

Save the date! [Android Dev Summit \(/dev-summit\)](/dev-summit) is coming to Mountain View, CA on November 7-8, 2018.

Sign your app

Android requires that all APKs be digitally signed with a certificate before they can be installed. This document describes how to sign your APKs using Android Studio, including creating and storing your certificate, signing different build configurations using different certificates, and configuring the build process to sign your APKs automatically.

Certificates and keystores

A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.

When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author. The key used to create this certificate is called the *app signing key*.

A keystore is a binary file that contains one or more private keys.

Every app must use the same certificate throughout its lifespan in order for users to be able to install new versions as updates to the app. For more about the benefits of using the same certificate for all your apps throughout their lifespans, see [Signing Considerations](#) (#considerations) below.

Sign your debug build

When running or debugging your project from the IDE, Android Studio automatically signs your APK with a debug certificate generated by the Android SDK tools. The first time you run or

debug your project in Android Studio, the IDE automatically creates the debug keystore and certificate in `$HOME/.android/debug.keystore`, and sets the keystore and key passwords.

Because the debug certificate is created by the build tools and is insecure by design, most app stores (including the Google Play Store) will not accept an APK signed with a debug certificate for publishing.

Android Studio automatically stores your debug signing information in a signing configuration so you do not have to enter it every time you debug. A signing configuration is an object consisting of all of the necessary information to sign an APK, including the keystore location, keystore password, key name, and key password. You cannot directly edit the debug signing configuration, but you can configure how you sign your release build (`#release-mode`).

For more information about how to build and run apps for debugging, see Build and Run Your App (<https://developer.android.com/tools/building/building-studio.html>).

Expiry of the debug certificate

The self-signed certificate used to sign your APK for debugging has an expiration date of 365 days from its creation date. When the certificate expires, you will get a build error.

To fix this problem, simply delete the `debug.keystore` file. The file is stored in the following locations:

- `~/.android/` on OS X and Linux
- `C:\Documents and Settings\<user>\.android\` on Windows XP
- `C:\Users\<user>\.android\` on Windows Vista and Windows 7, 8, and 10

The next time you build and run the debug build type, the build tools will regenerate a new keystore and debug key. Note that you must run your app, building alone does not regenerate the keystore and debug key.

Manage your key

Because your app signing key is used to verify your identity as a developer and to ensure seamless and secure updates for your users, managing your key and keeping it secure are very important, both for you and for your users. You can choose either to opt in to use Google Play

App Signing to securely manage and store your app signing key using Google's infrastructure or to manage and secure your own keystore and app signing key.

By opting in to Google Play App Signing, you will gain the following benefits:

- Ensure that the app signing key is not lost. Loss of the app signing key means that an app cannot be updated, so it is critical for it not to be lost.
- Ensure that the app signing key is not compromised. Compromise of the key would allow a malicious attacker to deploy a malicious version of your app as an update over an existing install. With Play App Signing, developers only manage an upload key which can be reset in the case of loss and compromise. In the event of compromise, an attacker also needs access to the developer account to be able to do anything malicious.

Use Google Play App Signing

When using Google Play App Signing, you will use two keys: the *app signing key* and the *upload key*. Google manages and protects the app signing key for you, and you keep the upload key and use it to sign your apps for upload to the Google Play Store.

When you opt in to use Google Play App Signing, you export and encrypt your app signing key using the Play Encrypt Private Key tool provided by Google Play, and then upload it to Google's infrastructure. Then you create a separate upload key and register it with Google. When you are ready to publish, you sign your app using the upload key and upload it to Google Play. Google then uses the upload certificate to verify your identity, and re-signs your APK with your app signing key for distribution as shown in figure 1. (If you do not already have an app signing key, you can generate one during the sign-up process.)

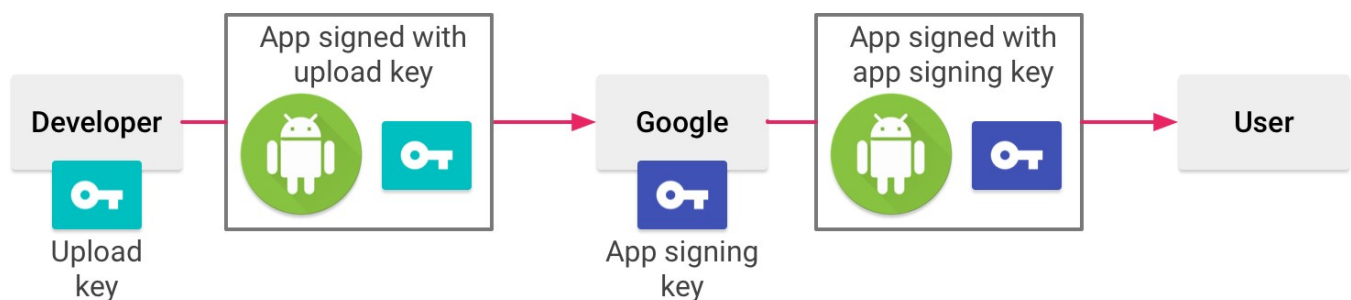


Figure 1. Signing an app with Google Play App Signing

When you use Google Play App Signing, if you lose your upload key, or if it is compromised, you can contact Google to revoke your old upload key and generate a new one. Because your app

signing key is secured by Google, you can continue to upload new versions of your app as updates to the original app, even if you change upload keys.

