

Signing Your Applications

This document provides information about signing your Android applications prior to publishing them for mobile device users.

Overview

The Android system requires that all installed applications be digitally signed with a certificate whose private key is held by the application's developer. The Android system uses the certificate as a means of identifying the author of an application and establishing trust relationships between applications. The certificate is not used to control which applications the user can install. The certificate does not need to be signed by a certificate authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates.

The important points to understand about signing Android applications are:

- All applications *must* be signed. The system will not install an application that is not signed.
- You can use self-signed certificates to sign your applications. No certificate authority is needed.
- When you are ready to release your application for end-users, you must sign it with a suitable private key. You can not publish an application that is signed with the debug key generated by the SDK tools.
- The system tests a signer certificate's expiration date only at install time. If an application's signer certificate expires after the application is installed, the application will continue to function normally.
- You can use standard tools — Keytool and Jarsigner — to generate keys and sign your application .apk files.
- Once you have signed the application, use the `zipalign` tool to optimize the final APK package.

The Android system will not install or run an application that is not signed appropriately. This applies wherever the Android system is run, whether on an actual device or on the emulator. For this reason, you must set up signing for your application before you will be able to run or debug it on an emulator or device.

The Android SDK tools assist you in signing your applications when debugging. Both the ADT Plugin for Eclipse and the Ant build tool offer two signing modes — *debug mode* and *release mode*.

- While developing and testing, you can compile in debug mode. In debug mode, the build tools use the Keytool utility, included in the JDK, to create a keystore and key with a known alias and password. At each compilation, the tools then use the debug key to sign the application .apk file. Because the password is known, the tools don't need to prompt you for the keystore/key password each time you compile.
- When your application is ready for release, you must compile in release mode and then sign the .apk **with your private key**. There are two ways to do this:

Signing quickview

- All Android apps *must* be signed
- You can sign with a self-signed key
- How you sign your apps is critical — read this document carefully
- Determine your signing strategy early in the development process

In this document

[Overview](#)

[Signing Strategies](#)

[Basic Setup for Signing](#)

[Signing in Debug Mode](#)

[Signing for Public Release](#)

[Obtain a suitable private key](#)

[Compile the application in release mode](#)

[Sign your application with your private key](#)

[Align the final APK package](#)

[Compile and sign with Eclipse ADT](#)

[Securing Your Private Key](#)

See also

[Versioning Your Applications](#)

[Preparing to Publish](#)

- Using Keytool and Jarsigner in the command-line. In this approach, you first compile your application to an *unsigned* .apk. You must then sign the .apk manually with your private key using Jarsigner (or similar tool). If you do not have a suitable private key already, you can run Keytool manually to generate your own keystore/key and then sign your application with Jarsigner.
- Using the ADT Export Wizard. If you are developing in Eclipse with the ADT plugin, you can use the Export Wizard to compile the application, generate a private key (if necessary), and sign the .apk, all in a single process using the Export Wizard.

Once your application is signed, don't forget to run `zipalign` on the APK for additional optimization.

Signing Strategies

Some aspects of application signing may affect how you approach the development of your application, especially if you are planning to release multiple applications.

In general, the recommended strategy for all developers is to sign all of your applications with the same certificate, throughout the expected lifespan of your applications. There are several reasons why you should do so:

- Application upgrade – As you release upgrades to your application, you will want to sign the upgrades with the same certificate, if you want users to upgrade seamlessly to the new version. When the system is installing an update to an application, if any of the certificates in the new version match any of the certificates in the old version, then the system allows the update. If you sign the version without using a matching certificate, you will also need to assign a different package name to the application — in this case, the user installs the new version as a completely new application.
- Application modularity – The Android system allows applications that are signed by the same certificate to run in the same process, if the applications so requests, so that the system treats them as a single application. In this way you can deploy your application in modules, and users can update each of the modules independently if needed.
- Code/data sharing through permissions – The Android system provides signature-based permissions enforcement, so that an application can expose functionality to another application that is signed with a specified certificate. By signing multiple applications with the same certificate and using signature-based permissions checks, your applications can share code and data in a secure manner.

Another important consideration in determining your signing strategy is how to set the validity period of the key that you will use to sign your applications.

- If you plan to support upgrades for a single application, you should ensure that your key has a validity period that exceeds the expected lifespan of that application. A validity period of 25 years or more is recommended. When your key's validity period expires, users will no longer be able to seamlessly upgrade to new versions of your application.
- If you will sign multiple distinct applications with the same key, you should ensure that your key's validity period exceeds the expected lifespan of *all versions of all of the applications*, including dependent applications that may be added to the suite in the future.
- If you plan to publish your application(s) on Android Market, the key you use to sign the application(s) must have a validity period ending after 22 October 2033. The Market server enforces this requirement to ensure that users can seamlessly upgrade Market applications when new versions are available.

