FREE eBook

LEARNING

android-gradle

#android-gradle

Table of Contents

About

Chapter 1: Getting started with android-gradle

Remarks

What is android-gradle

`android-gradle` is a `gradle` plugin officially maintained by Google Tools developer team and is the official build tool since the announcement in May 16, 2013 at the Google I/O.

Learn the basic by reading Configure your build with Gradle.

Main features

The main features of the Android Gradle Plugin are:

- Dependency management
- Modular Projects with libraries
- Variants through Flavors and Build Types
- IDE independent builds

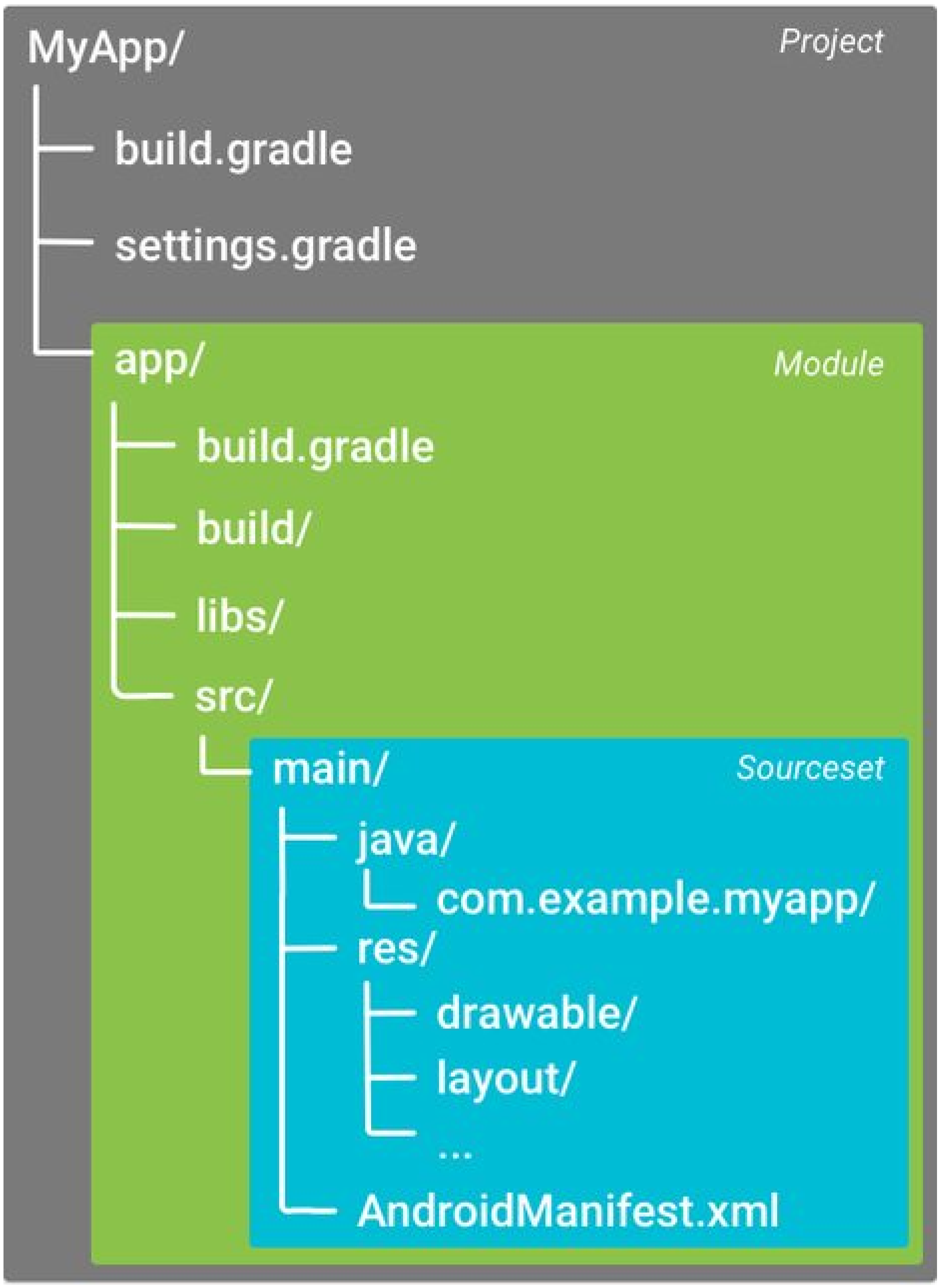
Overview

1. Download and install Android Studio
2. open it and create a new project with all default settings

   In theory you can install gradle directly, build the configuration files and directory structure by yourself. In practice no-one does that.