For more information about how to opt in to use Google Play App Signing, see [Manage your app signing keys](https://support.google.com/googleplay/android-developer/answer/7384423) (https://support.google.com/googleplay/android-developer/answer/7384423).

Manage your own key and keystore

Instead of using Google Play App Signing, you can choose to manage your own app signing key and keystore. If you choose to manage your own app signing key and keystore, you are responsible for securing the key and the keystore. You should choose a strong password for your keystore, and a separate strong password for each private key stored in the keystore. You must keep your keystore in a safe and secure place. If you lose access to your app signing key or your key is compromised, Google cannot retrieve the app signing key for you, and you will not be able to release new versions of your app to users as updates to the original app. For more information, see [Secure your key](#) (#secure-key), below.

If you manage your own app signing key and keystore, when you sign your APK, you will sign it locally using your app signing key and upload the signed APK directly to the Google Play Store for distribution as shown in figure 2.

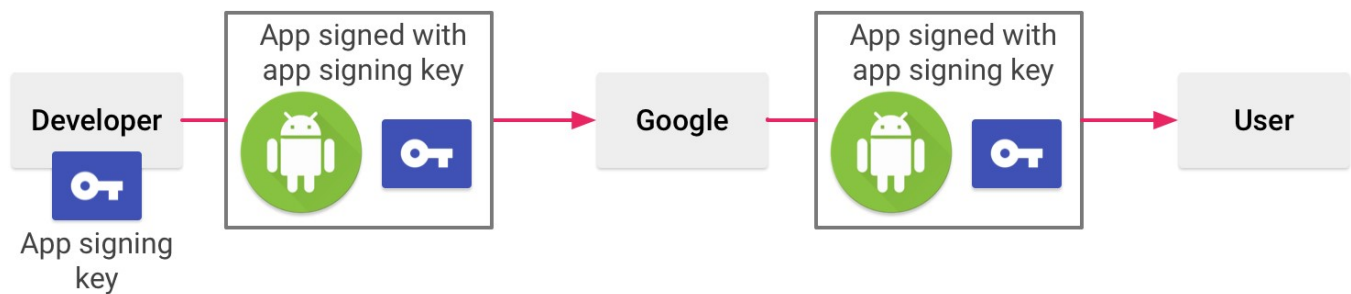


Figure 2. Signing an app when you manage your own app signing key

Sign an APK

Regardless of how you choose to manage your key and keystore, you can use Android Studio to sign your APKs (with either the upload key or the app signing key), either manually, or by configuring your build process to automatically sign APKs.

If you choose to manage and secure your own app signing key and keystore, you will sign your APKs with your app signing key. If you choose to use Google Play App Signing to manage and secure your app signing key and keystore, you will sign your APKs with your upload key.

Generate a key and keystore

You can generate an app signing or upload key using Android Studio, using the following steps:

1. In the menu bar, click **Build** > **Generate Signed APK**.
2. Select a module from the drop down, and click **Next**.
3. Click **Create new** to create a new key and keystore.
4. On the **New Key Store** window, provide the following information for your keystore and key, as shown in figure 3.



Figure 3. Create a new keystore in Android Studio.

Keystore

- **Key store path:** Select the location where your keystore should be created.
- **Password:** Create and confirm a secure password for your keystore.

Key

- **Alias:** Enter an identifying name for your key.
- **Password:** Create and confirm a secure password for your key. This should be different from the password you chose for your keystore

- **Validity (years):** Set the length of time in years that your key will be valid. Your key should be valid for at least 25 years, so you can sign app updates with the same key through the lifespan of your app.
- **Certificate:** Enter some information about yourself for your certificate. This information is not displayed in your app, but is included in your certificate as part of the APK.

Once you complete the form, click **OK**.

5. Continue on to Manually sign an APK (#release-mode) if you would like to generate an APK signed with your new key, or click **Cancel** if you only want to generate a key and keystore, not sign an APK.
6. If you would like to opt in to use Google Play App Signing, proceed to Manage your app signing keys (<https://support.google.com/googleplay/android-developer/answer/7384423>) and follow the instructions to set up Google Play App Signing.

Manually sign an APK

You can use Android Studio to manually generate signed APKs, either one at a time, or for multiple build variants at once. Instead of manually signing APKs, you can also configure your Gradle build settings to handle signing automatically during the build process. This section describes the manual signing process. For more about signing apps as part of the build process, see Configure the build process to automatically sign your APK (#sign-auto).

To manually sign your APK for release in Android Studio, follow these steps:

1. Click **Build > Generate Signed APK** to open the **Generate Signed APK** window. (If you just generated a key and keystore (#generate-key) as described above, this window will already be open.)
2. On the **Generate Signed APK Wizard** window, select a keystore, a private key, and enter the passwords for both. (If you just created your keystore in the previous section, these fields are already populated for you.) Then click **Next**.



Note: If you are using Google Play App Signing, you should specify your upload key here. If you are managing your own app signing key and keystore instead, you should specify your app signing key. For more information, see Manage your key (#manage-key) above.

