Android Application Development with Maven

Android is an open source operating system used for smartphones and tablet computers. The Android market is one of the biggest and fastest growing platforms for application developers, with over a million apps uploaded every day.

Right from the beginning, this book will cover how to set up your Maven development environment and integrate it with your favorite IDE. By sequentially working through the steps in each chapter, you will quickly master the plugins you need for every phase of the Android development process. You will learn how to use Maven to manage and build your project and dependencies, automate your Android application testing plans, and develop and maintain several versions of your application in parallel. Most significantly, you will learn how to integrate your project into a complete factory.

Who this book is written for
Android Application Development with Maven is intended for Android developers or devops engineers who want to use Maven to effectively develop quality Android applications. It would be helpful, but not necessary, if you have some previous experience with Maven.

What you will learn from this book
- Integrate Maven with your favorite Android IDE
- Install and configure Maven with your local development environment
- Create the proper Maven structure for both standalone Android applications or applications that are part of a bigger project
- Run unit tests using popular frameworks such as Robolectric and collect coverage information using Maven plugins
- Configure a variety of different tools such as Robotium, Spoon, and Selendroid to run integration tests
- Handle dependencies and different versions of the same application
- Manage and automate the release process of your application inside/outside Google Play
- Discover new tools such as Eclipse, IntelliJ IDEA/Android Studio, and NetBeans, which perfectly integrate with Maven and boost your productivity

Learn how to use and configure Maven to support all phases of the development of an Android application.
In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 2 'Starting the Development Phase'
- A synopsis of the book’s content
- More information on Android Application Development with Maven

About the Authors

Patroklos Papapetrou is a software architect addicted to software quality and an agile team leader with more than 15 years of experience in software engineering. His expertise lies in Android and Java development. He believes and invests in people and team spirit, seeking quality excellence. He's one of the authors of the book SonarQube in Action, Manning Publications and his next writing attempt will be The Art of Software Gardening. He treats software systems like flowers; that's why he prefers to call himself a software gardener.

He's also an occasional speaker, conducting talks about clean code, Android development, code quality, and software gardening.

I'd like to thank my loving and beautiful wife, Anna, for her patience all these months, especially during the weekends. Without her encouragement, I wouldn't have managed to finish my part of the book. Thanks to my sons, Panagiotis (age 6) and Charis (4-years old), who understood that sometimes, daddy couldn't play with them or go to the park. You can have me back now! Thanks to our families for their patience as well and for sometimes watching the kids to let me work on the book.
Jonathan LALOU is an engineer fascinated by new technologies, computer sciences, and the digital world since his childhood. A graduate of the École des Mines—one of the best French polytechnic institutes—Jonathan has more than 14 years of experience in Java and the JEE ecosystem.

Jonathan has worked for several global companies and financial institutions, such as Syred, Philips, Sungard, Ixis CIB, BNP Paribas, and Amundi AM. He has strong ties, daily contacts, and frequent trips in Western Europe, Northern America, Judea, and emerging Asia. During his career, Jonathan has successfully climbed many levels: developer, architect, Scrum master, team leader, and project manager.

Now, Jonathan is CTO at SayaSoft (http://www.sayasoft.fr), a digital company focused on very high value added projects he founded with two partners. SayaSoft brings Java environment, migration of small and large organizations to agility, and Android development to a new level. SayaSoft’s customers are ensured to get high-quality releases and quick ROI.

Jonathan's skills include a wide range of technologies and frameworks, such as Spring, JPA/Hibernate, GWT, Mule ESB, JSF/PrimeFaces, Groovy, Android, EJB, JMS, application servers, agile methods, and, of course, Apache Maven.


Android Application Development with Maven

During the months we were writing this book, a lot of people asked us what would make this book special and why someone should care to read it. The most powerful argument that I heard all this time was, "Hey, Google official supports only Gradle to build Android applications and the latest release of Android Studio makes extensive use of Gradle. Ant was already replaced and Maven is nowhere. Why do you spend time writing about developing Android applications with Maven?"

Good questions! The answers, however, is hidden within the question itself. First of all, there are no books out there that explain step by step about all the development phases and critical tasks to build and manage the life cycle of an Android Application with Maven. Maven is still—no matter if we like it or not—the most popular build tool. Many "traditional" software houses that have invested time and efforts to standardize their development process around Maven want to make the next step and expand their portfolio to the Android Market. Clearly, having another build tool only for Android development doesn't look very practical, although it's an option.