Project Structure

A project folder structure typically look like this:



`android-gradle` Plugin

A gradle project is usually divided in sub-project or *modules* each containing a dedicated build script.

The plugin dependency is usually declared in the main / top level `build.gradle` file:

```
buildscript {
    // maven repositories for dependencies
    repositories {
        jcenter()
    }
    // build script dependencies
    dependencies {
        // this is the dependency to the android build tools
        classpath 'com.android.tools.build:gradle:2.1.2'
    }
}
```

```
allprojects {
    // maven repositories for all sub-project / modules
    repositories {
        jcenter()
    }
}
```

In this example the `android-gradle` plugin version is `2.1.2` as you can see from this line:

```
classpath 'com.android.tools.build:gradle:2.1.2'
```

Modules

The Project is divided into *modules* each containing a dedicated `build.gradle` script. The `settings.gradle` file list these modules:

```
include ':app'
```

The colon `:` is used somewhat as a folder delimiter.

To use the plugin it has to be applied at the top of the `build.gradle` file of each module (`app` in the example).

For an Android Application:

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

For an Android Library:

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

And then configured in it's `android` tag:

```
android {
  // gradle-android plugin configuration
}
```

Basic Android application Configuration

The `build.gradle` generated by Android Studio for an application looks like this:

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

android {
    // setup which version of the SDK to build against and
    // with what version of the build tools
    compileSdkVersion 23
    buildToolsVersion "23.0.2"

    // default app configurations
    defaultConfig {
        // this is your app unique ID
        applicationId "com.example.myapp"

        // devices with lower SDK version can't install the app
        minSdkVersion 14
        // target SDK version, should be the last available one and
        // match the compile one
        targetSdkVersion 23

        // integer and string version of your app
        versionCode 1
        versionName "1.0"
    }

    // default build types are "debug" and "release"
    buildTypes {
        release {
            // enable / disable proguard optimization
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

// app dependencies
dependencies {
    // any jar file in the libs folder
    compile fileTree(dir: 'libs', include: ['*.jar'])
    // test dependency
    testCompile 'junit:junit:4.12'
    // runtime dependency on the support library
    compile 'com.android.support:appcompat-v7:24.0.0'
}
```

Configure your build with Gradle teach you more advanced Android Gradle Plugin settings and options and go deeper in the meaning of this setting.

The `defaultConfig` is called like that because it can be overridden with Product Flavors.

The `buildTypes` tag allow you to setup how to build your app enabling optimization (like proguard), you can learn more reading Build Types. It can also be used to setup signing of your app.

You should also learn more on how to Declare Dependencies. As you see the `dependencies` tag is outside the `android` one: this means it's not defined by the Android plugin but it's standard `gradle`.

The Gradle Wrapper

Android Studio will also, by default, install a gradle wrapper. This is a tool you can execute directly from the command line and it will download a local specific version of gradle the first time you execute it.

To launch compile the app you can then launch the gradle wrapper

Linux / Mac:

```
./gradlew assemble
```

Windows:

```
gradlew assemble
```

The script launch the wrapper, contained in a `gradle` folder in the root directory of your project:

- `gradle-wrapper.jar`: the code of the wrapper to download gradle and execute it
- `gradle-wrapper.properties` define which gradle version the wrapper should download

---

External Links:

- Official Android Build Tools documentation
- Official Android Gradle Plugin documentation
- Stackoverflow gradle documentation
- Official gradle documentation

Examples

Initial Setup with Android Studio

To setup for using Android Gradle Plugin you need many things:

- java
- gradle
- the Android project folder structure
- an Android Manifest
- initial plugin setup

The easiest way to get all of them is to follow these steps:

1. Donwload and Install Java OpenJDK version 6 or 7 (you can use 8 with additional settings of the gradle plugin)
2. Download and Install Android Studio
3. Create a new project (if you need help see Creating a New Project)

Check *Remarks* section for more informations.

Android Plugin for Gradle

As described in the remarks section the Android build system uses the Android Plugin for Gradle to support building Android applications with Gradle.

You can specify the Android Plugin for Gradle version in the top-level `build.gradle` file. The plugin version applies to all modules built in that Android Studio project.