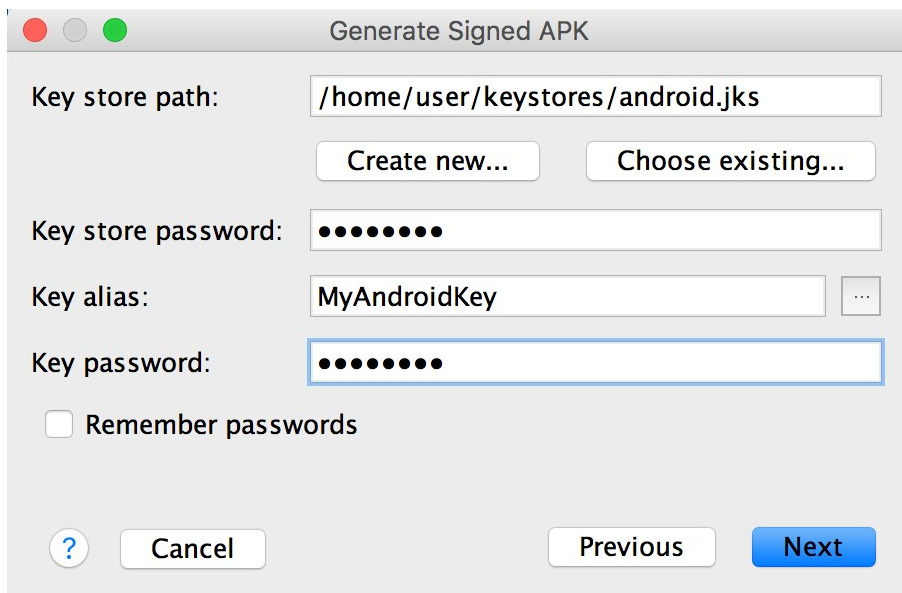


Figure 4. Select a private key in Android Studio.

3. On the next window, select a destination for the signed APK(s), select the build type, (if applicable) choose the product flavor(s), and click **Finish**.

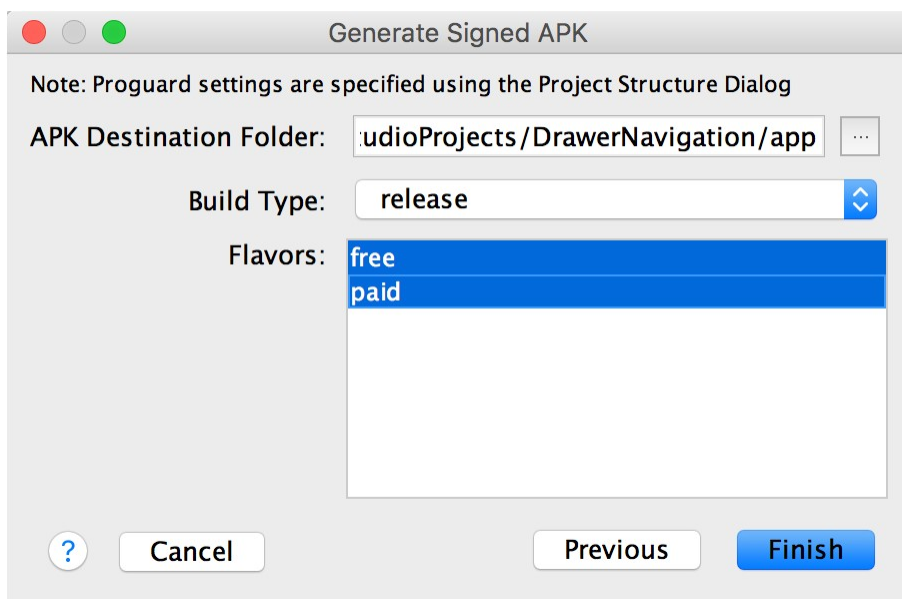


Figure 5. Generate signed APKs for the selected product flavors.


★ **Note:** If your project uses product flavors, you can select multiple product flavors while holding down the **Control** key on Windows/Linux, or the **Command** key on Mac OSX. Android Studio will generate a separate APK for each selected product flavor.

When the process completes, you will find your signed APK in the destination folder you selected above. You may now distribute your signed APK through an app marketplace like the Google Play Store, or using the mechanism of your choice. For more about how to publish your signed APK to the Google Play Store, see [Get Started with Publishing](https://developer.android.com/distribute/googleplay/start.html) (<https://developer.android.com/distribute/googleplay/start.html>). To learn more about other distribution options, read [Alternative Distribution Options](https://developer.android.com/distribute/tools/open-distribution.html) (<https://developer.android.com/distribute/tools/open-distribution.html>).

In order for users to successfully install updates to your app, you will need to sign your APKs with the same certificate throughout the lifespan of your app. For more about this and other benefits of signing all your apps with the same key, see [Signing Considerations](#) (#considerations) below. For more information about securing your private key and keystore, see [Secure your key](#) (#secure-key), below.

Configure the build process to automatically sign your APK

In Android Studio, you can configure your project to sign your release APK automatically during the build process by creating a signing configuration and assigning it to your release build type. A signing configuration consists of a keystore location, keystore password, key alias, and key password. To create a signing configuration and assign it to your release build type using Android Studio, use the following steps:

1. In the **Project** window, right click on your app and click **Open Module Settings**.
2. On the **Project Structure** window, under **Modules** in the left panel, click the module you would like to sign.
3. Click the **Signing** tab, then click **Add** .
4. Select your keystore file, enter a name for this signing configuration (as you may create more than one), and enter the required information.