Companies would save a lot of money if they could just plug their Android applications to the existing development life cycle, driven by Maven. At the same time, it's true that Maven is a very mature, flexible, and robust build tool. Its extensibility through plugins and the idea of descriptively configuring the build process without the need to write scripts made it the de-facto standard.

The reality, however, has shown us that it's not always that easy. Maven provides all the required plugins to do almost everything, but there are no instructions or well-structured documentation. You can find blog posts here and there that shortly cover some topics but this is not enough.

This book aims to fill that gap. It will not teach you how to write Android applications, although you will find some simple examples. It will guide you, however, from A to Z, about how to set up all the necessary Maven configuration to compile, run, test, deploy, release, and verify the quality of an Android application. It's convenient for both experienced and young Android developers because we provide all the example code to see Maven in action. This book is also for those of you who already have some Maven experience but feel lost when you try to integrate it with your Android development process.
You can read the book sequentially if you have little experience with Maven, but you can also use it as a reference and jump to any chapter you want as each one is dedicated to a particular topic. The provided code is separated in different folders per chapter so that you can easily run the examples and verify that you have correctly followed the instructions of the book.

We are confident that you will find the book useful and practical, and we hope that it will help you build your next Android application with Maven.

**What This Book Covers**

*Chapter 1, Beginning with the Basics*, introduces you to the basic concepts of Maven and guides you to install all the required software you need to develop an Android application with Maven.

*Chapter 2, Starting the Development Phase*, sets the pace for the rest of the book. It discusses the first step to set up a minimal Maven configuration to compile and deploy an Android application to a real device or emulator.

*Chapter 3, Unit Testing*, covers several ways to write and run unit tests using various tools. It also explains the differences between unit and integration testing and the important role that both of them playing when developing an Android application.

*Chapter 4, Integration Testing*, completes the discussion about testing and presents three alternatives of running Android instrumentation tests, and also provides guidance on properly configuring Maven.

*Chapter 5, Android Flavors*, discusses the hot topic of maintaining multiple versions (free, ads-supported, and paid) of the same application. It describes the problem and then presents two ways to handle the case using Maven.

*Chapter 6, Release Life Cycle and Continuous Integration*, is all about releasing and deploying an Android application to a Maven repository. A bonus topic in this chapter discusses about automating everything using Jenkins, the most popular continuous integration engine.

*Chapter 7, Other Tools and Plugins*, is the last chapter and covers two very important topics: code quality with SonarQube and speeding up the development life cycle using the non-standard emulators provided by Android.
Starting the Development Phase

In the previous chapter, we saw how to install a complete work area to develop an Android application, using Maven as the build and project management tool. In this chapter we will see how to take advantage of Maven during the development phase.

Reminders about Android projects

Before diving deeper into the subject, let's review some concepts and rules related to Android.

At first glance, an Android project looks like any other Java project. Yet, some differences are fundamental.

An Android project is made of sources (Java, XML, and property files) and code that is generated by the system. All of these are compiled into bytecode and compressed as an Android PacKage (APK) file. Some Android projects also can produce AAR files but we will have the chance to learn more about this in a following chapter.

Let's open the Android project we have created in the previous chapter, using an artifact. As seen, the hierarchy of folders and files is as following:

- assets/: This will be reviewed later.
- res/: This is a set of resources.
  - res/drawable-* pi/*.png: Due to the strong fractionation of Android system, and the high variety of devices, resources such as pictures, fonts, and so on have to be taken into account while releasing. Unless you do limit the portability of your application, you have no choice but to adapt your resources to the largest audience.
res/layout/activity_main.xml: This file describes the widgets; that is, the basic or complex graphical components and their position, padding, and so on. Keep in mind, however, that most Android projects have more than one activity. In this case, this folder will contain information for each activity: you can consider each activity as being a screen. The following code snippet shows the contents of the activity_main.xml file:

```xml
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">
  <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/hello_world"/>
</RelativeLayout>
```

res/menu/main.xml: This is the entry point of the application. It looks like any other XML file in res/layout/ folder but there is a big difference. This file is intended for use with a menuInflator parameter to create a menu in the onCreateOptionsMenu method of your activity.

res/values*: These files gather constants, such as strings, themes, and so on.

• src/main/java: Here is the folder for Java code specific to the application.