```
buildscript {
  ...
  dependencies {
    classpath 'com.android.tools.build:gradle:2.2.0'
  }
}
```

Gradle wrapper

As described in the remarks section you can specify the Gradle version used by each project editing the Gradle distribution reference in the `gradle/wrapper/gradle-wrapper.properties` file.

For example:

```
...
distributionUrl = https\://services.gradle.org/distributions/gradle-2.14.1-all.zip
...
```

Read Getting started with android-gradle online: https://riptutorial.com/android-gradle/topic/2092/getting-started-with-android-gradle

Chapter 2: Configure Build Types

Parameters

Parameter	Detail
applicationIdSuffix	Application id suffix applied to this base config
consumerProguardFiles	ProGuard rule files to be included in the published AAR
debuggable	Whether this build type should generate a debuggable apk
embedMicroApp	Whether a linked Android Wear app should be embedded in variant using this build type
jniDebuggable	Whether this build type is configured to generate an APK with debuggable native code
manifestPlaceholders	The manifest placeholders
minifyEnabled	Whether Minify is enabled for this build type
multiDexEnabled	Whether Multi-Dex is enabled for this variant
name	Name of this build type
proguardFiles	Returns ProGuard configuration files to be used
pseudoLocalesEnabled	Whether to generate pseudo locale in the APK
renderscriptDebuggable	Whether the build type is configured to generate an apk with debuggable RenderScript code
renderscriptOptimLevel	Optimization level to use by the renderscript compiler
shrinkResources	Whether shrinking of unused resources is enabled. Default is false
signingConfig	The signing configuration
testCoverageEnabled	Whether test coverage is enabled for this build type
versionNameSuffix	Version name suffix
zipAlignEnabled	Whether zipalign is enabled for this build type

Parameter	Detail
------	--------
Method	**Detail**
buildConfigField(type, name, value)	Adds a new field to the generated BuildConfig class
consumerProguardFile(proguardFile)	Adds a proguard rule file to be included in the published AAR
consumerProguardFiles(proguardFiles)	Adds proguard rule files to be included in the published AAR
proguardFile(proguardFile)	Adds a new ProGuard configuration file
proguardFiles(proguardFiles)	Adds new ProGuard configuration files
resValue(type, name, value)	Adds a new generated resource
resValue(type, name, value)	Adds a new generated resource
setProguardFiles(proguardFileIterable)	Sets the ProGuard configuration files
shrinkResources(flag)	Whether shrinking of unused resources is enabled. Default is false

Remarks

By default, the Android plugin for gradle automatically sets up the project to build both a debug and a release version of the application.

This configuration is done through an object called a `BuildType`

Official Documentation:

http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.BuildType.html

Examples

How to configure build types in the build.gradle

You can create and configure build types in the module-level `build.gradle` file inside the `android {}` block.

```
    android {
        ...
        defaultConfig {...}

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

            debug {
                applicationIdSuffix ".debug"
            }
        }
    }
```

Read Configure Build Types online: https://riptutorial.com/android-gradle/topic/3281/configure-build-types

Chapter 3: Configure Product Flavors

Remarks

The product flavors support the same properties as `defaultConfig` this is because defaultConfig actually belongs to the ProductFlavor class. This means you can provide the base configuration for all flavors in the `defaultConfig {}` block, and each flavor can override any of these default values, such as the a`pplicationId`.

Examples

How to configure the build.gradle file

```
android {
    ...
    defaultConfig {...}
    buildTypes {...}
    productFlavors {
        demo {
            applicationId "com.example.myapp.demo"
            versionName "1.0-demo"
        }
        full {
            applicationId "com.example.myapp.full"
            versionName "1.0-full"
        }
    }
}
```

Flavor Constants and Resources in build.gradle

You can use gradle to have `BuildConfig` constants and `res` values on a per flavor basis. Just add the value to the flavor you want to support.

```
android {
    defaultConfig {
        resValue "string", "app_name", "Full App"
        buildConfigField "boolean", "isDemo", "false"
    }
    productFlavors {
        demo {
            resValue "String", "app_name", "Demo App"
            buildConfigField "boolean", "isDemo", "true"
        }
        full {
            // use default values
        }
    }
}
```

Gradle will do all the merging / overriding for you. The generated code will also allow you to see where the values come from, e.g.