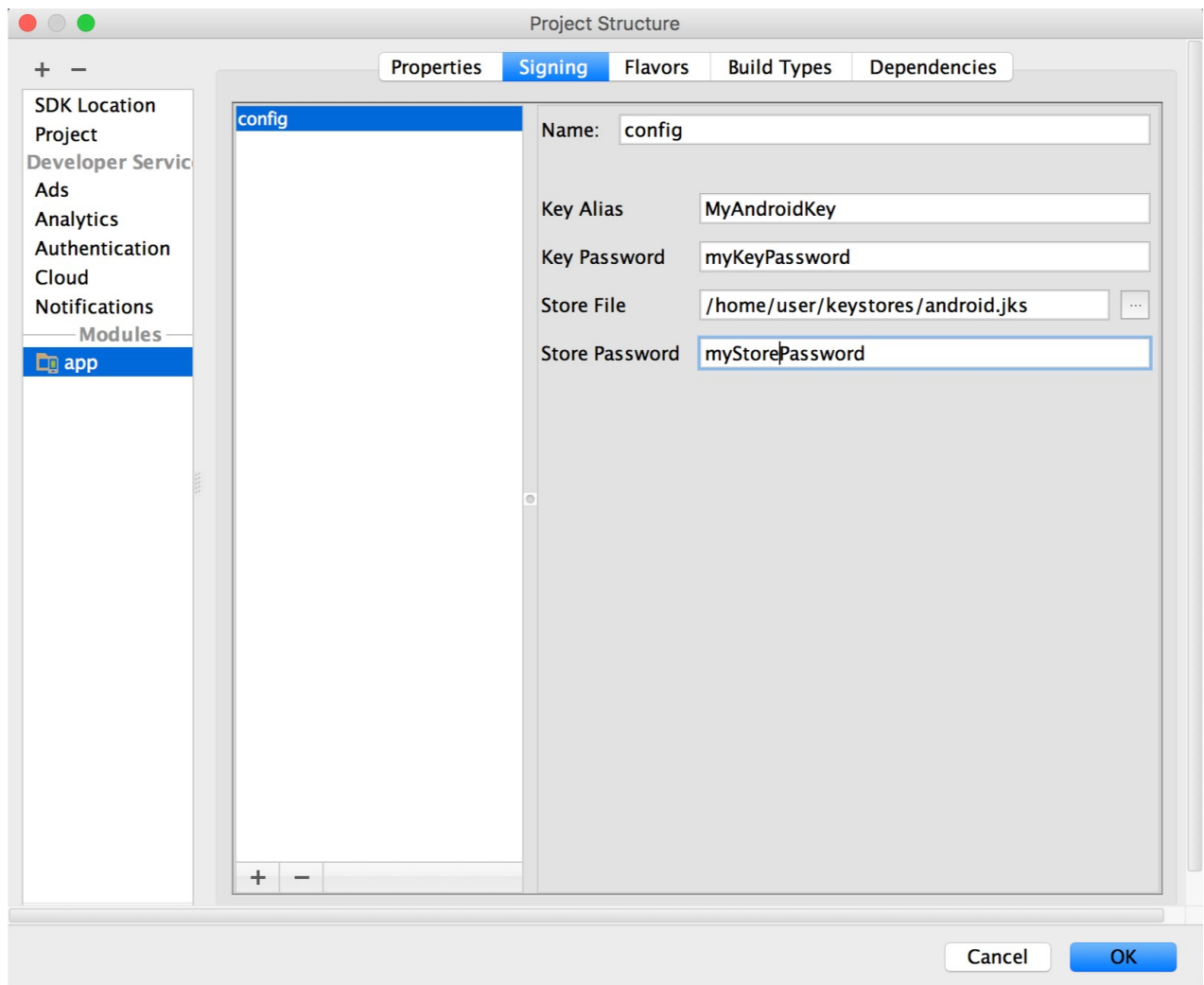


Figure 6. The window for creating a new signing configuration.

5. Click the **Build Types** tab.
6. Click the **release** build.
7. Under **Signing Config**, select the signing configuration you just created.

