50 Android Hacks

Carlos Sessa

Foreword by Jake Wharton

MANNING

Sample Chapter

50 Android Hacks

by Carlos Sessa

Chapter 12

brief contents

12 Building tools

Building software applications often requires custom processes such as adding dependencies, running tests, and deploying in a server. If building from Eclipse feels a bit limiting, you'll find this chapter interesting. We'll cover tips that provide some alternatives for building your applications.

Hack 48 *Handling dependencies with Apache Maven*

Android v1.6+

The Android SDK comes with a lot of classes and code that help you create your applications, but sometimes even this isn't enough. For example, if you want to add Google Analytics or you want to add a JSON parser, you'll have to add some kind of dependencies. The Android SDK doesn't provide a way to handle dependencies, other than placing JAR files in the /libs folder. Fortunately, it has other building tools. Even if you don't use third-party dependencies, you might want to separate your application in different modules and add dependencies between them in order to organize your code or create reusable components. What you can do to get around this issue is to use Apache Maven. In this hack you'll see how to use Apache Maven to build your application and run tests.

If you've used Maven for Java application dependencies, you'll agree that it's a powerful tool, but it takes some time to get used to it. In this case, we'll take a look at Manfred Moser's roboguice-calculator demo. In this project, Manfred used different dependencies, making it an excellent example to demonstrate how Maven works.

To understand how Maven works, we'll go through the different pom.xml sections. The pom.xml is the only Maven-related file your project will have. In it you'll tell Maven your application name, the build dependencies, the test dependencies, and how to create your APK. Maven first checks if you have the dependencies in the local repository, which is located at ~/.m2/repository by default. If they're not there, it will take care of downloading them from a central repository.

The first part has the following code:

```
<?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>org.roboguice</groupId>
  <artifactId>calculator</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>apk</packaging>
  <name>calculator</name>
```

As with every XML file, start with schemas and namespaces

groupId, artifactId, version, and packaging establish unique identifier for artifact in repository, and in general (like coordinates)

The final build will end up in $MVN_REPO/groupId/artifactId/version. The common example is to use the `groupId` as your project name and the `artifactId` as your module name. In this particular case, Manfred had used `org.roboguice` as `groupId` because it's an example for the roboguice project. The `artifactId`, `calculator`, identifies this example inside the project.

The last two attributes from this section are the `packaging` and the `name`. The `packaging` tells Maven the final output. Although the default is `jar`, Manfred had picked `apk` because he needs an Android application. The `name` in conjunction with the `version` will determine the output filename.

The second section to analyze is dependencies. Because the dependencies list is long, we'll analyze only a few of them. The dependencies section is the following:

```
<dependencies>
    <dependency>
      <groupId>org.roboguice</groupId>
      <artifactId>roboguice</artifactId>
      <version>2.0-SNAPSHOT</version>
    </dependency>

    ...

    <dependency>
      <groupId>com.google.android</groupId>
      <artifactId>android</artifactId>
      <version>2.3.3</version>
      <scope>provided</scope>
    </dependency>
```

1 Roboguice dependency

2 Android dependency

```
    ...
    <dependency>
      <groupId>com.pivotallabs</groupId>
      <artifactId>robolectric</artifactId>
      <version>1.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

❸ **Robolectric dependency**

Every dependency has four important attributes, `groupId`, `artifactId`, `version`, and `scope`. The first dependency is roboguice ❶. It has a `groupId`, `artifactId`, and `version`, which corresponds to a released version in some Maven repository. Remember what we learned in the first section? That information is required if someone needs to use your artifact as a dependency.

Although the roboguice dependency doesn't contain the `scope` attribute, you should know that `compile` is the default value. Compile dependencies are available in all classpaths of a project because they get included in the APK.

The next dependency is Android itself ❷. When you use Maven to build Android applications, you must always have Android as a dependency, but its `scope` is `provided`. `provided` is much like `compile`, but it indicates that you expect the JDK or a container to provide the dependency at runtime—in our case, the device running Android.

The last dependency is robolectric ❸. Robolectric is a test framework, so we only need that dependency when we're compiling/running the tests. That's what the `test` scope is for. This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.

After the dependencies section in the pom.xml file, we have the `build` section, which has the `plugins` section inside. This is where you'll configure the Android Maven plugin. Let's take a look at the following code to see how it's done:

```
<build>
    <plugins>
      <plugin>
        <groupId>
          com.jayway.Maven.plugins.android.generation2
        </groupId>
        <artifactId>
          android-Maven-plugin
        </artifactId>
        <version>
          3.0.0-SNAPSHOT
        </version>
        <configuration>
           <androidManifestFile>
            ${project.basedir}/AndroidManifest.xml
          </androidManifestFile>
          <assetsDirectory>
            ${project.basedir}/assets
          </assetsDirectory>
          <resourceDirectory>
```

❶ **groupId, artifactId, and version for android-Maven-plugin**

❷ **android-Maven-plugin configuration**

```
            ${project.basedir}/res
          </resourceDirectory>

          <sdk>
            <platform>10</platform>
          </sdk>
          <undeployBeforeDeploy>
            true
          </undeployBeforeDeploy>
        </configuration>
        <extensions>true</extensions>
      </plugin>

      ...

    </plugins>
  </build>
```

Build plugins works in a way similar to dependencies. The previous code shows how the android-Maven-plugin gets configured ❶. If we were configuring a dependency, we'd need to provide a `groupId`, an `artifactId`, and a `version`.

You'll notice that Apache Maven follows the convention-over-configuration paradigm, which results in decreasing the number of decisions that developers need to make, gaining simplicity, but not necessarily losing flexibility. A great example of this approach can be seen where the android-Maven-plugin gets configured ❷. You might want to place the AndroidManifest.xml somewhere else so you have an attribute to modify the default location.

When the pom.xml is ready, you can treat your Android application as a Maven artifact. If you run the `mvn package`, you'll get a target directory with the APK inside. If you want to get the application installed in all attached devices, you can run `mvn android:deploy`.

48.1 The bottom line

Apache Maven is a great build tool. It's true that it's somewhat complicated the first time you use it, but after you understand how it works, you'll start to create a project by generating the pom.xml file.

The best way to learn about it is to read how someone else is using it. For example, you can examine the roboguice's pom.xml. You'll notice it's not hard at all.

48.2 External links