```
<!-- Values from default config. -->
<string name="app_name" translatable="false">Default Name</string>
```

and

```
public final class BuildConfig {
    public static final String VERSION_NAME = "1.0";
    // Fields from product flavor: demo
    public static final boolean isDemo = true;
}
```

Using Flavor Dimension

When the app is based on more than one criteria, instead of creating a lot of flavors you can define flavor dimensions.

The flavor dimensions define the cartesian product that will be used to produce variants.

Example:

```
flavorDimensions("dimA", "dimB")

productFlavors {

    row1 {
        ...
        dimension = "dimA"
    }
    row2 {
        ...
        dimension = "dimA"
    }
    row3 {
         ...
        dimension = "dimA"
    }

    col1 {
        ...
        dimension = "dimB"
    }
    col2 {
        ...
        dimension = "dimB"
    }
    col3 {
         ...
        dimension = "dimB"
    }
}
```

This config will produce 18 (3*3*2) variants (if you have the 2 standard build types : `debug` and `release`). The following build variants will be created:

```
row1-col1-debug
row1-col2-debug
row1-col3-debug
row1-col1-release
row1-col2-release
row1-col3-release

row2-col1-debug
row2-col2-debug
row2-col3-debug
row2-col1-release
row2-col2-release
row2-col3-release

row3-col1-debug
row3-col2-debug
row3-col3-debug
row3-col1-release
row3-col2-release
row3-col3-release
```

The **order of the dimension** is defined by `android.flavorDimensions` and **drives which flavor override the other**, which is important for resources when a value in a flavor replaces a value defined in a lower priority flavor.

The flavor dimension is defined with higher priority first. So in this case:

```
dimA > dimB > defaultConfig
```

There is also a "flavor combination" source folder available when more than one flavor dimension is used. For instance `src/flavor1Flavor2/`.

- Note that this is for all combinations of all dimensions.
- Its priority is higher than single-flavor sourcesets, but lower than build-types.

Add dependencies for flavors

You can add different dependencies for a specific product flavor.

Just use the `<flavorName>Compile 'group:name:x.y.z'` syntax:

```
android {
    ...
    productFlavors {
        flavor1 {
           //.....
        }
        flavor2 {
           //.....
        }
    }
}

...
dependencies {
    compile 'com.android.support:appcompat-v7:24.2.0'

    // Add a dependency only for flavor1
    flavor1Compile 'group:name:x.y.z'

    // Add a dependency only for flavor2
    flavor2Compile 'group:name:x.y.z'
}
```

Develop and Production Product Flavors Example

```
productFlavors {
        // Define separate dev and prod product flavors.
        dev {
            // dev utilizes minSDKVersion = 21 to allow the Android gradle plugin
            // to pre-dex each module and produce an APK that can be tested on
            // Android Lollipop without time consuming dex merging processes.
            minSdkVersion 21
        }
        prod {
            // The actual minSdkVersion for the application.
            minSdkVersion 15
        }
    }
```

Read Configure Product Flavors online: https://riptutorial.com/android-gradle/topic/2929/configure-product-flavors

Chapter 4: Configure Signing Settings

Examples

Configure the build.gradle with signing configuration

You can define the signing configuration to sign the apk in the `build.gradle` file.

You can define:

- `storeFile` : the keystore file
- `storePassword`: the keystore password
- `keyAlias`: a key alias name
- `keyPassword`: A key alias password

You have to **define** the `signingConfigs` block to create a signing configuration:

```
android {
    signingConfigs {

        myConfig {
            storeFile file("myFile.keystore")
            storePassword "myPasswork"
            keyAlias "aKeyAlias"
            keyPassword "myAliasPassword"
        }
    }
    //....
}
```

Then you can **assign** it to one or more build types.

```
android {

    buildTypes {
        release {
            signingConfig signingConfigs.myConfig
        }
    }
}
```

Define the signing configuration in an external file

You can define the signing configuration in an external file as a `signing.properties` in the root directory of your project.

For example you can define these keys (you can use your favorite names):

```
STORE_FILE=myStoreFileLocation
STORE_PASSWORD=myStorePassword
```

```
KEY_ALIAS=myKeyAlias
KEY_PASSWORD=mykeyPassword
```

Then in your build.gradle file:

```
android {

    signingConfigs {
        release
    }

     buildTypes {
        release {
            signingConfig signingConfigs.release
        }
     }
}
```

Then you can introduce some checks to avoid gradle issues in the build process.