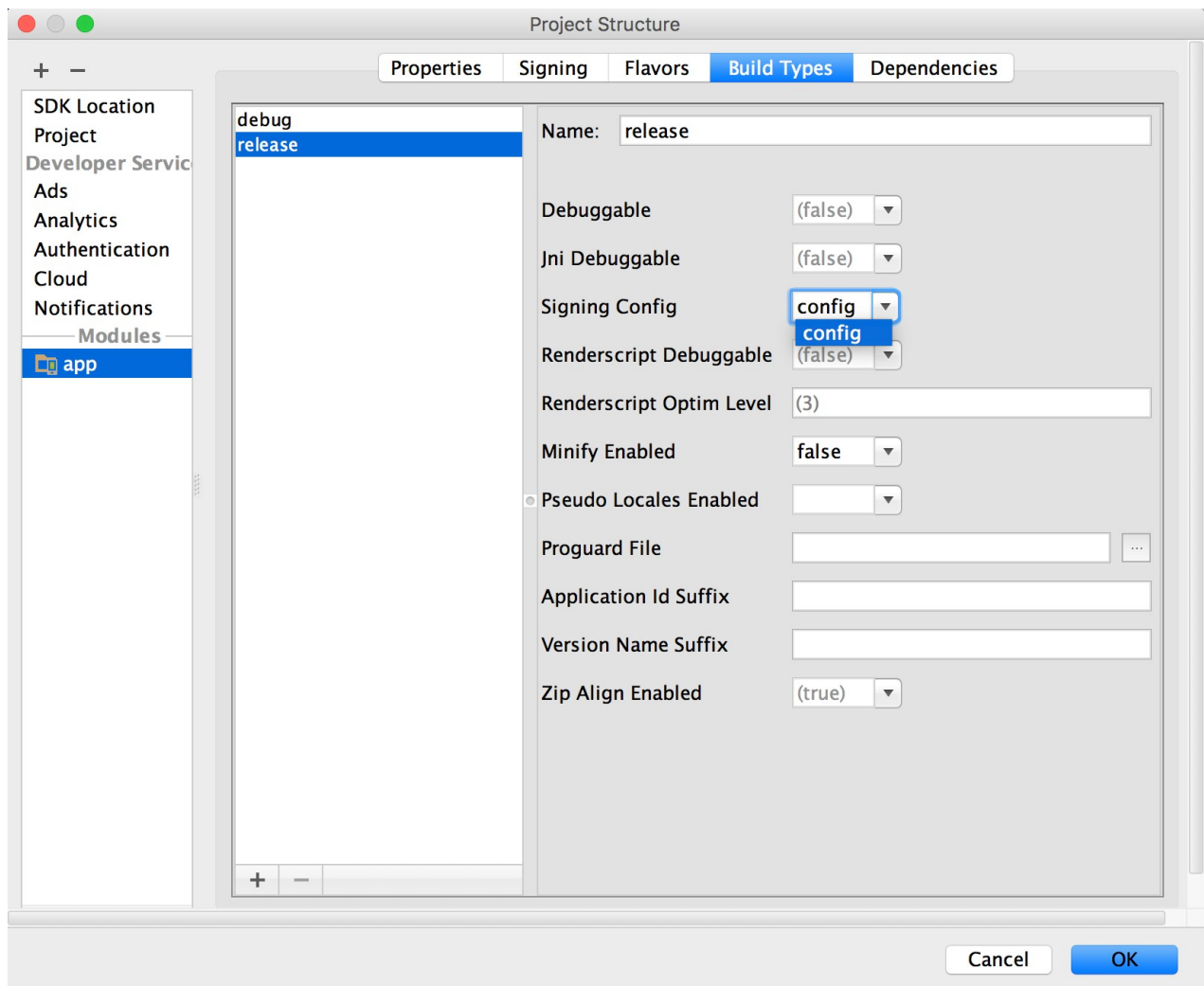


Figure 7. Select a signing configuration in Android Studio.


8. Click **OK**.

Now every time you build your release build type using Android Studio, the IDE will sign the APK automatically, using the signing configuration you specified. You can find your signed APKs in the `build/outputs/apk/` folder inside the project directory for the module you are building.

When you create a signing configuration, your signing information is included in plain text in your Gradle build files. If you are working in a team or sharing your code publicly, you should keep your signing information secure by removing it from the build files and storing it separately. You can read more about how to remove your signing information from your build files in [Remove Signing Information from Your Build Files](#) (#secure-shared-keystore). For more about keeping your signing information secure, read [Secure your key](#) (#secure-key).

Sign each product flavor differently

If your app uses product flavors and you would like to sign each flavor differently, you can create additional signing configurations and assign them by flavor:

1. In the **Project** window, right click on your app and click **Open Module Settings**.
2. On the **Project Structure** window, under **Modules** in the left panel, click the module you would like to sign.
3. Click the **Signing** tab, then click **Add** .
4. Select your keystore file, enter a name for this signing configuration (as you may create more than one), and enter the required information.

