IN THIS CHAPTER

- Creating a developer account
- Prepping your code
- Uploading your app

Chapter 1
Publishing Your App to the Google Play Store

First-time app publishing is both exciting and scary. It's exciting because, after months of development work, you're finally doing something "real" with your app. It's scary because you're exposing your app to the public. You're afraid of pressing the wrong button on the Google Play developer page and accidentally telling the world that your app kills kittens.

Well, you can relax about killing kittens. The Google Play developer page helps you get things right. And if you get something wrong, you can correct it pretty quickly. As for the excitement of publishing, there's nothing quite like it.

In this chapter, you take those courageous steps. You create a developer account and publish your first app.

Creating a Google Play Developer Account

Choosing Android for your app platform has some distinct advantages. To develop for the iPhone, you pay an annual $99 fee. Over a ten-year period, that's about a thousand bucks. As an Android developer, working over the same ten-year period, you pay $25. That's all.

To create a Google Play developer account, visit `https://play.google.com/apps/publish/signup`. On this page, you do the following:

- **Agree to the Google Play developer distribution rules.**
- **Pay the $25 fee.**
- **Provide your account details.** These details include your name, email, phone, and (optionally) your website URL.

  If you're working with a team, you can provide your coworkers' email addresses and set up each coworker's permissions on your account.

  If you plan to collect money through your app, you can set up a merchant account. When you do so, you provide your business name, contact name, address, phone, website URL, customer service email, and credit card statement name. There's also a What Do You Sell drop-down list. Do you sell Automotive and Marine supplies? Nutrients and Supplements? Timeshares? And what about Other? Do you sell Other?

  The information that you provide when you first sign up isn't cast in stone. You can change this information later using the Google Play's Settings page.

Preparing Your Code

At this point, you're probably tired of looking at your own app. You've written the basic app, tested the app, fixed the bugs, tested again, added features, done more testing, stayed up late at night, and done even more testing. But if you plan to publish your app, please follow this advice: After you've finished testing, test some more.

Ask yourself what sequences of buttons you avoided clicking when you did your "thorough" testing. Then muster the courage to click the buttons and use the widgets in those strange sequences. And while you're at it, tilt the device sideways, turn the device upside down, hold the device above your head, and try using the app. If your device is a phone, interrupt the app with an incoming call.

Are you finished testing? Not yet. Have your friends test the app on their devices. Whatever you do, don't give them any instructions other than the instructions you intend to publish. Better yet, don't even give them the instructions that you intend to publish. (Some users won't read those instructions anyway.) Ask your friends about their experiences running your app. If you sense that your friends are being too polite, press them for more details.

TIP

You can "publish" your app on Google Play so that only your designated friends can install the app. For more information, skip ahead to the section entitled "The App Releases page."

REMEMBER

When you test your app, be your app's worst enemy. Try as hard as you can to break your app. Be overly critical. Be relentless. If your app has a bug and you don't find it, your users will.

Un-testing the app

When you test an app, you find features that don't quite work. You check the logs, and you probably add code to help you diagnose problems. As you prepare to publish your app, remove any unnecessary diagnostic code, remove extra logging statements, and remove any other code whose purpose is to benefit the developer rather than the user.

In developing your app, you might have created some test data. (Is there a duck named "Donald" in your app's contact list?) If you've created test data, delete the data from your app.

Check your project's `AndroidManifest.xml` file. If the `<application>` element has an `android:debuggable="true"` attribute, remove that attribute. (The attribute's default value is `false`.)

Choosing Android versions

An app's `build.gradle` file specifies three SDK versions: `compileSdkVersion`, `minSdkVersion`, and `targetSdkVersion`. (In case you're wondering, Book 1, Chapter 4 tells you all about these SDK versions.) When you create a new project, Android Studio sets the compile and target SDK versions to the newest full release. You shouldn't change these values. Google released API Level 28 in August of 2018, and by November 2019, all new apps and updates to existing apps were required to target Level 28 or higher. This requirement will update each year with new versions of the Android SDK.

TIP

To read about the yearly version requirements, visit `https://developer.android.com/distribute/best-practices/develop/target-sdk` and `https://android-developers.googleblog.com/2019/02/expanding-target-api-level-requirements.html`.

Your app's `minSdkVersion` is a completely different story. When you create a new project, Android Studio asks you for a Minimum SDK version. This `minSdkVersion` number is important because it shouldn't be too low or too high.

- **If the minSdkVersion number is too low, your app isn't using features from newer Android versions.** If your app is very simple, this is okay. But if your app does anything that looks different in newer Android versions, your app's vintage look might turn users off.
- **If the `minSdkVersion` number is too high, Google's Play Store won't offer your app to users with older devices.** In fact, if your app's `minSdkVersion` is API 26, a user who visits the Play Store on an Android Nougat device doesn't even see your app. (You might have already encountered the `INSTALL_FAILED_OLDER_SDK` error message. Android Studio can't install an app on the emulator that you selected because the emulator's SDK version is lower than the app's `minSdkVersion`.)

  You don't want to eliminate users simply because they don't have the latest and greatest Android devices. So to reach more users, keep the `minSdkVersion` from being too high. If your app doesn't use any features that were introduced after API Level 23, set your `minSdkVersion` to 23.

Try running your app on emulators with many API levels. When you run into trouble (say, on an emulator with API Level 21) set your project's `minSdkVersion` to something higher than that troublesome level. You can change this number by editing the `build.gradle` file.


TIP

When you create a new project, the Minimum SDK drop-down list comes with its own Help Me Choose link. When you click this link, you see a chart showing the percentage of devices running Android 8, 9, 10, and other versions. This clickable chart describes the features in each Android version and (most important) shows the percentage of devices that are running each version. With information from this chart, you can choose the best compromise between the latest features and the widest user audience.


TIP

The AndroidX libraries allow devices with older Android versions to take advantage of newer Android features. For info, refer to Book 3, Chapter 1 and visit `https://developer.android.com/jetpack/androidx`.

Setting your app's own version code and version name

When you create a new project, Android Studio puts some default attributes in your `build.gradle` file. These attributes include the `versionCode` and `versionName` fields:

```
defaultConfig {
  ...
  versionCode 1
  versionName "1.0"
}
```

The version code must be an integer, and your app's code numbers must increase over time. For example, if your first published version has version code 42, your second published version must have a version code higher than 42.

Users never see the version code. Instead, users see your app's version name. You can use any string for your app's version name. Many developers use the `major-release.minor-release.point` system. For example, a typical version name might be `"1.2.2"`. But there are no restrictions.

Choosing a package name

Your app's package name should identify you or your company. If you have a domain name, start the package name with the domain name's parts reversed. Your domain name can't be `example.com`. But if it could be `example.com`, your first app's package name might be `com.example.earnmeamillion`. Every app must have its own package name, so your second app's package name would have to be different. Maybe it would be `com.example.secondtimeisacharm`.

TECHNICAL STUFF

The Internet Engineering Task Force reserves `example.com` as a placeholder name. Android Studio suggests `com.example` as part of a temporary package name for a new project. You also see `example.com` in Internet-related documentation.

REMEMBER

The Play Store has a few of its own restrictions on things you can use as package names. You can read all about this by visiting `https://developer.android.com/studio/build/application-id`.

Preparing Graphic Assets for the Play Store

When you publish an app to the Play Store, you interact with the Google Play Console. The essential step in this interaction is the step in which you upload your app's installation file. It's the essential step but by no means the only step. You also answer dozens of questions about your app, and you upload many graphic assets. This section describes those graphic assets.

Creating an icon

When you create a new project, Android Studio puts some default attributes in your `AndroidManifest.xml` file. One of them is the `android:icon` attribute:

```
<application android:icon="@mipmap/ic_launcher"
... >
```

In that attribute, the name `"@mipmap/ic_launcher"` refers to a bunch of `ic_launcher` files in various parts of your project's `res/mipmap` folder. Before publishing your app, you should replace this default icon name with your own icon's name.

The Play Store also requires you to submit a high-resolution icon — a high-quality version of your app's signature icon. The high-res icon appears here and there on the Play Store's pages.

Android Studio comes with a super-duper icon-building tool named Asset Studio. Here's a quick experiment to get you started using it:

1. **In the Project tool window, right-click the `res` branch.**

   As a result, a context menu appears.

2. **In the context menu, select New➪ Image Asset.**

   The Asset Studio window appears. The window's Name field contains the default name `ic_launcher`. (See Figure 1-1.)

