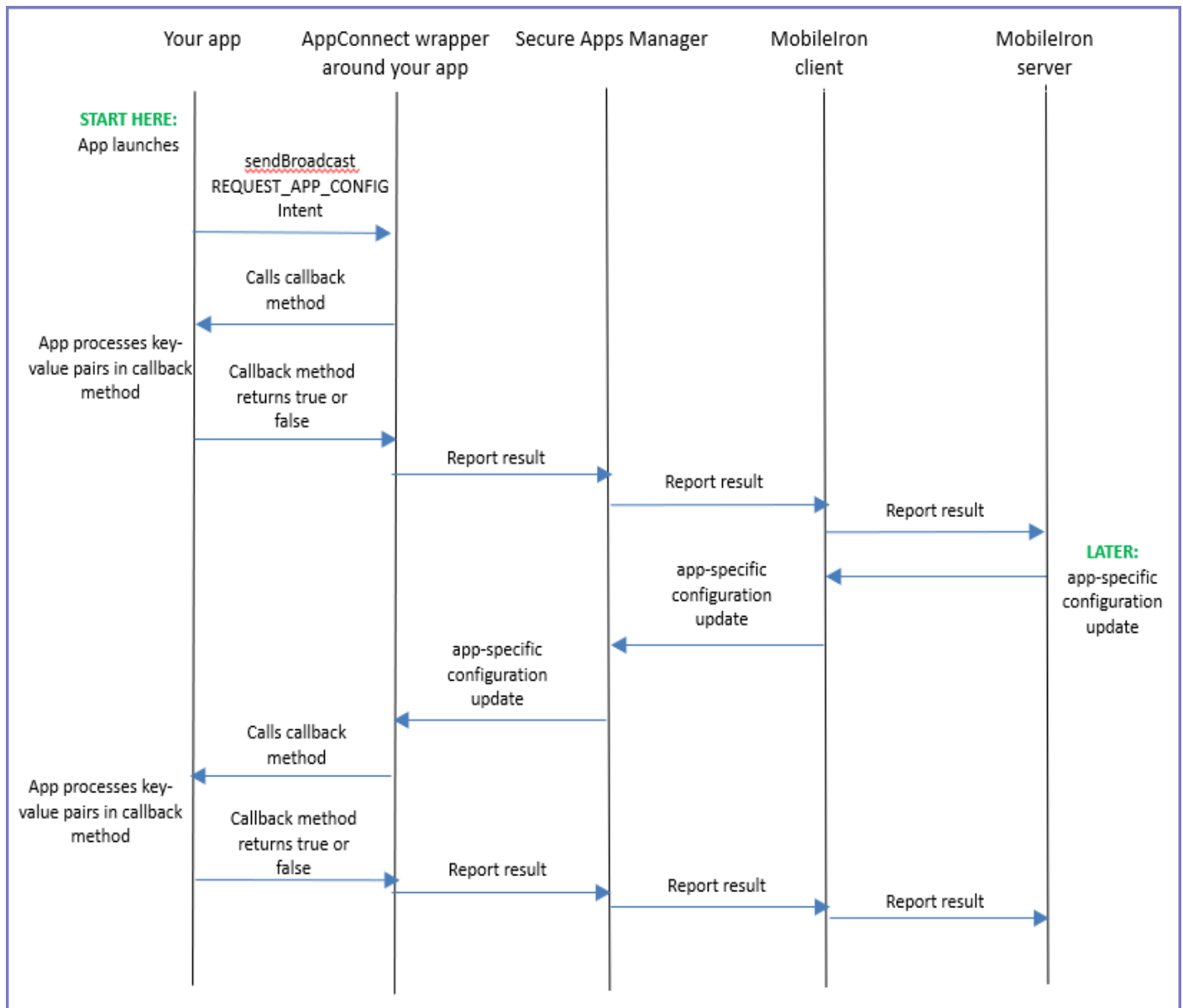
The following sequence diagram shows the flow of data between the MobileIron server, the MobileIron client app, the Secure Apps Manager, the AppConnect wrapper around your app, and your app. It shows the sequence when:

- Your app launches
- The MobileIron server administrator has updated the app-specific configuration on the server.

Before the sequence begins, when the app was installed, the AppConnect wrapper received the app-specific configuration from the MobileIron server, passed to the wrapper from the MobileIron client and the Secure Apps Manager.



FIGURE 3. APP-SPECIFIC CONFIGURATION DATA FLOW



Tasks for configuration handling

The code you add to your app to receive app-specific configuration is simple because the AppConnect wrapper around your app and the MobileIron client do most of the work. Your focus is in applying the configuration to your app according to your requirements.

To handle app-specific configuration in your app, do the following high-level tasks:

- [Check at runtime if your app is wrapped.](#)

This check is typically necessary if you are a third-party app developer using the same source code to create a Google Play app and an in-house AppConnect app. Only wrapped AppConnect apps can receive app-specific configuration from a MobileIron server.

If you are developing an app that will be distributed only as an in-house app, not from Google Play, you will not use this check.

- [Create a callback method to receive configuration updates](#)

You provide a callback method that receives app-specific configuration updates.

IMPORTANT: The callback thread runs on the main thread.

- [Request the configuration when your app starts.](#)

When your app starts, request the app-specific configuration, which your app will receive asynchronously in the callback method you provide.

- [Add callback information to AndroidManifest.xml](#)

You provide a callback method that receives app-specific configuration updates. Add information about your callback method to your app's AndroidManifest.xml file. You add this information as a `<meta-data>` element in your `<application>` element.

- [Specify app configuration and policies in .properties files.](#)

You can include .properties files in your app that list your app's key-value pairs and data loss prevention (DLP) policies. When the MobileIron server administrator uploads your app to the server, these files cause the server to automatically configure the key-value pairs and DLP policies.

If your app uses the legacy method for app-specific configuration handling, a summary of what to do to use the new method is in [Changing from the legacy configuration handling to the new mechanism](#).

Check at runtime if your app is wrapped

If you are a third-party developer, you sometimes develop an app in which the same source code is used in these ways:

- as a wrapped app distributed from the MobileIron server's App Catalog
This secure AppConnect app is for enterprise device users.
- as an unwrapped app distributed from Google Play
This unsecured app is for general distribution.

An app that serves both these markets typically behaves differently depending on whether it is a wrapped, secure AppConnect app.

For example:



- If a wrapped app expects key-value pairs from the MobileIron server, but does not receive the expected pairs or valid values, it should take appropriate actions.
As a best practice, if your app expects a login ID from the server, but does not receive one, do not allow the device user to enter the ID manually. See [Best practices for handling app-specific configuration](#).
- If an app is not wrapped, it cannot get its configuration from the MobileIron server. It gets configurable information another way, such as prompting the device user to enter it.
For example, the unwrapped app prompts the user to enter a login ID.

To determine at runtime whether the app is running as a wrapped app, check this Android system property:

```
"com.mobileiron.wrapped"
```

For example, use the following expression:

```
Boolean.parseBoolean(System.getProperty("com.mobileiron.wrapped", "false"))
```

The expression returns `true` if the app is wrapped. Otherwise, it returns `false`.

Create a callback method to receive configuration updates

Add a callback method to your app that receives the app-specific configuration when:

- the app requests the app-specific configuration after the app launches.
- the administrator has made updates to the app-specific configuration on the MobileIron server.

IMPORTANT: The callback method runs on the app's main thread.

Implementing the callback method

Create a callback method with the name of your choice in a class of your choice.

Example

```
public class AppConfigCallback {

    public boolean onConfigReceived(Context context, Bundle config) {
        // Extract the key-value pairs from the Bundle object. For example:
        Map<String, String> map = new HashMap<String, String>();
        for (String key:config.keySet()) {
            map.put(key, config.getString(key));
        }
        // Process the key-value pairs according to your app's requirements.
        // return true if app successfully processes the key-value pairs; otherwise false.
    }
}
```



Parameters of callback method

- **Context**
Receives the application context.
- **Bundle**
Bundle object that receives the app-specific configuration key-value pairs
If no key-value pairs are configured on the MobileIron server, calling the Bundle object's `keyset()` method returns an empty set.

Return value of callback method

boolean

Return true if the app successfully processed the configuration's key-value pairs.

Return false if your app failed to successfully process the key-value pairs. Some reasons for returning false are:

- A value is not valid for its key.
For example, if the key is "emailAddress", but the value does not include the @ character, return false.
- A value is empty.
Typically, if a key is included in the MobileIron server configuration for your app, your app expects a value. If the MobileIron server administrator did not enter a value, return false.
- Your app encounters a system error while processing a key-value pair.
Your app determines whether a system error impacts key-value processing to warrant returning false.

When the callback method returns false, how your app continues to operate depends on your app's design and requirements.

Request the configuration when your app starts

When your app starts, request the app-specific configuration. To request it, create an Intent object with the action "com.mobileiron.appconnect.action.REQUEST_APP_CONFIG"., and pass it to `sendBroadcast()`.

For example, in HelloAppConnect:

```
public class AppConfigCallback {

    private static final String ACTION_REQUEST_APP_CONFIG =
        "com.mobileiron.appconnect.action.REQUEST_APP_CONFIG";

    public static void requestConfig(Context context) {
        Intent intent = new Intent(ACTION_REQUEST_APP_CONFIG);
        context.sendBroadcast(intent);
    }
}
```

This code results in an asynchronous call to your callback method that handles app-specific configuration.



Although the app can request the configuration at any time, typically the app requests it only once, when the app launches. After that, whenever the MobileIron server administrator updates the configuration on the server, your callback method is automatically called.

Add callback information to AndroidManifest.xml

Add a `<meta-data>` element to your app's `AndroidManifest.xml` file as a child of your `<application>` element. The `<meta-data>` element contains name-value pairs that specify your callback class and method for handling app-specific configuration updates from the MobileIron server.

The `<meta-data>` element looks like this:

```
<application>
  <meta-data
    android:name="com.mobileiron.appconnect.config.callback_class"
    android:value="<fully-qualified-class-name>" />
  <meta-data
    android:name="com.mobileiron.appconnect.config.callback_method"
    android:value="<callback method>" />
</application>
```

For example, in `HelloAppConnect`, the lines in `AndroidManifest.xml` are:

```
<application>
  <meta-data
    android:name="com.mobileiron.appconnect.config.callback_class"
    android:value="com.mobileiron.helloappconnect.AppConfigCallback" />

  <meta-data
    android:name="com.mobileiron.appconnect.config.callback_method"
    android:value="onConfigReceived" />
</application>
```

Specify app configuration and policies in .properties files

You can include the following `.properties` files with your app:

- `appconnectconfig.properties`
This file specifies your app's configuration keys and their default values, if any. Providing this `.properties` file causes the MobileIron server to automatically configure the keys and their default values on the server.
- `appconnectpolicy.properties`
This file specifies the default data loss prevention policy for screen capture for the app. Specifically, it specifies whether screen capture is allowed in the app. The policy is enforced by the AppConnect wrapping technology.

If your app contains these `.properties` files, the MobileIron server automatically configures the key-value pairs and the screen capture policy that you specified. This automatic configuration occurs when the MobileIron server administrator uploads your app to the server's App Catalog.



The administrator can then change the default values on the server as necessary for that enterprise.

File location of the .properties files

Put the .properties files in this directory in your app:

<application root directory>/res/raw

Example of the appconnectconfig.properties file

An example of an appconnectconfig.properties file is available in HelloAppConnect.

It contains the following:

```
# This sample appconnectconfig.properties file uses rules found at
# http://en.wikipedia.org/wiki/.properties.

server=www.myCompanyApplicationServer.com
port=8080

# In the following example, the resulting property value contains only single spaces.
# It contains no other whitespace.
# Therefore, the value is: "I'm also demonstrating a multi-line property!"

name\ with\ spaces:I'm also demonstrating \
a multi-line property!

# Use an empty value for keys that have no default value.

nodefault=

! You can also start comments with exclamation marks.

# You can use these MobileIron Core variables for values:
# $USERID$, $EMAIL$, $PASSWORD$,
# $USER_CUSTOM1$, $USER_CUSTOM2$, $USER_CUSTOM3$, $USER_CUSTOM4$

# You can use these MobileIron Cloud variables for values:
# ${userID}, ${userEmailAddress}
# ${USER_CUSTOM1}, ${USER_CUSTOM2}, ${USER_CUSTOM3}, ${USER_CUSTOM4}

userid=$USERID$
email=$EMAIL$
user_custom1=$USER_CUSTOM1$
combined=$USERID$::$EMAIL$
```



Format of the appconnectconfig.properties file

Use the rules for well-formed Java property files given in the Java Properties class. For example, use the characters = or : or a space to separate the key from the value. Use \ before each of these characters if the character is part of the key.

The values that you specify are the default values for the key. If the value has no default, leave the value empty.

A value can be any string. The value can also use one of the following server variables:

TABLE 3. SERVER VARIABLES IN DEFAULT VALUES OF KEYS

MobileIron Core variable	MobileIron Cloud variable	Description
\$USERID\$	\${userID}	The device user's enterprise user ID, typically an LDAP ID.
\$PASSWORD\$	Not available	The device user's enterprise user password, typically an LDAP password.
\$EMAIL\$	\${userEmailAddress}	The device user's enterprise email address.
\$USER_CUSTOM1\$ \$USER_CUSTOM2\$ \$USER_CUSTOM3\$ \$USER_CUSTOM4\$	\${USER_CUSTOM1} \${USER_CUSTOM2} \${USER_CUSTOM3} \${USER_CUSTOM4}	Custom variables that the MobileIron server administrator sets up. Only use these variables if you are developing an app for a specific MobileIron customer. Contact the server administrator to determine the values of these variables.

You can also specify values that are combinations of text and server variables. For example, using MobileIron Core variables:

- \$USERID\$::\$EMAIL\$
- \$USERID\$@somedomain.com

Use server variables for default values in your appconnectconfig.properties only if you know what kind of server (MobileIron Core or MobileIron Cloud) your app will be used with. If you don't know, leave the value empty. The server administrator will fill in the value.

Example of the appconnectpolicy.properties file

An example of an appconnectpolicy.properties file is available in HelloAppConnect.

It contains the following:

```
# A sample appconnectpolicy.properties file
screencapture=disable
```

Format of the appconnectpolicy.properties file

To disable screen capture in the app, include the following line in appconnectpolicy.properties:

```
screencapture=disable
```



To allow screen capture:

```
screencapture=allow
```

Changing from the legacy configuration handling to the new mechanism

If you used the legacy mechanism for app-specific configuration handling, change your app to use the new mechanism.

To change your app to use the new mechanism, do the following:

- In the legacy method, you created a class that extends `IntentService()`, and implemented `onHandleConfig()` to handle the received `Intent` object with the action `"com.mobileiron.HANDLE_CONFIG"`. Replace this class with a class of your choice with a callback method with a name of your choice. Move the code from your legacy `onHandleConfig()` to your new callback method. See [Create a callback method to receive configuration updates](#).
- In the legacy method, when your app starts, it requests the app-specific configuration by calling `startService()` with an `Intent` object with the action `"com.mobileiron.REQUEST_CONFIG"`. Replace this code with a call to `sendBroadcast()` with an `Intent` object with the action `"com.mobileiron.appconnect.action.REQUEST_APP_CONFIG"`. See [Request the configuration when your app starts](#).
- In the legacy method, you added information to the app's `AndroidManifest.xml` file to add a service for handling configuration intents. Remove the `<service>` element. Add the information about your callback method. See [Add callback information to AndroidManifest.xml](#).

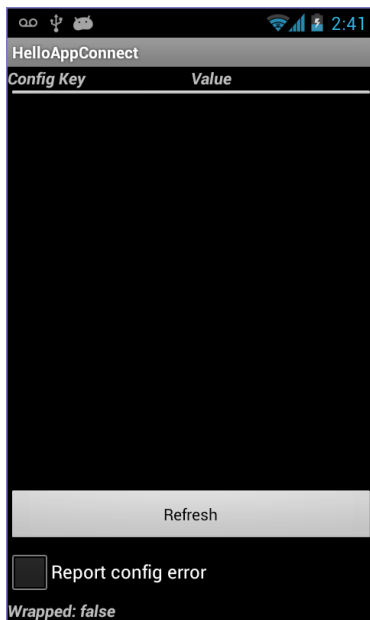
Sample Java app for handling app-specific configuration

MobileIron provides a sample Java app, `HelloAppConnect-newAPI`, which handles app-specific configuration. You can use the code from this app as a starting point for your app's configuration handling.