As you design your application, keep these points in mind and make sure to use a [suitable certificate](#) to sign your applications.

Basic Setup for Signing

Before you begin, you should make sure that Keytool is available to the SDK build tools. In most cases, you can tell the SDK build tools how to find Keytool by setting your `JAVA_HOME` environment variable to reference a suitable JDK. Alternatively, you can add the JDK version of Keytool to your `PATH` variable.

If you are developing on a version of Linux that originally came with GNU Compiler for Java, make sure that the system is using the JDK version of Keytool, rather than the gcj version. If Keytool is already in your `PATH`, it might be pointing to a symlink at `/usr/bin/keytool`. In this case, check the symlink target to be sure it points to the Keytool in the JDK.

If you will release your application to the public, you will also need to have the Jarsigner tool available on your machine. Both Jarsigner and Keytool are included in the JDK.

Signing in Debug Mode

The Android build tools provide a debug signing mode that makes it easier for you to develop and debug your application, while still meeting the Android system requirement for signing your `.apk`. When using debug mode to build your app, the SDK tools invoke Keytool to automatically create a debug keystore and key. This debug key is then used to automatically sign the `.apk`, so you do not need to sign the package with your own key.

The SDK tools create the debug keystore/key with predetermined names/passwords:

- Keystore name: "debug.keystore"
- Keystore password: "android"
- Key alias: "androiddebugkey"
- Key password: "android"
- CN: "CN=Android Debug,O=Android,C=US"

If necessary, you can change the location/name of the debug keystore/key or supply a custom debug keystore/key to use. However, any custom debug keystore/key must use the same keystore/key names and passwords as the default debug key (as described above). (To do so in Eclipse/ADT, go to **Windows > Preferences > Android > Build**.)

Caution: You *cannot* release your application to the public when signed with the debug certificate.

Eclipse Users

If you are developing in Eclipse/ADT (and have set up Keytool as described above in [Basic Setup for Signing](#)), signing in debug mode is enabled by default. When you run or debug your application, ADT signs the `.apk` with the debug certificate, runs `zipalign` on the package, then installs it on the selected emulator or connected device. No specific action on your part is needed, provided ADT has access to Keytool.

Ant Users

If you are using Ant to build your `.apk` files, debug signing mode is enabled by using the `debug` option with the `ant` command (assuming that you are using a `build.xml` file generated by the `android` tool). When you run `ant debug` to compile your app, the build script generates a keystore/key and signs the `.apk` for you. The script then also aligns the `.apk` with the `zipalign` tool. No other action on your part is needed. Read [Developing In Other IDEs: Building in debug mode](#) for more information.

Expiry of the Debug Certificate

The self-signed certificate used to sign your application in debug mode (the default on Eclipse/ADT and Ant builds) will have an expiration date of 365 days from its creation date.

When the certificate expires, you will get a build error. On Ant builds, the error looks like this:

```
debug:
[echo] Packaging bin/samples-debug.apk, and signing it with a debug key...
[exec] Debug Certificate expired on 8/4/08 3:43 PM
```

In Eclipse/ADT, you will see a similar error in the Android console.

To fix this problem, simply delete the `debug.keystore` file. The default storage location for AVDs is in `~/.android/avd` on OS X and Linux, in `C:\Documents and Settings*.android\` on Windows XP, and in `C:\Users*.android\` on Windows Vista.

The next time you build, the build tools will regenerate a new keystore and debug key.

Note that, if your development machine is using a non-Gregorian locale, the build tools may erroneously generate an already-expired debug certificate, so that you get an error when trying to compile your application. For workaround information, see the troubleshooting topic [I can't compile my app because the build tools generated an expired debug certificate](#).

Signing for Public Release

When your application is ready for release to other users, you must:

1. [Obtain a suitable private key](#)
2. [Compile the application in release mode](#)
1. [Sign your application with your private key](#)
4. [Align the final APK package](#)

If you are developing in Eclipse with the ADT plugin, you can use the Export Wizard to perform the compile, sign, and align procedures. The Export Wizard even allows you to generate a new keystore and private key in the process. So if you use Eclipse, you can skip to [Compile and sign with Eclipse ADT](#).

1. Obtain a suitable private key

In preparation for signing your application, you must first ensure that you have a suitable private key with which to sign. A suitable private key is one that:

- Is in your possession
- Represents the personal, corporate, or organizational entity to be identified with the application
- Has a validity period that exceeds the expected lifespan of the application or application suite. A validity period of more than 25 years is recommended.