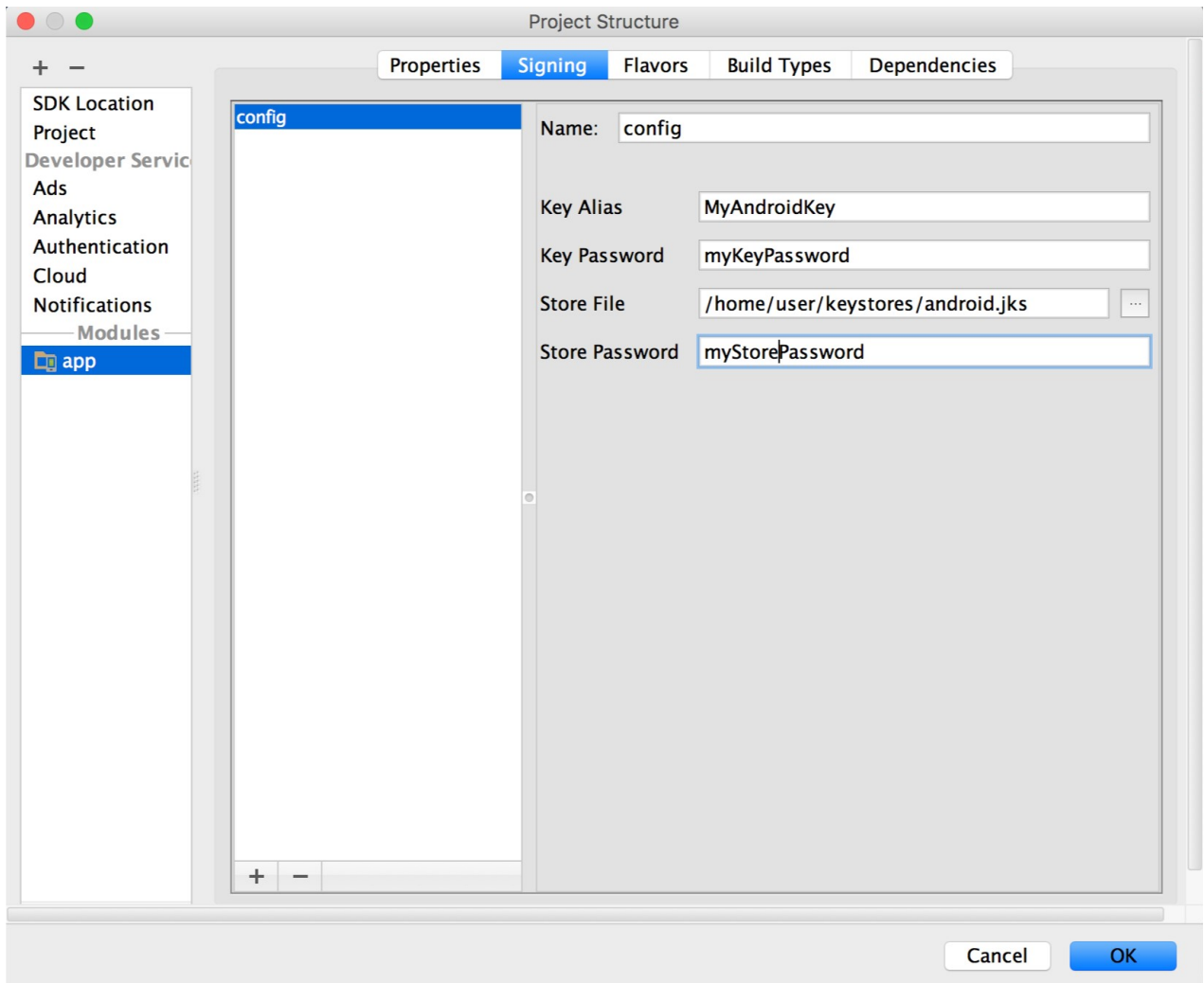


Figure 8. The window for creating a new signing configuration.

5. Repeat steps 3 and 4 as necessary until you have created all your signing configurations.

6. Click the **Flavors** tab.
7. Click the flavor you would like to configure, then select the appropriate signing configuration from the **Signing Config** dropdown menu.

