82. An Overview of Gradle in Android Studio

Up until this point it has, for the most part, been taken for granted that Android Studio will take the necessary steps to compile and run the application projects that have been created. Android Studio has been achieving this in the background using a system known as *Gradle*.

It is now time to look at how Gradle is used to compile and package together the various elements of an application project and to begin exploring how to configure this system when more advanced requirements are needed in terms of building projects in Android Studio.

82.1 An Overview of Gradle

Gradle is an automated build toolkit that allows the way in which projects are built to be configured and managed through a set of build configuration files. This includes defining how a project is to be built, what dependencies need to be fulfilled for the project to build successfully and what the end result (or results) of the build process should be.

The strength of Gradle lies in the flexibility that it provides to the developer. The Gradle system is a self-contained, command-line based environment that can be integrated into other environments through the use of plug-ins. In the case of Android Studio, Gradle integration is provided through the appropriately named Android Studio Plug-in.

Although the Android Studio Plug-in allows Gradle tasks to be initiated and managed from within Android Studio, the Gradle command-line wrapper can still be used to build Android Studio based projects, including on systems on which Android Studio is not installed.

The configuration rules to build a project are declared in Gradle build files and scripts based on the Groovy programming language.

82.2 Gradle and Android Studio

Gradle brings a number of powerful features to building Android application projects. Some of the key features are as follows:

82.2.1Sensible Defaults

Gradle implements a concept referred to as *convention over configuration*. This simply means that Gradle has a pre-defined set of sensible default configuration settings that will be used unless they are overridden by settings in the build files. This means that builds can be performed with the minimum of configuration required by the developer. Changes to the build files are only needed when the default configuration does not meet your build needs.

82.2.2Dependencies

Another key area of Gradle functionality is that of dependencies. Consider, for example, a module within an Android Studio project which triggers an intent to load another module in the project. The first module has, in effect, a dependency on the second module since the application will fail to build if the second module cannot be located and launched at runtime. This dependency can be declared in the Gradle build file for the first module so that the second module is included in the application build, or an error flagged in the event the second module cannot be found or built. Other examples of dependencies are libraries and JAR files on which the project depends in order to compile and run.

Gradle dependencies can be categorized as *local* or *remote.* A local dependency references an item that is present on the local file system of the computer system on which the build is being performed. A remote dependency refers to an item that is present on a remote server (typically referred to as a *repository*).

Remote dependencies are handled for Android Studio projects using another project management tool named *Maven*. If a remote dependency is declared in a Gradle build file using Maven syntax then the dependency will be downloaded automatically from the designated repository and included in the build process. The following dependency declaration, for example, causes the AppCompat library to be added to the project from the Google repository:

```
implementation 'com.android.support:appcompat-v7:26.0.2'
```

82.2.3Build Variants

In addition to dependencies, Gradle also provides *build variant* support for Android Studio projects. This allows multiple variations of an application to be built from a single project. Android runs on many different devices encompassing a range of processor types and screen sizes. In order to target as wide a range of device types and sizes as possible it will often be necessary to build a number of different variants of an application (for example, one with a user interface for phones and another for tablet sized screens). Through the use of Gradle, this is now possible in Android Studio.

82.2.4Manifest Entries

Each Android Studio project has associated with it an AndroidManifest.xml file containing configuration details about the application. A number of manifest entries can be specified in Gradle build files which are then auto-generated into the manifest file when the project is built. This capability is complementary to the build variants feature, allowing elements such as the application version number, application ID and SDK version information to be configured differently for each build variant.

82.2.5APK Signing

The chapter entitled *"Signing and Preparing an Android Application for Release"* covered the creation of a signed release APK file using the Android Studio environment. It is also possible to include the signing information entered through the Android Studio user interface within a Gradle build file so that signed APK files can be generated from the command-line.

82.2.6ProGuard Support

ProGuard is a tool included with Android Studio that optimizes, shrinks and obfuscates Java byte code to make it more efficient and harder to reverse engineer (the method by which the logic of an application can be identified by others through analysis of the compiled Java byte code). The Gradle build files provide the ability to control whether or not ProGuard is run on your application when it is built.