If you plan to publish your application(s) on Android Market, note that a validity period ending after 22 October 2033 is a requirement. You can not upload an application if it is signed with a key whose validity expires before that date.

- Is not the debug key generated by the Android SDK tools.

The key may be self-signed. If you do not have a suitable key, you must generate one using Keytool. Make sure that you have Keytool available, as described in [Basic Setup](#).

To generate a self-signed key with Keytool, use the `keytool` command and pass any of the options listed below (and any others, as needed).

Warning: Keep your private key secure. Before you run Keytool, make sure to read [Securing Your Private Key](#) for a discussion of how to keep your key secure and why doing so is critically important to you and to users. In particular, when you are generating your key, you should select strong passwords for both the keystore and key.

Keytool Option	Description
<code>-genkey</code>	Generate a key pair (public and private keys)
<code>-v</code>	Enable verbose output.
<code>-keystore <keystore-name>.keystore</code>	A name for the keystore containing the private key.
<code>-storepass <password></code>	A password for the keystore. As a security precaution, do not include this option in your command line unless you are working at a secure computer. If not supplied, Keytool prompts you to enter the password. In this way, your password is not stored in your shell history.
<code>-alias <alias_name></code>	An alias for the key. Only the first 8 characters of the alias are used.
<code>-keyalg <alg></code>	The encryption algorithm to use when generating the key. Both DSA and RSA are supported.
<code>-dname <name></code>	A Distinguished Name that describes who created the key. The value is used as the issuer and subject fields in the self-signed certificate. Note that you do not need to specify this option in the command line. If not supplied, Jarsigner prompts you to enter each of the Distinguished Name fields (CN, OU, and so on).
<code>-validity <valdays></code>	The validity period for the key, in days. Note: A value of 10000 or greater is recommended.
<code>-keypass <password></code>	The password for the key. As a security precaution, do not include this option in your command line unless you are working at a secure computer. If not supplied, Keytool prompts you to enter the password. In this way, your password is not stored in your shell history.

Here's an example of a Keytool command that generates a private key:

```
$ keytool -genkey -v -keystore my-release-key.keystore
  -alias alias_name -keyalg RSA -validity 10000
```

Running the example command above, Keytool prompts you to provide passwords for the keystore and key, and to provide the Distinguished Name fields for your key. It then generates the keystore as a file called `my-release-`

`key.keystore`. The keystore and key are protected by the passwords you entered. The keystore contains a single key, valid for 10000 days. The alias is a name that you — will use later, to refer to this keystore when signing your application.

For more information about Keytool, see the documentation at <http://java.sun.com/j2se/1.5.0/docs/tooldocs/#security>

2. Compile the application in release mode

In order to release your application to users, you must compile it in release mode. In release mode, the compiled application is not signed by default and you will need to sign it with your private key.

Caution: You can not release your application unsigned, or signed with the debug key.

With Eclipse

To export an *unsigned* .apk from Eclipse, right-click the project in the Package Explorer and select **Android Tools > Export Unsigned Application Package**. Then specify the file location for the unsigned .apk. (Alternatively, open your `AndroidManifest.xml` file in Eclipse, open the *Overview* tab, and click **Export an unsigned .apk**.)

Note that you can combine the compiling and signing steps with the Export Wizard. See [Compiling and signing with Eclipse ADT](#).

With Ant

If you are using Ant, you can enable release mode by using the `release` option with the `ant` command. For example, if you are running Ant from the directory containing your `build.xml` file, the command would look like this:

```
ant release
```

By default, the build script compiles the application .apk without signing it. The output file in your project `bin/` will be `<your_project_name>-unsigned.apk`. Because the application .apk is still unsigned, you must manually sign it with your private key and then align it using `zipalign`.

However, the Ant build script can also perform the signing and aligning for you, if you have provided the path to your keystore and the name of your key alias in the project's `build.properties` file. With this information provided, the build script will prompt you for your keystore and alias password when you perform `ant release`, it will sign the package and then align it. The final output file in `bin/` will instead be `<your_project_name>-release.apk`. With these steps automated for you, you're able to skip the manual procedures below (steps 3 and 4). To learn how to specify your keystore and alias in the `build.properties` file, see [Developing In Other IDEs: Building in release mode](#).

3. Sign your application with your private key

When you have an application package that is ready to be signed, you can do sign it using the Jarsigner tool. Make sure that you have Jarsigner available on your machine, as described in [Basic Setup](#). Also, make sure that the keystore containing your private key is available.

To sign your application, you run Jarsigner, referencing both the application's .apk and the keystore containing the private key with which to sign the .apk. The table below shows the options you could use.