3. **Type something different in the window's Name field . . . or don't!**

   If you don't, your new icon will replace Android's default `ic_launcher` icon. That way, you'll have no trouble finding the icon when you run the app. (Of course, the name `ic_launcher` might confuse other Android developers!)

4. **In the Asset Type radio group, select Image.**

   The alternatives are Clip Art and Text.

5. **To fill in the Path field, navigate to the location of an image on your development computer.**

   At this point, you've specified your icon's foreground layer.

6. **Repeat Step 5 with the dialog box's Background Layer tab selected.**

   For the Background Layer, the Asset Type radio group has only two choices: Color and Image.

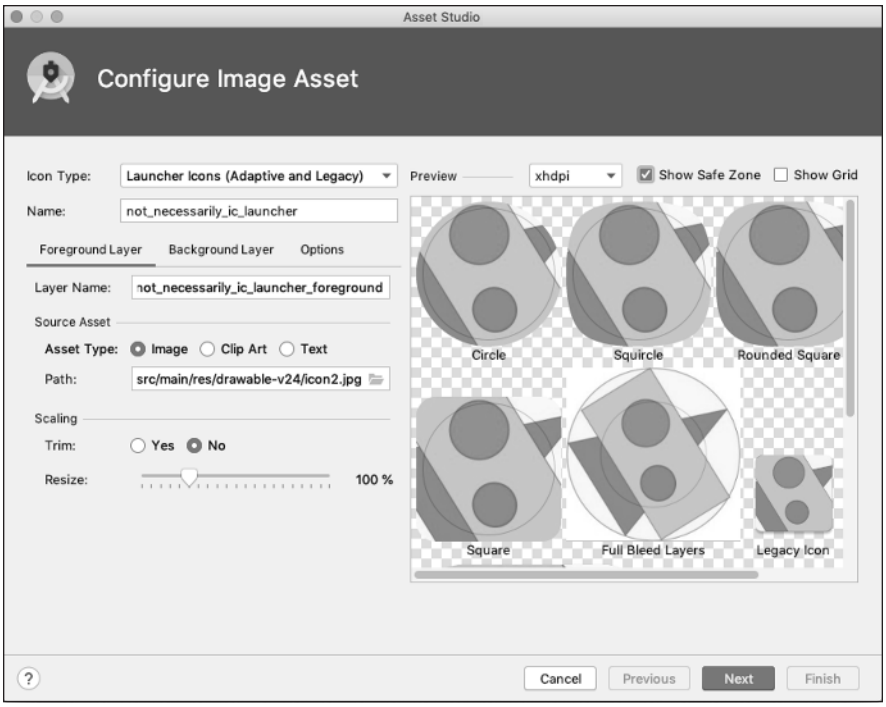


FIGURE 1-1: The Configure Image Asset dialog box.

7. **Press Next.**

   Asset Studio shows you a list of files and the directories to be created. When you select a file in the list, you see a preview of the file's contents. (See Figure 1-2.) If you like what you see, move on to the next step.

8. **Press Finish.**

   Presto! You're back to Android Studio's main window.

9. **Run your app and look for your brand-new icon.**

   Nice icon! Isn't it?

To create images for Asset Studio, you can use almost any drawing software. You can also create icons by visiting `https://romannurik.github.io/AndroidAssetStudio/`. In addition, you can download prepackaged icon packs from Google's Play Store.

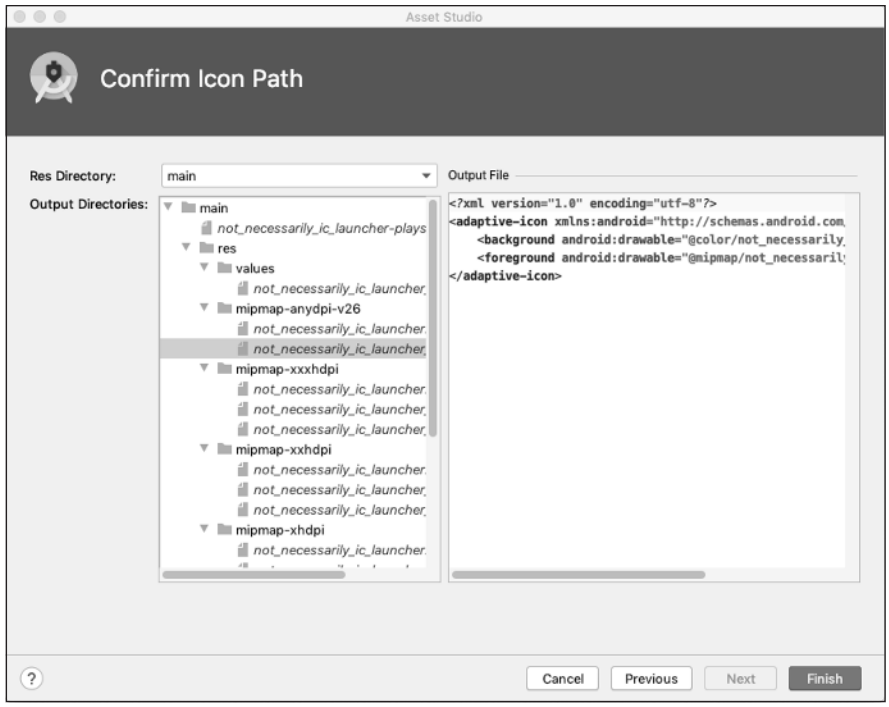


FIGURE 1-2: The Confirm Icon Path dialog box.

There's no shortage of documentation to help you create stunning Android icons:

- To read about the Play Store's design specs, visit `https://developer.android.com/google-play/resources/icon-design-specifications`.
- Android's adaptive icons look different depending on the device that's running your app. To find out about adaptive icons, visit `https://developer.android.com/guide/practices/ui_guidelines/icon_design_adaptive`.
- For guidelines on submitting graphic assets of any type, visit `https://support.google.com/googleplay/android-developer/answer/1078870`.

Creating screenshots

Along with every app that you submit to the Play Store, you must submit at least two screenshots. The Google Play Console has slots for phone screenshots, tablet screenshots, TV screenshots, and wearable screenshots. Each screenshot must be JPEG or 24-bit PNG with no alpha transparency. The minimum length for any side is 320 pixels, and the maximum length for any side is 3,840 pixels.

You have many ways to take screenshots of your running app. When you run the emulator, you get a camera icon on the emulator window's side menu bar. Another way to get a screenshot is to use Android Studio's built-in Screen Capture facility. Here's how you do it:

1. **Use Android Studio to run your app on an emulator or real device.**
2. **Look for a camera icon along the left side of the Logcat tool window.**

   If you don't see the camera icon, look instead for a tiny "more stuff" icon. That icon looks like two angle brackets (>>). When you select that icon, you see the camera icon. (See Figure 1-3.)

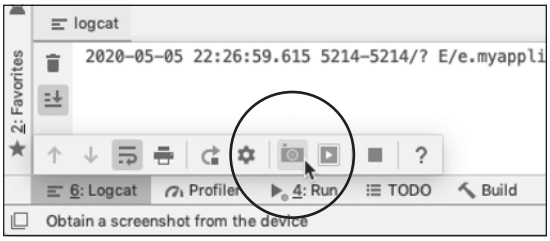


FIGURE 1-3: Getting a screenshot.

3. **Click the camera icon.**

   A new window appears. The window shows a screen capture of your device or emulator. You can click the Save button immediately. But you can also click Recapture or Rotate, or make other adjustments.

If Android Studio's Screen Capture facility isn't your thing, you have several alternatives. For example, you can use your operating system's screen capture facility to take a screenshot of a running emulator.

In Windows

1. **Launch your app in the emulator.**
2. **Click the emulator window (so that the emulator window is the active window).**
3. **While you hold down the Alt key, press the Print Screen key in the upper-right corner of the keyboard.**

   Depending on your keyboard's make and model, the key might be labeled PrintScr, PrtSc, or some other variant of Print Screen.

TIP

4. **Open an image-editing program.**

   If you don't already have a favorite image-editing program, try IrfanView (`www.irfanview.com`). It's full-featured and completely free.

5. **Press Ctrl-V to paste your new screenshot into the image-editing program.**

On a Mac

1. **Launch your app in the emulator.**
2. **Press Cmd+Shift+4.**

   This tells the Mac that you intend to take a screenshot.

3. **Press the spacebar.**

   This tells the Mac that the screenshot will capture a single window.

4. **Click anywhere inside the emulator window.**

   Your computer creates the screenshot and places it on the desktop.