```
//------------------------------------------------------------------------------
// Signing
//------------------------------------------------------------------------------
def Properties props = new Properties()
def propFile = file('../signing.properties')
if (propFile.canRead()) {

    if (props != null && props.containsKey('STORE_FILE') &&
props.containsKey('STORE_PASSWORD') &&
            props.containsKey('KEY_ALIAS') && props.containsKey('KEY_PASSWORD')) {

        android.signingConfigs.release.storeFile = file(props['STORE_FILE'])
        android.signingConfigs.release.storePassword = props['STORE_PASSWORD']
        android.signingConfigs.release.keyAlias = props['KEY_ALIAS']
        android.signingConfigs.release.keyPassword = props['KEY_PASSWORD']
    } else {
        android.buildTypes.release.signingConfig = null
    }
} else {
    android.buildTypes.release.signingConfig = null
}
```

Define the signing configuration setting environment variables

You can store the signing information setting environment variables.
These values can be accessed with `System.getenv("<VAR-NAME>")`

In your `build.gradle` you can define:

```
signingConfigs {
    release {
        storeFile file(System.getenv("KEYSTORE"))
        storePassword System.getenv("KEYSTORE_PASSWORD")
        keyAlias System.getenv("KEY_ALIAS")
        keyPassword System.getenv("KEY_PASSWORD")
    }
}
```

Define signing configuration in a separate gradle file

The simplest and cleanest way to add an external configuration is through a separate Gradle file

build.gradle

```
apply from: './keystore.gradle'
android{
    signingConfigs {
        release {
            storeFile file(keystore.storeFile)
            storePassword keystore.storePassword
            keyAlias keystore.keyAlias
            keyPassword keystore.keyPassword
        }
    }
}
```

keystore.gradle

```
ext.keystore = [
    storeFile    : "/path/to/your/file",
    storePassword: 'password of the store',
    keyAlias     : 'alias_of_the_key',
    keyPassword  : 'password_of_the_key'
]
```

The keystore.gradle file can exist anywhere in your file system, you can specify its location inside the `apply from: ''` at the top of your gradle file or at the end of your main project build.gradle file.

Typically its a good idea to ignore this file from version control system such as git if its located inside your repo.

It is also a good idea to provide a sample `keystore.gradle.sample` which developers entering the project would rename and populate on their development machine. This file would always be contained inside the repo at the correct location.

Read Configure Signing Settings online: https://riptutorial.com/android-gradle/topic/5249/configure-signing-settings

Chapter 5: Configure Your Build with Gradle

Remarks

The Android build system compiles app resources and source code, and packages them into APKs that you can test, deploy, sign, and distribute. Android Studio uses Gradle, an advanced build toolkit, to automate and manage the build process, while allowing you to define flexible custom build configurations.

Official Documentation

https://developer.android.com/studio/build/index.html

Examples

Why are there two build.gradle files in an Android Studio project?

`<PROJECT_ROOT>\app\build.gradle` is specific for app module.

`<PROJECT_ROOT>\build.gradle` is a "Top-level build file" where you can add configuration options common to all sub-projects/modules.

If you use another module in your project, as a local library you would have another build.gradle file: `<PROJECT_ROOT>\module\build.gradle`

The Top-level Build File

The top-level build.gradle file, located in the root project directory, defines build configurations that apply to all modules in your project. By default, the top-level build file uses the `buildscript {} block` to define the Gradle repositories and dependencies that are common to all modules in the project. The following code sample describes the default settings and DSL elements you can find in the top-level build.gradle after creating a new project.

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
       classpath 'com.android.tools.build:gradle:2.2.0'
       classpath 'com.google.gms:google-services:3.0.0'
    }
}

ext {
    compileSdkVersion = 23
    buildToolsVersion = "23.0.1"
}
```

The Module-level Build File

The module-level build.gradle file, located in each `<project>/<module>/` directory, allows you to configure build settings for the specific module it is located in. Configuring these build settings allows you to provide custom packaging options, such as additional build types and product flavors, and override settings in the `main/ app` manifest or top-level `build.gradle` file.