http://maven.apache.org/
https://github.com/mosabua/roboguice-calculator
http://code.google.com/p/maven-android-plugin/
https://github.com/roboguice/roboguice
www.robolectric.org
http://en.wikipedia.org/wiki/Convention_over_Configuration
www.simpligility.com

Hack 49 *Installing dependencies in a rooted device*
Android v1.6+

Android applications are commonly written in a dialect of Java and compiled to byte-code. Then they're converted from Java Virtual Machine–compatible .class files to Dalvik-compatible .dex files before installation on a device. Figure 49.1 (see section 49.5) illustrates the building process.

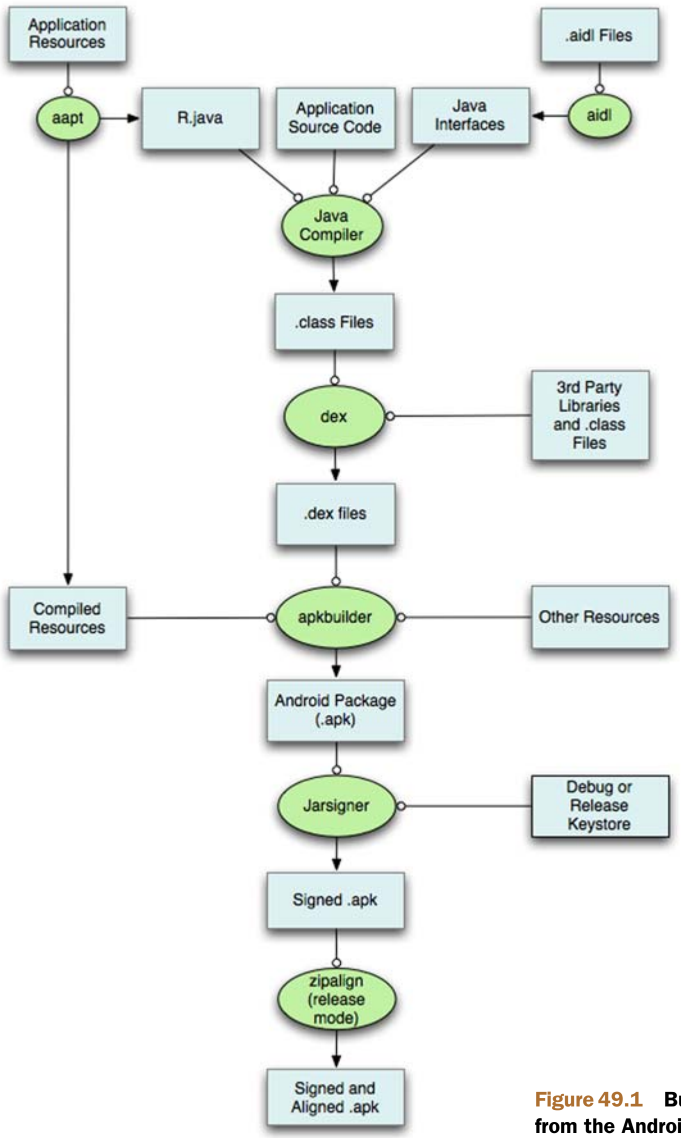


Figure 49.1 Building process taken from the Android documentation

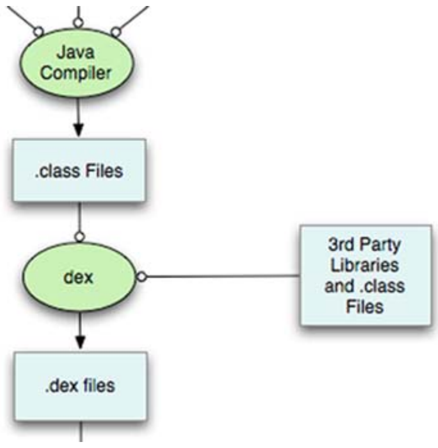


Figure 49.2 **Compilation procedure**

Apart from the Android SDK, many third-party libraries are available that we can use as dependencies. These dependencies can be useful for improving your application functionality, code organization, customs views, and so on. As we add dependencies to our application, we might notice the build time increases. Android supports adding JAR dependencies, but it first needs to convert the JAR file's .class files to .dex every time we want to build, and this takes time. From our earlier figure, we narrow our focus to this sequence in figure 49.2.

To give you an idea of how you can solve this, have you ever used Google's map library in Android? Remember how you added that dependency? The map library can be used from your application, but you never lose time indexing it. That's because the library is already installed on your device/emulator.

In this hack, we'll use the same approach, but with other libraries. We'll see how to install those dependencies in our developing device to make our build times faster, avoiding the dexing phase of the dependencies.

The first thing to understand from this hack is that we're installing dependencies on a rooted device. This means that this approach won't work for production. We're doing it to make our developing build times faster.

49.1 Predexing

The first step is predexing the dependencies. This means converting the JARs to dex. It can be done with the `dx` application inside the ANDROID_SDK/tools folder. For example, if our dependency is called `dep.jar`, we'll need to use the following line:

```
dx -JXmx1024M -JXms1024M -JXss4M
  --no-optimize --debug --dex
  --output=./dep_dex.jar dep.jar
```

The dep_dex.jar is the file that we'll upload to the device.

49.2 Creating the permissions XML

The second step is to create XML for each dependency with the permission to the library. If we think back to the Google maps dependency, when we want to use it we need to add a `use-library` tag in our AndroidManifest.xml file. The XML we'll create will be used for that specific line. Let's see an example:

```
<permissions>
  <library name="dep"                        ← ❶ Specifies library name
     file="/data/data/com.dep.package/files/dep_dex.jar"/>   ← ❷ Writes path for predexed file
 </permissions>
```

We first need to specify the library name ❶. This library name is the string that we should place in the `use-library` tag. We also need to write down the path for the predexed file inside the device ❷. We can upload the predexed file using `adb` or using an Android application. An example of an application doing the installation is placed in the sample code. The application is a modification of Johannes Rudolph's scala-android-libs source code.

49.3 Modifying AndroidManifest.xml

The last step is to modify the AndroidManifest.xml file to use the dependencies installed in the device. The example for the `dep` mentioned previously would be like the following:

```
<uses-library name="dep"/>
```

That's it. We're now using dependencies from the device instead of compiling them every time we want to run the application. Remember to change the build tool to avoid compiling the dependencies. For instance, in Apache Maven we can set the `scope` to `provided`.

49.4 The bottom line

Installing dependencies is a great way to improve your application build time. I've been using it for some applications and I'm getting them built twice as fast.

Although this hack is useful, two things might bother you. First, you need a rooted device. Unfortunately, not all the Android devices are rootable. You'll also need to modify your build script to avoid this behavior when you're targeting production. Apache Maven would be a useful tool to handle different types of builds.

49.5 External links

http://developer.android.com/tools/building/index.html
https://github.com/scala-android-libs/scala-android-libs
http://android-argentina.blogspot.com/2011/11/roboinstaller-install-roboguice.html

Hack 50 Using Jenkins to deal with device diversity

Android v1.6+

Contributed by Christopher Orr

Testing Android applications can be tough. With hundreds of manufacturers producing thousands of unique Android models, a device is available to suit nearly every need. But for software developers, this ubiquity represents a challenge: how to ensure your application works well on all of these devices, and across a variety of screen sizes, hardware configurations, and Android OS versions.

Buying hundreds of devices to develop and test isn't feasible. Thankfully, Android provides a great resource system that enables you to support a diversity of devices and OS versions with a single application package. But verifying that you've used this system correctly requires a lot of testing: Did you mistype a view ID in your layout XML for `layout-xhdpi-land`? Are you missing a string parameter in one of the Japanese translations? With the bundled SQLite version often changing between Android releases, have you written a SQL query that works only on certain versions?