TIP

You can use your development computer's screen capture facility to take a screenshot from a physical device. Start by searching for a program that mirrors an Android device's screen on your computer. After installing the program, follow this section's instructions for taking a screenshot in Windows or on a Mac.

In theory, you can press the Volume Down and Power buttons simultaneously to take a screenshot. The trouble is, the exact sequence of presses and button holds varies from one make and model to another. Check your device's documentation (and other sources) for more info.

Your options for creating screenshots are endless. If all else fails (and, in fact, all else seldom fails), you can get screenshots from running emulators and devices using your computer's command line. Search the web for `adb shell/system/bin/screencap`.

Providing other visual assets

If your app is featured on the Play Store (or rather, *when* your app is featured), a *feature graphic* appears on your store listing page. Your feature graphic must be a JPEG or 24-bit PNG file with no alpha transparency. Its dimensions must be 1,024 by 500 pixels.

The blog page at `http://android-developers.blogspot.com/2011/10/android-market-featured-image.html` has time-tested advice on creating feature graphics. The key is to create an eye-catching image that promotes your app

without replicating your app's screen. You should also make sure that the image looks good no matter what size screen displays it.

Why stop after you've uploaded screenshots and a feature graphic? You can also upload a 180-by-120-pixel promo graphic and a 1,280-by-720-pixel TV banner. If that's not enough, you can throw in a promotional video and a 4,096-by-4,096-pixel Daydream stereo image.

Who knows? Maybe next year you'll be able to upload a hologram!

Creating a Publishable File

When you create an app that runs on an emulator or a device, Android Studio packages your app in one of two ways: as an Android Package (APK) file or an Android App Bundle (AAB) file. You can upload either kind of file to Google's Play Store.

- **If you upload an APK file, the Play Store copies that file to users' devices.** The APK filename extension is `.apk`.
- **If you upload an AAB file, the Play Store creates custom APK files for the users' devices.** When a user wants to install your app, the Play Store picks and chooses the parts of the AAB that are required for that particular user's device. As a result, the Play Store sends a leaner APK file to the device. The custom APK file has only the components needed by that user's device.

  In addition, the Play Store can send a reduced-feature app — one that downloads and starts running very quickly. While the user explores the app's most basic features, the Play Store readies more components for download to the device. Without any special actions on the user's part, the app's feature set grows as needed. This is called *Dynamic Feature Module Delivery,* or *Dynamic Delivery* for short. (See `https://developer.android.com/guide/app-bundle/dynamic-delivery` for more details about Dynamic Delivery.)

  For any large, multifaceted app, AAB files are the way to go. But you can't deploy an AAB file directly to a device. For a simpler app, an APK file is good enough.

  The AAB file name extension is `.aab`. (Big surprise, right?)

Differences among builds

You may not be used to using *build* as a noun. In geek language, the noun *build* refers to a file or set of files that, in some way or another, are ready to run. When you ask Android Studio to run your app, your computer creates a build of the app and then *deploys* the build onto an AVD or a physical device.

As you develop an application, you create several different builds, each with its own characteristics and each for its own purposes. Android's official terminology classifies builds in two ways: by build variant and by flavor.

- **The differences among *build variants* are visible to the app developer.** By default, a new Android Studio project describes two different build variants — *debug* build and *release* build. When you create, test, and modify your app, you run debug builds. But to publish an app to the Google Play Store, you must create a release build.

  The next section lists some important characteristics of a release build.

- **The differences among *flavors* are visible to the user.** Android Studio doesn't assign flavors to a new project. The creation of flavors is up to each app's developer. Many apps come in two flavors: free and paid. Another breakdown by flavors is by country: one flavor for most countries, another for countries with special content restrictions.

To manage your app's build variants and flavors, go to Android Studio's main menu bar and choose Build➪Edit Build Types or Build➪Edit Flavors.

What is a release build? Funny that you should ask! Here are some facts about release builds:

- **A release build contains your digital signature.** A *digital signature* is a sequence of bits that only you (and no one else) can provide. If your APK or AAB file contains your digital signature, no one can pretend to be you. (No one can write a malicious version of your app and publish it on the Play Store site.)

  When you follow this section's instructions, you use Android Studio to create your own digital signature. This signature lives in a directory on your computer's hard drive. You can't examine this signature with an ordinary text editor (Notepad or TextEdit, for example), but you should treat that directory the way you treat any other confidential information. Do whatever you normally do with data to prevent the loss of the data and to keep others from using it.

  You can read more about digital signatures in this chapter's "Understanding digital signatures" sidebar.

TIP

UNDERSTANDING DIGITAL SIGNATURES

When you digitally sign a file, you add a sequence of bits that only you can add. You use sophisticated software to create the sequence and to embed the sequence in your file. The software to create this sequence uses techniques from number theory. (Sometime between 1777 and 1855, Carl Friedrich Gauss called number theory "the queen of mathematics," and he wasn't kidding!)

Digital signing actually involves two sequences of bits:

- **A private key:** A sequence that you don't share with others.
- **A public key:** A sequence that you do share with others.

The private key never leaves your office, but you can display the public key on a neon sign in Times Square. If you tell someone your private key, you'd have to . . . (well, you know). But you can hire a pilot to write your public key with white smoke in the sky over the Golden Gate Bridge.

To sign an app, you run software that adds a certificate to your app. A *certificate* is a bunch of information that includes your private key, your public key, some information to identify you, and some other information. Like your signature on a contract, a certificate's private key is difficult to fake. But with the certificate's public key, a program on the user's device verifies that your app is authentic.

A user gets keys from your app's certificate. But as a developer, you store keys apart from any certificate. You keep public and private keys in a place where your software can retrieve them — a *key store* file. When you digitally sign an app, software grabs keys from a key store file, uses the keys to create a certificate, and melds the certificate into your build file. If you visit your user home directory, and drill down to an `.android` subdirectory (starting with a dot), you probably find the `debug.keystore` file. When you test an app on an emulator or on your own device, Android Studio quietly signs your app with a simple key from this `debug.keystore` file. (For help finding your user home directory, see Book 1, Chapter 2.)

A key store file contains sensitive information, so every key store file is password-protected. Android's `debug.keystore` file is password-protected. But unlike most key store files, the `debug.keystore` file's password is freely available. The password is `android`. Anyone can sign any app using keys from the `debug.keystore` file. That's okay because the `debug.keystore` file's keys don't work for apps that you publish on the Play Store (or anywhere else, for that matter). So before publishing your app, Android Studio adds your own keys to the app.

(continued)

(continued)

After downloading your app, a user's software applies a public key to verify that your app is signed properly. And what does that prove? Well, if a hacker tampers with your app somewhere between publication and the user's downloading, the test for proper signing detects the tampering. "Sorry," says Android, "I refuse to install this app."

But what about a malicious hacker who creates a damaging app and uses Android's freely available tools to sign it? To the world in general, the app looks fine. Signing doesn't verify that an app's developer has good intentions.

The weak link in the chain is the fact that Android apps are *self-signed.* When you add a digital signature to your app, no one else signs with you.

For scenarios that require more security (scenarios not normally associated with mobile devices), a developer can get help from a certificate authority. A certificate authority is an organization that issues special digital signatures — signatures that the world recognizes as very trustworthy. To get such a signature, you convince a certificate authority that you're a good person, and you pay some money to the certificate authority. Some certificate authorities issue signatures for free. These free signatures are okay, but they aren't as trusted as the paid signatures, and they don't have the same clout as the paid signatures.

For more information about Android app signing, visit `http://developer.android.com/tools/publishing/app-signing.html`.

TIP

» **A release build's code is obfuscated.** Obfuscated code is confusing code. And, when it comes to foiling malicious hackers, confusing code is good code. If other people can make sense of your Kotlin code, they can steal it. They can make money off your ideas, or they can add snippets to your code to rob users' credit card accounts.

You want developers on your team to read and understand your code with ease, but you don't want some outsider (like our friend Joe S. Uptonogood) to understand your code. That's why, when you follow this chapter's instructions, Android Studio creates files with obfuscated code.

You can read more about obfuscated code in this chapter's "Don't wait! Obfuscate!" sidebar.

DON'T WAIT! OBFUSCATE!

Nestled quietly inside your project's directory is a `proguard-rules.pro` file. When you ready your project for upload to the Play Store, Android Studio creates two additional files named `proguard-android.txt` and `proguard-android-optimize.txt`. These three files contain configuration information for the compiler. The compiler minifies, preverifies, and obfuscates your code.