`HelloAppConnect-newAPI` displays this screen:



FIGURE 4. HELLOAPPCONNECT-NEWAPI SCREEN



The HelloAppConnect-newAPI app:

- Calls `sendBroadcast()` when you tap **Refresh**, passing it an Intent object with the action `"com.mobileiron.appconnect.action.REQUEST_APP_CONFIG"`.
- Provides a callback method `onConfigReceived()`.
The callback method displays the received key-value pairs.
- Displays whether the app is wrapped, based on the value of the system property `"com.mobileiron.wrapped"`

Related topics

[Sample apps, tester app, and Cordova plugin](#)

Best practices for handling app-specific configuration

The following are best practices when handling app-specific configuration in your app:

- [Provide documentation about your app to the MobileIron server administrator](#)
- [Use only a login ID from the MobileIron server if one is expected](#)

Provide documentation about your app to the MobileIron server administrator

Document each key and its valid values. Document the default value, if applicable, and document whether the value can be empty. Provide this documentation regardless of whether your app includes an `appconnectconfig.properties` file.

Use only a login ID from the MobileIron server if one is expected

If a wrapped app expects a key-value pair for the device user's login ID, it should not prompt the user to enter the login ID manually. Using only a login ID from the MobileIron server ensures that a user can use the app only if the enterprise has authenticated the user. If the app does not receive an expected valid user ID, display an error message to the device user.

Testing app-specific configuration handling

To test app-specific configuration handling, use MobileIron Core. If you are an in-house app developer and your enterprise uses MobileIron Core or Connected Cloud, you can use that Core. Otherwise, request a MobileIron Connected Cloud instance for your testing. Connected Cloud is the cloud offering of the on-premise server MobileIron Core. You then use a web portal called the Admin Portal to make configuration changes necessary for testing your app. All AppConnect apps require Mobile@Work on the device to interact with Core.

IMPORTANT: You must wrap your app before testing it with MobileIron Core.

NOTE: Apps that you test with MobileIron Connected Cloud and Mobile@Work will also work with MobileIron Cloud and supported versions of MobileIron Go. However, some AppConnect features are not supported by MobileIron Cloud and MobileIron Go.

The steps for testing app-specific configuration handling in your app are:

1. [Requesting a MobileIron Core test instance](#)
2. [Downloading Mobile@Work to the device](#)
3. [Logging in to the Admin Portal](#)
4. [Creating a label for testing your app](#)
5. [Enabling AppConnect on MobileIron Core](#)
6. [Configuring the AppConnect global policy](#)
7. [Uploading the Secure Apps Manager to MobileIron Core](#)
8. [Uploading your AppConnect app to MobileIron Core](#)
9. [Configuring app-specific settings in MobileIron Core](#)
10. [Registering the test device to MobileIron Core](#)
11. [Pushing Core configuration changes to the device](#)

Requesting a MobileIron Core test instance

To request a MobileIron Core test instance if one is not available to you, go to [Request and Setup a MobileIron Test Instance: MobileIron Core / Cloud / Sentry](#).



Downloading Mobile@Work to the device

Download Mobile@Work to the device from Google Play.

Logging in to the Admin Portal

If you requested a MobileIron Connected Cloud instance, MobileIron provides you with the following information about your test MobileIron Core:

- the URL for accessing the Core's Admin Portal
The Admin Portal is a web portal for configuring Core. The URL has the format:
`https://m.mobileiron.net/<app partner name>`
- a user ID and password for accessing the Admin Portal
You also use this user ID to register a device with Core.
- a port number for Core, used when you register a device with Core.
The port number is typically four or five digits.

If you are using your enterprise's MobileIron Core or Connected Cloud, get the URL, user ID and password, and port number (if using your enterprise's Connected Cloud) from your Core administrators. They might give you a separate user ID and password to use to register a device.

Procedure

1. Open a browser to the URL for accessing the Core's Admin Portal.
Use the URL of your test Core, appended with /mifs.
Connected Cloud example:
`https://m.mobileiron.net/myCompany/mifs`
On-premise Core example:
`https://mycore.mycompany.com/mifs`
2. Enter your Username and Password.
3. Click Sign In.
You are now in the Admin Portal.
When using a test instance from MobileIron, change your password when prompted.

Creating a label for testing your app

MobileIron Core uses labels to associate policies and apps with devices. For testing your app, create a new label so that your testing impacts only your test device.



Procedure

1. In the Admin Portal, go to **Devices & Users > Labels**.
2. Click **Add Label**.
3. Enter a name for the label.
For example: AppConnect Test
4. Enter a description.
For example: Use only for devices testing new AppConnect apps.
5. Select **Manual** for the **Type**.
6. Click **Save**.

Enabling AppConnect on MobileIron Core

To test your AppConnect app, ensure that AppConnect is enabled on MobileIron Core.

Procedure

1. In the Admin Portal, go to **Settings**.
2. Select **Additional Products > Licensed Products**.
3. Select **AppConnect For Third-party And In-house Apps** if it is not already selected.
4. Click **Save**.

Configuring the AppConnect global policy

Using AppConnect for Android requires that you configure an AppConnect global policy.

Procedure

1. In the Admin Portal, select **Policies & Configs > Policies**.
2. Select **Add New > AppConnect**.
3. Set the **AppConnect** field to **Enabled**.
4. Complete the form.
Most fields default to suitable values.
5. In the **Security Policies** section, select **Authorize** for the field **Apps without an AppConnect container policy**.
6. Click **Save**.
7. Select the policy.
8. Select **More Actions > Apply To Label**.



9. Select the label to which you want to apply this policy.
10. Click **Apply**.

Uploading the Secure Apps Manager to MobileIron Core

The Secure Apps Manager is required for running AppConnect apps. You upload the Secure Apps Manager to MobileIron Core as an in-house app. After uploading the Secure Apps Manager, you distribute it to your test device by applying the app to the label for your test device.

Procedure

1. Download the Secure Apps Manager app to your computer from help.mobileiron.com in **Software > Downloads**.
2. In the MobileIron Core Admin Portal, go to **Apps > App Catalog > Add+ > In-House**.
3. Click **Browse** and browse to the Secure Apps Manager.
4. Click **Next**.
5. Optionally make selections, clicking **Next**, and then **Finish**.
6. Select the Secure Apps Manager entry on the **Apps > Apps Catalog** screen.
7. Click **Actions > Apply to Labels**.
8. Select the appropriate label and click **Apply**.

Uploading your AppConnect app to MobileIron Core

You upload your Android AppConnect app to MobileIron Core as an in-house app. After uploading the app, you distribute the app to your test device by applying the app to the label for your test device.

IMPORTANT: Wrap your app before uploading it to MobileIron Core.

Procedure

1. In the MobileIron Core Admin Portal, go to **Apps > App Catalog > Add+ > In-House**.
2. Click **Browse** and browse to the AppConnect app.
3. Click **Next**.
4. Optionally make selections, clicking **Next**, and then **Finish**.
5. Select the app entry on the **Apps > Apps Catalog** screen.
6. Click **Actions > Apply to Labels**.
7. Select the appropriate label and click **Apply**.

Configuring app-specific settings in MobileIron Core

Using an AppConnect app configuration, you can configure settings that are specific to your AppConnect app. The configuration uses key-value pairs.



MobileIron Core automatically creates an AppConnect app configuration for the Android AppConnect app when you upload the wrapped app to the App Catalog. Core keeps in sync the labels that you apply to the app and the labels that you apply to the automatically-created AppConnect app configuration.

Procedure

1. In the Admin Portal, select **Policy & Configs > Configurations**.
2. Select the automatically-created AppConnect app configuration for the app.
3. Click **Edit**.
4. To add a key-value pair:
 - a. Click **Add+**.
 - b. Enter the key name.
 - c. Enter the key value.
5. Click **Save**.

Registering the test device to MobileIron Core

To run your AppConnect app, you must first launch Mobile@Work and follow its instructions to register with MobileIron Core. This procedure will install the Secure Apps Manager and your app on the device.

After you complete the registration procedure, you can run your AppConnect app and test its handling of the app-specific configuration you configured on MobileIron Core.

Pushing Core configuration changes to the device

If you change configuration on MobileIron Core, such as your app-specific key-value pairs, you can make Core send the changes to the device immediately.

Procedure

1. Open Mobile@Work.
2. Navigate to the **Device Status** screen if not already there.
3. Tap the refresh icon in the upper right-hand corner.
4. This action causes the Mobile@Work to check in with Core, which causes Core to deliver any configuration changes to the device.



Sample apps, tester app, and Cordova plugin

MobileIron provides several sample apps that demonstrate the correct way to implement various capabilities in apps to be wrapped. MobileIron also provides an app for testing app-specific configuration handling without a MobileIron server or client. For Phonegap apps, MobileIron provides a Cordova plugin for receiving app-specific configuration.

- [Summary of sample apps, tester app, and Cordova plugin](#)
- [Location of sample apps, tester app, and Cordova plugin](#)
- [Android API Usage Demo sample app overview](#)
- [HelloReact Demo sample app overview](#)
- [HelloFlutter sample app overview](#)

Summary of sample apps, tester app, and Cordova plugin

MobileIron provides the following sample apps, tester app, and Cordova plugin:

TABLE 4. SAMPLE APP, TESTER APP, AND CORDOVA PLUGIN DESCRIPTIONS

Sample app name	Description
HelloAppConnect-newAPI	A Java app that handles app-specific configuration. You can use the code from this app as a starting point for your app's configuration handling. See Sample Java app for handling app-specific configuration .
HelloAppConnect-oldAPI	A Java app that handles app-specific configuration using the legacy method for app-specific configuration handling. However, this mechanism is deprecated.. See Sample Java app for legacy app-specific configuration handling .
HelloAppTunnel	Demonstrates how an app uses the API's that AppTunnel with HTTP/S tunneling supports. See AppTunnel with HTTP/S tunneling .
ApiUsageDemo	Demonstrates the supported way for a wrapped app to use various Android APIs. MobileIron has verified that when you wrap apps that use these APIs as demonstrated, the behavior of the wrapped app is functionally the same as the behavior of the unwrapped app. Also demonstrates the behavior of attempts to share content depending on the value of the MI_AC_SHARED_CONTENT key on the Secure Apps Manager. See Android API Usage Demo sample app overview .



TABLE 4. SAMPLE APP, TESTER APP, AND CORDOVA PLUGIN DESCRIPTIONS (CONT.)

Sample app name	Description
HelloReact	<p>Demonstrates the supported way for a wrapped app built with the React Native development framework to use various React Native modules and components. The app includes how to use the MobileIron-provided files to get app-specification configurations from the MobileIron server.</p> <p>MobileIron has verified that when you wrap React Native apps that use these modules and components as demonstrated, the behavior of the wrapped app is functionally the same as the behavior of the unwrapped app.</p> <p>See HelloReact Demo sample app overview.</p>
HelloCordovaAppConnect and the Cordova plugin	<p>HelloCordovaAppConnect is a Phonegap app that uses the MobileIron-provided Cordova plugin. The app uses the plugin's APIs to get app-specific configuration from the MobileIron server. You can use this app as a starting point for your own app.</p> <p>See the README.md in the plugin's ZIP file for information on using the plugin.</p>
AppConnectTester	<p>If you use the legacy mechanism for handling app-specific configuration, use the AppConnectTester app for testing your app's handling of configuration changes without using the MobileIron server or the MobileIron client app.</p> <p>See App for testing legacy configuration handling.</p>
HelloFlutter	<p>The HelloFlutter sample app demonstrates the supported way for a wrapped app to use various Flutter plugins and components. MobileIron has verified that when you wrap apps that use these APIs as demonstrated, the behavior of the wrapped app is functionally the same as the behavior of the unwrapped app.</p> <p>See HelloFlutter sample app overview.</p>

IMPORTANT: Do not submit the sample apps to MobileIron for wrapping. If you use these apps as a starting part for your own app, be sure to change the package name to something that does not start with com.mobileiron before submitting.

Location of sample apps, tester app, and Cordova plugin

The sample apps, tester app, and Cordova plugin are located at

- <https://support.mobileiron.com/support/CDL.html>
- <https://help.mobileiron.com> in **Software > Downloads**

Look for the heading **AppConnect Sample Apps for Android**. Two ZIP files are provided:

- AndroidAppConnectSampleAndTester-w.x.y.z.n-src.zip, where w.x.y.z.n corresponds to the AppConnect version and build number

This ZIP file contains the source code, resources, manifest, and Eclipse project files for:



- HelloAppConnect-newAPI
- HelloAppConnect-oldAPI
- HelloAppTunnel
- AppConnectTester
- HelloCordovaAppConnect
- ApiUsageDemo
- HelloReact
- AndroidAppConnectSampleAndTesterAPKs-w.x.y.z.zip, where w.x.y.z corresponds to the AppConnect version for Android

This ZIP file contains:

- HelloAppConnect-newAPI (APKs for both the app and the wrapped app)
- HelloAppConnect-oldAPI (APKs for both the app and the wrapped app)
- HelloAppTunnel (APKs for both the app and the wrapped app)
- ApiUsageDemo (APKs for both the app and the wrapped app)
- HelloReact (APKs for both the React Native app and the wrapped React Native app)
- HelloCordovaAppConnect (APKs for both the Phonegap app and the wrapped Phonegap app)
- AppConnectCordovaConfigPlugin, a ZIP file containing the Cordova plugin for receiving app-specific configuration
- AppConnectTester APK

The provided wrapped apps are for use with Secure Apps Manager. They will not work with Secure Apps Manager for AppStation.

Android API Usage Demo sample app overview

A sample app called ApiUsageDemo demonstrates the supported way for a wrapped app to use various Android APIs. MobileIron has verified that when you wrap apps that use these APIs as demonstrated, the behavior of the wrapped app is functionally the same as the behavior of the unwrapped app.

Related topics

- [Location of sample apps, tester app, and Cordova plugin](#)
- [MediaPlayer and MediaMetadataRetriever Internet permission requirement](#)
- [Image selection from outside the AppConnect container](#)



Demonstrated APIs

The sample app ApiUsageDemo includes the following demonstrations:

- [Audio Recorder Demo](#)
- [Capture Image Demo](#)
- [Documents Demo](#)
- [Image Decoder Demo](#)
- [Media Meta Data Retriever Demo](#)
- [Media Player Demo](#)
- [Pick Image Demo](#)
- [Share Content Demo](#)
- [Video Recorder Demo](#)

IMPORTANT: Data loss protection policies on the MobileIron server determine whether wrapped apps have access to the device's camera and gallery. Be sure to allow access to the appropriate capability when testing the ApiUsageDemo app or your own wrapped app's capabilities.

Audio Recorder Demo

The code uses `MediaRecorder` APIs and `MediaPlayer` APIs to record and playback audio media content.

Recording illustrates using the `MediaRecorder` `setOutputFile()` method with:

- a file (absolute path)
- a file descriptor from a `RandomAccessFile`

Wrapping the `MediaPlayer` APIs requires the android Internet permission, described in [MediaPlayer and MediaMetadataRetriever Internet permission requirement](#).

Java files

- `AudioRecorderDemoActivity.java`
- `BaseRecorderDemoActivity.java`

Capture Image Demo

This code uses `FileProvider` APIs to extract a file URI and pass it to the Camera app. The Camera app writes the captured image to that file. The demo app then uses the `ImageView` `setImageURI` API to display the photo.

Allow the Camera DLP on the MobileIron server so that the app can access the camera to take a photo.



Java files

- ImageCaptureDemoActivity.java

Documents Demo

This code demonstrates creating documents and opening folders and documents either inside or outside the AppConnect container. The code uses the intents `ACTION_OPEN_DOCUMENT`, `ACTION_OPEN_DOCUMENT_TREE`, and `ACTION_CREATE_DOCUMENT`. The code illustrates handling text, image, and videos.

Note The Following:

- Accessing files on a USB OTG drive requires a key-value pair on the AppConnect app configuration for Secure Apps Manager. See [USB OTG support](#).
- Allow the Gallery DLP on the MobileIron server to select images from the device's gallery.
- To select an image from **outside** the AppConnect container, such as the device's gallery, the app must be granted the permission `Manifest.permission.READ_EXTERNAL_STORAGE`. This permission is necessary because of how the wrapper implements selecting an image from outside the AppConnect container. For details, see [Image selection from outside the AppConnect container](#).
- Only Generation 2 wrapped apps support `ACTION_CREATE_DOCUMENT` on internal storage or external storage such as an SD drive.

Java files

- Document/AbstractDocumentDemoActivity.java
- Document/DocumentsDemoActivity.java
- Document/DocumentTreeDemoActivity.java
- Document/ImageDocumentDemoActivity.java
- Document/TextDocumentDemoActivity.java
- Document/VideoDocumentDemoActivity.java

Image Decoder Demo

APIUsageDemo displays the Image Decoder demo option only when running on Android 9.0. The code uses ImageDecoder APIs to read an image and display its bitmap. The images are included in the ApiUsageDemo app.

The demo illustrates using the ImageDecoder methods `createSource(File file)` and `createSource(ContentResolver cr, Uri uri)` to get the image from either a file or a URL. The demo then uses the ImageDecoder method `decodeBitmap(ImageDecoder.source src)` to get the bitmap.

Source files

- ImageDecoderDemoActivity.java
- ExtendedImageCapture.kt



Media Meta Data Retriever Demo

The code uses MediaMetadataRetriever APIs to extract the duration time of video content. Allow the Camera DLP on the MobileIron server so that the app can access the camera to record the video.

