SAMPLE CHAPTER

Gradle IN ACTION

Benjamin Muschko

FOREWORD BY Hans Dockter

MANNING

Gradle in Action

by Benjamin Muschko

Chapter 9

brief contents

9 Integration and migration

This chapter covers

- Importing existing Ant projects
- Using Ant tasks from Gradle
- Converting Maven `pom.xml` into Gradle projects
- Migration strategies from Ant and Maven to Gradle
- Comparing build outcomes and upgrade testing

Long-running development projects are usually heavily invested in established build tool infrastructure and logic. As one of the first build tools, Gradle acknowledges that moving to a new system requires strategic planning, knowledge transfer, and acquisition, while at the same time ensuring an unobstructed build and delivery process. Gradle provides powerful tooling to integrate existing build logic and alleviate a migration endeavor.

If you're a Java developer, you likely have at least some experience with another build tool. Many of us have worked with Ant and Maven, either by choice or because the project we're working on has been using it for years. If you decide to move to Gradle as your primary build tool, you don't have to throw your existing knowledge overboard or rewrite all the existing build logic. In this chapter, we'll

look at how Gradle integrates with Ant and Maven. We'll also explore migration strategies to use if you decide to go with Gradle long-term.

Ant users have the best options by far for integrating with Gradle. The Gradle runtime contains the standard Ant libraries. Through the helper class `AntBuilder`, which is available to all Gradle build scripts, any standard Ant task can be used with a Groovy builder-style markup, similar to the way you're used to in XML. Gradle can also be pointed to an existing Ant build script and can reuse its targets, properties, and paths. This allows for smooth migrations in baby steps as you pick and choose which of your existing Ant build logic you want to reuse or rewrite.

The migration path from Maven to Gradle isn't as easy. At the time of writing, a deep integration with existing Maven project object model (POM) files isn't supported. To get you started, Gradle provides a conversion tool for translating a Maven `pom.xml` into a `build.gradle` file. Whether you're migrating an existing Maven build or starting from scratch, Maven repositories are ubiquitous throughout the build tool landscape. Gradle's Maven plugin allows for publishing artifacts to local and remote Maven repositories.

Migrating a build process from one build tool to another is a strategic, mission-critical endeavor. The end result should be a comparable, functional, and reliable build, without disruption to the software delivery process. On a smaller scale, migrating from one Gradle version to another can be as important. Gradle provides a plugin that compares the binary output of two builds—before and after the upgrade—and can make a deterministic statement about the result before the code is changed and checked into version control. In this chapter, you'll learn how to use the plugin to upgrade your To Do application from one Gradle version to another. Let's start by taking a closer look at Gradle's Ant capabilities.

9.1 Ant and Gradle

Gradle understands Ant syntax on two levels. On the one hand, it can import an existing Ant script and directly use Ant constructs from a Gradle build script as if they're native Gradle language elements. This type of integration between Gradle and Ant doesn't require any additional change to your Ant build. On the other hand, familiar Ant tasks (for example, Copy, FTP, and so on) can be used within your Gradle build script without importing any Ant script or additional dependency. Long-term Ant users will find themselves right at home and can reuse familiar Ant functionality with a convenient and easy-to-learn Groovy DSL notation. Chapter 2 demonstrated how to use the Echo task within a Gradle build.

Central to both approaches is the Groovy `groovy.util.AntBuilder`, a class packaged with the Groovy runtime. It allows for using Ant capabilities directly from Groovy in a concise fashion. Gradle augments the Groovy `AntBuilder` implementation by adding new methods. It does so by providing the class `org.gradle.api.AntBuilder`, which extends Groovy's `AntBuilder` implementation, as shown in figure 9.1.

