Build and release an Android app

Contents

During a typical development cycle, you test an app using `flutter run` at the command line, or by using the **Run** and **Debug** options in your IDE. By default, Flutter builds a *debug* version of your app.

When you're ready to prepare a *release* version of your app, for example to publish to the Google Play Store, this page can help. Before publishing, you might want to put some finishing touches on your app. This page covers the following topics:

- Adding a launcher icon
- Enabling Material Components
- Signing the app
- Shrinking your code with R8
- Reviewing the app manifest

- Reviewing the build configuration
- Building the app for release
- Publishing to the Google Play Store
- Updating the app's version number
- Android release FAQ

Note: Throughout this page, `[project]` refers to the directory that your application is in. While following these instructions, substitute `[project]` with your app's directory.

Adding a launcher icon

When a new Flutter app is created, it has a default launcher icon. To customize this icon, you might want to check out the flutter_launcher_icons package.

Alternatively, you can do it manually using the following steps:

1. Review the Material Design product icons guidelines for icon design.
2. In the `[project]/android/app/src/main/res/` directory, place your icon files in folders named using configuration qualifiers. The default `mipmap-` folders demonstrate the correct naming convention.
3. In `AndroidManifest.xml`, update the `application` tag's `android:icon` attribute to reference icons from the previous step (for example, `<application android:icon="@mipmap/ic_launcher" ...`).
4. To verify that the icon has been replaced, run your app and inspect the app icon in the Launcher.

Enabling Material Components

If your app uses Platform Views, you may want to enable Material Components by following the steps described in the Getting Started guide for Android.

For example:

1. Add the dependency on Android's Material in `<my-app>/android/app/build.gradle`:

```
dependencies {
    // ...
    implementation 'com.google.android.material:material:<version>'
    // ...
}
```
content_copy

To find out the latest version, visit Google Maven.

1. Set the theme in `<my-app>/android/app/src/main/res/values/styles.xml`:

```
-<style name="LaunchTheme" parent="Theme.AppCompat">
+<style name="LaunchTheme" parent="Theme.MaterialComponents.NoActionBar">
```
content_copy

Signing the app

To publish on the Play Store, you need to give your app a digital signature. Use the following instructions to sign your app.

On Android, there are two signing keys: deployment and upload. The end-users download the .apk signed with the 'deployment key'. An 'upload key' is used to authenticate the .aab / .apk uploaded by developers onto the Play Store and is re-signed with the deployment key once in the Play Store.

- It's highly recommended to use the automatic cloud managed signing for the deployment key. For more information, see the official Play Store documentation.

Create an upload keystore

If you have an existing keystore, skip to the next step. If not, create one by either:

- Following the Android Studio key generation steps
- Running the following at the command line:

  On Mac/Linux, use the following command:

  ```
  keytool -genkey -v -keystore ~/upload-keystore.jks -keyalg RSA -keysize
  2048 -validity 10000 -alias upload
  ```
  content_copy

  On Windows, use the following command:

  ```
  keytool -genkey -v -keystore c:\Users\USER_NAME\upload-keystore.jks
  storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias upload
  ```
  content_copy

  This command stores the `upload-keystore.jks` file in your home directory. If you want to store it elsewhere, change the argument you pass to the `-keystore` parameter. **However, keep the `keystore` file private; don't check it into public source control!**

Note:

- The `keytool` command might not be in your path—it's part of Java, which is installed as part of Android Studio. For the concrete path, run `flutter doctor -v` and locate the path printed after 'Java binary at:'. Then use that fully qualified path replacing `java` (at the end) with `keytool`. If your path includes space-separated names, such as `Program Files`, use platform-appropriate notation for the names. For example, on Mac/Linux use `Program\ Files`, and on Windows use `"Program Files"`.
- The `-storetype JKS` tag is only required for Java 9 or newer. As of the Java 9 release, the keystore type defaults to PKS12.

Reference the keystore from the app

Create a file named `[project]/android/key.properties` that contains a reference to your keystore:

```
storePassword=<password from previous step>
keyPassword=<password from previous step>
keyAlias=upload
storeFile=<location of the key store file, such as /Users/<user name>/upload-keystore.jks>
```

content_copy

Warning: Keep the `key.properties` file private; don't check it into public source control.

Configure signing in gradle

Configure gradle to use your upload key when building your app in release mode by editing the `[project]/android/app/build.gradle` file.