Extracting the duration time illustrates using the MediaMetadataRetriever setDataSource() method with these data sources:

- a file (absolute path)
- a URI
- a file descriptor from a ParcelFileDescriptor

Wrapping the MediaMetadataRetriever APIs requires the android Internet permission, described in [MediaPlayer](#) and [MediaMetaDataReader Internet permission requirement](#)..

Java files

- MediaMetaDataReaderDemoActivity.java

Media Player Demo

The code uses MediaPlayer APIs to playback video media content. Allow the Camera DLP on the MobileIron server so that the app can access the camera to record the video.

Playback illustrates using the MediaPlayer setDataSource() method to play back from these data sources:

- a file (absolute path)
- a URI
- a file descriptor from a ParcelFileDescriptor
- a file descriptor from a RandomAccessFile

Wrapping the MediaPlayer APIs requires the android Internet permission, described in [MediaPlayer](#) and [MediaMetaDataReader Internet permission requirement](#).

Java files

- MediaPlayerDemoActivity.java

Pick Image Demo

This code uses MediaStore APIs and the Glide library to select an image on the device from either inside or outside the AppConnect container. Allow the Gallery DLP on the MobileIron server to pick an image from the device's gallery.



To select an image from **outside** the AppConnect container, such as the device's gallery, the app must be granted the permission `Manifest.permission.READ_EXTERNAL_STORAGE`. This permission is necessary because of how the wrapper implements selecting an image from outside the AppConnect container. For details, see [Image selection from outside the AppConnect container](#).

Java files

- `PickImageDemoActivity.java`

Share Content Demo

This code demonstrates sharing content such as text, images, and video, with other apps. The app can share content with non-AppConnect apps only if the key `MI_AC_SHARE_CONTENT` on the Secure Apps Manager's AppConnect app configuration on MobileIron Core has the value **true**.

Java files

- `ShareContentDemoActivity.java`

Related topics

"Sharing content from AppConnect for Android apps to non-AppConnect apps" in the *MobileIron AppConnect Guide for Core*.

Video Recorder Demo

The code uses `MediaRecorder` APIs and `MediaPlayer` APIs to record and playback video media content.

Recording illustrates using the `MediaRecorder.setOutputFile()` method with:

- a file (absolute path)
- a file descriptor from a `RandomAccessFile`

Wrapping the `MediaPlayer` APIs requires the android Internet permission, described in [MediaPlayer and MediaMetadataRetriever Internet permission requirement](#).

Java files

- `VideoRecorderDemoActivity.java`
- `BaseRecorderDemoActivity.java`

HelloReact Demo sample app overview

A sample app called HelloReact demonstrates the supported way for a wrapped app built with the React Native development framework to use various React Native modules and components. MobileIron has verified that when



you wrap React Native apps that use these modules and components as demonstrated, the behavior of the wrapped app is functionally the same as the behavior of the unwrapped app.

Related topics

- [Location of sample apps, tester app, and Cordova plugin](#)
- [MediaPlayer and MediaMetadataRetriever Internet permission requirement](#)
- [Image selection from outside the AppConnect container](#)

Demonstrated functionality

The sample app HelloReact includes the following demonstrations:

- [AppConfig Demo](#)
- [Network API Demo](#)
- [Capture Image Demo](#)
- [Pick Image Demo](#)
- [Video View Demo](#)
- [Firebase Cloud Messaging Demo](#)

AppConfig Demo

The AppConfig demo shows how to request, receive, and process app-specific configurations from the MobileIron server. Within HelloReact, MobileIron provides files that provide the APIs to get the app-specific configurations from the MobileIron server.

NOTE: An AppConfig demo that demonstrates the legacy app-specific configuration handling is available only in releases prior to AppConnect 8.6.0.0 for Android.

See the README.txt for HelloReact for details on:

- Adding necessary files to your app.
- Adding ConfigServicePackage to the React Native packages that your app uses.
- Adding necessary permissions for using the package to your AndroidManifest.xml file.
- Using the ConfigService API to get and process the app-specific configurations.

Source files

- HelloReact/README.txt
- HelloReact/src/AppConfigScreen.js
- HelloReact/android/app/src/main/java/com/helloreact/MainApplication.java



Related topics

- [Handling app-specific configuration from the MobileIron server](#)
- [Overview of configuration handling](#)
- [App for testing legacy configuration handling](#)
- [Best practices for handling app-specific configuration](#)

Network API Demo

The Network API demo shows how to use network APIs available in the React Native framework.

You provide a URL in a text field and then choose whether the demo:

- loads the URL into a WebView.
- uses the Fetch API to do a GET request.
(<https://facebook.github.io/react-native/docs/network.html#using-fetch>)

Source files

- HelloReact/src/NetworkScreen.js

Capture Image Demo

The Capture Image demo shows how to use the React Native camera module at:

<https://github.com/lwansbrough/react-native-camera>

The demo shows how to use the camera's view, capture a photo, and receive the result with image properties. The camera module is responsible for storing the resulting image to the specified place.

Allow the Camera DLP on the MobileIron server so that the app can access the camera to take a photo. The image is available only to other wrapped apps.

Source files

- HelloReact/src/CaptureImageScreen.js

Pick Image Demo

The Pick Image demo shows how to use the image picker component at:

<https://github.com/react-community/react-native-image-picker>

The demo shows how to request a picker and process the response to display the chosen image.



Allow the Gallery DLP on the MobileIron server to pick images from the device's gallery. Allow the Camera DLP to take pictures.

Source files

- HelloReact/src/PickImageScreen.js

Video View Demo

The Video View demo shows how to use the video view component at:

<https://github.com/react-native-community/react-native-video>

The demo shows how to pick a video file from gallery storage, or record a video, and then play the video in the Video View.

Allow the Gallery DLP on the MobileIron server to pick video from the device's gallery. Allow the Camera DLP to record videos.

Source files

- HelloReact/src/VideoViewScreen.js

Firebase Cloud Messaging Demo

The Firebase Cloud Messaging (FCM) demo shows how to use the React Native module for FCM and local notifications at:

<https://github.com/evollu/react-native-fcm>

The demo shows how to verify required permissions, subscribe to topics, receive the FCM token, and catch the notifications.

Source files

- HelloReact/src/FCMScreen.js

HelloFlutter sample app overview

Flutter is a Google UI toolkit for building natively compiled applications for mobile, web, and desktop from a single codebase.

The HelloFlutter sample app demonstrates the supported way for a wrapped app to use various Flutter plugins and components. MobileIron has verified that when you wrap apps that use these APIs as demonstrated, the behavior of the wrapped app is functionally the same as the behavior of the unwrapped app.



Related topics

- [Location of sample apps, tester app, and Cordova plugin](#)

Demonstrated APIs

The sample app HelloFlutter includes the following demonstrations:

- [AppConfig Demo](#)

AppConfig Demo

The demo shows how to request, receive, and process app-specific configurations from the MobileIron server.

See the README.md for HelloFlutter for details on:

- Adding necessary files to your app.
- Adding ACConfigPlugin to the MainActivity.java file.
- Adding necessary permissions for using the package to your AndroidManifest.xml file.

Source files

- HelloFlutter/README.md
- HelloFlutter/android/app/src/main/java/com/helloflutter/MainActivity.java
- HelloFlutter/android/app/src/main/java/com/helloflutter/ACConfigCallback.java
- HelloFlutter/android/app/src/main/java/com/helloflutter/ACConfigPlugin.java



Before wrapping an Android app

To wrap an app, you either:

- Submit it to the MobileIron AppConnect Wrapping Portal. MobileIron signs the wrapped apps with the MobileIron private key. MobileIron also signs the Secure Apps Manager, the Secure Apps Manager for AppStation, and all MobileIron secure apps with the MobileIron private key.

The AppConnect Wrapping Portal is the simplest and most common way to wrap apps. It can be used for apps that are distributed by MobileIron Core, MobileIron Connected Cloud, or MobileIron Cloud.

- Use the MobileIron AppConnect for Android Wrapping Tool, a desktop app, to wrap and sign your apps. Use the wrapping tool only if you require that your apps are signed with your own enterprise private key. Signing apps with your enterprise private key instead of the MobileIron private key is a security decision that your enterprise makes.

The wrapping tool can be used only for apps that are distributed by MobileIron Core or MobileIron Connected Cloud. Apps wrapped with the wrapping tool cannot be distributed by MobileIron Cloud.

Before you wrap an Android app:

1. Determine whether your app uses only supported Android coding features.
See [Checking wrapping limitations](#).
2. Determine the wrapping mode to use to wrap the app.
See [Determining the wrapping mode](#).
3. If you are using the AppConnect Wrapping Portal, determine whether your app will be used on devices running AppStation.
See [About wrapping for AppStation](#).

IMPORTANT:

- Some apps cannot be wrapped, depending on the APIs or features they use. See [Capabilities and limitations of apps you can wrap](#).
- You cannot wrap an app (.apk file) that you get from Google Play.
- Do not wrap the sample apps that MobileIron provides. If you use these apps as a starting part for your own app, be sure to change the package name to something that does not start with com.mobileiron before wrapping.



Checking wrapping limitations

Some apps cannot be wrapped, depending on the APIs or coding capabilities they use. Some capabilities require the Generation 1 wrapper and some require the Generation 2 wrapper. Some of these limitations of app wrapping are listed in [Capabilities and limitations of apps you can wrap](#).

After reviewing the limitations with regard to your app, make adjustments to your app to avoid wrapping limitations.

Determining the wrapping mode

MobileIron provides two modes of wrappers: the Generation 1 wrapper and the Generation 2 wrapper. The type of app and the features that it uses determine the required wrapper Generation mode and wrapper release version. The app types are defined in [Apps that you can wrap](#). When you submit an app for wrapping, you must specify:

- whether to use the Generation 1 or Generation 2 wrapper
- the release version of the wrapper

NOTE: To wrap an app with a version of the Secure Apps wrapper that is no longer available in the AppConnect Wrapping Portal or wrapping tool, contact MobileIron Technical Support.

Consider the following in making your selection:

- **For the most comprehensive feature support, select the latest Generation 2 wrapper. MobileIron recommends that you use the Generation 2 wrapper.**
- Select Generation 2 wrapper if the app supports scoped storage and your target SDK is 30.
- Apps wrapped with the wrapper version X.Y.Z require Secure Apps Manager X.Y.Z. The same is true for Secure Apps Manager for AppStation.
- The Secure Apps Manager and the Secure Apps Manager for AppStation support apps wrapped with Generation 1 and 2 wrapper versions as listed in the *MobileIron AppConnect 9.0.0.0 for Android Release Notes and Upgrade Guide*.

Each version of the Secure Apps Manager and the Secure Apps Manager for AppStation supports apps wrapped with previous wrapper versions. Therefore, you can upgrade your Secure Apps Manager and still deploy apps wrapped with previous wrapper versions.

- Select the Generation 1 wrapper if the app was previously wrapped with the Generation 1 wrapper and deployed. **Upgrading a Generation 1 wrapped app to a Generation 2 wrapped version of the app is not supported.**
- Select the Generation 1 wrapper if the app requires support for the Android DocumentsProvider API.

NOTE: Apps wrapped with the Generation 2 wrapper can run, but DocumentsProvider API functionality is not supported.

The following table summarizes the Generation 1 and 2 wrapper support for:



- the various app types
- AppTunnel with HTTP/S or TCP tunneling
- Certificate authentication with AppTunnel with TCP tunneling

NOTE: For more information about capabilities and limitations of apps wrapped with the Generation 1 and 2 wrappers, see [Capabilities and limitations of apps you can wrap](#).

TABLE 5. GENERATION 1 AND 2 WRAPPER SUPPORT FOR APP TYPES AND APPTUNNEL

	Generation 1 wrapper	Generation 2 wrapper
Apps that you previously wrapped with the Generation 1 wrapper and deployed.	Supported	Not supported
Java apps	Supported	Supported
Xamarin apps	Not supported	Supported
Java apps with C or C++ code	Not supported	Supported
Hybrid web apps, including PhoneGap apps	Not supported	Supported
React Native apps	Not supported	Supported
AppTunnel with HTTP/S tunneling in Java apps using supported HTTP/S APIs* for network connections	Supported	Supported

TABLE 5. GENERATION 1 AND 2 WRAPPER SUPPORT FOR APP TYPES AND APPTUNNEL (CONT.)

	Generation 1 wrapper	Generation 2 wrapper
AppTunnel with HTTP/S tunneling in Xamarin apps using supported HTTP/S APIs* for network connections	Not supported	Supported
AppTunnel with TCP tunneling <ul style="list-style-type: none"> • in hybrid web apps, including Phonegap apps • in React Native apps • in Xamarin apps using unsupported HTTP/S APIs • in Java apps using the following for network connections: <ul style="list-style-type: none"> ◦ unsupported HTTP/S APIs ◦ C or C++ code 	Not supported	Supported
Certificate authentication with AppTunnel with TCP tunneling <ul style="list-style-type: none"> • in hybrid web apps, including Phonegap apps • in React Native apps • in Xamarin apps • in Java apps using the following for network connections: <ul style="list-style-type: none"> ◦ unsupported HTTP/S APIs ◦ C or C++ code 	Not supported	Supported

*The supported HTTP/S Java APIs are listed in [AppTunnel with HTTP/S tunneling](#).

The AppConnect wrapping portal

Use the AppConnect Wrapping Portal to wrap your Android apps unless you require that your apps are signed with your own enterprise private key. For that case, see [The AppConnect for Android wrapping tool](#). Most customers use the AppConnect Wrapping Portal.

IMPORTANT: Before using the AppConnect Wrapping Portal, see [Before wrapping an Android app](#).

Using the AppConnect Wrapping Portal, you can upload apps that are up to 200 MB and receive the wrapped app within minutes. The AppConnect Wrapping Portal does not keep either the unwrapped or wrapped version of your app.

The AppConnect Wrapping Portal is available at help.mobileiron.com in the **Developer > Wrapped Apps** tab.

Subscribe to <https://trust.mobileiron.com> for AppConnect Wrapping Portal system status and updates.

Using the AppConnect Wrapping Portal

Before you begin

See [Before wrapping an Android app](#).

Use the AppConnect Wrapping Portal to wrap your Android apps. Using the AppConnect Wrapping Portal involves these high-level tasks:

1. [Logging in to help.mobileiron.com](#)
2. [Uploading and wrapping an app](#)
3. [Downloading the wrapped app](#)

Next steps

After successfully wrapping your apps, do the steps in [After wrapping an Android app](#).

Logging in to help.mobileiron.com

Enter your login ID and password at <https://help.mobileiron.com>.

The home page displays.



Next steps

Go to [After wrapping an Android app](#).

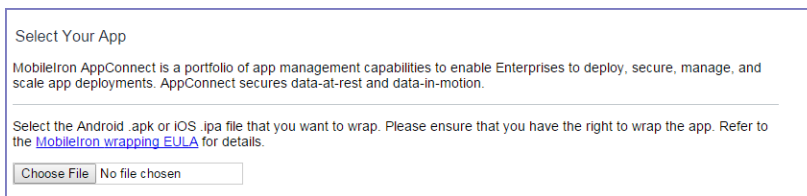
Uploading and wrapping an app

Upload and wrap your app in the wrapping portal.

Procedure

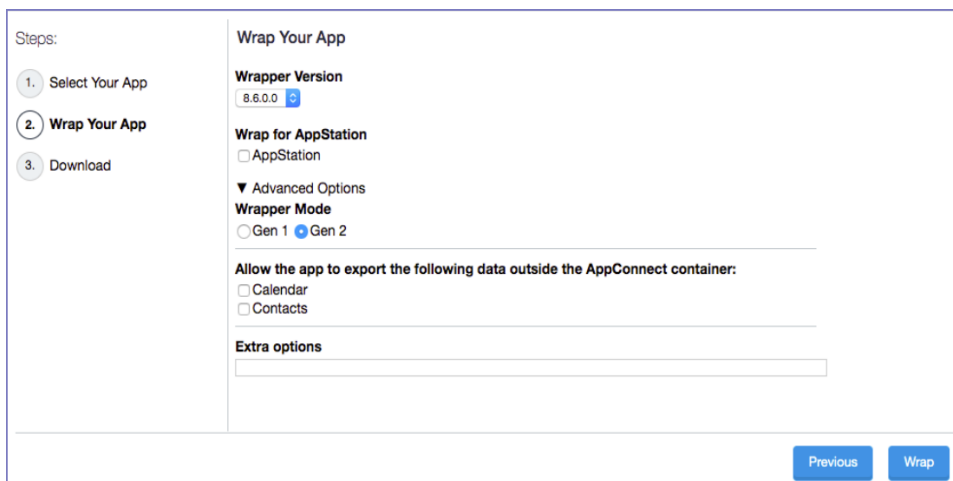
1. Click **Developer > Wrapped Apps** in the tab bar.
The **Wrapped Apps Home** page displays.
2. Click **Create New Wrapped App**.
3. Read and accept the license agreement, if presented.
The license agreement is presented the first time that you click **Create New Wrapped App**.
After accepting the license agreement, the **Select Your App** page displays.