82.3 The Top-level Gradle Build File

A completed Android Studio project contains everything needed to build an Android application and consists of modules, libraries, manifest files and Gradle build files.

Each project contains one top-level Gradle build file. This file is listed as *build.gradle (Project: <project name>)* and can be found in the project tool window as highlighted in Figure 82-1:

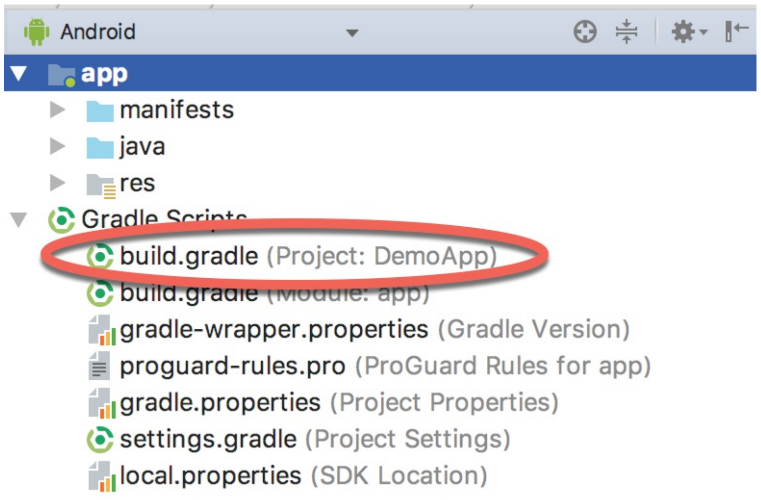
Android
app
manifests
java
res
Gradle Scripts
build.gradle (Project: DemoApp)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

Figure 82-1

By default, the contents of the top level Gradle build file read as follows:

```
// Top-
level build file where you can add configuration options common to al
projects/modules.

buildscript {

    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0'


        // NOTE: Do not place your application dependencies here; the
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

As it stands all the file does is declare that remote libraries are to be obtained using the jcenter repository and that builds are dependent on the Android plugin for Gradle. In most situations it is not necessary to make any changes to this build file.

82.4 Module Level Gradle Build Files

An Android Studio application project is made up of one or more modules. Take, for example, a hypothetical application project named GradleDemo which contains two modules named Module1 and Module2 respectively. In this scenario, each of the modules will require its own Gradle build file. In terms of the project structure, these would be located as follows:

- Module1/build.gradle
- Module2/build.gradle

By default, the Module1 build.gradle file would resemble that of the following listing:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 26
    buildToolsVersion "26.0.0"
    defaultConfig {
        applicationId "com.ebookfrenzy.module1"
        minSdkVersion 19
        targetSdkVersion 26
        versionCode 3
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.Androi
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-
android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.0.2'
    implementation 'com.android.support.constraint:constraint-
layout:1.0.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.0'
    androidTestImplementation 'com.android.support.test.espresso:espre
core:3.0.0'
}
```

As is evident from the file content, the build file begins by declaring the use of the Gradle Android application plug-in:

```
apply plugin: 'com.android.application'
```

The *android* section of the file then states the version of both the SDK and the Android Build Tools that are to be used when building Module1.

```
android {
    compileSdkVersion 26
    buildToolsVersion "26.0.0"
```

The items declared in the defaultConfig section define elements that are to be generated into the module's *AndroidManifest.xml* file during the build. These settings, which may be modified in the build file, are taken from the settings entered within Android Studio when the module was first created:

```
defaultConfig {
    applicationId "com.ebookfrenzy.module1"
    minSdkVersion 19
    targetSdkVersion 26
    versionCode 1
    versionName "1.0"
}
```