Jarsigner Option	Description
<code>-keystore <keystore-name>.keystore</code>	The name of the keystore containing your private key.
<code>-verbose</code>	Enable verbose output.
<code>-storepass <password></code>	The password for the keystore. As a security precaution, do not include this option in your command line unless you are working at a secure computer. If not supplied, Jarsigner prompts you to enter the password. In this way, your password is not stored in your shell history.
<code>-keypass <password></code>	The password for the private key. As a security precaution, do not include this option in your command line unless you are working at a secure computer. If not supplied, Jarsigner prompts you to enter the password. In this way, your password is not stored in your shell history.

Here's how you would use Jarsigner to sign an application package called `my_application.apk`, using the example keystore created above.

```
$ jarsigner -verbose -keystore my-release-key.keystore
my_application.apk alias_name
```

Running the example command above, Jarsigner prompts you to provide passwords for the keystore and key. It then modifies the `.apk` in-place, meaning the `.apk` is now signed. Note that you can sign an `.apk` multiple times with different keys.

To verify that your `.apk` is signed, you can use a command like this:

```
$ jarsigner -verify my_signed.apk
```

If the `.apk` is signed properly, Jarsigner prints "jar verified". If you want more details, you can try one of these commands:

```
$ jarsigner -verify -verbose my_application.apk
```

or

```
$ jarsigner -verify -verbose -certs my_application.apk
```

The command above, with the `-certs` option added, will show you the "CN=" line that describes who created the key.

Note: If you see "CN=Android Debug", this means the `.apk` was signed with the debug key generated by the Android SDK. If you intend to release your application, you must sign it with your private key instead of the debug key.

For more information about Jarsigner, see the documentation at <http://java.sun.com/j2se/1.5.0/docs/tooldocs/#security>

4. Align the final APK package

Once you have signed the .apk with your private key, run `zipalign` on the file. This tool ensures that all uncompressed data starts with a particular byte alignment, relative to the start of the file. Ensuring alignment at 4-byte boundaries provides a performance optimization when installed on a device. When aligned, the Android system is able to read files with `mmap()`, even if they contain binary data with alignment restrictions, rather than copying all of the data from the package. The benefit is a reduction in the amount of RAM consumed by the running application.

The `zipalign` tool is provided with the Android SDK, inside the `tools/` directory. To align your signed .apk, execute:

```
zipalign -v 4 your_project_name-unaligned.apk your_project_name.apk
```

The `-v` flag turns on verbose output (optional). `4` is the byte-alignment (don't use anything other than 4). The first file argument is your signed .apk (the input) and the second file is the destination .apk file (the output). If you're overriding an existing .apk, add the `-f` flag.

Caution: Your input .apk must be signed with your private key **before** you optimize the package with `zipalign`. If you sign it after using `zipalign`, it will undo the alignment.

For more information, read about the [zipalign](#) tool.

Compile and sign with Eclipse ADT

If you are using Eclipse with the ADT plugin, you can use the Export Wizard to export a *signed* .apk (and even create a new keystore, if necessary). The Export Wizard performs all the interaction with the Keytool and Jarsigner for you, which allows you to sign the package using a GUI instead of performing the manual procedures to compile, sign, and align, as discussed above. Once the wizard has compiled and signed your package, it will also perform package alignment with `zipalign`. Because the Export Wizard uses both Keytool and Jarsigner, you should ensure that they are accessible on your computer, as described above in the [Basic Setup for Signing](#).

To create a signed and aligned .apk in Eclipse:

1. Select the project in the Package Explorer and select **File > Export**.
2. Open the Android folder, select Export Android Application, and click **Next**.

The Export Android Application wizard now starts, which will guide you through the process of signing your application, including steps for selecting the private key with which to sign the .apk (or creating a new keystore and private key).

3. Complete the Export Wizard and your application will be compiled, signed, aligned, and ready for distribution.

Securing Your Private Key

Maintaining the security of your private key is of critical importance, both to you and to the user. If you allow someone to use your key, or if you leave your keystore and passwords in an unsecured location such that a third-party could find and use them, your authoring identity and the trust of the user are compromised.

If a third party should manage to take your key without your knowledge or permission, that person could sign and distribute applications that maliciously replace your authentic applications or corrupt them. Such a person could also sign and distribute applications under your identity that attack other applications or the system itself, or corrupt or steal user data.