1. Add the keystore information from your properties file before the `android` block:

```
    def keystoreProperties = new Properties()
    def keystorePropertiesFile = rootProject.file('key.properties')
    if (keystorePropertiesFile.exists()) {
        keystoreProperties.load(new
FileInputStream(keystorePropertiesFile))
    }

    android {
          ...
    }
```
content_copy

Load the `key.properties` file into the `keystoreProperties` object.

2. Find the `buildTypes` block:

```
    buildTypes {
        release {
            // TODO: Add your own signing config for the release build.
            // Signing with the debug keys for now,
            // so `flutter run --release` works.
            signingConfig signingConfigs.debug
        }
    }
```
content_copy

And replace it with the following signing configuration info:

```
    signingConfigs {
        release {
            keyAlias keystoreProperties['keyAlias']
            keyPassword keystoreProperties['keyPassword']
            storeFile keystoreProperties['storeFile'] ?
file(keystoreProperties['storeFile']) : null
            storePassword keystoreProperties['storePassword']
        }
    }
    buildTypes {
        release {
            signingConfig signingConfigs.release
        }
    }
```
content_copy

Release builds of your app will now be signed automatically.

Note: You may need to run `flutter clean` after changing the gradle file. This prevents cached builds from affecting the signing process.

For more information on signing your app, see Sign your app on developer.android.com.

Shrinking your code with R8

R8 is the new code shrinker from Google, and it's enabled by default when you build a release APK or AAB. To disable R8, pass the `--no-shrink` flag to `flutter build apk` or `flutter build appbundle`.

Note: Obfuscation and minification can considerably extend compile time of the Android application.

Reviewing the app manifest

Review the default App Manifest file, `AndroidManifest.xml`, located in `[project]/android/app/src/main` and verify that the values are correct, especially the following:

`application`
Edit the `android:label` in the `application` tag to reflect the final name of the app.

`uses-permission`

Add the `android.permission.INTERNET` permission if your application code needs Internet access. The standard template does not include this tag but allows Internet access during development to enable communication between Flutter tools and a running app.

Reviewing the build configuration

Review the default Gradle build file, `build.gradle`, located in `[project]/android/app` and verify the values are correct, especially the following values in the `defaultConfig` block:

`applicationId`
Specify the final, unique (Application Id)appid

`versionCode` & `versionName`
Specify the internal app version number, and the version number display string. You can do this by setting the `version` property in the pubspec.yaml file. Consult the version information guidance in the versions documentation.

`minSdkVersion`, `compilesdkVersion`, & `targetSdkVersion`
Specify the minimum API level, the API level on which the app was compiled, and the maximum API level on which the app is designed to run. Consult the API level section in the versions documentation for details. `buildToolsVersion`

Specify the version of Android SDK Build Tools that your app uses. Alternatively, you can use the Android Gradle Plugin in Android Studio, which will automatically import the minimum required Build Tools for your app without the need for this property.

Building the app for release

You have two possible release formats when publishing to the Play Store.

- App bundle (preferred)
- APK

Note: The Google Play Store prefers the app bundle format. For more information, see Android App Bundle and About Android App Bundles.

Warning: Recently, the Flutter team has received several reports from developers indicating they are experiencing app crashes on certain devices on Android 6.0. If you are targeting Android 6.0, use the following steps:

- If you build an App Bundle Edit `android/gradle.properties` and add the flag:`android.bundle.enableUncompressedNativeLibs=false`.
- If you build an APK Make sure `android/app/src/AndroidManifest.xml` doesn't set `android:extractNativeLibs=false` in the `<application>` tag.

For more information, see the public issue.

Build an app bundle

This section describes how to build a release app bundle. If you completed the signing steps, the app bundle will be signed. At this point, you might consider obfuscating your Dart code to make it more difficult to reverse engineer. Obfuscating your code involves adding a couple flags to your build command, and maintaining additional files to de-obfuscate stack traces.

From the command line:

1. Enter `cd [project]`
2. Run `flutter build appbundle`
   (Running `flutter build` defaults to a release build.)

The release bundle for your app is created at `[project]/build/app/outputs/bundle/release/app.aab`.

By default, the app bundle contains your Dart code and the Flutter runtime compiled for armeabi-v7a (ARM 32-bit), arm64-v8a (ARM 64-bit), and x86-64 (x86 64-bit).

Test the app bundle

An app bundle can be tested in multiple ways—this section describes two.

Offline using the bundle tool

1. If you haven't done so already, download `bundletool` from the GitHub repository.
2. Generate a set of APKs from your app bundle.
3. Deploy the APKs to connected devices.