• target/generated-sources/: This folder contains the Java code generated based on the different files as described in the preceding text. Notice this folder location is specific to projects built with Maven: by default, any IDE should create a gen/ folder at the root of the project. To be even more accurate, this folder and its content are generated by Maven when goal android:generate-source method is executed.

• On root.

• AndroidManifest.xml: This file summarizes different parameters, such as the Software Development Kit (SDK) needed to compile and/or to run the list of activities (among which the default activity), the rights needed by this application:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
Now, let's review our POM and add some comments.

The headers are classic: they allow to determine the unique way to characterize our artifact, thanks to the triplet groupId, artifactId, version parameters such as shown in the following code snippet:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.androidMaven</groupId>
  <artifactId>chapter1</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```

- default.properties: The file used by Eclipse to determine the target system. If you don't see this in your local environment and you're not using Eclipse, then this is fine.
The sole difference, compared to other Java project, is the packaging: here, our archive is neither a "jar" nor an "ejb" or a "war" files, but the Android specific format: APK:

```
<packaging>apk</packaging>
```

The next block is made of properties. This corresponds to a good practice to factorize constants, version numbers, and so on in a unique location, so as to limit and avoid conflicts of versions:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <platform.version>5.0.1_r2</platform.version>
  <android.plugin.version>3.8.2</android.plugin.version>
</properties>
```

A block dependencies lists the dependencies of our project. So far, we depend only on Android jar. The interesting point is that the dependency to Android jar is of scope provided. Therefore (and logically), the Android jar will not be included in the APK at the end of the compile because the intent is that the code will be run within a specific environment (device or emulator) where the android.jar file is expected to be in the classpath:

```
<dependencies>
  <dependency>
    <groupId>com.google.android</groupId>
    <artifactId>android</artifactId>
    <version>${platform.version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

At last, the build block references the Maven plugin for Android. This will be explored later.

You can add the following block in your settings.xml file, available in $M2_HOME/conf or ~/.m2 location:

```
<pluginGroups>
  <pluginGroup>
    com.jayway.maven.plugins.android.generation2
  </pluginGroup>
</pluginGroups>
```

Then, you will be able to call the plugin without writing the full qualified name (com.jayway.maven.plugins.android.generation2:android-maven-plugin).
Creating an AVD

Before running any application, you have to create an Android Virtual Device (AVD), a kind of software emulator for Android device. Alternatively, you can run directly on a physical device (phone, tablet, and watch), which is actually faster. However, creating AVD instances allows you to test your application in a variety of configurations (OS version, screen size, memory, and so on) which is almost impossible to do with real devices. You can perform this operation via the Graphical User Interface (GUI), or in command line. Both produce the same result.

With the GUI

To run the AVD Manager in windows, you need to execute the AVD Manager.exe file located in the $ANDROID_HOME root; Linux users need to navigate to their SDK’s tools/ directory and execute:

```
$ android avd
```

When you see the dialog—the list probably contains no emulators – click on New to add a new AVD. Fill out the fields relating to the device you want to emulate, as shown in the following screenshot:
Starting the Development Phase

A pop up will confirm the result and the details of the device.

By default, the AVD are stored in ~/.android/avd or %USERPROFILE%/.android/avd location. You can override this location by adding an environment variable named ANDROID_SDK_HOME, pointing to $ANDROID_HOME/.android for instance. Beware that AVD Manager will not create this folder if it does not yet exist; you have to create this folder before running AVD Manager!

In-command line

To create an AVD in-command line, you have to determine the list of AVD you can create owing to the configuration and content of your SDK.

Run the following command:

%ANDROID_HOME%/tools/android list target

You get the following output (only the first target is printed, as well as the headers of the others):

Available Android targets:
------------------------

id: 1 or "android-21"
   Name: Android 5.0.1
   Type: Platform
   API level: 21
   Revision: 2
   Skins: HVGA, QVGA, WQVGA400, WQVGA432, WVGA, WVGA800 (default), WVGA854, WXGA720, WXGA800, WXGA800-7in, AndroidWearRound, AndroidWearSquare, AndroidWearRound, AndroidWearSquare
   Tag/ABIs : android-tv/armeabi-v7a, android-tv/x86, android-wear/armeabi-v7a, android-wear/x86, default/armeabi-v7a, default/x86, default/x64
------------------------

id: 2 or "Google Inc.:Google APIs :21"
The general pattern to create an AVD is:
$ANDROID_HOME/tools/android --name <name> --target <target> [options]
The target with id 1 parameter corresponds to the platform android-21, and the default skin is WVGA800. So, to create the same AVD as in the preceding output, the command line will be as follows:

```
$ANDROID_HOME/tools/android create avd --name Nexus_7_2012 \  
  --target "android-21" \  
  --skin WVGA800 \  
  --abi default/armeabi-v7a \  
  --path ~/.android/avd/Nexus_7_2012
```

