Android studio build apk

You can perform all available build tasks for your Android project using the Gradle Wrapper command line tool. It is available as a batch file for Windows (gradlew.bat) and a shell script for Linux and Mac (gradlew.sh), and is accessible from the root of each project you create with Android Studio. To perform a task with the wrapper, use one of the following commands from a Terminal window (from Android Studio, select View &gt; Windows Tool &gt; Terminal): In Windows: gradlew task name No Mac or Linux: ./gradlew task-name To see a list of all the tasks available for your project, perform tasks: gradlew tasks The rest of this page describes the fundamentals for building and running your application with the Gradlele wrapper. For more information about setting up your Android build, see Set up your build. If you prefer to use the Android Studio tools instead of the command-line tools, see Build and run your app. About build types By default, there are two build types available for each Android app: one to debug your app—the debug build—and one to release your app to users—the release build. The resulting output of each build must be signed with a certificate before you can deploy your application to a device. The debug build is automatically signed with a debug key provided by the SDK tools (it's unsafe and you can't publish with it to the Google Play Store), and the release build must be signed with its own private key. If you want to build your app for launch, it's important that you also sign your app with the appropriate subscription key. If you're just getting started, however, you can quickly run your apps on an emulator or a connected device by building a debugging APK. You can also define a custom build type in your build.gradle file and configure it to be signed as a debug build, including the true debugable. For more information, see Set up build variants. Build and deploy an APK Although building an application package is the best way to package your application and send it to the Play Console, building an APK is best suited for when you want to quickly test a debug build or share your application as an deployable artifact with others. Build a debug APK For immediate application testing and debugging, you can build a debug APK. The debugging APK is signed with a debug key provided by the SDK tools and allows debugging through adb. To build a debugging APK, open a command line and navigate to the root of the project directory. To start a build of invoke the assembleDebug task: gradlew assembleDebug That creates an APK called module_name-debugging.apk in project_name/module_name/build/outputs/apk/. The file is already signed with the debug key and aligned with zipalign, so you can install it immediately on a device. Or to build the APK and install it immediately on a running emulator or connected device, instead of invoking the Debug installation: gradlew installDebug The Debug part in the task names above is just a camel case version of the build variant name, so it can be replaced by any build build variant you want to mount or install. For example, if you have a demo product flavor, then you can build the debug version with the assembleDemoDebug task. To see all available build and installation tasks for each variant (including uninstall tasks), run the task. See also the section on how to run your app in the emulator and run your app on a device. Build a release APK When you're ready to launch and distribute your app, you should build a signed launch APK with your private key. For more information, go to the section on how to sign your app from the command line. Deploy your app to the emulator To use the Android Emulator, you must create an Android Virtual Device (AVD) using Android Studio. Once you have an AVD, launch the Android Emulator and install your app as follows: On a command line, navigate to android_sdk/tools/ and start the emulator specifying your AVD: emulator -avd avd_name If you are unsure of the AVD name, run emulators -list-avds. You can now install your application using one of the Gradle installation tasks mentioned in the section on how to build a debug APK or the adb tool. If the APK is constructed using a developer preview SDK (if the targetSdkVersion is a letter instead of a number), you must include the -t option with the installation command to install a test APK. adb install path/your_app.apk All APKs you build are saved in project_name/module_name/build/outputs/apk/. For more information, see Run apps on the Android Emulator. Deploy your app to a physical device Before you can run your app on a device, you must enable USB debugging on your device. You can find the option in Settings &gt; developer options. Note: On Android 4.2 and newer, developer options are hidden by default. To make it available, go to Settings &gt; About Phone and tap Build Number seven times. Return to the previous screen to find Developer options. Once your device is configured and connected via USB, you can install your application using the gradle installation tasks mentioned in the section on how to build a debug APK or adb tool: adb -d install path/to/your_app.apk All APKs you build are saved in project_name/module_name/build/outputs/apk/. For more information, see Running apps on a hardware device. Create an App Package Android App Packs include all the compiled codes and features of your app, but defer aPK generation and sign up with Google Play. Unlike an APK, you cannot deploy an application package directly to a device. So if you want to test or quickly an APK with someone else, you should instead build an APK. The easiest way to build an app package is by using Android Studio. However, if you need to build an application package from the command line, you can do so using gradle or bundletool, as described in the sections below. Build an application package with Gradle If you prefer to generate an application package from the command line, run the BundleVariant Gradle task in your application's base module. For example, the following command builds a package of application applications