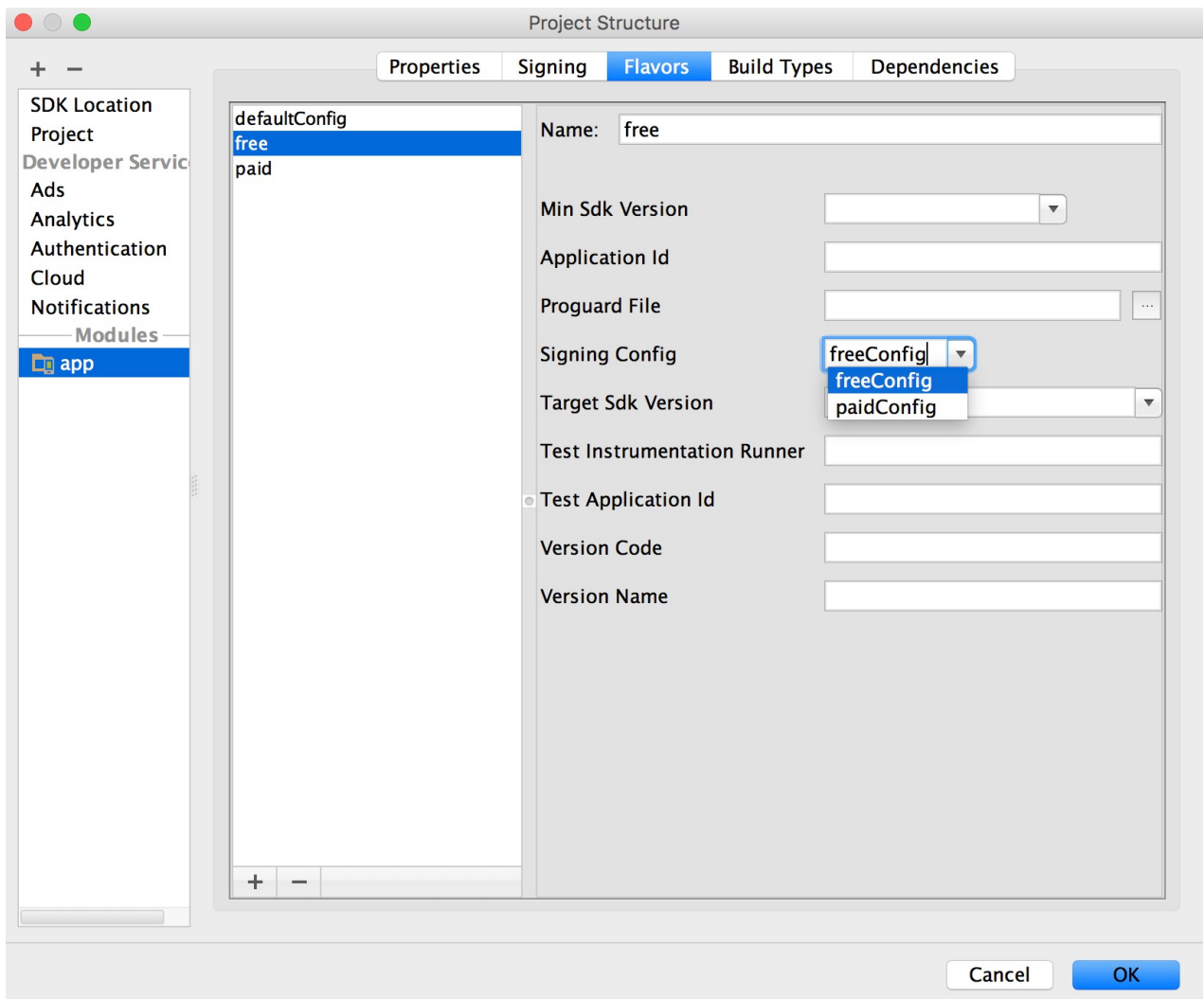


Figure 9. Configure signing settings by product flavor.

Repeat to configure any additional product flavors.

8. Click **OK**.

You can also specify your signing settings in Gradle configuration files. For more information, see [Configuring Signing Settings](#)

(<https://developer.android.com/studio/build/build-variants.html#signing>).

Sign Wear OS apps

If you are building an Wear OS app, the process for signing the app can differ from the process described on this page. See the information about [packaging and publishing Wear OS apps](https://developer.android.com/training/wearables/apps/packaging.html) (<https://developer.android.com/training/wearables/apps/packaging.html>).

Signing considerations

You should sign all of your APKs with the same certificate throughout the expected lifespan of your apps. There are several reasons why you should do so:

- **App upgrade:** When the system is installing an update to an app, it compares the certificate(s) in the new version with those in the existing version. The system allows the update if the certificates match. If you sign the new version with a different certificate, you must assign a different package name to the app—in this case, the user installs the new version as a completely new app.
- **App modularity:** Android allows APKs signed by the same certificate to run in the same process, if the apps so request, so that the system treats them as a single app. In this way you can deploy your app in modules, and users can update each of the modules independently.
- **Code/data sharing through permissions:** Android provides signature-based permissions enforcement, so that an app can expose functionality to another app that is signed with a specified certificate. By signing multiple APKs with the same certificate and using signature-based permissions checks, your apps can share code and data in a secure manner.

If you plan to support upgrades for an app, ensure that your app signing key has a validity period that exceeds the expected lifespan of that app. 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 app.

If you plan to publish your apps on Google Play, the key you use to sign those APKs must have a validity period ending after 22 October 2033. Google Play enforces this requirement to ensure that users can seamlessly upgrade apps when new versions are available. If you use [Google Play App Signing](#) ([#google-play-app-signing](#)), Google ensures your apps are correctly signed and able to receive updates throughout their lifespans.

Secure your key

If you choose to manage and secure your app signing key and keystore yourself (instead of opting in to [use Google Play App Signing](#) (`#google-play-app-signing`)), securing your app signing 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.

Note: If you use Google Play App Signing, your app signing key is kept secure using Google's infrastructure. You should still keep your upload key secure as described below. If your upload key is compromised, you can contact Google to revoke it and receive a new upload key.

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

Your private key is required for signing all future versions of your app. If you lose or misplace your key, you will not be able to publish updates to your existing app. You cannot regenerate a previously generated key.

Your reputation as a developer entity depends on your securing your app signing 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.
- Do not give or lend anyone your private key, and do not let unauthorized persons know your keystore and key passwords.
- Keep the keystore file containing your private key in a safe, secure place.

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

Remove signing information from your build files

When you create a signing configuration, Android Studio adds your signing information in plain text to the module's `build.gradle` files. If you are working with a team or open-sourcing your code, you should move this sensitive information out of the build files so it is not easily

accessible to others. To do this, you should create a separate properties file to store secure information and refer to that file in your build files as follows:

1. Create a signing configuration, and assign it to one or more build types. These instructions assume you have configured a single signing configuration for your release build type, as described in [Configure the Build Process to Automatically Sign Your APK](#) (#sign-auto), above.
2. Create a file named `keystore.properties` in the root directory of your project. This file should contain your signing information, as follows:

```
storePassword=myStorePassword
keyPassword=mykeyPassword
keyAlias=myKeyAlias
storeFile=myStoreFileLocation
```



3. In your module's `build.gradle` file, add code to load your `keystore.properties` file before the `android {}` block.

```
...

// Create a variable called keystorePropertiesFile, and initialize it to your
// keystore.properties file, in the rootProject folder.
def keystorePropertiesFile = rootProject.file("keystore.properties")

// Initialize a new Properties() object called keystoreProperties.
def keystoreProperties = new Properties()

// Load your keystore.properties file into the keystoreProperties object.
keystoreProperties.load(new FileInputStream(keystorePropertiesFile))

android {
    ...
}
```



★ **Note:** You could choose to store your `keystore.properties` file in another location (for example, in the module folder rather than the root folder for the project, or on your build server if you are using a continuous integration tool). In that case, you should modify the code above to correctly initialize `keystorePropertiesFile` using your actual `keystore.properties` file's location.