To *minify* code is to make the code smaller by removing unnecessary classes, fields, and methods. This includes classes belonging to the Kotlin and Android libraries.

Preverifying code means performing a certain kind of safety check on the code. This safety check looks for places where the code can escape from its virtual machine and start running wild on the rest of the device's operating system. Code that doesn't pass this check gets a failing grade from the compiler.

When you *obfuscate* something, you make it difficult to read. You scramble stuff and generally do the opposite of what you're supposed to do when you write clear, maintainable code.

Why do this? An obfuscated program can be executed without modification by an appropriate device. The device doesn't need a password and doesn't have to decrypt anything in order to run the code. In fact, an obfuscated program contains nothing unusual as far as the Android runtime (ART) is concerned. But for a person trying to reverse-engineer your code, the obfuscation is a nightmare. That's because the human mind doesn't process code mechanically. Instead, humans get the big picture; humans have to understand things in order to work with them; humans feel stress when they work with things that are terse, circuitous, and highly compressed.

So with obfuscated code, evil people can't easily figure out how your code works. They have trouble stealing your tricks, and (more important) they can't easily add viruses to your published code.

Before publishing on the Play Store, you must obfuscate your app's code. Fortunately, the steps in this chapter's "Creating the release build" section do the obfuscation for you. Android's compiler turns your code into a dizzying mess for anyone trying to tinker with it.

(In case you're wondering, the word `proguard` in the configuration filenames has nothing to do with the open source ProGuard project. In days gone by, Android used ProGuard to obfuscate code. Android moved away from ProGuard but didn't bother to change the filenames.)

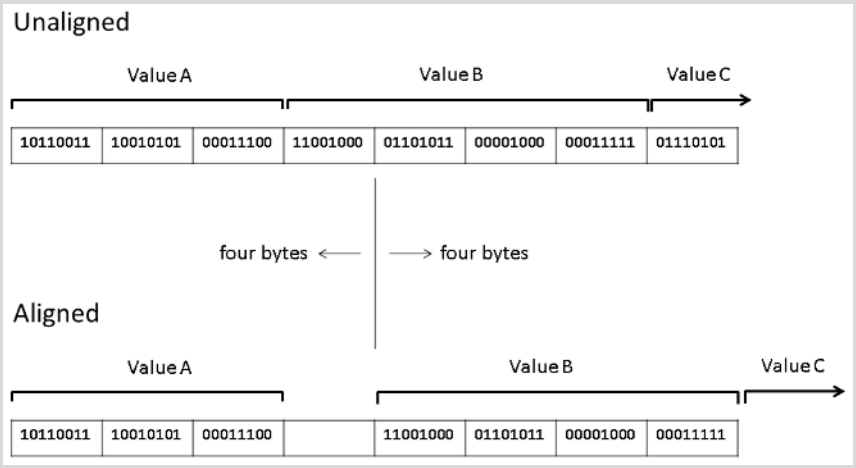
» **A release build's code is zipaligned.** Zipaligned code is easier to execute than code that's not zipaligned. Fortunately, when you follow this chapter's steps, Android Studio zipaligns your code (and does so behind your back, without any intervention on your part).

BYTE OFF MORE THAN YOU CAN VIEW

The Android operating system (along with all other UNIX-like systems) has a `mmap` program. The letters `mm` in `mmap` stand for *memory-mapped* input and output. The `mmap` program grabs data from a file and makes the data available to applications. The `mmap` program is a real workhorse, providing quick and efficient data access for many apps at once.

The nimbleness of `mmap` doesn't come entirely for free. For `mmap` to do its work, certain values must be stored so they start at four-byte boundaries. To understand four-byte boundaries, think about a chunk of data in your application's APK file. A *byte* is eight bits of data (each bit being a 0 or a 1). So four bytes is 32 bits. Now imagine two values (Value A and Value B) stored one after the other in your APK file. (See the figure in this sidebar.) Value A consumes three bytes, and Value B consumes four bytes.

Without four-byte alignment, the computer might store the first byte of Value B immediately after the last byte of Value A. If so, Value B starts on the last byte of a four-byte group. But `mmap` works only when each value starts at the beginning of a four-byte group. So Android's `zipalign` program moves data, as shown in the lower half of the figure below. Instead of using every available byte, `zipalign` wastes a byte in order to make Value B easy to locate.



TIP

You can read more about zipalignment in this chapter's "Byte off more than you can view" sidebar.

- **Android's build tools may reduce your release build's file size.** Maybe your code is somewhat bloated, or your `res` folders have more bytes than they need. Android Studio's build process can take care of all that. In Android Studio's Project tool window, double-click the `build.gradle` file (the one labeled *Module*). In that file, add two lines in `buildTypes`/`release` section, like so:

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile(
                'proguard-android-optimize.txt'),
                'proguard-rules.pro'
    }
}
```

With `minifyEnabled` set to `true`, the compiler puts your project's code on a diet. And with `shrinkResources` set to `true`, the compiler slims down your `res` folders' files.

WARNING

If you use something called JNI, setting `minifyEnabled` to `true` can break your code. For details and workarounds, visit `https://developer.android.com/studio/build/shrink-code`.

Creating the release build

Several paragraphs leading up to this section give lengthy descriptions of the ways release builds differ from debug builds. With all that chatter about release builds, you'll be surprised to find out that creating a release build isn't very complicated. Just follow these instructions:

1. **Make the changes described in this chapter's "Preparing Your Code" section.**

2. **In Android Studio's main menu, choose Build ➪ Generate Signed Bundle / APK.**

   The first page of a dialog box appears. The box's title is Generate Signed Bundle or APK. (See Figure 1-4.)

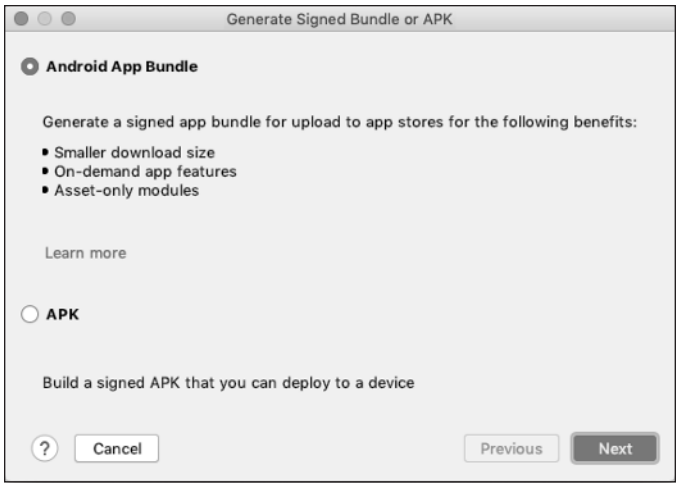


FIGURE 1-4: Bundle or APK?

3. **Select either Android App Bundle or APK.**

   If you need help choosing, refer to the start of the "Creating a Publishable File" section, earlier in this chapter.

4. **Click Next.**

   The top of the dialog box's next page has a drop-down list in which you select one of your project's modules. If your project has only one module (named app, for example) Android Studio grays out the drop-down list. (See Figure 1-5.)

   Your next task is to put something in the dialog box's Key Store Path field.

TIP

For some good bedtime reading about key stores, see the "Understanding digital signatures" sidebar.

At this point, it helps to understand the difference between a key store file and a single key. A *key* is what you use to digitally sign your Android app. A *key store file* is a place to store one or more keys. In this section's instructions, you create two passwords — one for the new key store file and another for the key that you'll be putting in the key store file. For details, see the sidebar entitled "Understanding digital signatures."

The Key Store Path field offers you three options: (a) Click a button to start creating a new key store file; (b) Click a button to choose an existing key store file (one that's already on your computer's hard drive); or (c) Type the full pathname of an existing key store file in the Key Store Path field. In what follows, you pursue the first option.

5. **Click the Create New button.**

   A New Key Store dialog box opens. (See Figure 1-6.)

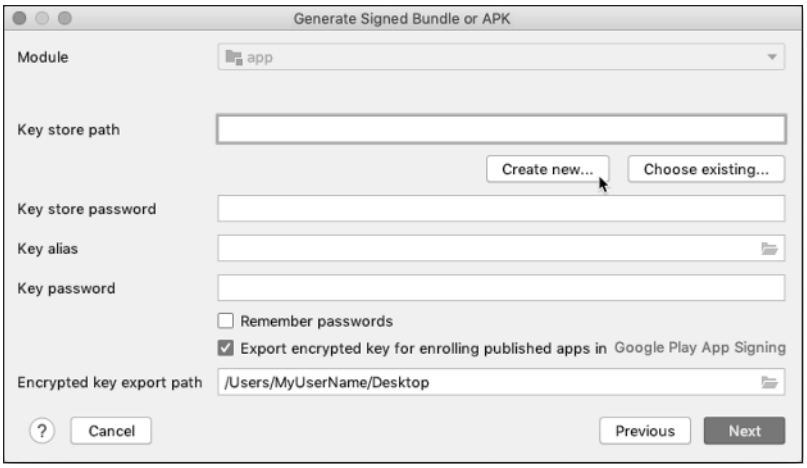


FIGURE 1-5: A bunch of questions about keys and key stores.

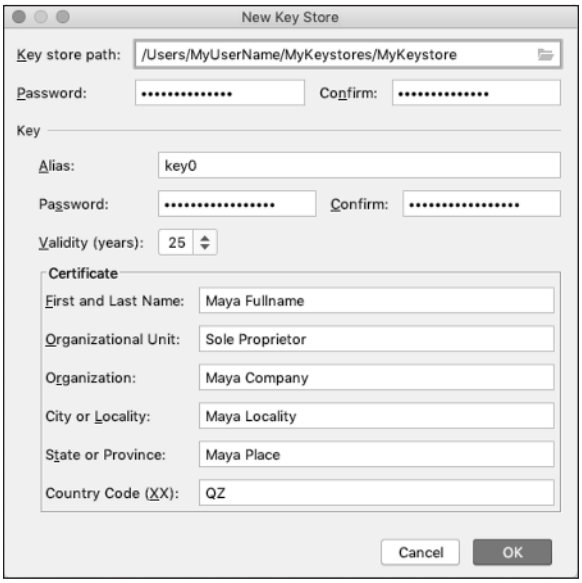


FIGURE 1-6: The New Key Store dialog box.

6. Click the folder icon at the rightmost end of the Key Store Path field. (Refer to Figure 1-6.)

In the resulting dialog box, you navigate to a folder on your computer's hard drive. In addition, you make up a name for your new key store file.

Where do you want to put your new key store file and what do you want to name the file? The choice is yours. One way or another, name the key store file *whatever_you_want*`.jks`. The extension `.jks` stands for *Java key store.*

TIP

Signing all your Android projects with the same key is generally a good idea. Android treats the key as a kind of fingerprint, and two apps with the same fingerprint can be trusted to communicate with one another. When two apps have the same key, you can easily get these apps to help one another out. But reusing a key has a potential downside. It's the same problem you have when you reuse a password. If your one and only key is compromised, all your apps are compromised.

If you decide on maintaining only one key, resist the temptation to put the key store file in your app's project directory. When you do that, you're hinting that the key store belongs exclusively to your current Android project. But when you publish more apps, you'll want to use this key store to sign other projects' files.

7. **Do whatever you must do to confirm your choice of a key store file's location and name.**

   Click OK, or something like that.

   Returning to the New Key Store dialog box, you see that the box has two Password fields and two Confirm fields. (Refer to Figure 1-6)

8. **Enter passwords in the Password and Confirm fields.**

   Do yourself and favor and make 'em strong passwords.

REMEMBER

(In the lingo of *For Dummies* books, this is a Remember icon.) Please remember to remember the passwords that you create when you fill in the Password and Confirm fields. You'll need to enter these passwords when you use this key to sign another app.

A key store file may contain several keys, and each key has a name (an *alias,* that is).

9. **Type a name in the Alias field.**

   The alias can be any string of characters, but don't be too creative when you make up an alias. Avoid blank spaces and punctuation. If you ever create a second key with a second alias, make sure that the second alias's spelling (and not only its capitalization) is different from the first alias's spelling.

10. **Accept the default validity period (25 years).**

    If you create a key on New Year's Day in 2021, the key will expire on New Year's Day in 2046. Happy New Year, everybody! According to the Play Store's rules, your key must not expire until sometime after October 22, 2033, so 25 years from 2021 is okay. (A recent Google search to find out how the creators of Android decided on the date October 22, 2033 came up empty. What kind of party will you throw when this day finally rolls around?)

11. **In the Certificate section, fill in the six fields. (Refer to Figure 1-6.)**

TECHNICAL STUFF

The items *First and Last Name, Organizational Unit,* and so on are part of the *X.500 Distinguished Name* standard. The probability of two people having the same name and working in the same unit of the same organization in the same locality is close to zero.

When you finish, your dialog box resembles Figure 1-6.

12. Click OK.

The Generate Signed Bundle or APK dialog box from Figure 1-5 reappears. This time, many of the box's fields are filled in for you. (See Figure 1-7.)

You may see a check box labeled Export Encrypted Key for Enrolling Published Apps in Google Play App Signing. If you do, make sure that the check box is selected. (If you opted to create an APK file rather than an Android App Bundle in Step 3, you don't see that check box.)

TIP

When you export an encrypted key, key's file extension is `.pepk`. To learn what `.pepk` files are all about, see the section entitled "The App Signing page," later in this chapter.

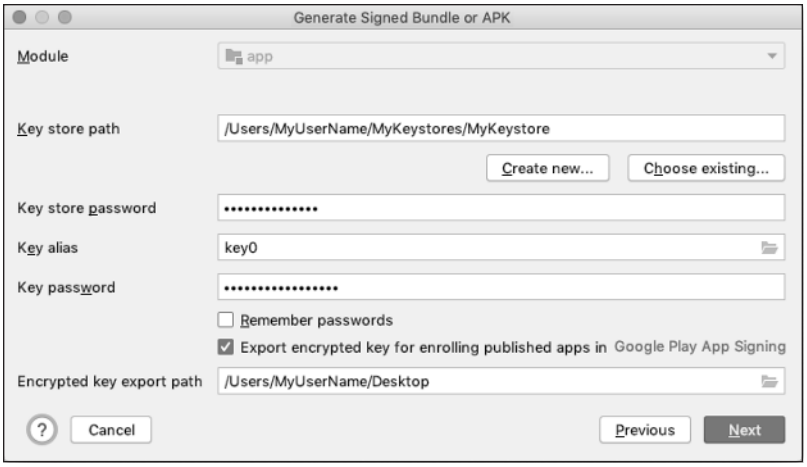


FIGURE 1-7: Filling in the fields of Figure 1-5.

13. Click Next.

One last Generate Signed Bundle or APK page appears. (See Figure 1-8.)

On this final Generate Signed Bundle or APK page, take note of the Destination Folder. Also, be sure to select Release in the Build Variants list. If you have a choice between V1 and V2 signature versions, select both. (To find out what the names V1 and V2 mean, visit `https://developer.android.com/about/versions/nougat/android-7.0.html#apk_signature_v2`.)

And finally . . .

FIGURE 1-8: The penultimate step.

14. Click Finish.

Android Studio offers to open the folder containing your shiny, new AAB or APK file. That's great! Open the folder and stare proudly at your work.

Congratulations! You've created a distributable version of your app and a reusable key store for future updates.

Running a new APK file

If you've created a release build and you want to try running it, don't click the usual Run icon in the toolbar, and don't select Run➪Run 'app'. Without first performing some extra steps, those things won't work. If the file that you've created is an APK file, follow these steps:

1. **Make sure that you're running at least one AVD.**

   If you're not, select Tools➪AVD Manager and get an AVD going.

2. **At the top of Android Studio's Project tool window, the word *Android* is one of several drop-down list items. Change the selection from Android to Project. (See Figure 1-9.)**

   For help finding the Project view, refer to Book 1, Chapter 4.

3. **Expand the Project view tree to find your project's app/release branch.**

   Chances are, the `app/release` folder has a file named app-release.apk. If it doesn't, look elsewhere for the `app-release.apk` file.

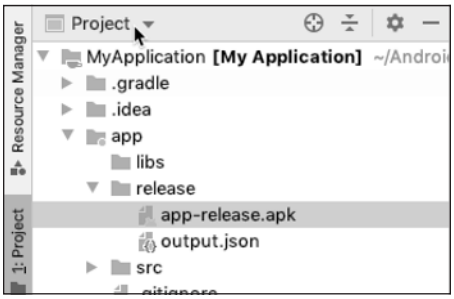


FIGURE 1-9: Finding the APK file.

4. Drag the app–`release.apk` file from the Project view to the AVD screen.

   When you do, Android installs your signed app on the AVD.

5. **Click the app's icon to launch the app on the AVD.**

   Watch your app run!

TIP

If you're comfortable using your operating system's command line, there's a quick way to make sure that your APK file has a digital signature. Look in a subdirectory of `Android/Sdk/build-tools` for a file named `apksigner.jar`. Then run the following command (all on one line that may wrap on its own):