Or, in short notation and default values:

```
$ANDROID_HOME/tools/android create avd -n Nexus_7_2012 \  
  -t 1 \  
  -b default/armeabi-v7a
```

The system confirms the creation of the AVD:

```
Android 5.0.1 is a basic Android platform.
Do you wish to create a custom hardware profile [no]
Created AVD 'Nexus_7_2012' based on Android 5.0.1, ARM (armeabi-v7a) processor,
with the following hardware config:
  hw.lcd.density=240
  hw.ramSize=512
  vm.heapSize=48
```

Nonetheless, this command-line-created AVD differs from the one created via the GUI, on the RAM size and heap size. You can edit the config.ini file within the folder where the AVD is stored (by default, in a subfolder of ~/.android/avd/), and manually change these settings (as well as many others):

```
  avd.ini.encoding=ISO-8859-1
  abi.type=armeabi-v7a
  hw.cpu.arch=arm
  hw.cpu.model=cortex-a8
  hw.lcd.density=240
  hw.ramSize=512
  image.sysdir.1=system-images\android-21\default\armeabi-v7a\  
  skin.name=WVGA800
  skin.path=platforms\android-21\skins\WVGA800
  tag.display=Default
  tag.id=default
  vm.heapSize=48
```
If you have to create multiple AVD with specific RAM and heap sizes, you can straightly edit the template, which is located in the skin you have chosen from the folder, for example, $ANDROID_HOME/platforms/android-19/skins/WVGA800/hardware.ini.

You can print a list of the installed AVDs by running the command:

```
$ANDROID_HOME/tools/android list avd
```

The expected output contains:

```
Available Android Virtual Devices:
  Name: Nexus_7_2012
  Path: C:\Users\jlalou\.android\avd\Nexus_7_2012.avd
  Target: Android 5.0.1 (API level 21)
  Tag/ABI: default/armeabi-v7a
  Skin: WVGA800
```

If you get such an error:

```
Error: Unable to find a 'userdata.img' file for ABI default/armeabi-v7a to copy into the AVD folder.
```

Then, check if the file `%ANDROID_HOME%/system-images/android-19/default/armeabi-v7a/userdata.img` does exist. If this does, this may be related to a known issue on certain versions of the SDK. The best to do is to update the SDK.

You can also rename, move, and delete AVD with the three command lines, respectively:

```
$ANDROID_HOME//tools/android move avd --name Nexus_7_2012 --rename Nexus_7_2012_bis
AVD 'Nexus_7_2012' moved.

$ANDROID_HOME/tools/android move avd --name Nexus_7_2012_bis --path $ANDROID_HOME/tmp
AVD 'Nexus_7_2012_bis' moved.

$ANDROID_HOME/tools/android delete avd --name Nexus_7_2012_bis
Deleting file ....\avd\Nexus_7_2012_bis.ini
Deleting folder ...android-sdk-r22.6\tmp
AVD 'Nexus_7_2012_bis' deleted.
```
Develop and Build

Now that we have learned how to manage emulators and we have understood the typical Maven project structure and the basic configuration settings found in pom.xml file, we are ready to start developing our simple application. Remember that this book's purpose is not to teach how to code in Android SDK, but to explain how you can effectively use maven to speed up the development process. We will explain the required Maven commands but we will not focus on a particular IDE. Each IDE has its own way of creating Maven command executions and it's out of the scope of this book. You can experiment with your favorite IDE but in any case, if you want to master Android Maven development you should be able to at least run all Maven commands from a terminal window, like that discussed in the following sections and throughout this whole book.