Online using Google Play

1. Upload your bundle to Google Play to test it. You can use the internal test track, or the alpha or beta channels to test the bundle before releasing it in production.
2. Follow these steps to upload your bundle to the Play Store.

Build an APK

Although app bundles are preferred over APKs, there are stores that don't yet support app bundles. In this case, build a release APK for each target ABI (Application Binary Interface).

If you completed the signing steps, the APK will be signed. At this point, you might consider obfuscating your Dart code to make it more difficult to reverse engineer. Obfuscating your code involves adding a couple flags to your build command.

From the command line:

1. Enter `cd [project]`
2. Run `flutter build apk --split-per-abi`
   (The `flutter build` command defaults to `--release`.)

This command results in three APK files:

- `[project]/build/app/outputs/apk/release/app-armeabi-v7a-release.apk`
- `[project]/build/app/outputs/apk/release/app-arm64-v8a-release.apk`
- `[project]/build/app/outputs/apk/release/app-x86_64-release.apk`

Removing the `--split-per-abi` flag results in a fat APK that contains your code compiled for *all* the target ABIs. Such APKs are larger in size than their split counterparts, causing the user to download native binaries that are not applicable to their device's architecture.

Install an APK on a device

Follow these steps to install the APK on a connected Android device.

From the command line:

1. Connect your Android device to your computer with a USB cable.
2. Enter `cd [project]`.
3. Run `flutter install`.

Publishing to the Google Play Store

For detailed instructions on publishing your app to the Google Play Store, see the Google Play launch documentation.

Updating the app's version number

The default version number of the app is `1.0.0`. To update it, navigate to the `pubspec.yaml` file and update the following line:

```
version: 1.0.0+1
```

The version number is three numbers separated by dots, such as `1.0.0` in the example above, followed by an optional build number such as `1` in the example above, separated by a `+`.

Both the version and the build number may be overridden in Flutter's build by specifying `--build-name` and `--build-number`, respectively.

In Android, `build-name` is used as `versionName` while `build-number` used as `versionCode`. For more information, see Version your app in the Android documentation.

After updating the version number in the pubspec file, run `flutter pub get` from the top of the project, or use the **Pub get** button in your IDE. This updates the `versionName` and `versionCode` in the `local.properties` file, which are later updated in the `build.gradle`file when you rebuild the Flutter app.

Android release FAQ

Here are some commonly asked questions about deployment for Android apps.

When should I build app bundles versus APKs?

The Google Play Store recommends that you deploy app bundles over APKs because they allow a more efficient delivery of the application to your users. However, if you're distributing your application by means other than the Play Store, an APK may be your only option.

What is a fat APK?

A fat APK is a single APK that contains binaries for multiple ABIs embedded within it. This has the benefit that the single APK runs on multiple architectures and thus has wider compatibility, but it has the drawback that its file size is much larger, causing users to download and store more bytes when installing your application. When building APKs instead of app bundles, it is strongly recommended to build split APKs, as described in build an APK using the `--split-per-abi` flag.

What are the supported target architectures?

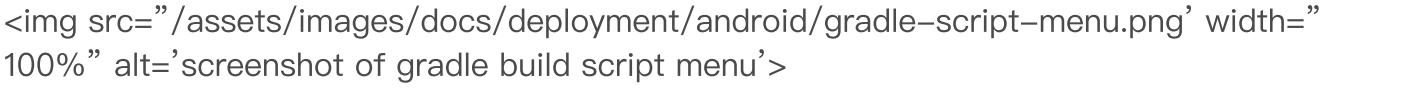
When building your application in release mode, Flutter apps can be compiled for armeabi-v7a (ARM 32-bit), arm64-v8a (ARM 64-bit), and x86-64 (x86 64-bit). Flutter does not currently support building for x86 Android (See Issue 9253).

How do I sign the app bundle created by `flutter build appbundle`?

See Signing the app.

How do I build a release from within Android Studio?

In Android Studio, open the existing `android/` folder under your app's folder. Then, select **build.gradle (Module: app)** in the project panel:



Next, select the build variant. Click **Build > Select Build Variant** in the main menu. Select any of the variants in the **Build Variants**panel (debug is the default):

<img src=”/assets/images/docs/deployment/android/build-variant-menu.png’ width=” 100%” alt=’screenshot of build variant menu’>

The resulting app bundle or APK files are located in `build/app/outputs` within your app’s folder.