FIGURE 5. SELECT YOUR APP



4. Click **Choose File**.
A dialog box for choosing the file opens.
5. Select the APK file in your computer's folders.
6. Click **Next** on the **Select Your App** page.
The portal uploads and evaluates the APK file, and then displays the **Wrap Your App** page.

FIGURE 6. WRAP YOUR APP



7. Select the **Wrapper Version**.

Keep in mind that the wrapped app will require a Secure Apps Manager or Secure Apps Manager for AppStation with at least the same version as the wrapper version.

NOTE: To wrap an app with an earlier version of the Secure Apps wrapper than the choices given, contact MobileIron Technical Support.

8. If you selected wrapper version 8.6.0.0 or later, the option **Wrap for AppStation** displays. Select **AppStation** to wrap the app for use with the Secure Apps Manager for AppStation.
9. If you are using the Generation 1 wrapper, select **Gen 1**.
10. Select **Calendar** to allow the app to export data to the device's calendar database. This option allows data export when the app uses the Calendar Provider Android API.
11. Select **Contacts** to allow the app to export data to the device's contact database. This option allows data export when the app uses the Contact Provider Android API.
12. Change the maximum heap size in **Max heap size in MBytes to run this tool** only if you encounter any issues when wrapping your app. The default heap size is 5500. The range is 4000 - 12000. Increasing the heap size may slow down the wrapping process, in very rare cases taking up to two hours.
13. In the **Extra options** field, enter the flag `-addInternetPermission` if both of the following are true:
 - Your app uses the `android.media.MediaPlayer` or `android.media.MediaMetadataRetriever` APIs.
 - Your app does not include `android.permission.INTERNET` in its `AndroidManifest.xml` file

IMPORTANT: Do not select any other **Advanced Settings** or make other modifications to **Custom Options** unless MobileIron Technical Support has instructed you to do so. The following flags are available:

Flag	Description
<code>-ignoreSqlCipher</code>	See Encryption of the SQLCipher database .
<code>-allowIntentAction</code>	See Receiving information from outside the AppConnect container .
<code>-enableCrashlytics</code>	Enables Crashlytics library.
<code>-disableArm64</code>	See 64-bit support .
<code>-keepJavaNativesLazyLinking</code>	See Linking native Java methods

14. Click **Wrap**.**Next steps**

Go to [Downloading the wrapped app](#).

Related topics

- [Determining the wrapping mode](#).
- [About wrapping for AppStation](#)

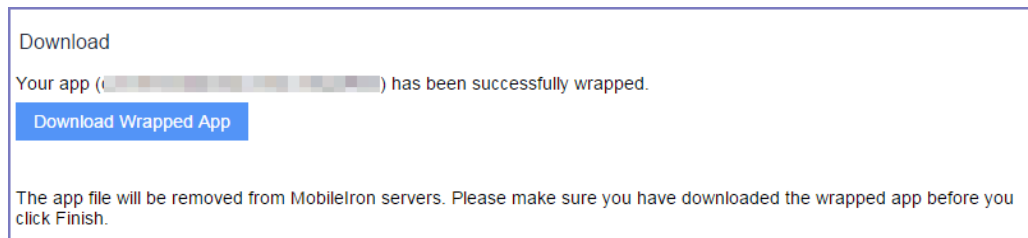


- [MediaPlayer and MediaMetadataRetriever Internet permission requirement](#)

Downloading the wrapped app

When you click **Wrap**, after a few moments, depending on the size of the app, the **Download** page displays.

FIGURE 7. DOWNLOAD WRAPPED APP



NOTE: if wrapping fails, the portal displays the reason. You can click **Open Support Ticket** if you need help.

1. Click **Download Wrapped App**.
The portal downloads the wrapped app to your computer.
2. Click **Finish**.
The portal removes both the wrapped and unwrapped version of the app.

Next steps

Go to [After wrapping an Android app](#).

The AppConnect for Android wrapping tool

The MobileIron AppConnect for Android Wrapping tool, also known as the wrapping tool, is a macOS and Windows app that MobileIron provides. Use the wrapping tool instead of the AppConnect Wrapping Portal only if both of the following are true:

- You require that your wrapped apps are signed with your own enterprise private key.
- You are distributing the wrapped apps with MobileIron Core or MobileIron Connected Cloud. You **cannot** distribute wrapped apps signed with your own enterprise private key with MobileIron Cloud.

If you are not using an enterprise private key, use [The AppConnect wrapping portal](#). The AppConnect Wrapping Portal signs wrapped apps with the MobileIron private key.

Signing apps with your enterprise private key instead of the MobileIron private key is a security decision that your enterprise makes.

Use the AppConnect for Android Wrapping Tool to:

- Wrap and sign an app.
The wrapping tool outputs a wrapped and signed APK file, which is the APK to be uploaded to MobileIron Core or MobileIron Connected Cloud. It also outputs a wrapped but unsigned APK file, which is useful if only specific people in your organization have access to the enterprise private key.
- Only sign the app.
The option to only sign the app is typically used in these cases:
 - For signing your wrapped apps when only specific people in your organization have access to the enterprise private key.
 - For re-signing apps provided by MobileIron with the enterprise private key.

Enterprise private key considerations with AppConnect for Android

By using the AppConnect for Android Wrapping Tool, you can distribute wrapped apps signed with your enterprise private key instead of the MobileIron private key.

Consider the following impact of using an enterprise private key:

You must use your enterprise private key to re-sign all the secure apps that you currently use.



These apps include the apps that MobileIron provides and any other secure apps that you use. You must also re-sign the Secure Apps Manager with your enterprise private key. For each future Secure Apps release, you will again have to re-sign the Secure Apps Manager and all updated apps provided by MobileIron.

You must secure your enterprise private key.

You must secure the enterprise private key to protect the secure apps that you deploy and the devices they are deployed on. If an unauthorized third party obtains the enterprise key without your permission, the third party can sign and distribute apps with your key, allowing them to maliciously replace your apps. These malicious apps could run in the AppConnect container, with access to your enterprise's sensitive data.

Therefore:

- Follow industry best practices for securing your enterprise key.
- Follow industry best practices against losing your key or forgetting the password for the keystore file or the key itself.

You must securely retain backup copies of your enterprise private key and password.

If you lose your enterprise private key or password and do not have a backup, you cannot deploy updates to your apps. Keep at least one secure backup of your key and password. MobileIron will not have a copy of your enterprise private key, and will not be able to assist you with restoring it.

Installing the new re-signed Secure Apps Manager on devices *deletes all existing secure apps data on the device.*

Your device users will lose all data relating to their secure apps. Going forward, as long as you use the same enterprise key, this loss will not reoccur. If your key is compromised and you have to create a new enterprise key, your device users will again lose all secure apps data.

AppConnect for Android Wrapping Tool supported platforms

You can run the wrapping tool on:

- macOS X
- Windows 7
- Windows 10
- Ubuntu 18.04.5

Preparing to use the wrapping tool

Do the following tasks before using the wrapping tool:



1. See [Before wrapping an Android app](#).
2. See [Enterprise private key considerations with AppConnect for Android](#).
3. Install the Java Development Kit (JDK) on your Windows or macOS computer.
See <http://www.oracle.com/technetwork/java/javase/downloads>.
4. Install Android Studio on your Windows or macOS computer as an easy way to get the Android Software Development Kit (SDK) build-tools that the wrapping tool requires.
See <https://developer.android.com/studio/index.html> to download Android Studio. After downloading and installing Android Studio, launch it to install the standard Android SDK component and tools.
The build-tools must be version 24.0.1 through the most recently released version as supported by MobileIron.
5. Obtain a private key for signing secure apps.
You will upload the keystore file to the wrapping tool, and you will upload the matching public certificate to MobileIron Core or MobileIron Connected Cloud.
Use the Java keytool command to generate the public and private key pair. See <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html>.
6. Get the Secure Apps Manager and any other MobileIron-provided apps that you distribute. You will need to re-sign these apps with your enterprise private key. The apps are available at <https://help.mobileiron.com> in the Software tab. For example, if you use Web@Work, Docs@Work, or Email+, you must re-sign them.

NOTE: Do not re-wrap MobileIron apps, including the sample apps. You only re-sign the Secure Apps Manager and MobileIron apps.
7. Download the wrapping tool, a JAR file, to your computer from <https://help.mobileiron.com> in the Software tab.

Using the AppConnect for Android Wrapping Tool in UI mode

Using the wrapping tool in UI mode involves these high-level tasks:

Before you begin

- [Preparing to use the wrapping tool](#)

Overview

1. [Launching the wrapping tool](#)
2. [Providing developer settings to the wrapping tool](#)
3. [Selecting wrapping options in the wrapping tool](#)
4. [Wrapping and signing an app with the wrapping tool](#)



Next steps

- [After wrapping an Android app](#)
- [Distributing wrapped apps with an enterprise key \(Core\)](#)

Related topics

- [Signing an app with the wrapping tool](#)
- [Using the AppConnect for Android Wrapping Tool in UI mode](#)

Launching the wrapping tool

To use the wrapping tool, launch it, and accept the license agreement.

Procedure

1. Enter the following at the command prompt to launch the app:

```
java -jar path\wrap-tool.jar
```

Where:

- *path\jar wrap-tool* is the location and name of the wrapping tool jar file.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar
```

2. Accept the license agreement.

NOTE: You are not prompted to accept the license agreement if you had accepted it in a previous version of the tool.

Providing developer settings to the wrapping tool

Before wrapping or signing an app, provide the necessary developer settings to the wrapping tool.

Procedure

1. In the wrapping tool, go to **Developer Settings**.
2. Browse to the Android SDK directory or enter its path.

Example

On Windows: C:\Users\username\AppData\Local\Android\SDK

On macOS: /Users/username/Library/Android/SDK

NOTE: If the ANDROID_HOME environment variable is set, filling in this field is unnecessary.

3. In **Developer Settings**, if you are using Windows, enter Java VM options if necessary. This option is necessary if the Windows computer has less than 8GB of RAM.



Example`-Xmx5000M`

4. Drag and drop or browse to the keystore file that contains your enterprise private key.
5. Enter the keystore password, the key alias, and the key password.
6. Click **Save**.
The wrapping tool displays that your keystore has been successfully uploaded.
7. Click **Done**.

Selecting wrapping options in the wrapping tool

Before wrapping an app, select the appropriate wrapping options in the wrapping tool. These steps are not necessary when you are signing the app, but not wrapping it.

Procedure

1. Select the wrapper version.
Keep in mind that the wrapped app will require a Secure Apps Manager with at least the same version as the wrapper version.

NOTE: To wrap an app with an earlier version of the Secure Apps wrapper than the choices given, contact MobileIron Technical Support.

2. Select either Generation 1 or 2.
3. Select **Calendar Access** to allow the app to export data to the device's calendar database.
This option allows data export when the app uses the Calendar Provider Android API.
4. Select **Contacts Access** to allow the app to export data to the device's contact database.
This option allows data export when the app uses the Contact Provider Android API.s
5. Select **Show** next to **Advanced Settings** field, scroll down to **Custom Options**, and enter the flag `-addInternetPermission` if both of the following are true:
 - Your app uses the `android.media.MediaPlayer` or `android.media.MediaMetadataRetriever` APIs.
 - Your app does not include `android.permission.INTERNET` in its `AndroidManifest.xml` file
 If you turn on **Custom Options**, the default custom options are:
 - `-allowAccessGoogle`, which allows the app to use Google Play services
 - `-allowNativeCode`, which allows the app to use native libraries

IMPORTANT: Do not select any other **Advanced Settings** or make other modifications to **Custom Options** unless MobileIron Technical Support has instructed you to do so. The following flags are available:



TABLE 6. AVAILABLE FLAGS

Flag	Description
-ignoreSqlCipher	See Encryption of the SQLCipher database .
-allowIntentAction	See Receiving information from outside the AppConnect container .
-enableCrashlytics or -allowCrashlytics	See Firebase Cloud Messaging and Crashlytics support . Earlier versions of AppConnect supported the -allowCrashlytics option and not the -enableCrashlytics option.
-disableArm64	See 64-bit support .
-keepJavaNativesLazyLinking	See Linking native Java methods

Related topics[Determining the wrapping mode](#)

Wrapping and signing an app with the wrapping tool

After selecting developer settings and wrapping options in the wrapping tool, you can wrap and sign an app.

Procedure

1. Drag and drop or browse to the unwrapped app's APK file.
The wrapping and signing process begins.
2. If wrapping and signing succeed, the wrapping tool displays that wrapping and signing was successful. It provides a link to the same directory as the unwrapped APK file, and places the following files in the directory:
 - the wrapped and signed APK file, named *<file name>.wrapped.signed.apk*
You will upload this file to the MobileIron server as an in-house app for distribution to devices.
 - the wrapped and unsigned file, named *<file name>.wrapped.apk*
asdf
 - a log file about wrapping, named *<file name>.apk.result.json*
3. If wrapping or signing fail, the wrapping tool displays that it failed. It provides a link to the same directory as the unwrapped APK file, and places the following files in the directory:
 - a log file about wrapping, named *<file name>.apk.result.json*
 - a signing error file, named *<file name>.apksigner.errors*

Related topics[Using the AppConnect for Android Wrapping Tool in UI mode.](#)

Signing an app with the wrapping tool

After selecting developer settings in the wrapping tool, you can sign a wrapped app with your enterprise private key. The app can be unsigned or already signed. You can also re-sign the Secure Apps Manager. Use this procedure to:

- Sign your own wrapped apps.
- Re-sign MobileIron-provided apps with your enterprise private key. Re-sign the apps each time you get a new release of the app from MobileIron.
- Re-sign your own apps and MobileIron-provided apps with a new enterprise private key when, for example, the previous enterprise private key had been compromised.

Procedure

1. Select **Sign Only** in the wrapping tool where you drag and drop your app. It can be unsigned or already signed.
2. Drag and drop or browse to the app's APK file.
The signing process begins.
3. If signing succeeds, the wrapping tool displays that signing was successful.
It places the wrapped and signed file in the same directory as the submitted APK file, and provides a link to that directory. The file is named:
The file is named:
<file name>.signed.apk
4. If signing fails, the wrapping tool displays that signing failed. It provides a link to the same directory as the unwrapped APK file, and places in the directory a signing error file named *<file name>.apksigner.errors*.

Related topics

- [Enterprise private key considerations with AppConnect for Android](#)
- [Providing developer settings to the wrapping tool](#)

Using the AppConnect for Android Wrapping Tool in CLI mode

Using the wrapping tool in CLI mode involves these high-level tasks:

Before you begin

- [Preparing to use the wrapping tool](#)

Overview

Using the wrapping tool involves these high-level tasks:



1. [Providing developer settings](#)
2. [Setting the keystore](#)
3. [Wrapping and signing the app](#)

Next steps

After successfully wrapping and signing your apps, do the following:

- [After wrapping an Android app](#)
- [Distributing wrapped apps with an enterprise key \(Core\)](#)

Related topics

- [Additional wrapping tasks using CLI](#)
- [Troubleshooting the wrapping tool](#)
- [Determining the wrapping mode](#)

The paths are documented using the Windows format. Use the format that is appropriate to your OS.

Example

- On Windows: C:\Users\username\AppData\Local\Android\SDK
- On macOS: /Users/username/Library/Android/SDK

Providing developer settings

When you first use the wrapping tool, provide the necessary developer settings to the wrapping tool.

Enter the following at the command prompt:

```
java -jar path\wrap-tool.jar -android-SDK-path SDKpath
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.
- *SDKpath* is the location of the Android SDK.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar -android-SDK-path  
C:\Users\testuser\AppData\Local\Android\Sdk
```

Accept the license agreement presented.

NOTE: You are not prompted to accept the license agreement if you had accepted it in a previous version of the tool.



Example

```
Do you agree with all terms and conditions? [Y/n] y
Android SDK path update: success!
### To run wrapping tool in UI mode
Usage: java -jar wrap-tool.jar
### To run wrapping tool in CLI mode please specify proper action
Usage: java -jar wrap-tool.jar (-wrap || -sign-only || -help) -in appName.apk [wrapping arguments]
```

Setting the keystore

Set the keystore to sign the wrapped app with your enterprise private key. The keystore file is saved for all subsequent app signing.

Before you set the keystore, create a *keystore.txt* file that contains the location and name of the keystore file, the store password, the key alias, and the key password. It is recommended to create the keystore text file so that the passwords are not visible in the CLI.

The keystore text file contains the following content:

```
path\fileName.keystore ks-pass ks-key-alias key-pass
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.
- *path\fileName* is the location and name of the keystore file.
- *ks-pass* is the store password.
- *ks-key-alias* is the key alias.
- *key-pass* is the key password.