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


android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion
}

dependencies {
    //.....
}
```

Top Level File example

```
/**
 * The buildscript {} block is where you configure the repositories and
 * dependencies for Gradle itself--meaning, you should not include dependencies
 * for your modules here. For example, this block includes the Android plugin for
 * Gradle as a dependency because it provides the additional instructions Gradle
 * needs to build Android app modules.
 */

buildscript {

    /**
     * The repositories {} block configures the repositories Gradle uses to
     * search or download the dependencies. Gradle pre-configures support for remote
     * repositories such as JCenter, Maven Central, and Ivy. You can also use local
     * repositories or define your own remote repositories. The code below defines
     * JCenter as the repository Gradle should use to look for its dependencies.
     */

    repositories {
        jcenter()
    }

    /**
     * The dependencies {} block configures the dependencies Gradle needs to use
     * to build your project. The following line adds Android Plugin for Gradle
     * version 2.0.0 as a classpath dependency.
     */

    dependencies {
        classpath 'com.android.tools.build:gradle:2.0.0'
    }
}

/**
 * The allprojects {} block is where you configure the repositories and
 * dependencies used by all modules in your project, such as third-party plugins
 * or libraries. Dependencies that are not required by all the modules in the
 * project should be configured in module-level build.gradle files. For new
 * projects, Android Studio configures JCenter as the default repository, but it
 * does not configure any dependencies.
 */

allprojects {
   repositories {
       jcenter()
   }
}
```

The module file example

```
/**
 * The first line in the build configuration applies the Android plugin for
 * Gradle to this build and makes the android {} block available to specify
 * Android-specific build options.
 */

apply plugin: 'com.android.application'

/**
 * The android {} block is where you configure all your Android-specific
 * build options.
 */

android {

  /**
   * compileSdkVersion specifies the Android API level Gradle should use to
   * compile your app. This means your app can use the API features included in
   * this API level and lower.
   *
   * buildToolsVersion specifies the version of the SDK build tools, command-line
   * utilities, and compiler that Gradle should use to build your app. You need to
   * download the build tools using the SDK Manager.
   */

  compileSdkVersion 23
  buildToolsVersion "23.0.3"

  /**
   * The defaultConfig {} block encapsulates default settings and entries for all
   * build variants, and can override some attributes in main/AndroidManifest.xml
   * dynamically from the build system. You can configure product flavors to override
   * these values for different versions of your app.
   */

  defaultConfig {

    /**
     * applicationId uniquely identifies the package for publishing.
     * However, your source code should still reference the package name
     * defined by the package attribute in the main/AndroidManifest.xml file.
     */

    applicationId 'com.example.myapp'
    // Defines the minimum API level required to run the app.
    minSdkVersion 14

    // Specifies the API level used to test the app.
    targetSdkVersion 23

    // Defines the version number of your app.
    versionCode 1

    // Defines a user-friendly version name for your app.
    versionName "1.0"
  }

  /**
   * The buildTypes {} block is where you can configure multiple build types.
   * By default, the build system defines two build types: debug and release. The
   * debug build type is not explicitly shown in the default build configuration,
   * but it includes debugging tools and is signed with the debug key. The release
   * build type applies Proguard settings and is not signed by default.
   */

  buildTypes {

    /**
     * By default, Android Studio configures the release build type to enable code
     * shrinking, using minifyEnabled, and specifies the Proguard settings file.
     */

    release {
        minifyEnabled true // Enables code shrinking for the release build type.
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
  }

  /**
   * The productFlavors {} block is where you can configure multiple product
   * flavors. This allows you to create different versions of your app that can
   * override defaultConfig {} with their own settings. Product flavors are
   * optional, and the build system does not create them by default. This example
   * creates a free and paid product flavor. Each product flavor then specifies
   * its own application ID, so that they can exist on the Google Play Store, or
   * an Android device, simultaneously.
   */

  productFlavors {
    free {
      applicationId 'com.example.myapp.free'
    }

    paid {
      applicationId 'com.example.myapp.paid'
    }
  }
}

/**
 * The dependencies {} block in the module-level build configuration file
 * only specifies dependencies required to build the module itself.
 */

dependencies {
    compile project(":lib")
    compile 'com.android.support:appcompat-v7:24.1.0'
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

Use archivesBaseName to change the apk name

You can use the `archivesBaseName` to set the name of apk.