4. You can refer to properties stored in `keystoreProperties` using the syntax `keystoreProperties['propertyName']`. Modify the `signingConfigs` block of your module's `build.gradle` file to reference the signing information stored in `keystoreProperties` using this syntax.

```
android {  
    signingConfigs {  
        config {  
            keyAlias keystoreProperties['keyAlias']  
            keyPassword keystoreProperties['keyPassword']  
            storeFile file(keystoreProperties['storeFile'])  
            storePassword keystoreProperties['storePassword']  
        }  
    }  
    ...  
}
```



5. Open the **Build Variants** tool window and ensure that the release build type is selected.
6. Click **Build > Build APK** to build your release build, and confirm that Android Studio has created a signed APK in the `build/outputs/apk/` directory for your module.

Because your build files no longer contain sensitive information, you can now include them in source control or upload them to a shared codebase. Be sure to keep the `keystore.properties` file secure. This may include removing it from your source control system.

Build and sign your app from command line

You do not need Android Studio to sign your app. You can sign your app from the command line using the `apksigner` tool or configure Gradle to sign it for you during the build. Either way, you need to first generate a private key using [keytool](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html)

(<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html>). For example:

```
keytool -genkey -v -keystore my-release-key.jks  
-keyalg RSA -keysize 2048 -validity 10000 -alias my-alias
```



Note: `keytool` is located in the `bin/` directory in your JDK. To locate your JDK from Android Studio, select **File > Project Structure**, and then click **SDK Location** and you will see the **JDK location**.

This example prompts you for 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.jks`, saving it in the current directory (you can move it wherever you'd like). The keystore contains a single key that is valid for 10,000 days.

Now you can [build an unsigned APK and sign it manually](#) (#sign-manually) or instead [configure Gradle to sign your APK](#) (#gradle-sign).

Build an unsigned APK and sign it manually

1. Open a command line and navigate to the root of your project directory—from Android Studio, select **View > Tool Windows > Terminal**. Then invoke the `assembleRelease` task:

```
gradlew assembleRelease
```



This creates an APK named `module_name-unsigned.apk` in `project_name/module_name/build/outputs/apk/`. The APK is *unsigned* and unaligned at this point—it can't be installed until signed with your private key.

2. Align the unsigned APK using [zipalign](https://developer.android.com/studio/command-line/zipalign.html) (<https://developer.android.com/studio/command-line/zipalign.html>):

```
zipalign -v -p 4 my-app-unsigned.apk my-app-unsigned-aligned.apk
```



`zipalign` ensures that all uncompressed data starts with a particular byte alignment relative to the start of the file, which may reduce the amount of RAM consumed by an app.

3. Sign your APK with your private key using [apksigner](https://developer.android.com/studio/command-line/apksigner.html) (<https://developer.android.com/studio/command-line/apksigner.html>):

```
apksigner sign --ks my-release-key.jks --out my-app-release.apk my-app-unsigned-aligned.apk
```



This example outputs the signed APK at `my-app-release.apk` after signing it with a private key and certificate that are stored in a single KeyStore file: `my-release-key.jks`.

The `apksigner` tool supports other signing options, including signing an APK file using separate private key and certificate files, and signing an APK using multiple signers. For

more details, see the **apksigner**

(<https://developer.android.com/studio/command-line/apksigner.html>) reference.

★ **Note:** To use the **apksigner** tool, you must have revision 24.0.3 or higher of the Android SDK Build Tools installed. You can update this package using the [SDK Manager](https://developer.android.com/studio/intro/update.html#sdk-manager) (<https://developer.android.com/studio/intro/update.html#sdk-manager>).

4. Verify that your APK is signed:

```
apksigner verify my-app-release.apk
```



Configure Gradle to sign your APK

1. Open the module-level `build.gradle` file and add the `signingConfigs {}` block with entries for `storeFile`, `storePassword`, `keyAlias` and `keyPassword`, and then pass that object to the `signingConfig` property in your build type. For example:

```
android {  
    ...  
    defaultConfig { ... }  
    signingConfigs {  
        release {  
            storeFile file("my-release-key.jks")  
            storePassword "password"  
            keyAlias "my-alias"  
            keyPassword "password"  
        }  
    }  
    buildTypes {  
        release {  
            signingConfig signingConfigs.release  
            ...  
        }  
    }  
}
```



Because Gradle reads paths relative to the `build.gradle`, the above example works only if `my-release-key.jks` is in the same directory as the `build.gradle` file.



Note: In this case, the keystore and key password are visible directly in the `build.gradle` file. For improved security, you should [remove the signing information from your build file](#) (#secure-key).

2. Open a command line in your project root directory and invoke the `assembleRelease` task:

```
gradlew assembleRelease
```



This creates an APK named `module_name-release.apk` in `project_name/module_name/build/outputs/apk/`. This APK file is signed with the private key specified in your `build.gradle` file and aligned with `zipalign`.

And now that you've configured the release build with your signing key, the "install" task is available for that build type. So you can build, align, sign, and install the release APK on an emulator or device all with the `installRelease` task.

An APK signed with your private key is ready for distribution, but you should first read more about how to [publish your app](https://developer.android.com/studio/publish/index.html) and review the [Google Play launch checklist](https://developer.android.com/distribute/tools/launch-checklist.html).

Content and code samples on this page are subject to the licenses described in the [Content License](#) (/license). Java is a registered trademark of Oracle and/or its affiliates.

Last updated June 29, 2018.



Twitter

Follow @AndroidDev on
Twitter



Google+

Follow Android Developers on
Google+



YouTube

Check out Android Developers
on YouTube