Your reputation as a developer entity depends on your securing your private key properly, at all times, until the key is expired. Here are some tips for keeping your key secure:

- Select strong passwords for the keystore and key.
- When you generate your key with Keytool, *do not* supply the `-storepass` and `-keypass` options at the command line. If you do so, your passwords will be available in your shell history, which any user on your computer could access.
- Similarly, when signing your applications with Jarsigner, *do not* supply the `-storepass` and `-keypass` options at the command line.
- Do not give or lend anyone your private key, and do not let unauthorized persons know your keystore and key passwords.

In general, if you follow common-sense precautions when generating, using, and storing your key, it will remain secure.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 14 May 2010 15:20

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

[↑ Go to top](#)

Versioning Your Applications

Versioning is a critical component of your application upgrade/maintenance strategy.

- Users need to have specific information about the application version that is installed on their devices and the upgrade versions available for installation.
- Other applications — including other applications that you publish as a suite — need to query the system for your application's version, to determine compatibility and identify dependencies.
- Services through which you will publish your application (s) may also need to query your application for its version, so that they can display the version to users. A publishing service may also need to check the application version to determine compatibility and establish upgrade/downgrade relationships.

Versioning quickview

- Your application *must* be versioned
- You set the version in the application's manifest file
- How you version your applications affects how users upgrade
- Determine your versioning strategy early in the development process, including considerations for future releases.

In this document

[Setting Application Version](#)

[Specifying Your Application's System API Requirements](#)

See also

[Preparing to Publish Your Application](#)

[Publishing On Android Market](#)

[The AndroidManifest.xml File](#)

The Android system itself *does not ever* check the application version information for an application, such as to enforce restrictions on upgrades, compatibility, and so on. Instead, only users or applications themselves are responsible for enforcing any version restrictions for applications themselves.

The Android system *does* check any system version compatibility expressed by an application in its manifest, in the `minSdkVersion` attribute. This allows an application to specify the minimum system API with which is compatible. For more information see [Specifying Minimum System API Version](#).

Setting Application Version

To define the version information for your application, you set attributes in the application's manifest file. Two attributes are available, and you should always define values for both of them:

- `android:versionCode` — An integer value that represents the version of the application code, relative to other versions.

The value is an integer so that other applications can programmatically evaluate it, for example to check an upgrade or downgrade relationship. You can set the value to any integer you want, however you should make sure that each successive release of your application uses a greater value. The system does not enforce this behavior, but increasing the value with successive releases is normative.

Typically, you would release the first version of your application with `versionCode` set to 1, then monotonically increase the value with each release, regardless whether the release constitutes a major or minor release. This means that the `android:versionCode` value does not necessarily have a strong resemblance to the application release version that is visible to the user (see `android:versionName`, below). Applications and publishing services should not display this version value to users.

- `android:versionName` — A string value that represents the release version of the application code, as it should be shown to users.

The value is a string so that you can describe the application version as a <major>.<minor>.<point> string, or as any other type of absolute or relative version identifier.

As with `android:versionCode`, the system does not use this value for any internal purpose, other than to enable applications to display it to users. Publishing services may also extract the `android:versionName` value for display to users.

You define both of these version attributes in the `<manifest>` element of the manifest file.