the debug version of the base module: ./gradlew :base:bundleDebug If you want to build a signed package for upload to the Play Console, you need to first configure the build.gradle file of the base module with the subscription information of your application. To learn more, go to the section on how to set up Gradle to subscribe to your app. You can then, for example, build the release version of your application, and Gradle automatically generates an application package and signs it with the subscription information that you provide in the build.gradle file. If you want to sign an application package as a separate step, you can use jarsigner to sign your application package from the command line. Note: You cannot use the apksigner to sign your app package. Building an app package using bundletool bundletool is a command-line tool that Android Studio, the Android Gradle plugin, and Google Play use to convert your app's compiled code and features into app packages and generate deployable APKs from those packages. So while it's useful to test app packages with package tools and recreate locally as Google Play generates APKs, you typically don't need to invoke bundletool to build your own app package—you should instead use Android Studio or Gradle tasks, as described in previous sections. However, if you don't want to use Android Studio or Gradle tasks to create packages—for example, if you use a custom build tool—you can use the command-line bundletool to build an application package from precompiled codes and features. If you haven't already, download the package from the GitHub repository. This section describes how to package the compiled code and resources of your application and how to use the command-line bundletool to convert them to an Android App Pack. Generating the manifest and resources in the proto bundletool format requires certain information about your application's design, such as the manifest and application features, to be in the Google Protocol Buffer format—which is also known as protobuf and uses the *.pb file extension. Protobufs provide a platform-neutral, platform-neutral, extensible mechanism for serializing structured data—it's similar to XML, but smaller, faster, and simpler. Download AAPT2 You can generate your app's manifest file and protobuf-format resource table using the latest version of AAPT2 from the Google Maven repository. Note: Do not use the version of AAPT2 that is included in the Android build tool pack—this version of AAPT2 does not support package tools. To download AAPT2 from google's Maven repository, proceed as follows: Browse to com.android.tools.build &gt; aapt2 in the repository index. the name of the latest version of AAPT2. Enter the copied version name in the following URL and specify your target operating system: windows | linux | osx].jar For example, to download version 3.2.0-alpha18-4804415 for Windows, you would use: Navigate to the URL in a browser — AAPT2 should start downloading soon. Unpack the JAR file you just downloaded. Use AAPT2 to compile your application resources with the following command: aapt2 compile project_root/module_root/src/main/res/drawable/Image1.png project_root/module_root/src/main/res/drawable/Image2.png -o compiled_resources/ Note: Although you can pass resource directories to AAPT2 using the flag -dir-, doing this i recompisto all the files in the directory, regardless of how many files actually changed. During the link phase, where AAPT2 links its various compiled resources into a single APK, instruct AAPT2 to convert your application manifest and resources compiled into protobuf format, including the flag --proto-format, as shown below: aapt2 link --proto-format -the output.apk\-I android_sdk/platforms/android_version/android.jar--manifest project_root/module_root/src/main/AndroidManifest.xml\-R compiled_resources/*.flat\ --auto-add-overlay Note: Alternatively, when specifying resources compiled with the -R flag, you can specify a text file that includes the absolute path for each of your compiled resources—with each path separated by a single space. You can then pass this text file to AAPT2 as follows: link aapt2 ... -R @compiled_resources.txt. You can then extract content from the output APK, such as AndroidManifest from your app.xml, resources.pb, and other resource files—now in protobuf format. You need these files when preparing the input package that the tool requires to build your application package, as described in the following section. Pack code and precompiled resources Before you use bundletool to generate an application package for your application, you must first provide ZIP files that contain compiled code and resources for a given application module. The content and organization of the ZIP file of each module is very similar to that of the Android App Bundle format. For example, you should create a base file.zip for the base module of your application, and organize its contents as follows: File description manifest or directory/AndroidManifest.xml The module manifest in protobuf format. dex/... A directory with one or more DEX files compiled from your application. These files should be named as follows: classes.dex, classes2.dex, classes3.dex, etc. res/... Contains the features of the protobuf format module for all device configurations. Subdirectories and files should be organized similar to those of a typical APK. root/..., active/..., and lib/... These directories are identical to those described in the section on the Android app package format. resources.pb Table of your application's features in protobuf format. After you prepare the ZIP file for each module of your application, you pass them to package tool to build your application package, as described in the following section. Build your application package using the bundletool To build your application package, you use the bundletool build-bundle command, as shown below: bundletool build-bundle --modules=base.zip --output=mybundle.aab Note: If you you to publish the application package, you need to sign it using jarsigner. You cannot use apksigner to sign your app package. The