An instance of the class `org.gradle.api.AntBuilder` is implicitly available to all Gradle projects, as well as to every class that extends `DefaultTask` through the property

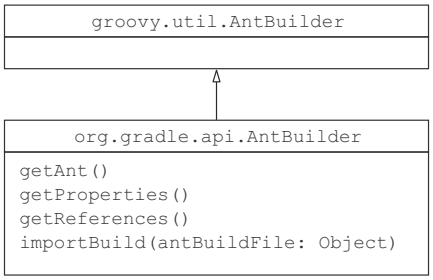

Figure 9.1 Access to Ant functionality from Gradle is provided through the class `AntBuilder`

`ant`. In regular class files that don't have access to a project or task, you can create a new instance. The following code snippet demonstrates how to do that in a Groovy class:

```
def ant = new org.gradle.api.AntBuilder()
```

Let's look at some use cases for Gradle's `AntBuilder`. You're going to take the Ant build script from chapter 1, import it into a Gradle build script, reuse its functionality, and even learn how to manipulate it.

9.1.1 Using Ant script functionality from Gradle

Importing an Ant script and reusing its functionality from Gradle is dead simple. All you need to do is use the method `importBuild` from Gradle's `AntBuilder` and provide it with the target Ant build script, as shown in figure 9.2.

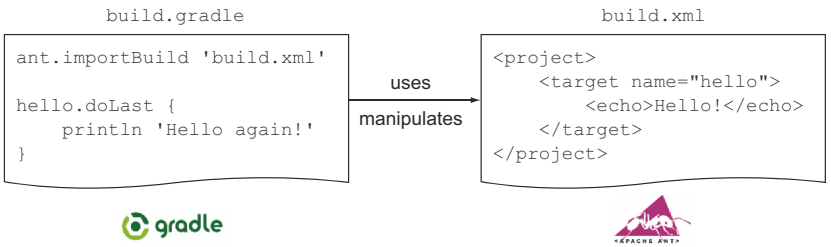

Figure 9.2 Importing an existing Ant script into Gradle

To see the import functionality, you'll take the directory structure of your Ant build from chapter 1 and put it under the directory name `ant`. Parallel to this directory, create another directory named `gradle`, which holds your Gradle build script responsible for importing the Ant script. The end result should look similar to the following directory tree:

```
.
├── ant
│   ├── build.xml
│   ├── lib
│   │   └── commons-lang3-3.1.jar
│   └── src
│       └── main
```

Original Ant build script including multiple properties and tasks

```
│           └── java
│               └── com
│                   └── mycompany
│                       └── app
│                           └── Main.java
└── gradle
    └── build.gradle
```

Gradle build script consuming existing Ant build script

Let's look at the Ant build script shown in listing 9.1. It's a simple script for compiling Java source code and creating a JAR file from the class files. Required external libraries are stored in the directory `lib`. This directory only holds a single library, the Apache Commons language API. The script also defines a target for initializing the build output directory. Another target named `clean` makes sure that existing class and JAR files can be deleted.

Listing 9.1 Original Ant build script

```
<project name="my-app" default="dist" basedir=".">
   <property name="src" location="src"/>
   <property name="build" location="build"/>
   <property name="lib" location="lib"/>
   <property name="dist" location="dist"/>
   <property name="version" value="1.0"/>

   <target name="init">
      <mkdir dir="${build}"/>
   </target>

   <target name="compile" depends="init" description="compile the source">
      <javac srcdir="${src}" destdir="${build}"
             ➥ classpath="${lib}/commons-lang3-3.1.jar"
             ➥ includeantruntime="false"/>
   </target>

   <target name="dist" depends="compile"
           ➥ description="generate the distribution">
      <mkdir dir="${dist}"/>
      <jar jarfile="${dist}/my-app-${version}.jar" basedir="${build}"/>
   </target>

   <target name="clean" description="clean up">
      <delete dir="${build}"/>
      <delete dir="${dist}"/>
   </target>
</project>
```

Ant properties for defining directories and the version of your JAR file

Ant tasks for compiling Java source code and creating the JAR file

Now, let's look at the Gradle build file. This is where the `AntBuilder` comes into play. Use the implicit property named `ant`, call the method `importBuild`, and provide the path to the Ant build script, as shown in the following code snippet:

```
ant.importBuild '../ant/build.xml'
```

Listing the tasks available to the Gradle build script reveals that Ant targets are treated as Gradle tasks:

```
$ gradle tasks --all
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Help tasks
----------
...

Other tasks
-----------
clean - clean up
dist - generate the distribution
compile - compile the source
init
```

Top-level Ant targets are listed.