Here's an example manifest that shows the `android:versionCode` and `android:versionName` attributes in the `<manifest>` element.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.package.name"
    android:versionCode="2"
    android:versionName="1.1">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        ...
    </application>
</manifest>
```

In this example, note that `android:versionCode` value indicates that the current .apk contains the second release of the application code, which corresponds to a minor follow-on release, as shown by the `android:versionName` string.

The Android framework provides an API to let applications query the system for version information about your application. To obtain version information, applications use the [getPackageInfo\(java.lang.String, int\)](#) method of [PackageManager](#).

Specifying Your Application's System API Requirements

If your application requires a specific minimum version of the Android platform, or is designed only to support a certain range of Android platform versions, you can specify those version requirements as API Level identifiers in the application's manifest file. Doing so ensures that your application can only be installed on devices that are running a compatible version of the Android system.

To specify API Level requirements, add a `<uses-sdk>` element in the application's manifest, with one or more of these attributes:

- `android:minSdkVersion` — The minimum version of the Android platform on which the application will run, specified by the platform's API Level identifier.
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — The maximum version of the Android platform on which the application is designed to run, specified by the platform's API Level identifier. **Important:** Please read the [<uses-sdk>](#) documentation before using this attribute.

When preparing to install your application, the system checks the value of this attribute and compares it to the system version. If the `android:minSdkVersion` value is greater than the system version, the system aborts the installation of the application. Similarly, the system installs your application only if its `android:maxSdkVersion` is compatible with the platform version.

If you do not specify these attributes in your manifest, the system assumes that your application is compatible with all platform versions, with no maximum API Level.

To specify a minimum platform version for your application, add a `<uses-sdk>` element as a child of `<manifest>`, then define the `android:minSdkVersion` as an attribute.

For more information, see the [<uses-sdk>](#) manifest element documentation and the [API Levels](#) document.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

[↑ Go to top](#)

Android 2.2 r1 - 14 May 2010 15:20

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)



Preparing to Publish: A Checklist

Publishing an application means testing it, packaging it appropriately, and making it available to users of Android-powered mobile devices.

If you plan to publish your application for installation on Android-powered devices, there are several things you need to do, to get your application ready. This document highlights the significant checkpoints for preparing your application for a successful release.

If you will publish your application on Android Market, please also see [Publishing on Android Market](#) for specific preparation requirements for your application.

For general information about the ways that you can publish an applications, see the [Publishing Your Applications](#) document.

Before you consider your application ready for release:

1. Test your application extensively on an actual device
2. Consider adding an End User License Agreement in your application
3. Specify an icon and label in the application's manifest
4. Turn off logging and debugging and clean up data/files

Before you do the final compile of your application:

5. Version your application
6. Obtain a suitable cryptographic key
7. Register for a Maps API Key, if your application is using MapView elements

Compile your application...

After compiling your application:

8. Sign your application
9. Test your compiled application

Before you consider your application ready for release

1. Test your application extensively on an actual device

It's important to test your application as extensively as possible, in as many areas as possible. To help you do that, Android provides a variety of testing classes and tools. You can use [Instrumentation](#) to run JUnit and other test cases, and you can use testing tools such as the [UI/Application Exerciser Monkey](#).

- To ensure that your application will run properly for users, you should make every effort to obtain one or more physical mobile device(s) of the type on which you expect the application to run. You should then test your application on the actual device, under realistic network conditions. Testing your application on a physical device is very important, because it enables you to verify that your user interface elements are sized correctly (especially for touch-screen UI) and that your application's performance and battery efficiency are acceptable.
- If you can not obtain a mobile device of the type you are targeting for your application, you can use emulator options such as `-dpi`, `-device`, `-scale`, `-netspeed`, `-netdelay`, `-cpu-delay` and others to model the emulator's

screen, network performance, and other attributes to match the target device to the greatest extent possible. You can then test your application's UI and performance. However, we strongly recommend that you test your application on an actual target device before publishing it.

- If you are targeting the [T-Mobile G1](#) device for your application, make sure that your UI handles screen orientation changes.

2. Consider adding an End User License Agreement in your application

To protect your person, organization, and intellectual property, you may want to provide an End User License Agreement (EULA) with your application.

3. Specify an icon and label in the application's manifest

The icon and label that you specify in an application's manifest are important because they are displayed to users as your application's icon and name. They are displayed on the device's Home screen, as well as in Manage Applications, My Downloads, and elsewhere. Additionally, publishing services may display the icon and label to users.

To specify an icon and label, you define the attributes `android:icon` and `android:label` in the `<application>` element of the manifest.

As regards the design of your icon, you should try to make it match as much as possible the style used by the built-in Android applications.

4. Turn off logging and debugging and clean up data/files

For release, you should make sure that debug facilities are turned off and that debug and other unnecessary data/files are removed from your application project.

- Remove the `android:debuggable="true"` attribute from the `<application>` element of the manifest.
- Remove log files, backup files, and other unnecessary files from the application project.
- Check for private or proprietary data and remove it as necessary.
- Deactivate any calls to [Log](#) methods in the source code.

Before you do the final compile of your application

5. Version your application

Before you compile your application, you must make sure that you have defined a version number for your application, specifying an appropriate value for both the `android:versionCode` and `android:versionName` attributes of the `<manifest>` element in the application's manifest file. Carefully consider your version numbering plans in the context of your overall application upgrade strategy.

If you have previously released a version of your application, you must make sure to increment the version number of the current application. You must increment both the `android:versionCode` and `android:versionName` attributes of the `<manifest>` element in the application's manifest file, using appropriate values.

For detailed information about how to define version information for your application, see [Versioning Your Applications](#).

6. Obtain a suitable cryptographic key

If you have read and followed all of the preparation steps up to this point, your application is compiled and ready for signing. Inside the .apk, the application is properly versioned, and you've cleaned out extra files and private data, as described above.

Before you sign your application, you need to make sure that you have a suitable private key. For complete information about how to obtain (or generate) a private key, see [Obtaining a Suitable Private Key](#).

Once you have obtained (or generated) a suitable private key, you will use it to:

- Register for a Maps API Key (see below), if your application uses MapView elements.
- Sign your application for release, later in the preparation process

7. Register for a Maps API Key, if your application is using MapView elements

If your application uses one or more Mapview elements, you will need to register your application with the Google Maps service and obtain a Maps API Key, before your MapView(s) will be able to retrieve data from Google Maps. To do so, you supply an MD5 fingerprint of your signer certificate to the Maps service.

For complete information about getting a Maps API Key, see [Obtaining a Maps API Key](#).

During development, you can get a temporary Maps API Key by registering the debug key generated by the SDK tools. However, before publishing your application, you must register for a new Maps API Key that is based on your private key.

If your application uses MapView elements, the important points to understand are:

1. You *must* obtain the Maps API Key before you compile your application for release, because you must add the Key to a special attribute in each MapView element — `android:apiKey` — in your application's layout files. If you are instantiating MapView objects directly from code, you must pass the Maps API Key as a parameter in the constructor.
2. The Maps API Key referenced by your application's MapView elements must be registered (in Google Maps) to the certificate used to sign the application. This is particularly important when publishing your application — your MapView elements must reference a Key that is registered to the release certificate that you will use to sign your application.
3. If you previously got a temporary Maps API Key by registering the debug certificate generated by the SDK tools, you *must* remember to obtain a new Maps API Key by registering your release certificate. You must then remember to change the MapView elements to reference the new Key, rather than the Key associated with the debug certificate. If you do not do so, your MapView elements will not have permission to download Maps data.
4. If you change the private key that you will use to sign your application, you *must* remember to obtain a new Maps API Key from the Google Maps service. If you do not get a new Maps API Key and apply it to all MapView elements, any MapView elements referencing the old Key will not have permission to download Maps data.

Compile your application

When you've prepared your application as described in the previous sections, you can compile your application for release.

After compiling your application

8. Sign your application

Sign your application using your private key and then align it with the `zipalign` tool. Signing your application correctly is critically important. Please see [Signing Your Applications](#) for complete information.

9. Test your compiled and signed application

Before you release your compiled application, you should thoroughly test it on the target mobile device (and target network, if possible). In particular, you should make sure that any MapView elements in your UI are receiving maps data properly. If they are not, go back to [Register for a Maps API Key](#) and correct the problem. You should also ensure that the application works correctly with any server-side services and data that you are providing or are relying on and that the application handles any authentication requirements correctly.

After testing, you are now ready to publish your application to mobile device users.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 14 May 2010 15:20

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

[↑ Go to top](#)

Publishing Your Applications

Publishing an application means testing it, packaging it appropriately, and making it available to users of Android-powered mobile devices for download or sideload.

If you've followed the steps outlined in [Preparing to Publish Your Applications](#), the result of the process is a compiled .apk that is signed with your release private key. Inside the .apk, the application is properly versioned and any MapView elements reference a Maps API Key that you obtained by registering the MD5 fingerprint of the same certificate used to sign the .apk. Your application is now ready for publishing.

The sections below provide information about publishing your Android application to mobile device users.

Publishing on Android Market

Android Market is a hosted service that makes it easy for users to find and download Android applications to their Android-powered devices, and makes it easy for developers to publish their applications to Android users.

To publish your application on Android Market, you first need to register with the service using your Google account and agree to the terms of service. Once you are registered, you can upload your application to the service whenever you want, as many times as you want, and then publish it when you are ready. Once published, users can see your application, download it, and rate it using the Market application installed on their Android-powered devices.

To register as an Android Market developer and get started with publishing, visit the Android Market:

<http://market.android.com/publish>

If you plan to publish your application on Android Market, you must make sure that it meets the requirements listed below, which are enforced by the Market server when you upload the application.

Requirements enforced by the Android Market server:

1. Your application must be signed with a cryptographic private key whose validity period ends after **22 October 2033**.
2. Your application must define both an `android:versionCode` and an `android:versionName` attribute in the `<manifest>` element of its manifest. The server uses the `android:versionCode` as the basis for identifying the application internally and handling updates, and it displays the `android:versionName` to users as the application's version.
3. Your application must define both an `android:icon` and an `android:label` attribute in the `<application>` element of its manifest.

Publishing quickview

- You can publish your application using a hosted service such as Android Market or through a web server.
- Before you publish, make sure you have prepared your application properly.
- Android Market makes it easy for users of Android-powered devices to see and download your application.

In this document

[Publishing on Android Market](#)

[Publishing Updates on Android Market](#)

[Using Intents to Launch the Market Application](#)

See also

[Preparing to Publish](#)



Interested in publishing your app on Android Market?

[Go to Android Market »](#)

Publishing Updates on Android Market

At any time after publishing an application on Android Market, you can upload and publish an update to the same application package. When you publish an update to an application, users who have already installed the application will automatically receive a notification that an update is available for the application. They can then choose to update the application to the latest version.

Before uploading the updated application, be sure that you have incremented the `android:versionCode` and `android:versionName` attributes in the `<manifest>` element of the manifest file. Also, the package name must be the same and the .apk must be signed with the same private key. If the package name and signing certificate do *not* match those of the existing version, Market will consider it a new application and will not offer it to users as an update.

Using Intents to Launch the Market Application on a Device

Android-powered devices include a preinstalled Market application that gives users access to the Android Market site. From Market, users can browse or search available applications, read ratings and reviews, and download/install applications.

You can launch the Market application from another Android application by sending an Intent to the system. You might want to do this, for example, to help the user locate and download an update to an installed application, or to let the user know about related applications that are available for download.

To launch Market, you send an `ACTION_VIEW` Intent, passing a Market-handled URI string as the Intent data. In most cases, your application would call `startActivity()` to send the `ACTION_VIEW` Intent with the Market-handled URI.

The URI that you supply with the Intent lets the system route the intent properly and also expresses the type of action that you want Market to perform after launch. Currently, you can have Market take these actions:

- Initiate a search for applications on Android Market, based on the query parameters that you provide, or
- Load the Details page for a specific application on Android Market, based on the application's package name.

Initiating a search

Your application can initiate a search on Android Market for applications that match the query parameters that you provide. To do so, your application sends an `ACTION_VIEW` Intent that includes a URI and query parameters in this format:

```
market://search?q=<paramtype>:<value>
```

Using this URI format, you can search for applications by:

- Package name
- Developer name
- String match across application name, developer name, and description, or
- Any combination of the above

The table at the bottom of this page specifies the `paramtypes` and `values` that correspond to each of these types of search.

When you send an intent to initiate a search for applications, Market sends the search query to the server and displays the result. To the user, the experience is something like this:

1. The user presses a link or button in your application.



Searches on Android Market

When you initiate a search, Android Market returns results from matches in the public metadata supplied by developers in their Android Market profiles or application publishing information, but not from the developer's private account or from the certificate used to sign the application.

2. The Market application launches and takes control of the screen, displaying a progress indicator labeled "Searching" until it receives the search results.
3. Market receives the search results and displays them. Depending on the query parameters, the search results may include a list of one or more applications.
4. From the results page, the user can select an app to go to its Details page, which offers information about the app and lets the user download/purchase the app.

Loading an application's Details page

In Android Market, every application has a Details page that provides an overview of the application for users. For example, the page includes a short description of the app and screen shots of it in use, if supplied by the developer, as well as feedback from users and information about the developer. The Details page also includes an "Install" button that lets the user trigger the download/purchase of the application.

If you want to refer the user to a specific application, your application can take the user directly to the application's Details page. To do so, your application sends an `ACTION_VIEW` Intent that includes a URI and query parameter in this format:

```
market://details?id=<packagename>
```

In this case, the `packagename` parameter is target application's fully qualified package name, as declared in the `package` attribute of the `manifest` element in the application's manifest file. For example:

```
market://details?id=com.example.android.jetboy
```

Android Market URIs

The table below provides a list of URIs and actions currently supported by the Market application.

Note that these URIs work only when passed as Intent data — you can't currently load the URIs in a web browser, either on a desktop machine or on the device.

For this Result	Pass this URI with the ACTION_VIEW Intent	Comments
Display the Details screen for a specific application, as identified by the app's fully qualified package name.	<code>http://market.android.com/details?id=<packagename></code> or <code>market://details?id=<packagename></code>	Note that the package name that you specify is <i>not</i> specific to any version of an application. Therefore, Market always displays the Details page for the latest version of the application.
Search for an application by its fully qualified Java package name and display the result.	<code>http://market.android.com/search?q=pname:<package></code> or <code>market://search?q=pname:<package></code>	Searches only the Java package name of applications. Returns only exact matches.
Search for applications by developer name and display the results.	<code>http://market.android.com/search?q=pub:<Developer Name></code> or <code>market://search?q=pub:<Developer Name></code>	Searches only the "Developer Name" fields of Market public profiles. Returns exact matches only.
Search for applications by substring and display the results.	<code>http://market.android.com/search?q=<substring></code> or <code>market://search?q=<substring></code>	Searches all public fields (application title, developer name, and application description) for all applications. Returns exact and partial matches.
Search using multiple query parameters and display the results.	Example: <code>http://market.android.com/search?q=world pname=com.android.hello pub:Android</code>	Returns a list of applications meeting all the supplied parameters.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 14 May 2010 15:20

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

[↑ Go to top](#)