```
java -jar path_to_apksigner.jar verify --print-certs path_to_build.apk
```

The output may be overzealous with its warnings, but you should see information about the signature that you created.

Running the app in a new AAB file

You can't deploy an AAB file to an Android device. Instead, you need a tool that sifts an APK file out of the AAB file and then deploys the APK file to the device. As of May 2020, the appropriate tool runs only from your development computer's command line — the Windows Command Prompt or the macOS Terminal application. The instructions that follow are intentionally sketchy because the details are likely to change over time. Anyway, if you're determined, these instructions can get you started:

1. **Visit `https://github.com/google/bundletool/releases` and download the latest bundletool jar file.**

2. **Collect all the information you need.**

   Table 6-1 lists the things you need to know to use `bundletool`.

The Information You Need	This Chapter's Nickname for the Information
On your computer's file system, the location of:	
The `java` program (`java.exe` or simply `java`)	*JAVA_COMMAND*
The `bundletool` jar file	*BUNDLETOOL_JAR*
Your AAB file	*AAB_FILE*
Your key store file	*KEYSTORE_JKS_FILE*
Your key store's password	*KEYSTORE_PASSWORD*
Your key's alias	*KEY_ALIAS*
Your key's password	*KEY_PASSWORD*
A new name (ending in `.apks`) for the collection of APK files that will be generated when you issue the `bundletool` command	*APKS_FILE*

3. **Launch your computer's command-line application — the `cmd` app in Windows or the Terminal app on a Mac.**

4. **In the command-line window, type the following command (all on one line that wraps a few times on its own):**

```
JAVA_COMMAND -jar BUNDLETOOL_JAR build-apks
  --bundle=AAB_FILE
  --output=APKS_FILE
  --ks=KEYSTORE_JKS_FILE
  --ks-pass=pass:KEYSTORE_PASSWORD
  --ks-key-alias=KEY_ALIAS
  --key-pass=pass:KEY_PASSWORD
```

   If all goes well, the result is a shiny, new `.apks` file. That file encodes some APK files that have been extracted from your AAB file.

   The only remaining task is to deploy an APK file on your Android emulator.

5. **Make sure that one (and only one) AVD is running, and that no physical devices are connected to your computer.**

   Doing so keeps the next command from being a bit more complicated.

6. **In the command-line window, type the following command (all on one line that may wrap on its own):**