Ant targets with dependencies are shown in indented form.

As shown in the command-line output, Gradle inspects the available Ant targets, wraps them with Gradle tasks, reuses their description, and even keeps their dependencies intact. You can now execute the translated Ant targets as you're used to in Gradle:

```
$ gradle dist
:init
:compile
:dist
```

After executing the `dist` Ant target, you'll find that the source code was compiled and the JAR file was created—exactly as if running the script directly from Ant. The following directory tree shows the end result:

```
.
├── ant
│   ├── build
│   │   └── com
│   │       └── mycompany
│   │           └── app
│   │               └── Main.class
│   ├── build.xml
│   ├── dist
│   │   └── my-app-1.0.jar
│   ├── lib
│   │   └── commons-lang3-3.1.jar
│   └── src
│       └── main
│           └── java
│               └── com
│                   └── mycompany
│                       └── app
│                           └── Main.java
└── gradle
    └── build.gradle
```

Java class file

Created JAR file

Accessing imported Ant properties and paths from Gradle

Ant targets translate one-to-one to Gradle tasks and can be invoked from Gradle with exactly the same name. This makes for a very fluent interface between both build tools. The same can't be said about Ant properties or paths. To access them, you'll need to use the methods `getProperties()` and `getReferences()` from Gradle's `AntBuilder` reference. Keep in mind that the Gradle task `properties` won't list any Ant properties.

Importing an Ant script into Gradle can be a first step toward a full migration to Gradle. In section 9.1.3, we'll discuss various approaches in more detail. The command-line output of Gradle tasks wrapping Ant targets is pretty sparse. As an Ant user, you may want to see the information you're used to seeing when executing the targets from Ant. Next, we'll see how to ease that pain.

LOGGING ANT TASK OUTPUT

At any time, you can render the Ant task output from Gradle by executing the Gradle build with the INFO log level (`-i` command-line parameter). As always, this command-line parameter renders more information than you actually want to see. Instead of using this command-line parameter, you can directly change the logging level for the Gradle task that wraps the Ant target. The following assignment changes the logging level to INFO for all Ant targets:

```
[init, compile, dist, clean]*.logging*.level = LogLevel.INFO
```

Listing imported Ant targets manually can be tedious and error-prone. Unfortunately, there's no easy way around it, because you can't distinguish the origin of a task. Run the `dist` task again to see the appropriate output from Ant:

```
$ gradle dist
:init
[ant:mkdir] Created dir: /Users/Ben/books/gradle-in-action/code/
➥ ant-import/ant/build
:compile
[ant:javac] Compiling 1 source file to /Users/Ben/books/
➥ gradle-in-action/code/ant-import/ant/build
:dist
[ant:mkdir] Created dir: /Users/Ben/books/gradle-in-action/code/
➥ ant-import/ant/dist
[ant:jar] Building jar: /Users/Ben/books/gradle-in-action/code/
➥ ant-import/ant/dist/my-app-1.0.jar
```

You'll see the output from Ant that you're familiar with. To indicate that the output originates from Ant, Gradle prepends the message `[ant:<ant_task_name>]`.

Gradle's integration with Ant doesn't stop here. Often you'll want to further modify the original Ant script functionality or even extend it. This could be the case if you're planning a gradual transition from your existing Ant script to Gradle. Let's look at some options.

MODIFYING ANT TARGET BEHAVIOR

When you import an existing Ant script, its targets are effectively treated as Gradle tasks. In turn, you can make good use of all of their features. Remember when we discussed adding actions to existing Gradle tasks in chapter 4? You can apply the same behavior to imported Ant targets by declaring `doFirst` and `doLast` actions. The following listing demonstrates how to apply this concept by adding log messages to Ant target `init` before and after the actual Ant target logic is executed.