Enter the following at the command prompt to set the keystore:

```
java -jar path\wrap-tool.jar -keystore @path\keystore-file.txt
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.
- *path\keystore-file* is the location and name for the keystore text file.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar -keystore
@C:\Users\testuser\Downloads\keystore-file.txt
checking keystore settings...
applying keystore settings...
Keystore update: success!
### To run wrapping tool in UI mode
```



Usage: java -jar wrap-tool.jar

To run wrapping tool in CLI mode please specify proper action

Usage: java -jar wrap-tool.jar (-wrap || -sign-only || -help) -in appName.apk [wrapping arguments]

Wrapping and signing the app

The wrapping tool in command line interface (CLI) mode uses the following defaults:

- **Wrapper version:** The current AppConnect version. If the AppConnect version is 9.2.0, the associated AppConnect wrapping tool defaults to the 9.2.0 wrapper version.
- **Wrapping mode:** Generation 2.
- **Wrapping options:**
 - -allowAccessGoogle, which allows the app to use Google Play services
 - -allowNativeCode, which allows the app to use native libraries
 - -allowUnwrappedAPIs, which allows

Enter the following command at the command prompt to wrap and sign an app:

```
java -jar path\wrap-tool.jar -wrap -in path\appName.apk [wrapping options]
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.
- *path\appName* is the location and name of the APK file.
- *wrapping options* are any optional arguments you want to use for wrapping the file.

The following table describes the available arguments to use for wrapping an app.

TABLE 7. AVAILABLE ARGUMENTS

Flag	Description
-ignoreSqlCipher	See Encryption of the SQLCipher database .
-allowIntentAction	See Receiving information from outside the AppConnect container .
-enableCrashlytics	Enables Crashlytics library. See Firebase Cloud Messaging and Crashlytics support .
-disableArm64	See 64-bit support .
-keepJavaNativesLazyLinking	See Linking native Java methods

Example :

```
C:\Users\testuser>java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar -
wrap -in C:\Users\testuser\Downloads\unTransformedApps_9.2.0.0.14\Box.apk
selected the version 9.2
```



```
running transformer version 9.2 for the file C:\Users\testuser\Downloads\unTransformedApps_
9.2.0.0.14\Box.apk
wrapping exit value: 0
```

```
...<JSON output>
```

```
zipalign exit value: 0
signing exit value: 0
Signed
```

Additional wrapping tasks using CLI

The following additional tasks allow you to re-sign an app, change the wrapper mode, change the wrapper version, and view the available wrapper arguments for a wrapper version:

- [Signing an app](#)
- [Using the Generation 1 wrapper](#)
- [Wrapping with a different allowed wrapper version](#)
- [Viewing wrapper arguments for a wrapper version](#)
- [Using the -help command](#)

Signing an app

Use this procedure to:

- Sign your own wrapped apps.
- Re-sign MobileIron apps with your enterprise private key. Re-sign the apps each time you get a new release of the app from MobileIron.
- Re-sign your own apps and MobileIron apps with a new enterprise private key when, for example, the previous enterprise private key had been compromised.

Before signing an app, set the keystore. See [Additional wrapping tasks using CLI](#).

To sign an app in the CLI mode, enter the following command at the prompt:

```
java -jar path\wrap-tool.jar -sign-only -in path\appName.apk
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.
- *path\appName* is the location and name of the APK file.



Using the Generation 1 wrapper

By default, the CLI mode uses the Generation 2 wrapper. To use the Generation 1 wrapper, specify `-gen1` when wrapping the app.

Enter the following at the command prompt:

```
java -jar path\wrap-tool.jar -wrap -gen1 -in path\appName.apk [wrapping arguments]
```

Where:

- `path\wrap-tool` is the location and name of the wrapping tool jar file.
- `path\appName` is the location and name of the APK file.
- `wrapping arguments` are any optional arguments you want to use for wrapping the file.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar -wrap -gen1 -in C:\Users\testuser\Downloads\unTransformedApps_9.2.0.0.14\Box.apk
```

Wrapping with a different allowed wrapper version

By default, in CLI mode, an app is wrapped using the latest wrapper version that the tool supports. The wrapping tool also supports previous versions of the wrapper. Therefore, you can specify a previous wrapper version to wrap an app.

Enter the following command at the command prompt to view the supported previous wrapper versions:

```
java -jar path\wrap-tool.jar -help -transformer-version
```

Where:

- `path\wrap-tool` is the location and name of the wrapping tool jar file.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar -help -transformer-version
### To run wrapping tool in UI mode
Usage: java -jar wrap-tool.jar
### To run wrapping tool in CLI mode please specify proper action
Usage: java -jar wrap-tool.jar (-wrap || -sign-only || -help) -in appName.apk [wrapping arguments]
### Allowed versions of transformer:
1) 9.2
2) 9.1
3) 9.0
To receive list of all allowed wrapping arguments need to specify transformer version:
java -jar wrap-tool.jar -help -transformer-version 9.2
```



Enter the following command to wrap an app with a previous wrapper version:

```
java -jar path\wrap-tool.jar -wrap -transformer-version allowed-version -in path\appName.apk
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.
- *path\appName* is the location and name of the APK file.
- *allowed-version* is an allowed wrapper version.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.18-9.2.0.0.4.jar -wrap -transformer-
version 9.1 -in C:\Users\testuser\Downloads\unTransformedApps_9.2.0.0.14\Box.apk\
selected the version 9.1
running transformer version 9.1 for the file C:\Users\testuser\Downloads\unTransformedApps_
9.2.0.0.14\Box.apk
wrapping: |
```

Viewing wrapper arguments for a wrapper version

Enter the following command at the command prompt to view the allowed wrapper versions:

```
java -jar path\ wrap-tool.jar -help -transformer-version transformer-version
```

Where:

- *path\ wrap-tool* is the location and name of the wrapping tool jar file.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.14-9.2.0.0.4.jar -help -transformer-
version 9.2
### To run wrapping tool in UI mode
Usage: java -jar wrap-tool.jar
### To run wrapping tool in CLI mode please specify proper action
Usage: java -jar wrap-tool.jar (-wrap || -sign-only || -help) -in appName.apk [wrapping
arguments]
### Allowed versions of transformer:
1) 9.2
2) 9.1
3) 9.0
transformer help exit value: 1
Usage: java -Xmx4096M -jar Transformer.jar input.apk [arguments]
Version: 9.2.0.0.4-0
Arguments:
Name: -allowExternalMailToAccess
Type: Boolean
Description: Secure apps normally cannot respond to the requests of
```



non-secure apps. This flag lets the wrapped app respond to 'send email' requests from unwrapped apps.

Name: -allowNativeCode

Type: Boolean

Description: Lets an app be wrapped even if it contains native code.

...

Using the -help command

The -help command displays a list of available commands and descriptions.

Enter the following command at the command prompt to view a complete list of available commands and descriptions:

```
java -jar path\wrap-tool.jar -help
```

Where:

- *path\wrap-tool* is the location and name of the wrapping tool jar file.

Example

```
java -jar C:\Users\testuser\Downloads\wrap-tool-1.14.14-9.2.0.0.4.jar -help
### To run wrapping tool in UI mode
Usage: java -jar wrap-tool.jar
### To run wrapping tool in CLI mode please specify proper action
Usage: java -jar wrap-tool.jar (-wrap || -sign-only || -help) -in appName.apk [wrapping arguments]
Arguments:
Name: -help
Is required for wrapping: false
Description: Display help and exit
Name: -transformer-version
Is required for wrapping: false
Description: Can be used with or without specified transformer version. In wrapping mode will choose specific version for wrapping application.
Example: java -jar wrap-tool.jar -wrap -transformer-version 9.2
In help mode will display all available version of transformer. With specified exact version of wrapper all the arguments for exact transformer will be displayed
Example: java -jar wrap-tool.jar -help -transformer-version 9.2
...
```

Wrapping tool CLI

Use the following command to wrap an app in CLI mode.