Testing your application on a few chosen devices—whether manually or using your automated test suite—is a possibility, but it's time-consuming and quickly becomes impractical as your application grows, adding more features plus support for further screen densities, device classes, and languages.

To reduce this burden, in this hack you'll automatically generate multiple Android emulators with various software and hardware properties and run your automated test suite on a number of them, allowing you to pinpoint potential problems on certain device configurations.

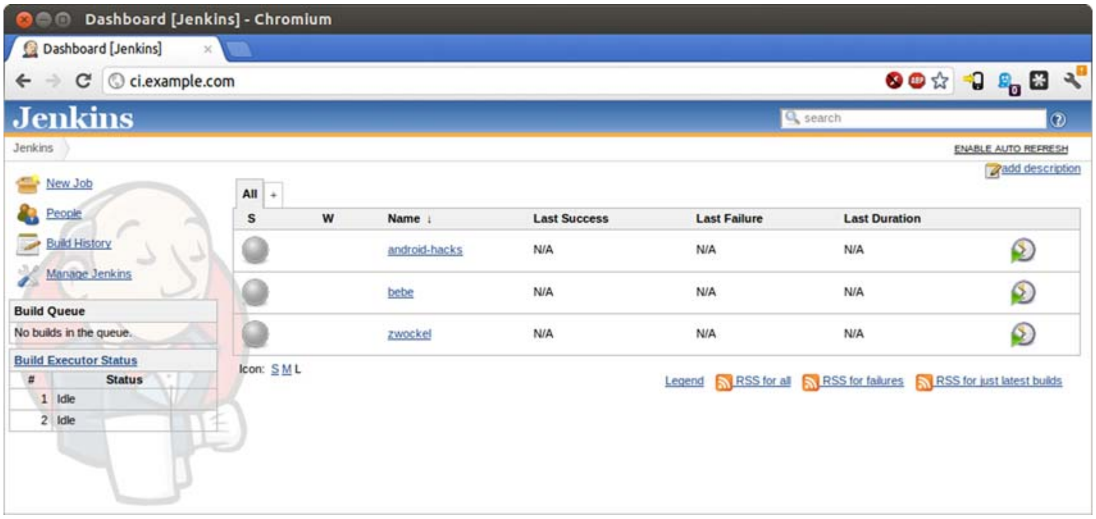
Although emulators can't fully replace testing on real hardware, they're a fast and flexible way to test how your application copes with a variety of hardware properties, such as whether the device has a front camera, is missing an SD card, has a hardware keyboard, is equipped with limited RAM, and so on.

You'll use a piece of software called Jenkins—a popular, open source continuous integration server, along with its Android Emulator plugin. The web-based dashboard of Jenkins can be seen in figure 50.1.

The strategy for this hack is to create a Jenkins "matrix" job and, for every check-in of your source code, you'll let Jenkins build your application, automatically generate some emulators, run your automated test suite on each of them, and then report on the results.

If you don't have an automated test suite already, you can create one relatively quickly using a library like Robotium—even starting with a few rudimentary smoke tests is helpful, such as ensuring that a few key activities open and that the expected UI elements are shown.

Assuming you have Jenkins running with the Android Emulator plugin installed, with a code repository containing both your application and test code that can be



Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration
●		android-hacks	N/A	N/A	N/A
●		bebe	N/A	N/A	N/A
●		zwockel	N/A	N/A	N/A

Icon: S M L

Legend RSS for all RSS for failures RSS for just latest builds

Figure 50.1 **Jenkins dashboard UI**

accessed by Jenkins (all of which is available in the sample code for this hack), the first thing to do is to choose the set of emulated devices you want to test with. As a minimum, you should test on each major Android OS version between your `minSdkVersion` and the latest version available. Other factors to think about are screen density, supported locales, and any hardware properties that are important to your application (e.g., camera, accelerometer).

50.1 Creating a Jenkins job

In Jenkins, click New Job, enter a job name, and select Build Multi-configuration Project" (also known as a "matrix" job) and click OK. Matrix jobs allow you to run the same set of steps—in your case, starting an Android emulator, building an application, and testing it—but with slight differences in configuration each time, such as changing the OS version used by the emulator.

In the job configuration, first enter the Source Code Management information to let Jenkins check out your application and test the code repository. Depending on the source control system you use, this may require you to install an extra plugin, such as the Git or Subversion plugin, via Jenkins' built-in plugin manager.

So that Jenkins monitors your repository for changes, enable the Build Periodically option and enter a cron-style syntax; for example, to poll for changes every two minutes on weekdays enter this:

```
*/2 * * * 1-5
```

Under the Configuration Matrix heading, click Add Axis, choose User-defined Axis, and in the Name field enter `os`. As the values, enter the following:

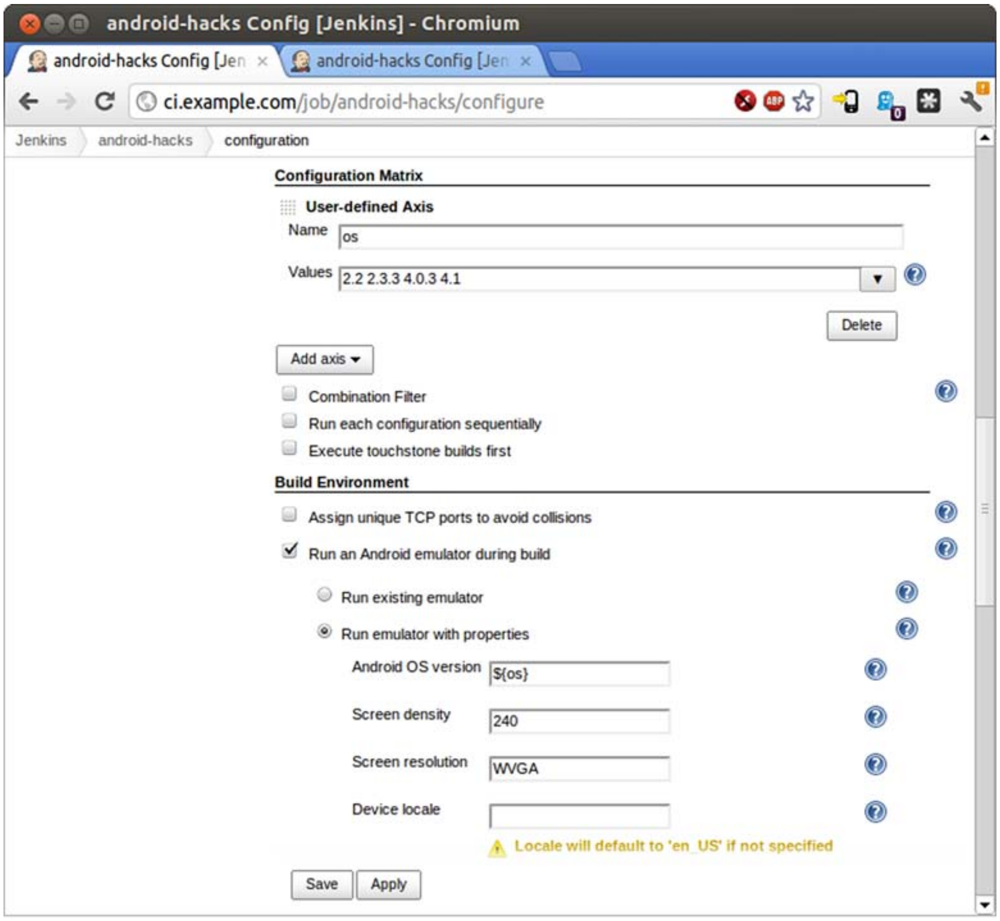
```
2.2 2.3.3 4.0.3 4.1
```

As you might be thinking, each value represents an Android version to test on. You could later add further axes for screen density, locales, and so on, but for now let's stick with just one. By entering four distinct values here, Jenkins will run four individual builds each time you start this job, with each build seeing a different value in the `os` environment variable.

Next, click Run an Android Emulator During Build, and enter the following values under Run Emulator with Properties:

- Android OS version: `${os}`
- Screen density: `240`
- Screen resolution: `WVGA`

You can leave the other fields unchanged, but you should uncheck the Show Emulator Window option. By setting the value `${os}` as the Android version, this ensures a different Android emulator will be created in each of the four builds that will occur. The complete configuration can be seen in figure 50.2.



android-hacks Config [Jenkins] - Chromium

ci.example.com/job/android-hacks/configure

Jenkins › android-hacks › configuration

Configuration Matrix

User-defined Axis

Name: os

Values: 2.2 2.3.3 4.0.3 4.1

Delete

Add axis

☐ Combination Filter

☐ Run each configuration sequentially

☐ Execute touchstone builds first

Build Environment

☐ Assign unique TCP ports to avoid collisions

☑ Run an Android emulator during build

○ Run existing emulator

◉ Run emulator with properties

Android OS version: ${os}

Screen density: 240

Screen resolution: WVGA

Device locale:

Locale will default to 'en_US' if not specified

Save Apply

Figure 50.2 **Configuring the axes and the emulator to create**

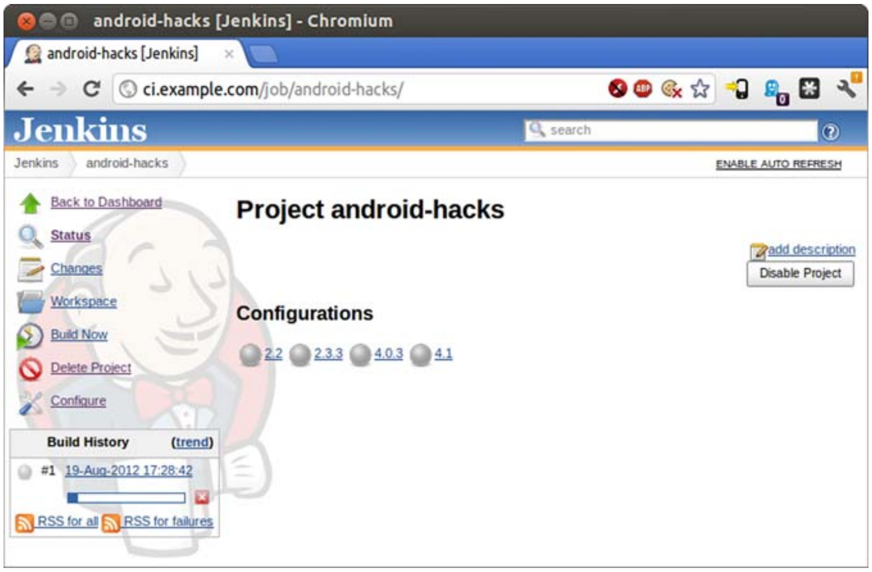


Figure 50.3 Project page showing the configurations and a build in progress

In the Build section, add the build steps Install Android Project Prerequisites and Invoke Ant, assuming that you have used the android tool to generate Ant build scripts for your application and test projects. As the targets, enter `clean debug install test`. Click Advanced, and for the build file enter `tests/build.xml` (assuming `tests` is the directory name you've used for your test suite). Add a property: `sdk.dir=$ANDROID_HOME`.

If you have your Android test suite configured to output results in JUnit XML format (e.g., using the android-junit-report project), you can also check the Publish JUnit Test Result Report option under the Post-build Actions section.

Press Save to finalize the job configuration. You now have a Jenkins job that will run multiple times, each time checking out your source code, starting a different Android emulator, and then building your application and running its test suite. The job page should look like figure 50.3, with each ball representing one configuration (that is, OS version). They're gray to indicate that a build hasn't yet occurred.

50.2 Running the job

Click Build Now on the left side of the job page and, after a few seconds, you'll see a couple of the balls start to flash to indicate that a couple of the configurations are building.

Meanwhile, you can observe the build in progress by clicking on one of the flashing balls, and then clicking the blue progress bar on the left. This shows the Console Output, revealing that the source code has been checked out, an emulator has been automatically generated, and that Jenkins is waiting for the emulator to boot up.

By default, Jenkins runs two builds in parallel, so you'll have to wait a few minutes before everything completes. In any case, the first builds will take a little longer as the emulators have to be generated and booted for the first time. Furthermore, if you don't have the Android SDK installed on the machine where Jenkins is running, it will be automatically installed for you, which will add to the initial build time.