For example:

```
  defaultConfig {
      ....
      project.ext.set("archivesBaseName", "MyName-" + defaultConfig.versionName);

  }
```

You will obtain this output.

```
MyName-X.X.X-release.apk
```

Read Configure Your Build with Gradle online: https://riptutorial.com/android-gradle/topic/2161/configure-your-build-with-gradle

Chapter 6: Declare Dependencies

Examples

How to add dependencies

The example below describes how to declare three different types of direct dependencies in the app/ module's `build.gradle` file:

```
   android {...}
    ...
    dependencies {
        // The 'compile' configuration tells Gradle to add the dependency to the
        // compilation classpath and include it in the final package.

        // Dependency on the "mylibrary" module from this project
        compile project(":mylibrary")

        // Remote binary dependency
        compile 'com.android.support:appcompat-v7:24.1.0'

        // Local binary dependency
        compile fileTree(dir: 'libs', include: ['*.jar'])
    }
```

How to add a repository

To download dependencies, declare the repository so Gradle can find them. To do this, add a `repositories { ... }` to the app/ module's `build.gradle` in the top-level file.

```
repositories {
  // Gradle's Java plugin allows the addition of these two repositories via method calls:
  jcenter()
  mavenCentral()

  maven { url "http://repository.of/dependency" }

  maven {
      credentials {
          username 'xxx'
          password 'xxx'
      }

  url 'http://my.maven
  }
}
```

Module dependencies

In a multi-project `gradle build`, you can have a dependency with another module in your build.

Example:

```
  dependencies {
        // Dependency on the "mylibrary" module from this project
        compile project(":mylibrary")
  }
```

The `compile project(':mylibrary')` line declares a local Android library module named "mylibrary" as a dependency, and requires the build system to compile and include the local module when building your app.

Local binary dependencies

You can have a dependency with a single jar or multiple jar files.

With a single jar file you can add:

```
dependencies {
    compile files('libs/local_dependency.jar')
}
```

It's possible to add a directory of jars to compile.

```
dependencies {
        compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

The compile `fileTree(dir: 'libs', include: ['*.jar']`) line tells the build system to include any JAR files inside the `app/libs/` directory in the compilation classpath and in the final package of your app.

If you have modules that require local binary dependencies, copy the JAR files for these dependencies into `<moduleName>/libs` inside your project.

If you need to add an **aar files** you can read more details here.

Remote binary dependencies

You can add remote dependencies in Gradle usign this structure:

```
compile 'group:name:version'
```

or this alternative syntax:

```
compile group: 'xxx', name: 'xxxxx', version: 'xxxx'
```

For example:

```
compile 'com.android.support:appcompat-v7:24.1.0'
```

The `compile 'com.android.support:appcompat-v7:24.1.0`' line declares a dependency on version

24.1.0 of the Android Support Library.

Declare Dependencies for Configurations

Dependencies can be added for specific configuration like test/androidTest

```
androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.1'
testCompile 'junit:junit:3.8.1'
```

Alternatively create your own configuration

```
configurations {
    myconfig
}
```

And then download dependency for this config

```
myconfig group: 'com.mycompany', name: 'my_artifact', version: '1.0.0'
```

Declare dependencies for flavors

Dependencies can be added for specific product flavors in a similar fashion as build configurations
.

```
android {
    ...
    productFlavors {
        flavor1 {
            //...
        }
        flavor2 {
            //...
        }
    }
}

dependencies {
    flavor1Compile 'com.android.support:appcompat-v7:24.1.1'
    flavor1Compile 'com.google.firebase:firebase-crash:9.4.0'

    flavor2Compile 'com.android.support:appcompat-v7:24.1.1'
}
```

Declare dependencies for build types

Dependencies can be added for specific Build types:

```
android {
    ...
    buildTypes {
        release {
            //...
        }

        debug {
            //....
        }
    }
}

dependencies {
    debugCompile 'com.android.support:appcompat-v7:24.1.1'
    releaseCompile 'com.google.firebase:firebase-crash:9.4.0'
}
```

Read Declare Dependencies online: https://riptutorial.com/android-gradle/topic/3289/declare-dependencies

Chapter 7: Gradle - Information of Tags

Examples

Gradle - Information of Tags

Gradle: It is used to make build for any software, it is a Domain specific language used to configure and fulfill all plugins, libraries downloaded from repositories.

Use Plugins:

```
Apply plugin: ‘com.android.application’
```

Plugin is property in key value form. In above statement plugin denotes to key and right side string in single coats becomes its value.

Gradle is DSL (Domain specific language):

It contains different `blocks:Tags`

```
repositories { }
dependencies {}
android {}
```

Repositories and dependencies are used to configure requirements for application code. Android block is used to add android specific code or information into application. We also generate our custom tags and define our own custom code, library and information.

By using `“task” tag :`

```
task genrateTestDb (depends on: ….) {
 }
```

Gradle files for any application

`Build.gradle` -These file is working for all project. `Settings.gradle` – define all sub directories or projects are included in application.

`Build.gradle` contains below:

```
repositories {
mavenCentral()
}
```

Above repositories tag hold `mevenCentral()` it means all dependencies are downloaded from `mevenCentral()` .we can use `jcenter()` or any other source too. Dependencies block holds all **compile time dependencies** that’s should be downloaded from `repositories`.