```
java -jar wrap-tool.jar (-wrap || -sign-only || -help) -in appName.apk [wrapping arguments]
```



TABLE 8. WRAPPING COMMANDS

Action	Command	Description
Wrap an app	<code>java -jar wrap-tool.jar -wrap -in <i>appName.apk</i> [<i>wrapping arguments</i>]</code>	Using the AppConnect for Android Wrapping Tool in CLI mode
Sign an app	<code>java -jar wrap-tool.jar -sign-only <i>appName.apk</i></code>	Additional wrapping tasks using CLI
Help	<code>java -jar wrap-tool.jar -help</code>	<p>Displays a list of available commands and description.</p> <ul style="list-style-type: none"> • Use the following command to view the supported transformer versions: <code>java -jar <i>path</i>\ wrap-tool.jar -help -transformer-version</code> • Use the following command to view the supported wrapping arguments for a specific transformer version <code>java -jar <i>path</i>\ wrap-tool.jar -help -transformer-version <i>transformer-version</i></code> <p>Additional wrapping tasks using CLI</p>

Troubleshooting the wrapping tool

When you run the wrapping tool to both wrap and sign an app, or to only sign an app the tool places the following files in the same directory as the APK file you are wrapping and signing:

- `<file name>.wrapped.signed.apk`
 The wrapped and signed APK file. It is available only when you use the tool for both wrapping and signing, and both actions succeed. You will upload this file to the MobileIron server as an in-house app for distribution to devices.
- `<file name>.wrapped.apk`
 The wrapped but unsigned APK file. It is available only when you use the tool for both wrapping and signing, and wrapping succeeds. This file is useful when someone else in your organization signs the apps.
- `<file name>.signed.apk`
 The signed APK file. Available only when you use the tool for signing only, having provided it a wrapped app, and signing succeeds. You will upload this file to the MobileIron server as an in-house app for distribution to devices.



- `<file name>.wrapped.result.json`
The log file about the wrapping process.
If wrapping fails, open this file and scroll to the end to see the error.
- `<file name>.apksigning.errors`
The log file about the signing process.
If signing fails, open this file to see the error.

Distributing wrapped apps with an enterprise key (Core)

After you have signed all wrapped apps and the Secure Apps Manager with your enterprise private key, you can distribute them to your enterprise's device users.

The steps in this section are applicable if your UEM is Core or Connected Cloud.

IMPORTANT: If you are upgrading device users to use secure apps signed with your enterprise private key, installing the re-signed Secure Apps Manager on devices deletes all existing secure apps data on the device.

Do the steps in these tasks:

1. [Uploading the apps to the App Catalog.](#)
2. [Configuring the enterprise public key.](#)
3. [Applying labels to the new apps.](#)
4. [Removing labels from the old apps.](#)

Related topics

- [The device user experience when upgrading](#)
- [Behavior when the device does not have the enterprise public certificate](#)

Uploading the apps to the App Catalog

Use the Admin Portal to upload the newly signed secure apps to the Core app distribution library just as you would any in-house app. Go to **Apps > App Catalog > Add+ > In-House**. For details on in-house apps for Android, see “Working with Apps for Android Devices” in the *MobileIron Core Apps@Work Guide*.

Configuring the enterprise public key

To run secure apps signed with the enterprise private key, configure MobileIron Core or Connected Cloud to provide the matching public certificate to the devices.



Procedure

1. In the Admin Portal, select **Policies & Configs > Configurations**.
2. Select **Add New > Certificates**.
3. Enter a **Name** and **Description** for the new Certificate Setting.
4. Upload the public certificate that matches your enterprise private key.
No entries are necessary for the password settings since this is the public certificate.
5. Click **Save**.
6. Select the app configuration for the new Secure Apps Manager.
7. Click **Edit**.
8. In **App-specific Configurations**, click **Add+**.
9. For the **Key**, enter **AC_PUBLIC_KEY**.
10. For the **Value**, select the certificate setting that you just create from the drop-down list.
11. Click **Save**.

Applying labels to the new apps

Apply the appropriate labels to the newly signed apps, including the Secure Apps Manager. These labels determine to which devices the apps will be downloaded.

IMPORTANT: Installing the re-signed Secure Apps Manager on devices deletes all existing secure apps data on the device.

Procedure

1. In the **Apps** tab of the Admin Portal, select **Android** for **Select Platform**.
2. Select all the newly signed apps, including the Secure Apps Manager.
3. Select **Actions > Apply To Label**.
4. Select the appropriate labels.
5. Click **Apply**.

Removing labels from the old apps

If devices already had secure apps signed with the MobileIron private key (or some other enterprise private key), remove the appropriate labels from the old secure apps, including the Secure Apps Manager.

Do the following for the old secure apps, including the Secure Apps Manager:

1. In the **Apps** tab of the Admin Portal, select **Android** for **Select Platform**.
2. Select all the old secure apps, including the old Secure Apps Manager.



3. Select **Actions > Remove From Label**.
4. Select the appropriate labels.
5. Click **Remove**.

The device user experience when upgrading

After you have completed the steps to upgrade to secure apps signed with a enterprise private key, the device user experiences the following:

1. Mobile@Work prompts the device user to update secure apps.
2. When the user begins the update process, Mobile@Work warns the user of the consequences of the pending update. Specifically, users are warned that they will lose their secure apps data (including email settings) and will need to create a new secure apps passcode.
3. If the user continues with the update process, the old secure apps are uninstalled, and the new secure apps are installed. On devices that support silent installation, silent uninstall and install are used.

NOTE: An uninstall followed by an install is necessary instead of an app upgrade. The reason is because the Android operating system does not allow app upgrades when the signing keys do not match.

Behavior when the device does not have the enterprise public certificate

Using an enterprise private key to sign secure apps and the Secure Apps Manager requires that you configure the Secure Apps Manager's app configuration with the enterprise public certificate. Consider the following situations relating to this requirement:

- You do not configure Secure Apps Manager's app configuration with the enterprise public certificate.
In this case, the Secure Apps Manager defaults to using the MobileIron private key. Therefore, if secure apps signed with the enterprise private key are on the device, the secure apps cannot run due to a signature mismatch.
- You later remove the enterprise public certificate.
Consider the situation when secure apps signed with the enterprise private key are running on the device. Later, you remove the public certificate from the device. For example, you remove the certificates setting from MobileIron Core, or remove the related key-value pair from the Secure Apps Manager's app configuration. The secure apps signed with an enterprise private key can no longer run due to a signature mismatch. However, no secure data is lost. When the key-value pair is added back to the Secure Apps Manager's app configuration, the secure apps can once again run.
- You configure Secure Apps Manager's app configuration with the enterprise public certificate, but the installed apps are signed with the MobileIron private key.
In this case, the secure apps cannot run due to a signature mismatch.



After wrapping an Android app

After you have wrapped an Android app:

1. If the app uses Google APIs that use a Google API key, add the wrapped app's key to the Google API Console.
See [Adding the wrapped app's key to the Google API Console](#).
2. Document your app's requirements.
See [Inform the server administrator of your app's requirements](#).

Adding the wrapped app's key to the Google API Console

Some Google APIs, such as the Google Maps API, can use a Google API key. The key comprises a SHA1 certificate fingerprint and package name. If the app you wrapped uses a Google API key, you have an additional step to perform before deploying the wrapped app. Wrapping the app changes the wrapped app's API key. You must add the new API key to the Google API Console.

Wrapped app's Google API key format

The new API key has the following format:

<MobileIron-provided SHA1 certificate fingerprint>;<your wrapped app package name>

Your wrapped app's package name is one of the following:

- `forgepond.<your app package name>` if you wrapped the app for use with the Secure Apps Manager
- `appstation.<your app package name>` if you wrapped the app for use with the Secure Apps Manager for AppStation

The MobileIron-provided SHA1 certificate fingerprint is:

`D1:F0:BB:0F:7B:E9:91:6F:0C:08:C1:96:0B:E2:E0:BF:A5:76:1D:60`

Therefore, a wrapped app's new API key, is, for example:

`D1:F0:BB:0F:7B:E9:91:6F:0C:08:C1:96:0B:E2:E0:BF:A5:76:1D:60;forgepond.com.myco.myapp`

Adding the new API key to Google API console

If the app you wrapped uses a Google API key, wrapping the app changes the wrapped app's API key. Add the new API key to the Google API Console.



Procedure

1. Go to <https://code.google.com/apis/console>.
2. Login with your Google account.
3. Click **API Manager > Credentials** in the left menu.
4. Create **Credentials > API key > Android key**.
5. Name your key.
6. Add the package name with this format: `forgepond.<your package name>` or `appstation.<your package name>`
7. And the MobileIron-provided SHA1 certificate fingerprint.

Inform the server administrator of your app's requirements

Provide the MobileIron server administrator the following information so that the administrator can correctly configure and deploy your app:

- Documentation about app-specific configuration.
See [Handling AppConnect app-specific configuration](#).
- Whether to allow the app to ignore the auto-lock time.
See [Ignoring the auto-lock time](#).
- Whether the app requires AppTunnel with HTTP/S tunneling.
See [AppTunnel with HTTP/S tunneling](#).
- Whether the app requires AppTunnel with TCP tunneling.
See [AppTunnel with TCP tunneling](#).
- Whether the app requires certificate authentication with AppTunnel with TCP tunneling.
See [Certificate authentication with AppTunnel with TCP tunneling](#).
- Whether the app requires the secure FileManager.
See [DownloadManager API considerations](#).



Capabilities and limitations of apps you can wrap

You can wrap most apps with the AppConnect wrapper with no app development. However some app capabilities require special attention, some capabilities are supported with the Generation 1 or Generation 2 wrapper but not both, and some app capabilities cannot be wrapped.

Therefore, before wrapping an app, review the following:

- [AppConnect wrapping considerations](#)
- [Wrapping support of commonly used app capabilities](#)
- [Known wrapper limitations](#)

Also, see these sample apps for demonstrations of implementing common capabilities in Java apps and React Native apps to be wrapped.

- [Android API Usage Demo sample app overview](#)
- [HelloReact Demo sample app overview](#)

AppConnect wrapping considerations

- [SQLCipher considerations](#)
- [DownloadManager API considerations](#)
- [Google Cloud Messaging considerations](#)
- [Firebase Cloud Messaging and Crashlytics support](#)
- [MediaPlayer and MediaMetadataRetriever Internet permission requirement](#)
- [Image selection from outside the AppConnect container](#)
- [External storage permissions](#)
- [Receiving information from outside the AppConnect container](#)
- [USB OTG support](#)
- [Preference API usage](#)
- [64-bit support](#)
- [Linking native Java methods](#)



SQLCipher considerations

- [SQLCipher library version](#)
- [Using both SQLCipher and SQLite is not supported](#)
- [Encryption of the SQLCipher database](#)

SQLCipher library version

The Secure Apps Manager, the Secure Apps Manager for AppStation, and the AppConnect wrapper in wrapped apps all use SQLCipher 4.0.1, which is a 64-bit library.

Using both SQLCipher and SQLite is not supported

Using SQLCipher and Android SQLite databases at the same time is not supported. Developers should move all databases to either SQLCipher or Android SQLite.

Encryption of the SQLCipher database

If your app uses SQLCipher calls to encrypt data, the wrapping process makes changes so that the Secure Apps Manager or Secure Apps Manager for AppStation manages the encryption key for the SQL data. Therefore, the wrapping process encrypts the SQLCipher database a second time, because it is already encrypted by SQLCipher. This second layer of encryption can have performance impact on your app.

To avoid the performance impact, you can use the wrapping flag `-ignoreSqlcipher` with Generation 2 wrapped apps. Use this flag **only for new installations, not upgrades**, of your app where the app is wrapped with wrapper version 8.2.1 through the most recently released version as supported by MobileIron. Do not use the flag if your wrapped app was previously installed on devices. If you do, your database will not be accessible.

When you use the `-ignoreSqlcipher` flag, continue to use it every time you wrap the app.

NOTE: You can use the `-ignoreSqlcipher` flag for wrapping an app that uses the Room Persistence Library.

DownloadManager API considerations

If your app uses the Android DownloadManager API, the secure FileManager that MobileIron provides must also be installed on the device. The FileManager ensures downloaded files remain in the secure container. Only secure apps can access the downloaded files.

Inform the MobileIron server administrator that your app requires the FileManager.

Google Cloud Messaging considerations

Secure Apps for Android supports Google Cloud Messaging for Android (GCM) in wrapped apps. GCM allows you to send data from an app server to these apps.



Secure Apps supports using GCM for sending HTTP messages from an app server to wrapped apps (HTTP downstream messages). The feature requires that the app uses the `GoogleCloudMessaging` class. The deprecated class `GCMRegistrar` is not supported.

Unsupported GCM features

Secure Apps does not support these GCM features:

- XMPP
- upstream messaging

Situations when GCM messages are discarded

In the following situations, the app does not receive GCM messages, and the messages are discarded:

- The device user is logged out of secure apps after a device restart or Secure Apps Manager (or Secure Apps Manager for AppStation) termination.
- In this situation, the user has a notification that says “Logged out. No app activity. Tap to log in.”
- The app is unauthorized (blocked) on the device.

Firebase Cloud Messaging and Crashlytics support

A wrapped app can use Firebase Cloud Messaging (FCM) the same as any Android app. To support FCM in a wrapped app, on the Firebase web console, you create a Firebase project that specifies your unwrapped app. The unwrapped app and the wrapped app can each then receive the FCM messages. If both the wrapped app and the unwrapped app are on the same device, they both receive the FCM messages. Unauthorized wrapped apps do not receive the FCM messages.

NOTE: Generation 1 wrapping does not support the Firebase Cloud Messaging library starting with `com.google.firebase.firebase-messaging:17.0.0`.

If the `-enableCrashlytics` or `-allowCrashlytics` option is used when wrapping the app, the crash data is available on the Firebase Crashlytics console. The app must support Firebase Crashlytics. The crash reports appear for the unwrapped app package name on the console. However, the reports include the label (**wrapped app**), which identifies the report as an AppConnect wrapped app crash report.

MediaPlayer and MediaMetadataRetriever Internet permission requirement

You can wrap apps that use the `android.media.MediaPlayer` and `android.media.MediaMetadataRetriever` APIs. However, wrapper support of these APIs require that wrapped apps set the Android Internet permission.

You can set the Android Internet permission in your app in one of the following ways:

- Add the permission `android.permission.INTERNET` to your app's `AndroidManifest.xml` file before you submit it to the AppConnect Wrapping Portal.



- In the AppConnect Wrapping Portal, add the following flag in the **Advanced Option** section in the **Extra options** field:
`-addInternetPermission`

MobileIron provides a sample app, ApiUsageDemo, that demonstrates the necessary code.

Related topics

- [Preparing to use the wrapping tool](#) (when using the AppConnect for Android Wrapping Tool)
- [Android API Usage Demo sample app overview](#)

Image selection from outside the AppConnect container

To select an image from **outside** the AppConnect container, such as the device's gallery, the app must be granted the permission `Manifest.permission.READ_EXTERNAL_STORAGE`. This permission is necessary because of how the wrapper implements selecting an image from outside the AppConnect container.

Do the following in your app if it selects images from outside the AppConnect container:

1. Add the permission `android.permission.READ_EXTERNAL_STORAGE` to your app's `AndroidManifest.xml` file before you submit it to the AppConnect Wrapping Portal.
2. Provide code to grant the permission at runtime.

MobileIron provides a sample app, ApiUsageDemo, that demonstrates the necessary code.

Related topics

- [Sample apps, tester app, and Cordova plugin](#)
- [Android API Usage Demo sample app overview](#)

External storage permissions

When a wrapped app saves data to its own app-specific directory on external storage, such as an SD card, the wrapper saves the data to a particular location depending on the wrapper version. The storage location, which is returned by `getExternalFilesDir()`, determines whether the wrapped app requires the external storage permission (`WRITE_EXTERNAL_STORAGE`). The following table provides the storage location and permission requirements.

TABLE 9. EXTERNAL STORAGE PERMISSIONS

Wrapper version	External storage location	Does the wrapped app require the <code>WRITE_EXTERNAL_STORAGE</code> permission?
Prior to 8.2	<code>/<sdcard_path>/AppConnect/</code>	Yes. The permission is necessary because the storage location is not specific to the wrapped app's package name.
8.2 through the most recently released version as supported by MobileIron	<code>/<sdcard_path>/Android/data/<wrapped_pkg>/</code>	No. The permission is not necessary because the storage location is specific to the wrapped app's package name.

However, consider the case when you are re-wrapping an app that was previously wrapped with a wrapper version prior to 8.2. In this case, **the wrapped app still requires the `WRITE_EXTERNAL_STORAGE` permission**. The wrapped app still needs the permission so that the wrapper can move the data from the pre-8.2 storage location to the new location. The wrapper moves the data after the wrapped app finishes its initialization, so **make sure the permission is granted during initialization**. After the wrapper has moved the data, later versions of the wrapped app do not require the `WRITE_EXTERNAL_STORAGE` permission to save data to the app-specific external storage directory.

NOTE: Apps request permissions at runtime when running on Android 6.0 through the most recently released version as supported by MobileIron. When running Android versions prior 6.0, apps request permissions at installation time. Therefore, the runtime permission requirements for moving the data do not apply on Android versions prior to 6.0.

Support for scoped storage

AppConnect supports scoped storage. If the `targetSdkVersion` of the app is set to 30, ensure that the app is wrapped with the Generation 2 Wrapper. No additional setup is required.

IMPORTANT: If you set `targetSdkVersion` to 29, you can wrap the app with the Generation 1 wrapper. However, ensure that `android:requestLegacyExternalStorage="true"` in `AndroidManifest.xml`. Otherwise, Android activates the `ScopedStorage` by default.

Receiving information from outside the AppConnect container

Sometimes wrapped apps need to receive information from outside the AppConnect container that is different than what is supported by the MobileIron server policies and configurations. For example, an app might need to receive a system broadcast event.



If your wrapped app has this requirement, you can add a wrapping flag that specifies which Android intent action you want to allow the app to receive. The wrapping flag is `-allowIntentAction`. The flag is followed by a list of actions to allow. Separate actions with spaces.

For example:

```
-allowIntentAction android.intent.action.TIMEZONE_CHANGED
```

USB OTG support

Wrapped apps can create, read, and write unencrypted image and text files to a USB OTG (on-the-go) drive that uses a FAT32 file system.

For a wrapped app to be able to take these actions, a key-value pair is required on the Secure Apps Manager's AppConnect app configuration on MobileIron Core. The new key is `MI_AC_ENABLE_USB_UNENCRYPTED_ACCESS`. Access to files on a USB OTG drive is allowed only if the value of the key is **true**. The value defaults to **false**.

NOTE: This feature is supported only with MobileIron Core and Connected Cloud. It is not supported with MobileIron Cloud.

Preference API usage

If your app uses the `PreferenceActivity` or `PreferenceFragment` APIs:

- If you wrap with the Generation 1 wrapper, change your app to use the `PreferenceFragmentCompat` APIs for the app to behave correctly on Android 9.0 through the most recently released version as supported by MobileIron.
- If you wrap with the Generation 2 wrapper, MobileIron recommends that you change your app to use the `PreferenceFragmentCompat` APIs, because Google has deprecated the `PreferenceActivity` and `PreferenceFragment` APIs.

Google deprecated the `PreferenceActivity` API in Android 3.0 and the `PreferenceFragment` API in Android 9.0 (API level 28).

64-bit support

- Generation 1 wrapper provides 64-bit support by default.
- Generation 2 wrapper wraps apps with native libraries in both 32-bit and 64-bit versions. However, if the app does not use native libraries, the app is wrapped only with 64-bit support.

You can disable 64-bit mode by entering the `-disableArm64` option. For more information about using the option see [Uploading and wrapping an app](#) and [Preparing to use the wrapping tool](#).



Linking native Java methods

In wrapped apps, native java methods are linked as soon as the client native library loads. In unwrapped apps, the methods are linked on their first call.

Configure the flag `-keepJavaNativesLazyLinking` to preserve the old linking mechanism for backward compatibility. However, do not use this option for wrapping Xamarin apps.

You can disable 64-bit mode by entering the `-disableArm64` option. For more information about using the option see [Uploading and wrapping an app](#) and [Preparing to use the wrapping tool](#).

Wrapping support of commonly used app capabilities

The following sections summarize the app capabilities that both the Generation 1 and 2 wrappers support, and the app capabilities supported by only one wrapper or the other.

Related topics

- For information about the types of apps supported with each wrapper mode, see [Determining the wrapping mode](#)
- For information about supported processors and native libraries, see [Android devices supporting AppConnect apps](#).
- For information about APIs supported and not supported with AppTunnel with HTTP/S tunneling, see [AppTunnel with HTTP/S tunneling](#).

Generation 1 and Generation 2 support for commonly used app capabilities

The following commonly used app capabilities are supported by both the Generation 1 wrapper and the Generation 2 wrapper:

- PHONE_STATE, SCREEN_ON, and SCREEN_OFF broadcasts can be received in wrapped apps.
- URL intents from apps outside the container can be allowed into the container.
- Apps containing multiple dex files are supported.
- Badge count APIs (android.intent.action.BADGE_COUNT_UPDATE) are supported.
- Apps can have as many methods as the system allows.
- Using the Google Maps API is supported.
- Using the Google Maps APIs Premium Plan is supported.
- Wrapping obfuscated code is supported.
- Using DocumentsProvider APIs are supported.

For apps wrapped with the Generation 2 wrapper, the app can run but the DocumentsProvider API functionality is not supported.



- Identifiers in DEX bytecodes can contain characters that are legal in Kotlin or Java.
- Using a service which extends `android.app.job.JobService` is supported.
- Using the `javax.xml.parsers.SAXParser.parse()` method is supported.
- Using the `com.squareup.okhttp.OkHttpClient` class is supported.
- Using `android.settings.LOCATION_SOURCE_SETTINGS` is supported.
- Using Google Play Services 9.0 APIs is supported.
- Apps which target Android API levels through 30.
- The `android.support.v4.content.FileProvider` API is supported.
- The `android.provider.MediaStore` API is supported for media on the device.
- The `android.media.MediaRecorder` API is supported with the device's native media recorder.
MobileIron recommends calling `MediaRecorder.stop()` in a thread other than the user interface thread, and showing a progress indicator when stopping the recording. This recommendation is because when the media recorder stops recording, the wrapper encrypts the recording. Therefore, saving the recorded video can be slow, causing a noticeable delay after the device user stops recording. This delay is most noticeable with video recordings when using the Generation 1 wrapper.
- The `android.media.MediaPlayer` API is supported with the device's native media player. Streaming media from a remote source with this API is not supported when using AppTunnel with HTTP/S tunnel or AppTunnel with TCP tunneling.
- The `android.media.MediaMetadataRetriever` API is supported with the device's native media player.
- The `android.speech.RecognizerIntent` class is supported for starting an activity with the action `ACTION_RECOGNIZE_SPEECH`.
- The intents `android.intent.action.OPEN_DOCUMENT`, `CREATE_DOCUMENT`, and `OPEN_DOCUMENT_TREE` are supported for OTG USB storage as described in [USB OTG support](#).
- The `<layout>` manifest element is supported.
- The action `com.android.camera.action.CROP` is supported to share cropped images with non-AppConnect apps. This feature requires setting the key `MI_AC_SHARE_CONTENT` to `True` in the Secure App Manager's AppConnect app configuration in MobileIron Core. This feature is supported only when the Google Photo app is used to crop the image.
- Using the Crashlytics library is supported by using the wrapping flag `-enableCrashlytics`. If you are using the AppConnect Wrapping Portal, see [The AppConnect wrapping portal](#). If you are using the AppConnect wrapping tool, see "Selecting wrapping options in the wrapping tool" in [Preparing to use the wrapping tool](#).
- Using the intent `ACTION_MEDIA_SCANNER_SCAN_FILE` in a broadcast action to save images into the Android gallery or other media database is supported when the Gallery data loss prevention policy on the MobileIron server is set to allowed.
- Kotlin coroutines.
- `BlobStoreManager.Session.allowPublicAccess()` which, in wrapped apps, behaves like the `allowSameSignatureAccess()` call. This prevents wrapped apps from providing access to unwrapped apps.
- Realm Database library.



- The following ThumbnailUtils deprecated methods are supported: createAudioThumbnail, createImageThumbnail, createVideoThumbnail.

Related topics

- [Android API Usage Demo sample app overview](#)

Generation 1 wrapper support for commonly used app capabilities

- The Generation 1 wrapper supports data encryption only for Java file I/O.
- The intents android.intent.action.OPEN_DOCUMENT and OPEN_DOCUMENT_TREE are supported for device storage and for SD card storage.

CREATE_DOCUMENT is not supported.

Generation 2 Wrapper support for commonly used app capabilities

The following commonly used app capabilities are supported by only the Generation 2 wrapper:

- Data Encryption of all file I/O
- AppTunnel with TCP Tunneling as described in [AppTunnel with TCP tunneling](#).
- Java Native Interface (JNI) call support:
 - Calling from Java to native code
 - Calling from native code to Java
- Passing file descriptors with Parcels across processes
- Calls to Runtime.exec are supported with a special flag. Contact MobileIron Technical Support. This will function only if the binary being executed is not performing any direct file I/O.
- AccountManager APIs getAccounts(), getAccountsByType(), and getAccountsByTypeAndFeatures() are supported.
- The intents android.intent.action.OPEN_DOCUMENT, CREATE_DOCUMENT, and OPEN_DOCUMENT_TREE are supported for device storage and for SD card storage.
- Calls (by reflection) to the private method java.lang.Runtime.nativeLoad()
- Wrapping apps that use 64-bit native libraries (C or C++ libraries)
- Scoped storage
See [Support for scoped storage](#).

Known wrapper limitations

The following sections summarize the limitations of app capabilities of both the Generation 1 and 2 wrappers, and the limitations of only one wrapper or the other.



Related topics

- For information about the types of apps supported with each wrapper mode, see [Determining the wrapping mode](#)
- For information about supported processors and native libraries, see [Android devices supporting AppConnect apps](#).
- For information about APIs supported and not supported with AppTunnel with HTTP/S tunneling, see [AppTunnel with HTTP/S tunneling](#).

Generation 1 and Generation 2 wrapper limitations

Apps wrapped with either the Generation 1 or Generation 2 wrapper have limitations regarding app capabilities.

The following limited app capabilities have been identified:

- Accessing Google Play from the AppConnect app is not supported.
- Google Cloud Messaging XMPP Mode is not supported, and the deprecated GCM APIs (`com.google.android.gcm`) are not supported.
See also [Google Cloud Messaging considerations](#).
- Embedded SQLite databases are not supported.
If you embed an SQLite database in your app as an asset or raw resource, the app cannot be wrapped. Similarly, you cannot download an SQLite database from over the network and use it in your app. However, programmatically creating SQLite databases **is** supported.
- Creating home screen shortcuts is not supported.
- Hidden APIs are not supported. Using hidden APIs can cause undefined behavior in your app, including possible data leakage.
- Android widgets are not supported.
- `android.security.KeyChain` APIs are not supported.
- Launcher icons must be in RGB format.
- Launcher icons must be square.
- Android Device Administration APIs are not supported.
- Some non-standard libraries are not supported with AppTunnel with HTTP/S tunneling.
See [AppTunnel with HTTP/S tunneling on page 22](#).
- `java.util.jar.JarFile` is not supported.
- `java.net.JarURLConnection` is not supported.
- Using SQLCipher and Android SQLite databases at the same time is not supported.
Developers should move all databases to SQLCipher or Android SQLite.



- Creating documents using the DocumentsProvider action `android.intent.action.CREATE_DOCUMENT` is not supported.
- Installing other apps is not supported.
- Using multiple JAR files using the Android Support Library is not supported.
- The `DexClassLoader` class is not supported.
- The deprecated contacts authority (`android.provider.Contacts`) is not supported. Only the newer `android.provider.ContactsContract` is supported.
- Using packages starting with an upper case character is not supported.
- Access to `content://media/` is not supported.
As a result, camera access in PhoneGap apps is not supported using the PhoneGap APIs.
- If an app reads and writes application directories on external storage, the `[READ|WRITE]_EXTERNAL_STORAGE` permission must be specified in the Android manifest.
- URIs are rewritten in WebView objects, which potentially impacts Content-Security-Policies:
 - File:// URIs are rewritten to allow access to encrypted files. For example, the URI `file://someFileName` is rewritten as `content://com.forgepond.0.<pkg_name>/<data>`.
 - Content:// URIs are rewritten to restrict access to apps within the container. For example, the URI `content://foo/bar` is rewritten as `content://forgepond.foo/bar`.
- PrintManager APIs are not supported.
- Google Analytics APIs are not supported.
- Kiosk mode (screen pinning) is not supported.
- Wrapped Cordova apps cannot read from the photo gallery using `com.synconset.MultiImageChooserActivity` or `com.synconset.ImagePicker` because they do not create an `ACTION_GET_CONTENT` intent. Ensure Cordova plugins use `ACTION_GET_CONTENT` intent for image selection.
- Using `JobIntentService` and manifest permission "`android.permission.BIND_JOB_SERVICE`" is not supported unless you add the following code to Proguard:


```
-keep class android.support.v4.app.JobIntentService {
    *;
}
```
- Wrapped Cordova apps cannot read from the photo gallery using `com.synconset.MultiImageChooserActivity` or `com.synconset.ImagePicker` because they do not create an `ACTION_GET_CONTENT` intent. Ensure Cordova plugins use the `ACTION_GET_CONTENT` intent for image selection.
- When a wrapped app attempts to retrieve a file from external storage using an Intent with the action `ACTION_PICK`, only content in the Gallery app with types `image/` or `video/` are presented to the user to



choose from.

- Wrapping apps that load unwrapped Java code is not supported. Although wrapping the app succeeds, the app will fail when it runs.

Generation 1 wrapper limitations

Apps wrapped with the Generation 1 wrapper have limitations regarding app capabilities.

The following limited app capability has been identified:

- AppTunnel with TCP tunneling is not supported.

Generation 2 wrapper limitations

Apps wrapped with the Generation 2 wrapper have limitations regarding app capabilities.

- Upgrading a Generation 1 wrapped app to a Generation 2 wrapped version of the app is not supported.
- Creating multiple processes is not supported.
- Calling `exec()` via native libraries are not supported.
Java's `Runtime.exec()` is supported, with some limitations as described in [Generation 2 Wrapper support for commonly used app capabilities on page 94](#).
- AppTunnel with TCP tunneling is not supported for apps that use OOB (Out-of-band messaging) and TCP urgent packets.
- Wi-Fi proxies are not supported with AppTunnel with TCP tunneling.
- Calling `mmap()` to map private pages (the `MAP_PRIVATE` flag) is not supported.
- Calling `mremap()` with the `MREMAP_FIXED` flag is not supported on file descriptors.
- The `DocumentsProvider` API functionality is not supported. However, the app can be wrapped.
- Although Generation 2 wrapping supports wrapping apps that load native libraries, it does not support loading native libraries by any means other than system calls such as `System.load`, `System.loadLibrary`, or `dlopen()`.



Legacy mechanism for handling AppConnect app-specific configuration

IMPORTANT:

- This mechanism for handling AppConnect app-specific configuration is deprecated, although still supported.
- This mechanism is replaced by the mechanism described in [Handling AppConnect app-specific configuration](#).
- If you have already used this legacy mechanism, modify your app to use the new mechanism as soon as possible.
- If you are adding app-specific configuration handling to your app for the first time, use the new mechanism.

For information about the legacy mechanism for handling AppConnect app-specific configuration, see:

- [Overview of legacy configuration handling](#)
- [Communicating with the MobileIron client app using intents for legacy configuration handling](#)
- [App-specific configuration legacy data flow](#)
- [Contents of the Intent objects in legacy configuration handling](#)
- [Tasks for legacy configuration handling](#)
- [Sample Java app for legacy app-specific configuration handling](#)
- [App for testing legacy configuration handling](#)
- [Best practices for handling app-specific configuration](#)

Overview of legacy configuration handling

The MobileIron server administrator can set up app-specific configuration on the server for AppConnect for Android apps. This configuration is in the form of key-value pairs. Your app can receive these key-value pairs. Specifically, when you implement configuration handling in your app, your app:

- requests the current configuration when it first runs.
Your app then receives an asynchronous response containing the key-value pairs.
- receives updates to the configuration.

MobileIron provides the following apps to help you add configuration handling to your Java app:



- a sample AppConnect app, HelloAppConnect-oldAPI, that implements configuration handling. You can use this sample app's code as a starting point for your own.
- an app for testing your app's configuration handling, without using the MobileIron server, or the Mobile@Work or MobileIron Go app, and without submitting your app to MobileIron for wrapping. The testing app that supports the legacy configuration handling is available only in releases prior to AppConnect 8.6.0.0 for Android.

This chapter describes how to receive app-specific configuration in Java apps using the legacy mechanism

Phonegap developers

You can implement app-specific configuration in a Phonegap app by using a MobileIron-provided Cordova plugin. This plugin provides the necessary APIs to receive the app-specific configuration from the MobileIron server.

The Cordova plugin and sample Phonegap app that uses the legacy mechanism for configuration handling is available **only in releases prior to AppConnect 8.6.0.0 for Android**. They can be found in

- AppConnectCordovaConfigPlugin-w.x.y.z.zip, which contains the the Cordova plugin. (w.x.y.z corresponds to the AppConnect version for Android)
See the README.md in the ZIP file for information on using the plugin.
- HelloCordovaAppConnect-w.x.y.z.zip, which contains a sample Phonegap app that uses the plugin, available as a starting point for your own app.

React Native developers

You can implement app-specific configuration in a React Native app by using MobileIron-provided files that make up a React Native package called ConfigServicePackage.

The package and sample demo app that use the legacy mechanism for configuration handling are available **only in releases prior to AppConnect 8.6.0.0 for Android**.

The files provide the necessary APIs to receive the app-specific configuration from the MobileIron server. MobileIron provides a sample React Native app called HelloReact that includes:

- all files relating to getting app-specific configuration
- a README.txt with instructions for using the files
- sample code for using the files

Related topics

- [Sample Java app for legacy app-specific configuration handling.](#)
- [App for testing legacy configuration handling](#)



Communicating with the MobileIron client app using intents for legacy configuration handling

The MobileIron server passes the app-specific configuration to the MobileIron client app (Mobile@Work when using MobileIron Core or Connected Cloud, and MobileIron Go when using MobileIron Cloud). The MobileIron client app in turn passes the configuration to your app. The communication between the MobileIron client app and your app uses intents (android.content.Intent objects). The MobileIron client app and your app each have services to handle the Intent objects. The Intent objects are implicit intents, so Intent filters and action values in the Intent object tell the system which services handle the intent.

NOTE: On Android 5.x or newer devices, you must use explicit intents. To use an explicit intent, use `PackageManager.resolveService(intent, 0);` and then call `startService()`.

The Intent objects passed between the MobileIron client app and your app are:

TABLE 10. INTENT OBJECTS PASSED BETWEEN MOBILEIRON CLIENT AND APP

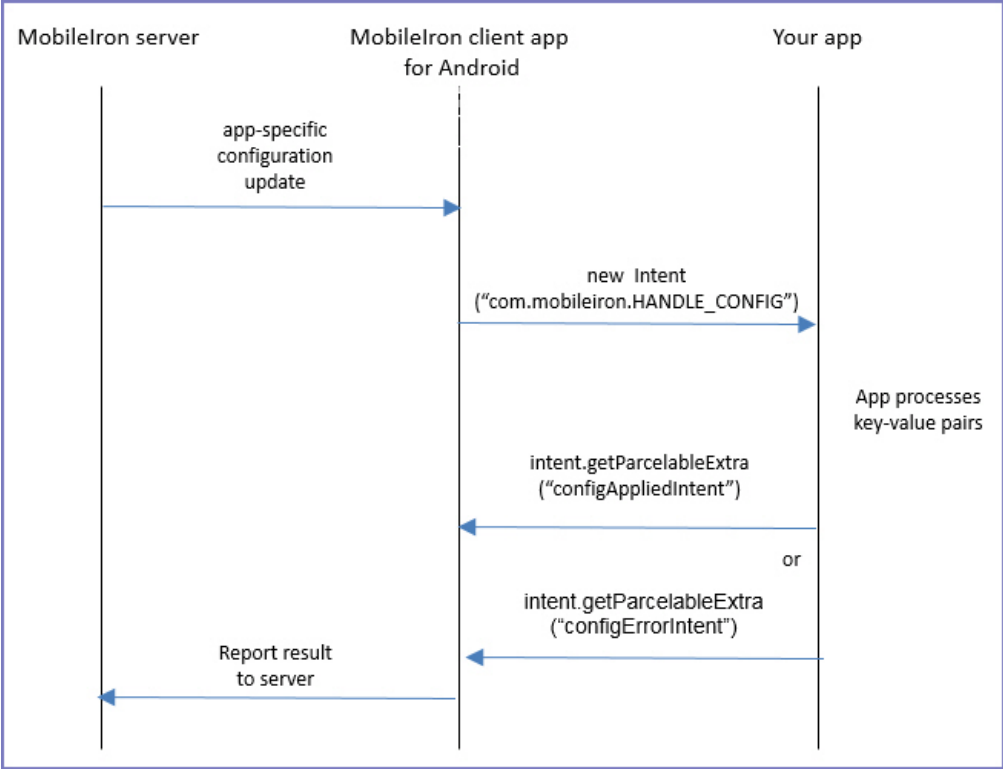
Intent object	Description
Intent object with the action: "com.mobileiron.REQUEST_CONFIG"	When your app makes a request for the current configuration, it starts a service in the MobileIron client app. Specifically, the app calls <code>startService()</code> , passing an Intent object with this action.
Intent object with the action: "com.mobileiron.HANDLE_CONFIG"	When the MobileIron client app has a configuration update for your app, it starts a service in your app. Specifically, the MobileIron client app calls <code>startService()</code> , passing an Intent object with this action.
Intent object in extended data named: "configAppliedIntent" in the Intent object with the action: "com.mobileiron.HANDLE_CONFIG".	When your app successfully handles the key-value pairs sent in a configuration update, it notifies the MobileIron client app of the success. Specifically, your app calls <code>startService()</code> , passing this Intent object.
Intent object in extended data named: "configErrorIntent" in the Intent object with the action: "com.mobileiron.HANDLE_CONFIG".	When your app fails to successfully handle the key-value pairs sent in a configuration update, it notifies the MobileIron client app of the failure. Specifically, your app calls <code>startService()</code> , passing this Intent object.

App-specific configuration legacy data flow

The following sequence diagram shows the flow of data between the MobileIron server, the MobileIron client app, and your app when the MobileIron server administrator has updated the app-specific configuration on the server:



FIGURE 8. APP-SPECIFIC CONFIGURATION DATA FLOW



NOTE: When your app first runs and requests the configuration, the sequence begins with your app sending a "com.mobileiron.REQUEST_CONFIG" intent to the MobileIron client app. After that, the sequence continues as shown in the diagram.

Contents of the Intent objects in legacy configuration handling

The following table shows the contents that you work with in each Intent object:



TABLE 11. CONTENTS OF EACH INTENT OBJECT

Intent object	Other data in an Intent object with this action
Intent object with the action: <code>"com.mobileiron.REQUEST_CONFIG"</code>	<ul style="list-style-type: none"> your app's package name, in extended data named <code>"packageName"</code>. The extended data has a string value that is the name of your app's package.
Intent object with the action: <code>"com.mobileiron.HANDLE_CONFIG"</code>	<ul style="list-style-type: none"> a Bundle object, in extended data named <code>"config"</code>. The Bundle object contains the key-value pairs of the configuration. an Intent object to send to the MobileIron client app after successfully processing the key-value pairs. This Intent object is in extended data named <code>"configAppliedIntent"</code> an Intent object to send to the MobileIron client app after failing to process the key-value pairs. This Intent object is in extended data named <code>"configErrorIntent"</code>.
Intent object in extended data named: <code>"configAppliedIntent"</code> in the Intent object with the action <code>"com.mobileiron.HANDLE_CONFIG"</code> .	Your app does not add any further data to this Intent object.
Intent object in extended data named: <code>"configErrorIntent"</code> in the Intent object with the action <code>"com.mobileiron.HANDLE_CONFIG"</code> .	<ul style="list-style-type: none"> an error string, in extended data named <code>"errorString"</code>. The string describes the error that occurred when your app failed to process the key-value pairs.

Tasks for legacy configuration handling

To handle app-specific configuration using the legacy mechanism in your app, do the following high-level tasks:

- [Check at runtime if your app is wrapped.](#)

This check is typically necessary if you are a third-party app developer using the same source code to create a Google Play app and an in-house AppConnect app. Only wrapped AppConnect apps can receive app-specific configuration from a MobileIron server.

If you are developing an app that will be distributed only as an in-house app, not from Google Play, you will not use this check.

- [Add a service to AndroidManifest.xml.](#)

Add a service for handling configuration intents to your app's AndroidManifest.xml file. In the service, you specify an intent-filter for the intent with the action `"com.mobileiron.HANDLE_CONFIG"`.

- [Create a class that extends IntentService.](#)



The name of the class matches the `android:name` attribute of the service you add to the `AndroidManifest.xml` file. You implement the `onHandleIntent()` method, in which you asynchronously receive the configuration's key-value pairs when they change or you request them.

- [Request the configuration when your app starts.](#)

When your app starts, request the configuration, which your app will receive asynchronously in its `onHandleIntent()` method.

- [Specify app configuration and policies in .properties files.](#)

You can include `.properties` files in your app that list your app's key-value pairs and data loss prevention (DLP) policies. When the MobileIron server administrator uploads your app to the server, these files cause the server to automatically configure the key-value pairs and DLP policies.

Check at runtime if your app is wrapped

If you are a third-party developer, you sometimes develop an app in which the same source code is used in these ways:

- as a wrapped app distributed from the MobileIron server's App Catalog
This secure AppConnect app is for enterprise device users.
- as an unwrapped app distributed from Google Play
This unsecured app is for general distribution.

An app that serves both these markets typically behaves differently depending on whether it is a wrapped, secure AppConnect app.

For example:

- If a wrapped app expects key-value pairs from the MobileIron server, but does not receive the expected pairs or valid values, it should take appropriate actions.
As a best practice, if your app expects a login ID from the server, but does not receive one, do not allow the device user to enter the ID manually. See [Use only a login ID from the MobileIron server if one is expected.](#)
- If an app is not wrapped, it cannot get its configuration from the MobileIron server. It gets configurable information another way, such as prompting the device user to enter it.

For example, the unwrapped app prompts the user to enter a login ID.

To determine at runtime whether the app is running as a wrapped app, check this Android system property:

```
"com.mobileiron.wrapped"
```

For example, use the following expression:

```
Boolean.parseBoolean(System.getProperty("com.mobileiron.wrapped", "false"))
```

The expression returns `true` if the app is wrapped. Otherwise, it returns `false`.



Add a service to AndroidManifest.xml

Add a `<service>` element to your app's `AndroidManifest.xml` file. This service is for handling the configuration intent sent from the MobileIron client app.

For example, in `HelloAppConnect`, the lines in `AndroidManifest.xml` are:

```
<service
  android:permission="com.mobileiron.CONFIG_PERMISSION"
  android:enabled="true"
  android:name=".AppConnectConfigService">
  <intent-filter>
    <action android:name="com.mobileiron.HANDLE_CONFIG"/>
  </intent-filter>
</service>
```

Do the following in the `<service>` element:

1. Set the attribute `android:permission="com.mobileiron.CONFIG_PERMISSION"` to ensure that only the MobileIron client app can start this service.
This permission is necessary only if you are a third-party app developer planning to distribute an unwrapped version of your app. For more information, see [App for testing legacy configuration handling](#).
2. Ensure that the attribute `android:exported` is `"true"`, which it is by default.
3. Ensure that the attribute `android:enabled` is `"true"`, which it is by default.
4. Ensure that the attribute `android:enabled` of the `<application>` element that contains the `<service>` element is `"true"`, which it is by default.
5. Set the `android:name` attribute of the `<service>` element to the name of the class in your app that extends `IntentService`. This class in your app implements `onHandleIntent()` to handle configuration updates.
6. Add an `<intent-filter>` element that contains an `<action>` element.
7. Set the `android:name` attribute of the `<action>` element to `"com.mobileiron.HANDLE_CONFIG"`.

Create a class that extends IntentService

Create a class that extends `IntentService`. This class handles the intent with the action `"com.mobileiron.HANDLE_CONFIG"`.

Do the following:

1. Name the class the same name you specified in the `android:name` attribute of the `<service>` element in the `AndroidManifest.xml` file.
2. Implement `onHandleConfig()` to handle the received `Intent` object with the action `"com.mobileiron.HANDLE_CONFIG"`.

Implement onHandleConfig()

Implementing `onHandleConfig()` involves the following steps. Code samples are from `HelloAppConnect`.



1. Get the Bundle object contained in the received Intent object.

For example:

```
Bundle config = intent.getBundleExtra("config");
```

NOTE: If no configuration exists on server for your app, the Bundle object is null. If a configuration exists on server, but contains no key-value pairs, the Bundle object is not null, but its `keySet()` method returns an empty set.

2. Extract the key-value pairs in the Bundle object.

For example:

```
Map<String, String> map = new HashMap<String, String>();
for (String key: config.keySet()) {
    map.put(key, config.getString(key));
}
```

3. Process the key-value pairs according to your app's requirements.
4. If the app successfully processes the key-value pairs, start a service. Pass the service the "success" Intent object. The "success" Intent object is in the received Intent object, and has the extended data name "configAppliedIntent".

For example:

```
startService ((Intent)intent.getParcelableExtra("configAppliedIntent"));
```

5. If the app fails to process the key-value pairs, start a service. Pass the service the "error" Intent object. The "error" Intent object is in the received Intent object, and has the extended data name "configErrorIntent". Include a string in the "error" Intent object, where the string value describes the error condition.

For example:

```
Intent i = (Intent)intent.getParcelableExtra("configErrorIntent");
i.putExtra("errorString", "This is a sample error message.");
startService(i);
```

Reasons for returning an error

If your app fails to process the key-value pairs, it starts a service, passing the "error" Intent object. Some reasons for failure are:

- A value is not valid for its key.
For example, if the key is "emailAddress", but the value does not include the @ character, return an error.
- A value is empty.
Typically, if a key is included in the MobileIron server configuration for your app, your app expects a value. If the MobileIron server administrator did not enter a value, return an error.
- Your app encounters a system error while processing a key-value pair.
Your app determines whether a system error impacts key-value processing to warrant an error return.

When the app returns an error, how it continues to operate depends on your app's design and requirements.



Request the configuration when your app starts

When your app starts, request the app-specific configuration. Do the following:

1. Create an Intent object.
2. Set the Intent object's action to `com.mobileiron.REQUEST_CONFIG`.
3. Add your app's package name as extended data to the Intent object.
4. Call `startService()` with the Intent object.
5. Handle the asynchronous response in `onHandleConfig()`.

For example, based on code in `HelloAppConnect`:

```
Intent intent = new Intent("com.mobileiron.REQUEST_CONFIG");
intent.putExtra("packageName", ctx.getPackageName());
ctx.startService(intent);
```

NOTE: Request the configuration only once, when your app starts. After that, whenever the MobileIron server administrator updates the configuration on the server, your app automatically receives the Intent object with the action `com.mobileiron.HANDLE_CONFIG`.

Specify app configuration and policies in .properties files

You can include the following .properties files with your app:

- `appconnectconfig.properties`
This file specifies your app's configuration keys and their default values, if any. Providing this .properties file causes the MobileIron server to automatically configure the keys and their default values on the server.
- `appconnectpolicy.properties`
This file specifies the default data loss prevention policy for screen capture for the app. Specifically, it specifies whether screen capture is allowed in the app. The policy is enforced by the AppConnect wrapping technology.

If your app contains these .properties files, the MobileIron server automatically configures the key-value pairs and the screen capture policy that you specified. This automatic configuration occurs when the MobileIron server administrator uploads your app to the server's App Catalog.

The administrator can then change the default values on the server as necessary for that enterprise.

File location of the .properties files

Put the .properties files in this directory in your app:

`<application root directory>/res/raw`



Example of the appconnectconfig.properties file

An example of an appconnectconfig.properties file is available in HelloAppConnect.

It contains the following:

```
# This sample appconnectconfig.properties file uses rules found at
# http://en.wikipedia.org/wiki/.properties.

server=www.myCompanyApplicationServer.com
port=8080

# In the following example, the resulting property value contains only single spaces.
# It contains no other whitespace.
# Therefore, the value is: "I'm also demonstrating a multi-line property!"

name\ with\ spaces:I'm also demonstrating \
a multi-line property!

# Use an empty value for keys that have no default value.

nodefault=

! You can also start comments with exclamation marks.

# You can use these MobileIron Core variables for values:
# $USERID$, $EMAIL$, $PASSWORD$,
# $USER_CUSTOM1$, $USER_CUSTOM2$, $USER_CUSTOM3$, $USER_CUSTOM4$

# You can use these MobileIron Cloud variables for values:
# ${userID}, ${userEmailAddress}
# ${USER_CUSTOM1}, ${USER_CUSTOM2}, ${USER_CUSTOM3}, ${USER_CUSTOM4}

userid=$USERID$
email=$EMAIL$
user_custom1=$USER_CUSTOM1$
combined=$USERID$::$EMAIL$
```

Format of the appconnectconfig.properties file

Use the rules for well-formed Java property files given in the Java Properties class. For example, use the characters = or : or a space to separate the key from the value. Use \ before each of these characters if the character is part of the key.

The values that you specify are the default values for the key. If the value has no default, leave the value empty.



A value can be any string. The value can also use one of the following server variables:

TABLE 12. SERVER VARIABLES IN DEFAULT VALUES OF KEYS

MobileIron Core variable	MobileIron Cloud variable	Description
\$USERID\$	\${userUID}	The device user's enterprise user ID, typically an LDAP ID.
\$PASSWORD\$	Not available	The device user's enterprise user password, typically an LDAP password.
\$EMAIL\$	\${userEmailAddress}	The device user's enterprise email address.
\$USER_CUSTOM1\$ \$USER_CUSTOM2\$ \$USER_CUSTOM3\$ \$USER_CUSTOM4\$	\${USER_CUSTOM1} \${USER_CUSTOM2} \${USER_CUSTOM3} \${USER_CUSTOM4}	Custom variables that the MobileIron server administrator sets up. Only use these variables if you are developing an app for a specific MobileIron customer. Contact the server administrator to determine the values of these variables.

You can also specify values that are combinations of text and server variables. For example, using MobileIron Core variables:

- \$USERID\$::\$EMAIL\$
- \$USERID\$@somedomain.com

Use server variables for default values in your appconnectconfig.properties only if you know what kind of server (MobileIron Core or MobileIron Cloud) your app will be used with. If you don't know, leave the value empty. The server administrator will fill in the value.

Example of the appconnectpolicy.properties file

An example of an appconnectpolicy.properties file is available in HelloAppConnect.

It contains the following:

```
# A sample appconnectpolicy.properties file
screencapture=disable
```

Format of the appconnectpolicy.properties file

To disable screen capture in the app, include the following line in appconnectpolicy.properties:

```
screencapture=disable
```

To allow screen capture:

```
screencapture=allow
```

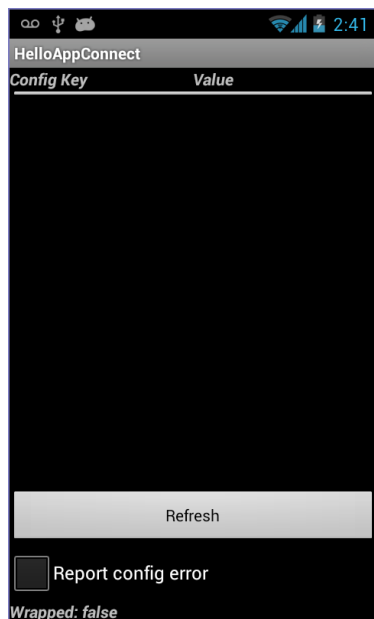


Sample Java app for legacy app-specific configuration handling

MobileIron provides a sample Java app, HelloAppConnect-oldAPI, which handles app-specific configuration using the legacy mechanism for app-specific configuration handling. You can use the code from this app as a starting point for your app's configuration handling.

HelloAppConnect-oldAPI displays this screen:

FIGURE 9. HELLOAPPCONNECT-OLDAPI SCREEN



The HelloAppConnect-oldAPI app:

- sends a `"com.mobileiron.REQUEST_CONFIG"` Intent object when you tap Refresh.
- handles the `"com.mobileiron.HANDLE_CONFIG"` Intent object in `onHandleIntent()`. It displays the received key-value pairs, and sends either a `"configAppliedIntent"` or `"configErrorIntent"` Intent object. The choice depends on whether you select Report Config Error. When it sends an error, it includes the error string `"This is a sample error message."`
- displays whether the app is wrapped, based on the value of the system property `"com.mobileiron.wrapped"`

App for testing legacy configuration handling

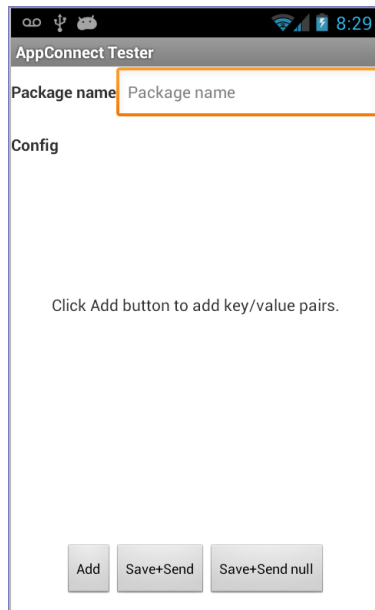
To test your app's configuration handling without using a MobileIron server or MobileIron client app, you can use the AppConnectTester app. Use the AppConnectTester to test your app before you submit it to MobileIron for wrapping. You can enter key-value pairs into the AppConnectTester. It then passes the key-value pairs to your app just as the MobileIron client app would.

IMPORTANT: The testing app that supports the legacy configuration handling is available only in releases prior to AppConnect 8.6.0.0 for Android. No similar app is available for testing the mechanism described in [Handling AppConnect app-specific configuration](#).

Using AppConnectTester

The AppConnectTester app displays the following:

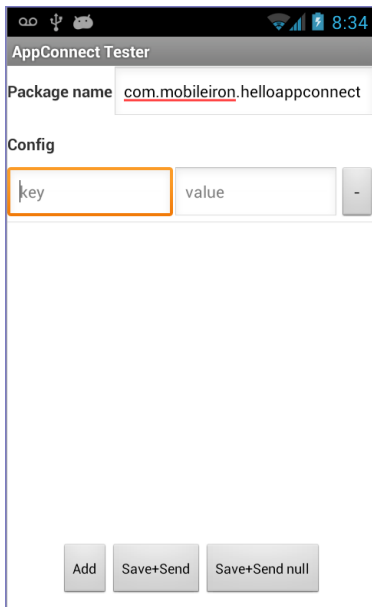
FIGURE 10. APPCONNECT TESTER



To use AppConnectTester to test your app:

1. Launch AppConnectTester.
2. Enter the package name of your app. For example:
com.mobileiron.helloappconnect
3. Tap **Add**.

FIGURE 11. APPCONNECT TESTER KEY-VALUE PAIR



4. Enter a key and value.
5. Repeat steps 3 and 4 as needed.
6. Tap **Save+Send**.
AppConnectTester saves your entries, and sends the key-value pairs to your app in the "HANDLE_CONFIG" Intent object.
7. If your app returns the "configAppliedIntent" Intent object, AppConnectTester displays "Success Received!"
8. If your app returns the "configErrorIntent" Intent object, AppConnectTester displays the error string that your app specified in the Intent object.

If you tap the button **Save+Send null**, AppConnectTester saves your entries but sends a null Bundle object in the "HANDLE_CONFIG" Intent object. In a production environment, the Bundle object is null when the MobileIron server administrator has not yet configured the AppConnect app configuration for your app on the server.

Protecting the unwrapped version of your app

If you are a third-party developer, and you are developing both a wrapped version and unwrapped version of your app with the same source code, consider the following scenario.

Your unwrapped version, distributed on Google Play, does not receive configuration from a MobileIron server. However, you use the unwrapped version to test configuration handling with AppConnectTester. Therefore, you must protect your unwrapped app from configuration sent from a potentially malicious app. This additional protection is not necessary for your wrapped app, because the wrapper provides this protection for you.

You provide the protection by using a permission that is granted to apps only if they have the same signature as your app. The AppConnectTester, by using the same signature as your app, can use the permission. When you receive the configuration intent in your `onHandleConfig()` method, when your app is not wrapped, your app checks whether it owns the permission:

- If your app owns the permission, process the intent because your app knows it is receiving the configuration intent from the AppConnectTester app.
- If your app does not own the permission, do not process the intent. Another app, installed before yours, must have declared the permission first, but without requiring apps to have the same signature to be granted the permission. Therefore, although the system granted your app the permission so that your app receives the intent, your app does not process the intent, since it was sent by a potentially malicious app.

Use the following steps to protect your unwrapped app from potentially malicious configuration intents:

1. In your app's `AndroidManifest.xml`, declare the following permission in the `<manifest>` element:

```
<permission
    android:name="com.mobileiron.CONFIG_PERMISSION"
    android:protectionLevel="signature" >
```

The "signature" protection level means the system will grant the permission to apps only if they are signed with the same certificate as your app.

2. In `AndroidManifest.xml`, request the permission in the `<manifest>` element:


```
<uses-permission android:name="com.mobileiron.CONFIG_PERMISSION" />
```
3. In `AndroidManifest.xml`, in your service that handles the configuration intents, require that the intent comes only from an app with the permission:


```
android:permission="com.mobileiron.CONFIG_PERMISSION"
```
4. In your `onHandleConfig()` implementation, process the configuration intent only if your app owns the permission:

```
if (!Boolean.parseBoolean(System.getProperty("com.mobileiron.wrapped", "false"))) &&
    !getPackageName().equals(getPackageManager().getPermissionInfo(
        "com.mobileiron.CONFIG_PERMISSION", 0).packageName)) {
    // Do not handle the Intent
    return;
}
```

5. Re-sign the AppConfigTester app so that it has the same key as your app.
When Eclipse builds an app in debug mode, by default it uses the same key for all apps. Therefore, rebuild the AppConfigTester app in Eclipse in debug mode so that it uses the same key as your app in debug mode.

Best practices for handling app-specific configuration

The following are best practices when handling app-specific configuration in your app:



- [Provide documentation about your app to the MobileIron server administrator](#)
- [Use only a login ID from the MobileIron server if one is expected](#)

Provide documentation about your app to the MobileIron server administrator

Document each key and its valid values. Document the default value, if applicable, and document whether the value can be empty. Provide this documentation regardless of whether your app includes an `appconnectconfig.properties` file.

Use only a login ID from the MobileIron server if one is expected

If a wrapped app expects a key-value pair for the device user's login ID, it should not prompt the user to enter the login ID manually. Using only a login ID from the MobileIron server ensures that a user can use the app only if the enterprise has authenticated the user. If the app does not receive an expected valid user ID, display an error message to the device user.