When the progress bars disappear from the Jenkins sidebar, the build is complete.

So within a few minutes you've automatically tested your software on four different versions of Android—and Jenkins will continue to do this automatically each time it finds a new commit in your code repository.

After you have the basics running, you can refine your Jenkins job configuration by adding further axes. For example, add an axis for different screen resolutions, allowing you to automatically create emulators to test layouts designed for different phone or tablet devices.

The Android Emulator plugin also lets you run the Android monkey tool to stress-test your UI. You could set up a Jenkins job that runs nightly, rather than for every commit, and that builds your APK, installs it onto an emulator, and then runs monkey against your application to check for instabilities.

50.3 The bottom line

Running your Android tests automatically means you can spend a lot less time manually testing your applications and lets you have greater confidence in the quality of your applications.

The samples for this hack include a basic Android application, test suite, and pre-configured Jenkins installation with which you can experiment.

Because Jenkins isn't only for automated testing, you can go beyond the basics of this hack and do things like integrating monkey testing into your workflow, check and monitor Android lint issues over time, automatically sign your APK, publish beta builds to a web server for testers, and much more.

50.4 External links

http://opensignalmaps.com/reports/fragmentation.php
http://jenkins-ci.org/
https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin
https://wiki.jenkins-ci.org/display/JENKINS/Android+Lint+Plugin
https://github.com/jsankey/android-junit-report

ANDROID/MOBILE

50 Android Hacks

Carlos Sessa

Hacks. Clever programming techniques to solve thorny little problems. Ten lines of code that save you two days of work. The little gems you learn from the old guy in the next cube or from the geniuses on Stack Overflow. That's just what you'll find in this compact and infinitely useful book.

The name **50 Android Hacks** says it all. Ranging from the mundane to the spectacular, each self-contained, fully illustrated hack is just a couple pages long and includes annotated source code. These practical techniques are organized into twelve collections covering layout, animations, patterns, and more.

What's Inside

- Hack 3 Creating a custom ViewGroup
- Hack 8 Slideshow using the Ken Burns effect
- Hack 20 The Model-View-Presenter pattern
- Hack 23 The SyncAdapter pattern
- Hack 31 Aspect-oriented programming in Android
- Hack 34 Using Scala inside Android
- Hack 43 Batching database operations
- Plus 43 more!

Most hacks work with Android 2.x and greater. Version-specific hacks are clearly marked.

Carlos Sessa is a passionate professional Android developer. He's active on Stack Overflow and is an avid hack collector.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/50AndroidHacks

Free eBook
SEE INSERT

"How to solve common problems that arise in Android development."
—From the Foreword by Jake Wharton, Android Engineer

"One of the best how-to books I've read!"
—Christian Badenas
Android and .NET Developer

"Packed with useful Android development tidbits not found in the official documentation."
—Matthias Käppler, SoundCloud

"A great resource for creating nontrivial user experiences for the Android platform."
—Frank Ableson
Coauthor of *Android in Action*

ISBN 13: 978-1-617290-56-5
ISBN 10: 1-617290-56-4
53499
9781617290565

MANNING $34.99 / Can $36.99 [INCLUDING eBOOK]