Cleaning

Our first step is to clean the project from any generated source code or other artifacts. Typically, the Maven clean goal is included among any other target command, but for clarity, since it's the first time we explain this, we will run separate commands. During the next chapters, you will notice that the clean goal is executed with other Maven goals. Open a terminal window and navigate to the root of our example project. Then, run the following command:

mvn clean

Generating sources

Run mvn android:generate-sources command. Have a look at the expected output:

[INFO] Scanning for projects...
[INFO]
[INFO] --- android-maven-plugin:3.8.2:generate-sources (default-cli) @ chapter1 ---
[INFO] ANDROID-904-002: Found aidl files: Count = 0
[INFO] Manifest merging disabled. Using project manifest only
C:\dev\android\sdk\build-tools\21.1.2\aapt.exe [package, -f, -
### Starting the Development Phase

```bash
-no-crunch, -I, C:\dev\android\sdk\platforms\android-21\android.jar, 
-M, C:\dev\android\packt\BookSteps\chapter1\AndroidManifest.xml, -S, 
C:\dev\android\packt\BookSteps\chapter1\res, -A, 
C:\dev\android\packt\BookSteps\chapter1\target\generated-sources\combined-assets\assets, -m, 
-J, C:\dev\android\packt\BookSteps\chapter1\target\generated-sources\r, 
--output-text-symbols, C:\dev\android\packt\BookSteps\chapter1\target, 
--auto-add-overlay]

[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time: 1.709 s
[INFO] Finished at: 2015-01-18T18:23:01+02:00
[INFO] Final Memory: 13M/310M

```
```

Basically, as hinted, `android:generate-sources` goal calls the `aapt` tool from the Android SDK.

As expected, a `target/generated-sources` file has been created. It contains the regular files already seen in the preceding text, such as the `R.java` file.

### Build

To build with Maven, simply run `mvn clean install`.

This compiles the project and generates several artifacts under `target` directory:

- **classes.dex**: This archive gathers the compressed bytecode generated by compilation, in a format understandable by Dalvik, which is the Java Virtual Machine executed below Android system. Actually, Dalvik is quite more limited than HotSpot or JRockit (the major JVM in desktop computers and servers), but is more adapted to short-resource devices such as smartphones and tablets.

- **{artifactId}.ap_**: This archive gathers the resources of the application: XML, picture files, and so on.

- **{artifactId}.apk**: This Android Package, compressed and signed. Basically, the APK file merges the `{artifactId}.ap_` and `classes.dex` files.
• `{artifactId}.jar`: A Java archive containing zipped bytecode (.class files).

Actually, in the operation of building, our aim is to get an APK file; therefore, running another goal such as `mvn clean android:apk` may have obtained the same result.

## Emulator

You can start and stop one or many emulators with Maven commands:

### Start

`mvn android:emulator-start` starts up an AVD. By default, Maven searches for an AVD named **Default**. This can be overridden via changing some options:

- Either in the POM: add a block similar to:
  
  ```xml
  <plugin>
  <groupId>com.jayway.maven.plugins.android.generation2</groupId>
  <artifactId>android-maven-plugin</artifactId>
  <version>${android.plugin.version}</version>
  <extensions>true</extensions>
  <configuration>
  <sdk>
  <platform>21</platform>
  </sdk>
  <emulator>
  <!-- Name of the AVD to start/stop-->
  <avd>Nexus_7_2012</avd>
  <!-- Timeout to consider whether or not the AVD is successful or failed. Do not be stingy on this value, since your material configuration may affect and influence on startup speed-->
  <wait>30000</wait>
  <!-- Any other option-->
  <options></options>
  </emulator>
  </configuration>
  </plugin>
  ```
Starting the Development Phase

- In the command line:

```
mvn android:emulator-start -Dandroid.emulator.avd=Nexus_7_2012 \
-Dandroid.emulator.wait=30000 \
-Dandroid.emulator.options=-no-skin
```

The options available are those of $ANDROID_HOME/tools/emulator: you can display them by running $ANDROID_HOME/tools/emulator -help, or, alternatively, by consulting this Gist: https://gist.github.com/JonathanLalou/180c87554d8278b0e6d7

The expected output is like:

```
[INFO] ----------------------------------------------------------
[INFO] Building chapter1 1.0-SNAPSHOT
[INFO] ----------------------------------------------------------
[INFO] --- android-maven-plugin:3.8.2:emulator-start (default-cli) @ helloWorld ---
[INFO] Android emulator command: ""C:\win32app\android-sdk-r22.6\tools\emulator"" -avd Nexus_7_2012