```
dependencies {
compile ‘org.codehous.groovy:groovy-all:2.3.2’
}
```

Above is `meven` library : syntax:

`org.codehous.groovy` - > group id

`groovy-all` - > order fact id , that’s is a name gradle used to identify library .

`2.3.2’` - > version

`Settings.gradle` – it’s include tag for all sub projects that’s is added into project.

```
Include ‘googlechart’, ‘chuckgroovy’
```

Read Gradle - Information of Tags online: https://riptutorial.com/android-gradle/topic/9439/gradle---information-of-tags

Chapter 8: How to include aar files in a project in Android

Examples

How to add .aar dependency in a module?

In a module (library or application) where you need the aar file you have to add in your `build.gradle` the repository:

```
repositories {
    flatDir {
        dirs 'libs'
    }
}
```

and add the dependency:

```
dependencies {
    compile(name:'nameOfYourAARFileWithoutExtension', ext:'aar')
}
```

Pay attention to the relative path of the libs folder that you are using in the module.

The aar file doesn't include the transitive dependencies

The **aar** file **doesn't contain the transitive dependencies** and doesn't have a pom file which describes the dependencies used by the library.

It means that, if you are importing a aar file using a `flatDir` repo **you have to specify the dependencies also in your project**.

You should use a **maven repository** (you have to publish the library in a private or public maven repo), you will not have the same issue.
In this case, gradle downloads the dependencies using the pom file which will contains the dependencies list.

This works with aar libraries that are published to a remote or local maven repository, In your case it sounds like the library will not be published to even a local maven repository. I can't find any definitive information as to if it will work in your circumstances, but you should give it a shot.

Read How to include aar files in a project in Android online: https://riptutorial.com/android-gradle/topic/3037/how-to-include-aar-files-in-a-project-in-android

Chapter 9: Shrink Code and Resources

Remarks

To make your APK file as small as possible, you should enable shrinking to remove unused code and resources in your release build.

Examples

Shrink the code with ProGuard

To enable code shrinking with ProGuard, add `minifyEnabled` true to the appropriate build type in your `build.gradle` file.

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

where:

- `minifyEnabled true` : enable code shrinking
- The `getDefaultProguardFile(‘proguard-android.txt')` method gets the default ProGuard settings from the Android SDK
- The `proguard-rules.pro` file is where you can add custom ProGuard rules

Shrink the resources

To enable resource shrinking, set the `shrinkResources` property to true in your `build.gradle` file.

```
android {
    ...

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

Pay attention because resource shrinking **works only** in conjunction with code shrinking.

You can customize which resources to keep or discard creating an XML file like this:

```
<?xml version=1.0" encoding="utf-8"?>
<resources xmlns:tools="http://schemas.android.com/tools"
    tools:keep="@layout/mylayout,@layout/custom_*"
    tools:discard="@layout/unused" />
```

Save this file in `res/raw` folder.

Remove unused alternative resources

All libraries come with resources that are not necessary useful to your application. For example Google Play Services comes with translations for languages your own application don't even support.

You can configure the build.gradle file to specify which resource you want to keep.
For example:

```
defaultConfig {
    // ...

    resConfigs "en", "de", "it"
    resConfigs "nodpi", "xhdpi", "xxhdpi", "xxxhdpi"
}
```

Read Shrink Code and Resources online: https://riptutorial.com/android-gradle/topic/5257/shrink-code-and-resources

Credits

S. No	Chapters	Contributors
1	Getting started with android-gradle	Community, Daniele Segato, Gabriele Mariotti
2	Configure Build Types	Gabriele Mariotti
3	Configure Product Flavors	David Medenjak, Gabriele Mariotti, piotrek1543, Tarek El-Mallah
4	Configure Signing Settings	DArkO, Gabriele Mariotti
5	Configure Your Build with Gradle	Gabriele Mariotti
6	Declare Dependencies	4444, cricket_007, Gabriele Mariotti, jitinsharma
7	Gradle - Information of Tags	Chetan Joshi
8	How to include aar files in a project in Android	Gabriele Mariotti, JBirdVegas
9	Shrink Code and Resources	Gabriele Mariotti