```
JAVA_COMMAND -jar BUNDLETOOL_JAR install-apks --apks=APKS_FILE
```

This command analyzes your emulated device, decides which APK (or combination of APKs) to install on that device, and then proceeds to install your app.

7. **On the emulated device, find your app's launch icon and run the app.**

Good work!

TIP

If you get stuck trying to use `bundletool`, visit the online reference page: `https://developer.android.com/studio/command-line/bundletool`.

Another way to build and run an AAB file

If you want to test an AAB file and `bundletool` isn't your cup of tea, you can follow this section's steps. With these steps, you can also fine-tune a build to target specific release versions and specific flavors of your app. You can use one key to sign your app's free version and another key to sign the app's paid version.

Here's what you do:

1. **Create a signing key by following Steps 1 through 12 in the section entitled "Creating the release build."**

Hey! That's most of the section's steps, isn't it?

2. **In the Generate Signed Bundle or APK dialog box, click Cancel.**

Lo and behold! You're back to Android Studio's main window.

3. **Click the Build Variants tool button.**

You can find that button along the left edge of the Android Studio window. (See Figure 1-10.)

Clicking that button brings the Build Variants tool window out of hiding. This tool window has an Active Build Variant drop-down list.

4. **In the Active Build Variable drop-down list, select Release. (Refer to Figure 1-10.)**

Now, when you select Run➪Run 'app', Android Studio will try to build and run your project's release version. The only problem is, Android Studio won't try to run a signed version. Before that can happen, you have a few more steps to follow.

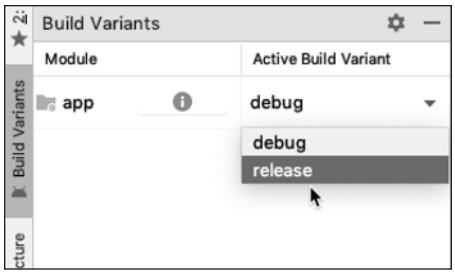


FIGURE 1-10: Selecting a build variant.

5. **In Android Studio's main menu bar, choose File➪Project Structure.**

   The Project Structure dialog box appears. (What else would you expect?)

   You're about to create something called a *signing configuration.* The signing configuration says, "One way to sign a build is to use the key that was created in Step 1."

6. **In the Project Structure dialog box, choose Modules➪Signing Configs.**

   The dialog box now contains two plus sign icons — one below the word Modules and another in the Signing Configs tab. (See Figure 1-11.)

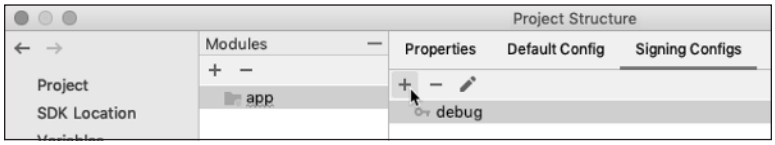


FIGURE 1-11: Creating a signing configuration.

7. **Click the Signing Configs tab's plus sign icon.**

   A new message box requests that you enter a signing config name.

8. **In the message box, type** release **and then click OK.**

WARNING

   Using the name `release` doesn't automatically connect this signing configuration with your project's release build variant. Making that connection comes later in these steps.

9. **In the list near the top of the Signing Configs tab, check to make sure that the release item is selected.**

   If not, select it.

10. **In the body of the Signing Configs tab, fill in the information about the signing key that you created in Step 1.**

    Android Studio wants the path to the key store file, the file's password, a particular key's alias, and that key's password. (See Figure 1-12.)

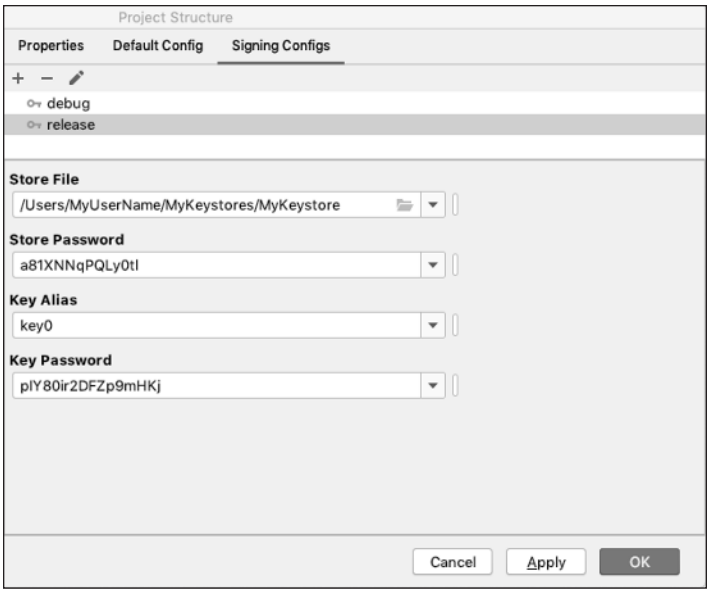


FIGURE 1-12: Don't even think about it! This screenshot's passwords are fake.

11. Click Apply.

At this point, you've created a signing configuration. As a final step, you have to associate that configuration with your project's release build.

12. In the Project Structure dialog box, select Build Variants➪Build Types. (See Figure 1-13.)

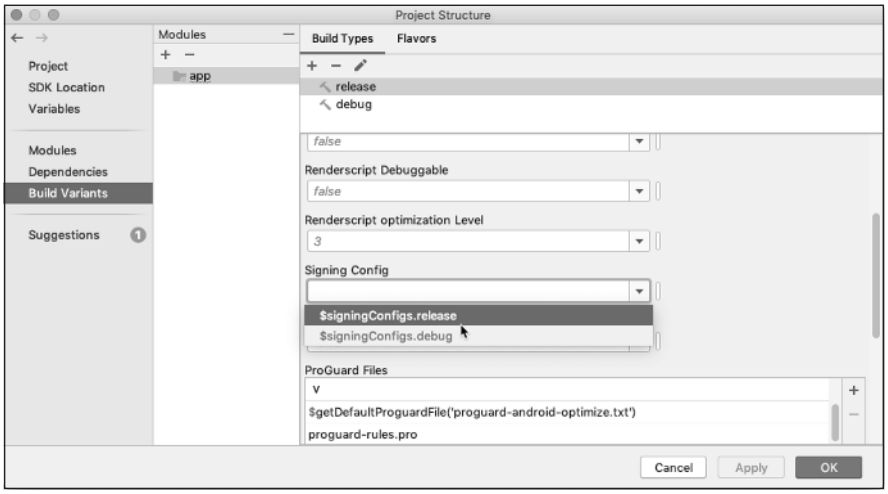


FIGURE 1-13: Connecting a signing configuration to a build.

13. **In the list near the top of the Build Types tab, check to make sure that the release item is selected.**

14. **Look for the Signing Config drop-down list in the Build Types tab. In that list, select SigningConfigs.release. (Refer to Figure 1-13.)**

15. **Click OK.**

    Well, whaddya know? You've returned to Android Studio's main window.

16. **Check the Build Variants tool window to make sure that it's still in release mode.**

17. **In Android Studio's toolbar, click the Run icon.**

    And that's it! Your app runs in the AVD window.

Publishing Your App

To start this section's adventure, visit `https://play.google.com/apps/publish`. Look for a button or link with words like Create Application. Click that button or link and get ready to roll.

REMEMBER

Nothing permanent happens until you click the Rollout button. If the Rollout button makes you nervous, there's also a friendly Save Draft button. So you can pause your work, think about things for a while, and log on again later. For more extreme situations (severe cases of Developer's Remorse), there's an Unpublish link.

TIP

Neither the Rollout nor the Unpublish requests take effect immediately. In particular, a Rollout request triggers a review of your app by the folks at the Play Store. You must wait a few days for their final approval.

If Google's website is anything like its May 2020 version, clicking Create Application takes you to a place with a big navigation bar along the left side. Pages accessible from the navigation bar include App Signing, Store Listing, Content Rating, Pricing & Distribution, Translation Service, and many others. This section describes a few of the pages in detail.

The App Releases page

In your visit to the Google Play Store, the grandest of all events is the uploading of the APK file. To make this happen, select the navigation bar's App Releases item. That's where you find buttons and links for uploading APK or AAB files. In particular, the page offers you a few different upload tracks.

- **Tracks intended for app testing:**
  - **Internal track,** with up to 100 testers. You supply the testers' email addresses. Each email address must be associated with a Google account.
  - **Closed (alpha) track,** with up to 2,000 testers. You supply the testers' email addresses or the names of Google Groups.
  - **Open (beta) track,** with an unlimited number of testers. Anyone with a compatible device can be a tester.
- **Track intended for general release: Production track,** with users instead of testers. Your app is listed on the Google Play Store.

TIP

For more information about each of the tracks, visit `https://support.google.com/googleplay/android-developer/answer/3131213`.

WHAT? MORE TESTING?

When testing your app with friends and relatives, you don't always get reliable results. Sure, your friends are polite, but they might also know something about your app — something that other users won't already know. So your casual acquaintances living on other continents should test your app, too. In fact, you should test so much that you'd have trouble mailing your APK file to all your testers.

That's why the Play Store allows you to publish an app with invitation-only access. After uploading an app to the internal or closed track, you provide a list of testers — a relatively small number of people who can download the app from the Play Store. When these people use your app, the Play Store keeps track of crashes, application not responding (ANR) occurrences, and other unwanted events. The Google Play Console can filter its reporting of these events by Android version and device type.

If you want more testing, you can use Android's built-in testing tools. Book 3, Chapter 2 has the details. Other popular testing frameworks include Robotium, UI Animator, Selendroid, Testdroid, and Dynatrace.

And, if you want even more testing, you can enroll your app with a *testing farm* — a service that automates the testing and tests your app on many different devices simultaneously. Amazon Web Services (AWS) has its own full-featured testing farm.

For each of these tracks, you supply an APK or AAB upload, a name for this release, and an explanation of what's new in this release. When you follow the steps in the "Creating a Publishable File" section, earlier in this chapter, Android Studio creates a file named `app-release.apk` or `app-release.aab` and puts the file in your project's `app/release` subdirectory. So, when the big upload moment comes, drag that `app-release` file to the Drop Your File Here box, or click the Browse button and use your File Explorer or Finder to navigate to this `app-release` file.

The Store Listing page

On the Google Play Console's Store Listing page, you describe your app to potential users. You answer many questions and upload several files. This section lists several of the items on the page.

TIP

As you scroll down among these items, a Save Draft button stays at the bottom of your web browser's screen. Click the Save Draft button frequently to keep from losing any work that you've done.

- **Title of your app:** A great title can jump-start an app's popularity, so make your title something snappy.
- **Short description:** You may enter up to 80 characters.
- **Full description:** You may enter up to 4,000 characters.
- **Graphic assets:** Refer to the section entitled "Preparing Graphic Assets for the Play Store,"earlier in this chapter.
- **Your app's category:** Is it a game? If so, what type of game? (Choose from Action, Adventure, Card, Puzzle, Racing, Role Playing, and many other types.) If it's not a game, what type of app is it? (Choose from Business, Communication, Education, Finance, Health, and a bunch of other types.)
- **Tags to associate with your app:** For some suggestions, click the Save Draft button and then click the page's Manage Tags button.
- **Content rating:** Complete a questionnaire to determine your app's content rating under several standards (international and otherwise). Does your app involve violence, sexuality, potentially offensive language, or references to illegal drugs? Does your app involve the exchange of any personal information?
- **Your contact details:** You must supply an email address, and users have access to this address. Optionally, you can supply a website and a phone number.

- **Language translations:** You specify the default language for your listing on the Play Store. You can provide translations in other languages, or have Google Translate furnish its own guesses. (This can accidentally lead to some fairly amusing results.) You can also purchase translations straight from the Google Play Console. For a very simple app, translation costs as little as $5 per target language.

The App Signing page

The "Creating a Publishable File" section, earlier in this chapter, draws a sharp distinction between APK files and AAB files. An APK file represents an installable app, and an AAB file represents several APK files. You can upload either kind of file to the Google Play Store. If you upload an APK file, the Play Store downloads that file to users' devices. If you upload an AAB file, the Play Store manages the creation of your app's actual APK files and downloads those files to users' devices.

Whatever you send, the file must be digitally signed. So says the same "Creating a Publishable File" section. But if you upload an AAB file and the Play Store creates all the APK files, who signs those APK files?

The App Signing page presents you with the option of letting the Play Store add your signature to each of the APK files. To make this happen, you can upload the `.pepk` file that you create in Step 12 of the section entitled "Creating the release build," earlier in this chapter. Yes, you're handing your very own digital signature over to Google. It's a bit like telling someone your banking PIN or your mother's maiden name. But in return, you're reducing the complexity of delivering a sleek, customized app to each of your users. If you want the benefits of Dynamic Delivery, you have to let Google manage your app signing key.

Other pages

The Google Play Console has other pages, too, and here are some of them:

- **Device catalog:** On the Device catalog page, you specify which devices are explicitly supported for running your app, and any devices that are explicitly excluded. In May 2020, the Device catalog lists over 16,000 devices.
- **App content:** On the App content page, you describe your app's privacy policy. You specify the app's target age group. You tell Google if your app contains ads.