[INFO] Found 0 devices connected with the Android Debug Bridge
[INFO] Starting android emulator with script: C:\Users\jlalou\AppData\Local\Temp\ConsolePortableTemp\android-maven-plugin-emulator-start.vbs
[INFO] Waiting for emulator start:300000
[INFO] Emulator is up and running.
```

Actually, under Windows the plugin only calls a VB script to launch the emulator.

### Stop

`mvn android:emulator-stop` stops up an AVD. By default, Maven searches for an AVD named Default. Like `emulator-start` goal, this default behavior can be overridden if the POM contains a `<configuration>` block pointing at the AVD, or if the command line includes the option `-Dandroid.emulator.avd=<name_of_the_AVD>`

### Stop all

`mvn android:emulator-stop` stops all AVD running on the system.
Deploy

With Maven, provided that you have already built the project through `mvn clean install`, run the command:

```
mvn android:deploy.
```

This output is similar to:

```
[INFO] --- android-maven-plugin:3.8.2:deploy (default-cli) @ helloWorld ---
[INFO] Waiting for initial device list from the Android Debug Bridge
[INFO] Found 1 devices connected with the Android Debug Bridge
[INFO] android.device parameter not set, using all attached devices
[INFO] Emulator emulator-5554_Nexus_7_2012_unknown_sdk found.
[INFO] emulator-5554_Nexus_7_2012_unknown_sdk : Successfully installed C:\Users\jialou==PRIVATE==\GDrive\packt\6101\sources\helloWorldWithoutIDEA\target\chapter1.apk to emulator-5554_Nexus_7_2012_unknown_sdk
```

Then, in the emulator, the APK that you have compiled and deployed appears in the application list (as shown in the following screenshot: fourth line, second column):
Starting the Development Phase

By default, Maven deploys the application on all active emulators; to deploy on a single target, add the `-Dandroid.device=<name_of_the_AVD>` parameter, as given here:

```
mvn android:deploy -Dandroid.device=Nexus_7_2012
```

The output contains:

```
[INFO] --- android-maven-plugin:3.8.2:deploy (default-cli) @ AndroidTier ---
[INFO] Waiting for initial device list from the Android Debug Bridge
[INFO] Found 1 devices connected with the Android Debug Bridge
[INFO] android.device parameter set to Nexus_7_2012
```

```
[INFO] emulator-5554_Nexus_7_2012_unknown_sdk : Successfully installed C:\Users\jlalou\==PRIVATE==\GDrive\packt\6101\sources\helloWorldWithoutIDEA\target\chapter1.apk to emulator-5554_Nexus_7_2012_unknown_sdk
```

**Undeploy**

To undeploy the application, run:

```
mvn android:undeploy
```

The output contains:

```
[INFO] --- android-maven-plugin:3.8.2:undeploy (default-cli) @ helloWorld ---
[INFO] Waiting for initial device list from the Android Debug Bridge
[INFO] Found 1 devices connected with the Android Debug Bridge
[INFO] android.device parameter not set, using all attached devices
```

```
[INFO] Emulator emulator-5554_Nexus_7_2012_unknown_sdk found.
```
The application does not appear anymore.

At this point, you may not be convinced of the advantage of such goals. Anyway, step back and think of the industrialization of deployments and tests on a great diversity of devices.
Architecture principles
Now that we are able to build and deploy on an AVD, let's resume the development phase itself.

Standalone application
In "simple cases", the Android application has no contact with any other tier. The application is self-dependent. Therefore, the principles of architecture common to any other standalone application apply.

The application has to be divided based on functional criteria, for instance: domain or model, Data Access Object (DAO), service, and view. In Android terminology, views are activities. A graph of direct dependencies is shown in the following image:

In other terms, the view may have direct access to the services, but not to the DAO layer. Yet, any layer is aware of the model.

This best practice allows developing many applications built on common basic blocks, with a frontend and a behavior being totally different.

The management of external dependencies is regular: dependencies are defined mainly by the triplet groupId, artifactId, and version parameters; the dependency scope can be compile, provided, runtime, or system. If your application depends on another Android application, do no forget that the dependency type is apk, and not jar.
Since you add external dependencies to your Android application project, you should get the following error, repeated as many times as inner or anonymous classes appear in your dependency graph:

[INFO] warning: Ignoring InnerClasses attribute for an anonymous inner class
[INFO] (any.class.from.any.package) that doesn't come with an
[INFO] associated EnclosingMethod attribute. This class was probably produced by a
[INFO] compiler that did not target the modern .class file format. The recommended
[INFO] solution is to recompile the class from source, using an up-to-date compiler
[INFO] and without specifying any "-target" type options. The consequence of ignoring
[INFO] this warning is that reflective operations on this class will incorrectly
[INFO] indicate that it is *not* an inner class.

This error is not to worry about. If it does bore you, you can get the source code of the dependencies in the case of open source projects, and then include them in your own source code (anyway, beware of licenses such as General Public License (GPL) that may require you to make your application source code become GPL, too). More seriously, this problem means your project, or one of its dependencies or even sub-dependencies, requires Java 1.3 back-compatibility. As stated, you can ignore this warning.

As an example, let's consider the project com.packt.androidMaven.:sampleProject where you can download from here: https://github.com/ppapapetrou76/AndroidMavenDevelopment/tree/master/Chapter2. It is a regular project, with three submodules:

```xml
<modules>
  <module>model</module>
  <module>service</module>
  <module>AndroidTier</module>
</modules>
```
Starting the Development Phase

We haven’t discussed Maven modules so far; so, it's a good idea to explain their purpose. The idea behind modules is to split a large project into smaller functional pieces of projects. Each module should have a concrete sets of responsibilities and low level module should not be dependent to high level modules. When Maven builds a multimodule project, it uses a mechanism that is called a reactor.

Actually, here’s what maven does:

- Collects all the available modules to build
- Sorts the projects into the correct build order
- Builds the selected projects in order

Our first module, `model`, consists of a single file, representing an entity `Book`, as a Java bean, with mere properties, getters and setters, and an overridden `toString()` method:

```java
public class Book {
    private Integer id;
    private String title;
    private String format;
    private String color;
    private Integer numberOfPages;
    private Boolean brandNew;

    public Book() {
    }
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", title=" + title + \\
                "", format=" + format + \\
                ", color=" + color + \\
                ", numberOfPages=" + numberOfPages + \\
                ", brandNew=" + brandNew + \\
                "}";
    }
    // plus getters and setters
}
```

Module `service` depends only on `model` parameter:

```xml
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://maven.apache.org/POM/4.0.0"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
</project>
```
A service interface BookService is declared, with one method to implement:

Book createBook(String title, String format, String color, Integer numberOfPages);

The implementation creates an instance of Book class:

public class BookServiceImpl implements BookService {
    @Override
    public Book createBook(String title, String format, String color, Integer numberOfPages) {
        final Book book = new Book();
        book.setTitle(title);
        book.setFormat(format);
        book.setColor(color);
        book.setNumberOfPages(numberOfPages);
        return book;
    }
}

The Android application, com.packt.androidMaven:AndroidTier, depends on model and service parameter. We may have not declared the dependency on model parameter, because of the transitive and implicit dependencies induced because of service. Besides, the Android application obviously depends on com.google.android:android jar:

<dependencies>
<dependency>
    <groupId>${project.parent.groupId}</groupId>
    <artifactId>model</artifactId>
    <version>${project.parent.version}</version>
</dependency>
<dependencies>
Starting the Development Phase

```xml
<groupId>com.google.android</groupId>
<artifactId>android</artifactId>
<version>${platform.version}</version>
<scope>provided</scope>
</dependency>
<dependency>
  <!-- Add a dependency on 'model' -->
  <groupId>${project.groupId}</groupId>
  <artifactId>model</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <!-- Add a dependency on 'service' -->
  <groupId>${project.groupId}</groupId>
  <artifactId>service</artifactId>
  <version>${project.version}</version>
</dependency>
</dependencies>

The Android application consists of a single activity, calling the service and displaying the result:

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    final TextView textview;
    final Book book;
    book = bookService.createBook("Maven and Android", "eBook", "black", 150);
    textview = new TextView(this);
    textview.setText(book.toString());
    setContentView(textview);
}
```

You can build the project from the parent POM folder:

```bash
mvn clean install
```

Then, go to AndroidTier module folder. Start up Android emulator and deploy the APK:

```bash
mvn android:emulator-start
mvn android:deploy
```
Chapter 2

The result appears in the emulator as shown in the following screenshot:
Starting the Development Phase

Android application within an n-tier architecture

Standalone Android applications are more the exception than the rule: the trend for Android applications is to be part of two or three-tier architecture: this way, the Android application is only one view (among others), connected to one (or more) servers through HTTP (JSON or SOAP web services), linked to a backend such as a database, as shown in the following screenshot:

An Android application integrated within a larger project must follow the same rules as any other multi-tier project:

The Android application artifact should contain only the resource and source code specific to the application. In other terms, matching with Model-View-Controller (MVC) design pattern, concentrate the look and feel the information in the project that will generate the APK. Business treatments, common to the Android application and a web application for instance, must be factorized in a common business layer.
Many advantages and benefits result therefrom: first, the code is factorized. Second, you can move business intelligence and logic in a computer far more powerful than an Android device, which is, by definition, limited in CPU and RAM. Third, you protect your business algorithms from reverse-engineering: an Android APK, although encrypted and signed, remains breakable.

Android applications often are in dialog with a server. On development time, the emulator has to access the web server (in general, a servlet container such as Tomcat or Jetty) that is deployed on the same physical machine as the emulator. To access the local web server, getting localhost or 127.0.0.1 is not used: rather, use the IP 10.0.2.2 (do not forget the port number, such as the default 8080) which is mapped to the localhost on which the emulator is running.

Another best practice is to expose the entities and interfaces needed by the Android application in a minimal set of dependencies.

In the following schema, the Android application depends only on an artifact named contract, gathering entities and declared service interfaces. You may notice that the service implementation web application artifacts also depend only on contract as well as on dao:
Here is how Maven manages such a situation:

The parent POM contains a `<packaging>pom</packaging>` tag and declares its submodules:

```xml
<modules>
  <module>entity</module>
  <module>service</module>
  <module>contract</module>
  <module>dao</module>
  <module>service-impl</module>
  <module>webtier</module>
  <module>AndroidTier</module>
</modules>
```

The `entity` parameter has no special dependency, unlike `service` parameter that depends on `entity` parameter.

`contract` declares itself as `pom`, and propagates the two dependencies onto `entity` and `service` parameters:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.packt.androidMaven.chapter2</groupId>
    <artifactId>multitiers</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <packaging>pom</packaging>
  <artifactId>contract</artifactId>
  <version>1.0-SNAPSHOT</version>
  <!-- This POM declares and propagates two dependencies -->
  <dependencies>
    <dependency>
      <groupId>com.packt.androidMaven.chapter2</groupId>
      <artifactId>entity</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>com.packt.androidMaven.chapter2</groupId>
      <artifactId>service</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```
<version>1.0-SNAPSHOT</version>
</dependency>
</dependencies>
</project>

AndroidTier and webtier parameters behave in a similar manner: they depend only on `contract.service-impl` declares a dependency on `dao` parameter in more of `contract` parameter. Beware! The dependency on the `contract` parameter must be declared with type `pom`:

```xml
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>contract</artifactId>
  <version>1.0-SNAPSHOT</version>
  <type>pom</type>
</dependency>
```

Android with dependencies on SDK Add-ons

You may need some dependencies, external of core Android, but yet linked to it. The most common case is when you depend on Google Maps. The JAR file of Google Maps is installed with the other add-ons by default install configuration.

If your project depends on Google Maps (or, similarly, to `usb` or `effects` JARs), firstly you have to install the dependencies in your local repository:

```bash
mvn install:install-file \
-Dfile=%ANDROID_HOME%\add-ons\addon-google_apis-google-19\libs \
-DgroupId=com.google.android \
-DartifactId=maps \
-Dversion=19_r3 \
-Dpackaging=jar \
-DgeneratePom=true
```

Then, declare the dependency, specifying the scope as provided:

```xml
<dependency>
  <groupId>com.google.android.maps</groupId>
  <artifactId>maps</artifactId>
  <version>19_r3</version>
  <scope>provided</scope>
</dependency>
```
The `<version>` tag of artifact maps differs from that of "Android core" JAR: 19_r3 versus 4.4.2_r3. In other terms, for Android add-ons, the version to write follows the API level, whereas "Android core" follows the release version. In a further chapter, we will deal with maven-android-sdk-deployer plugin, which eases the installation process of Android dependencies.

**Summary**

Thus, in this second chapter, we learned or revised concepts related to Android SDK and tools. We reviewed how to create AVDs in the GUI and in the command line. We saw how to start/stop an emulator from Maven, and how to clean, build, deploy, and undeploy an APK onto an emulator. Finally, we described best practices in the architecture of Android projects managed by Maven.

In the next chapter, we will begin exploring unit testing.
Where to buy this book

You can buy Android Application Development with Maven from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.