following table describes flags for the build-bundle command in more detail: Flag description --modules=path to base.zip, path-to-module2.zip,path to module3.zip Specifies the list of module ZIP files that the zip tool should use to build your application package. --output=path-to-output.aab Specifies the path and file name for the output file *.aab. --config=path-to-BundleConfig.json Specifies the path to an optional configuration file that you can use to customize the build process. To learn more, see the section on customizing downstream APK generation. --metadata-file=target-bundle-path:local-file-path Instructs bundletool to package an optional metadata file within your application package. You can use this file to include data, such as ProGuard mappings or the full list of your app's DEX files, which may be useful for other steps in your tool chain or in an app store. the target package path specifies a path relative to the root of the application package where you would like the metadata file to be packed, and the local file path specifies the path to the local metadata file itself. Customize packages of downstream APK generation applications include a BundleConfig.pb file that provides metadata that app stores, such as Google Play, require when generating package APKs. Although bundletool creates this file for you, you can configure some aspects of metadata in a BundleConfig.json file and pass it to the bundletool build-bundle command—bundletool later converts and merges this file with the protobuf version included in each application package. Note: To learn more about how JSON maps the protobuf format, read JSON Mapping. However, this information is more advanced than necessary for this page. For example, you can control which categories of configuration APKs to enable or disable. The following example of a BundleConfig.json file disables configuration APKs that each targets a different language (that is, resources for all languages are included in their respective base or resource APKs): { optimizations: { splitsConfig: { splitDimension: {{ value: LANGUAGE, negate: true }} } } } in your bundleconfig.json file, you can also specify which file types to leave uncompressed when packing APKs using glob patterns, as follows: { compression: { UncompressedGlob: [res/raw/**, assets/**.uncompressed] } } } Keep in mind, by default, bundletool does not compress your app's native libraries (on Android 6.0 or higher) and resource table (resources.arsc). For a complete description of what you can set up in your inspect the config.proto bundletool file, which is written using proto3 syntax. Deploy your app from an app package If you built and signed an app package, use the bundletool to generate APKs and deploy them to a device. Subscribe to your app on the command line You don't need Android Studio to subscribe to your app. You can subscribe to your app from the line, using apksigner for APKs or jarsigner for application packages, or configure Gradle to sign it during compilation. Either way, you need to first generate a private key using keytool, as shown below: keytool -genkey -v -keystore my-release-key.jks -keyalg RSA -keysize 2048 -validity 10000 -alias my-alias The example above prompts passwords for key storage and key, and for the Distinguished Name fields for your key. It then generates the key store as a file called my-release-key.jks, saving it to the current directory (you can move it wherever you want). The keystore contains a single key valid for 10,000 days. You can now sign with you APK or app bundle manually, or configure Gradle to sign your app during the build process, as described in the sections below. Subscribe to your application manually from the command line If you want to sign a command-line application package, you can use jarsigner. If, instead, you want to sign an APK, you need to use zipalign and apksigner as described below. Open a command line —from Android Studio, select View &gt; Windows &gt; Terminal Tool—and navigate to the directory where your unsigned APK is located. Align the unsigned APK using zipalign: zipalign -v -p 4 my-app-unigned.apk my-app-un-ign-aligned.apk zipalign ensures that all uncompressed data starts with a specific byte alignment relative to the beginning of the file, which can reduce the amount of RAM consumed by an application. Sign your APK with your private key using apksigner: apksigner sign --ks my-release-key.jks --out my-app-release.apk my-app-unigned-aligned.apk This example produces the signed APK in my app-release.apk after signing it with a private and certified key 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 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. Make sure your APK is signed: apksigner check my version of the app.apk

Configure Gradle to sign your app Open module-level build.gradle file and add the Configs signature block {} with entries for storeFile, storePassword, keyAlias, and keyPassword, and then pass that object to the Config signature property in the build type. For example: android { ... defaultConfig { ... } signatureConfigs { release { // You need to specify an absolute path or include the file // keystore in the same directory as the file build.gradle. file (my-release-key.jks) storePassword password keyAlias my-alias keyPassword password } } buildTypes { release { signingConfig signingConfigs.release ... } } Note: In this case, the key store and key password are visible directly in the build.gradle file. To improve security, you must remove the signature signature from your build file. Now, when you build your application by invoking a Gradle task, Gradle signs your application (and runs zipalign) for you. Additionally, because you configured the version build with your signature key, the install task is available for this type of build. So you can build, align, sign, and install the launch APK on an emulator or device, all with the installRelease task. An app signed with your private key is ready for distribution, but you should first read more about publishing your app and reviewing the Google Play release checklist. List.