- **Pricing & distribution:** In which countries can your app be distributed? You can pick and choose from more than 150 countries.

  Will you charge for your app, or will it be free? For many developers, this question requires some serious thinking so the next chapter delves deeply into the alternatives. For now, the only thing you have to know is that changing your mind is a one-way street. You can change an app from being paid to free, but you can't change an app from being free to being paid. Of course, you can publish a new app that's very much like the original free app, except that the new app is a paid app.

- **In-app products:** Will you sell products or offer subscriptions through your app? For a discussion about this possibility, see the next chapter.

- **Services & APIs:** A *back-end service* is computing done on the cloud. And why would your app need to deal with the cloud? Maybe your game has a leaderboard, and you want to compare the scores of users around the world. To make this comparison, you need information that's stored outside the user's own device. This function, and many other functions that apps perform, require access to a server.

  Maybe you want to send data to your users using Google Cloud Messaging. Maybe you want Google's search engine to look for content within your app. Maybe you want Google to handle your in-app billing. All these things involve back-end services.

  Licensing is another very commonly used back-end service. Licensing protects your app from illegal use. For more info, see the "About app licensing" sidebar.

ABOUT APP LICENSING

If you license your app, no device can run your app unless the device checks in with a server. The server ensures that the device has permission to run your app. Here are some scenarios for an app (free or paid), with and without licensing:

- **Best case scenario with licensing:** A user buys your app and copies the `.apk` file to another user's device. The other user hasn't paid for your app. The other user tries to run the app but can't run it because of the licensing restrictions.
- **Worst case scenario without licensing:** A user buys your app and copies the `.apk` file to a file-sharing website. People download and install your `.apk` file and run the code for free. (Ooo! That's bad!)

- **Worst case scenario with licensing:** A user buys your app, cracks the licensing, and copies the `.apk` file to a file-sharing website. People download and install the cracked version of your `.apk` file and run the code for free. (That's bad, too.)
- **Best case scenario without licensing:** No one ever tries to steal your app. Or, if someone steals your app, the additional distribution of your app works to your advantage.

All things considered, it's best to do licensing with any paid app. Licensing is also a good precaution with a free app (to help you maintain ownership of the app's concept).

To enable licensing in your app, you must install the *Google Play Licensing Library* (also known as *LVL* — the *Licensing Verification Library*) using the Android SDK Manager. You must add that library to your app's project. You must obtain the app's licensing key (a sequence of about 400 gibberish characters) from the Google Play Console and add the key to your main activity. You must add additional code in your app to check a device's license and to respond (based on the result of the check). The additional code implements one of three possible policies:

- **Strict policy:** Whenever the user tries to launch your app, the device asks the Google Play server for approval to run the app. If the user tries to launch your app when the device has no connectivity, the user is out of luck. Life's tough.
- **Server-managed policy:** The user's device stores a copy of the user's license. The device uses the copy when network connectivity is unavailable. The license is obfuscated (so it's tamper-resistant), and the license keeps track of trial periods, expiration dates, and other stuff. This is the default policy, and it's the policy that Google highly recommends.
- **Custom policy:** Create your own policy or modify either of the preceding policies by adding Kotlin code to your app.

This chapter's "What? More testing?" sidebar describes how you can use the Google Play Console to register testers for your soon-to-be-published app. You can also name some special testers for your app's licensing scheme. Your testers attempt to run the app when (as they know darn well) they shouldn't get permission. The Google Play Console keeps track of successes and failures so that you can find out whether your licensing scheme works correctly.

For all the details about the licensing of apps, visit `http://developer.android.com/google/play/licensing/`.

Leave No Stone Unturned

Do lots of homework before you publish on Google's Play Store by checking out these resources:

- Visit `https://developer.android.com/studio/publish/preparing` for a comprehensive list of required steps.
- Visit `https://developer.android.com/docs/quality-guidelines/core-app-quality` for an exhaustive list of criteria that your app must satisfy.
- Visit `https://developer.android.com/distribute/play-policies` for a glimpse at the Play Store's upcoming changes.

And finally . . .

- Visit `https://developer.android.com/distribute/best-practices/launch/launch-checklist` to make sure that you're ready for your app's big rollout.

Publishing Elsewhere

Google's Play Store isn't the only game in town. (It's a very important game, but it's not the only game.) You can also publish on the Amazon Appstore, on several independent websites, or on your own website.

The Amazon Appstore

This section has a few notes about publishing on Amazon Appstore. The steps for publishing with Amazon resemble the steps for publishing with Google. Publishing on Amazon's Appstore is less expensive than publishing on Google's Play Store (if you call not paying a one-time $25 developer fee "less expensive"). The Amazon Developer Portal pages look a bit different from the Google Play Console pages, but the basic ideas are almost all the same. Amazon's focus is primarily on tablets and Fire TV, but the store lists apps for phones as well.

Digital rights management

When you publish an app, you have the option of applying Amazon's *digital rights management (DRM)* to your app. This is the Amazon equivalent of Google's Licensing Verification Library. Like the DRM for Amazon Kindle books, the Appstore's DRM electronically grants permission to run each app on a device-by-device basis. And like any other scheme for managing users' privileges, the Appstore's DRM inspires vast waves of controversy in blogs and online forums.

Amazon doesn't publish gobs of information about the workings of its DRM scheme. But one thing is clear: Without digital rights management, any user can run your application. With DRM, a user can replace his or her device and, by logging on to the new device as the same Amazon user, have access to his or her rightfully purchased apps. Users can run paid apps without having an Internet connection because when a user buys an app, the user receives an offline token for that app. There's no doubt about it: When you publish a paid app, DRM is the way to go.

Amazon answers some questions about DRM in a blog post with the following unwieldy URL: `https://developer.amazon.com/public/community/post/Tx16GPJPAW8IKLC/Amazon-Appstore-Digital-Rights-Management-simplifies-life-for-developers-and-cus`. For reference, you can also check `https://developer.amazon.com/docs/app-submission/understanding-submission.html#about_drm`.

A few other differences

Amazon's graphic assets requirements are different from Google's. The image sizes are different, and the number of images that you must submit are different. Fortunately, when you're submitting your app and you encounter these differences, you can save your Developer Portal work, set the Developer Portal aside, and create more images. You can find Amazon's image requirements at `https://developer.amazon.com/docs/app-submission/asset-guidelines.html`.

Amazon's app-signing procedure is a bit different from Google's. By default, Amazon applies its own certificate to each app published on the Amazon Appstore. The certificate is unique to your account. But otherwise, it's a boilerplate certificate.

Then there's the optional SKU. When you submit an app, Amazon's Developer Portal lets you supply a SKU. The acronym *SKU* stands for *Stock Keeping Unit*. It's a way that you, the seller, keep track of each kind of thing that you sell. For example, imagine that you sell only two kinds of shirts: green ones and blue ones. When the customer buys a shirt, the only thing the customer decides is whether to buy a green shirt or a blue shirt. Then you might assign SKU number 000001 to your green shirts and 000002 to your blue shirts. It's up to you. Instead of 000001 and 000002, you might assign 23987823 to your green shirts and 9272 to your blue shirts. Anyway, when you submit an app, you can create your own SKU number for that app.

TIP

For a nice summary of the Amazon Appstore's guidelines and recommendations, visit `https://developer.amazon.com/docs/app-submission/faq-submission.html`.

Other venues

Some of the lesser known Android app websites offer apps that consumers can't get through Google Play or Amazon Appstore. In addition, many sites offer reviews,

ratings, and links to the Google and Amazon stores. You can search for these sites yourself, or you can find lists of such sites. To get started, visit Android Authority (`https://www.androidauthority.com/best-app-stores-936652/`) and Joy of Android (`http://joyofandroid.com/android-app-store-alternatives`).

Sites differ from one another in several ways. Does the site specialize in any particular kind of app? Is the site linked to a particular brand of phone? Are the site's reviews more informative than those of other sites? Does the site vet its apps? Is the site's interface easy to use? And here's a big one: How does a user install one of the site's apps?

Before there were app stores, there were websites with files that you could download and install. Installing meant clicking an icon, issuing some commands, or doing other things. That model is still alive in the desktop/laptop world. But for mobile devices, for which installation procedures can be cumbersome, the one-stop app store model dominates.

Some Android app sites still use the download-and-install-it-yourself model. For a patient (or a truly determined) consumer, the install-it-yourself model is okay. But most mobile-device owners are accustomed to the one-step app store installation process. Besides, some mobile service providers put up roadblocks to keep users from installing unrecognized apps. On many phones, the user has to dig into the Settings screen to enable installation of apps from unknown sources. On some phones, that Settings option is either hidden or unavailable.

For users who know and trust your work, there's always one oft-forgotten alternative. Post a link to your app's APK file on your own website. Invite users to visit the page with their mobile phones' browsers and download the APK file. After downloading the file, the user can click the download notification to have Android install your app.

Here's one thing to consider. To preload the Google Play Store on a device, the device manufacturer must obtain Google's approval. The approval comes in the form of a certification, which asserts that the device and its software meet certain compatibility standards. Because of the expense in meeting the standards or in obtaining certification, some device manufacturers don't bother to apply. Their devices don't have the Play Store preloaded. Many of these devices have alternative app stores preloaded on their home screens but the Play Store app is conspicuously absent. Some users find workarounds and manage to install the Play Store app, but many users rely on apps from other sources. To reach these users, you have to find alternative publishing routes. (The percentage of users who live in this uncertified world could be very small, or it could be quite large. The stats aren't readily available. One way or another, these people deserve to have access to your app.)