The buildTypes section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```
buildTypes {
    release {
        runProguard false
        proguardFiles getDefaultProguardFile('proguard-
android.txt'),
                      'proguard-rules.pro'
    }
}
```

As currently configured, ProGuard will not be run when Module1 is built. To enable ProGuard, the *runProguard* entry needs to be changed from *false* to *true*. The *proguard-rules.pro* file can be found in the module directory of the project. Changes made to this file override the default settings in the *proguard-android.txt* file which is located on the Android SDK installation directory under *sdk/tools/proguard*.

Since no debug buildType is declared in this file, the defaults will be used (built without ProGuard, signed with a debug key and with debug symbols enabled).

An additional section, entitled *productFlavors* may also be included in the module build file to enable multiple build variants to be created.

Finally, the dependencies section lists any local and remote dependencies on which the module is dependent. The first dependency reads as follows:

```
implementation fileTree(dir: 'libs', include: ['*.jar'])
```

This is a standard line that tells the Gradle system that any JAR file located in the module's lib sub-directory is to be included in the project build. If, for example, a JAR file named myclasses.jar was present in the GradleDemo/Module1/lib folder of the project, that JAR file would be treated as a module dependency and included in the build process.

The last dependency lines in the above example file designate that the Android Support and Design libraries need to be included from the Android Repository:

```
implementation 'com.android.support:appcompat-v7:26.0.0'
implementation 'com.android.support:design:26.0.0'
```

Note that the dependency declaration can include version numbers to indicate which version of the library should be included.

82.5 Configuring Signing Settings in the Build File

The *"Signing and Preparing an Android Application for Release"* chapter of this book covered the steps involved in setting up keys and generating a signed release APK file using the Android Studio user interface. These settings may also be declared within a *signingSettings* section of the build.gradle file. For example:

```
apply plugin: 'android'

android {
    compileSdkVersion 26
    buildToolsVersion "26.0.0"

    defaultConfig {
        applicationId "com.ebookfrenzy.gradledemo.module1"
        minSdkVersion 19
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
    }
    signingConfigs {
        release {
            storeFile file("keystore.release")
            storePassword "your keystore password here"
            keyAlias "your key alias here"
            keyPassword "your key password here"
        }
    }
    buildTypes {
.
.
.
}
```

The above example embeds the key password information directly into the build file. Alternatives to this approach are to extract these values from system environment variables:

```
signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.getenv("KEYSTOREPASSWD")
        keyAlias "your key alias here"
        keyPassword System.getenv("KEYPASSWD")
    }
}
```

Yet another approach is to configure the build file so that Gradle prompts for the passwords to be entered during the build process:

```
signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.console().readLine
                ("\nEnter Keystore password: ")
        keyAlias "your key alias here"
        keyPassword System.console().readLIne("\nEnter Key password: '
    }
}
```

82.6 Running Gradle Tasks from the Command-line

Each Android Studio project contains a Gradle wrapper tool for the purpose of allowing Gradle tasks to be invoked from the command line. This tool is located in the root directory of each project folder. While this wrapper is executable on Windows systems, it needs to have execute permission enabled on Linux and macOS before it can be used. To enable execute permission, open a terminal window, change directory to the project folder for which the wrapper is needed and execute the following command:

```
chmod +x gradlew
```

Once the file has execute permissions, the location of the file will either need to be added to your $PATH environment variable, or the name prefixed by ./ in order to run. For example:

```
./gradlew tasks
```

Gradle views project building in terms of number of different tasks. A full listing of tasks that are available for the current project can be obtained by running the following command from within the project directory (remembering to prefix the command with a ./ if running in macOS or Linux):

```
gradlew tasks
```

To build a debug release of the project suitable for device or emulator testing, use the assembleDebug option:

```
gradlew assembleDebug
```

Alternatively, to build a release version of the application:

```
gradlew assembleRelease
```

82.7 Summary

For the most part, Android Studio performs application builds in the background without any intervention from the developer. This build process is handled using the Gradle system, an automated build toolkit designed to allow the ways in which projects are built to be configured and managed through a set of build configuration files. While the default behavior of Gradle is adequate for many basic project build requirements, the need to configure the build process is inevitable with more complex projects. This chapter has provided an overview of the Gradle build system and configuration files within the context of an Android Studio project.