Listing 9.2 Adding behavior to existing Ant target functionality

```
init {
   doFirst {
      logger.quiet "Deleting the directory '${ant.properties.build}'."
   }

   doLast {
      logger.quiet "Starting from a clean slate."b
   }
}
```

Adds a Gradle action executed before any Ant target code is run

Adds a Gradle action that's executed after Ant target code is run

Renders a message to inform the user about the directory you're about to delete by accessing Ant property `build`

Now when you execute the task `init`, the appropriate messages are rendered in the terminal:

```
$ gradle init
:init
Deleting the directory '/Users/Ben/Dev/books/gradle-in-action/
➥ code/ant-import/ant/build'.
[ant:mkdir] Created dir: /Users/Ben/Dev/books/gradle-in-action/
➥ code/ant-import/ant/build
Starting from a clean slate.
```

Importing Ant targets into a Gradle build is often only the starting point when working in the setting of a conjunct build. You may also want to extend the existing model by functionality defined in the Gradle build script. In listing 9.2, you saw how to access the Ant property `build` via the `AntBuilder` method `getProperties()`. Imported Ant properties aren't static entities. You can even change the value of an Ant property to make it fit your needs. You can also make changes to the task graph by hooking in new tasks. With the regular instruments of Gradle's API, a dependency can be defined between an Ant target and a Gradle task or vice versa.

Let's look at code that pulls together all of these concepts in a concise example. In the next listing, you'll make heavy use of existing Ant properties, change the value of an existing Ant property, and let an imported Ant target depend on a new Gradle task.

Listing 9.3 Seamless interaction between Ant and Gradle builds

```
ext.antBuildDir = '../ant/build'
ant.properties.build = "$antBuildDir/classes"
ant.properties.dist = "$antBuildDir/libs"
```

Changes value of an Ant property

```
task sourcesJar(type: Jar) {
   baseName = 'my-app'
   classifier = 'sources'
   version = ant.properties.version
   destinationDir = file(ant.properties.dist)
   from new File(ant.properties.src, 'main/java')
}

dist.dependsOn sourcesJar
```

Creates JAR file containing source files

Adds a task dependency on Ant target

The new task `sourcesJar` shouldn't look foreign. It simply creates a new JAR file containing the Java source files in the destination directory `ant/build/libs`. Because it should be part of your distribution, you declared a dependency on the Ant target `dist`. Executing the build automatically invokes the task as part of the task graph:

```
$ gradle clean dist
:clean
:init
:compile
:sourcesJar
:dist
```

Executes task sourcesJar as dependency of task dist

The resulting JAR files can be found in a new distribution directory:

```
.
└── ant
    ├── build
    │   ├── classes
    │   │   └── ...
    │   └── libs
    │       ├── my-app-1.0-sources.jar
    │       └── my-app-1.0.jar
    ├── ...
```

Redefined classes output directory

Redefined distribution directory

Generated JAR file containing Java source files

JAR file containing production class files

So far, you've learned how to apply Gradle's feature set to simplify the integration with an existing Ant build script. Next you'll apply one of Gradle's unique and powerful features: incremental builds.

ADDING INCREMENTAL BUILD CAPABILITIES TO AN ANT TARGET

The build tool Ant doesn't support incremental build functionality because Ant targets can't determine their own state. Sure, you can always implement it yourself by applying homegrown (and potentially error-prone) techniques to prevent the execution of unnecessary targets (for example, with the help of time-stamped files). But why put in all this effort if Gradle provides a built-in mechanism for it? The following listing demonstrates how to define inputs and outputs for the compilation target imported from the Ant script.

Listing 9.4 Defining inputs and outputs for imported Ant target

```
compile {
   inputs.dir file(ant.properties.src)
   outputs.dir file(ant.properties.build)
}
```

Defines compilation input directory ../ant/src

Defines compilation output directory ../ant/build/classes

This looks pretty straightforward, right? Try it out. First, clean up the existing class and JAR files and run through the whole generation process:

```
$ gradle clean dist
:init
:compile
:sourcesJar
:dist
```

As expected, the Java source code is compiled and the JAR files are created. When you run the `dist` task again without first deleting the files, Gradle realizes that the source files haven't changed and that output files already exist. The compilation task is automatically marked as `UP-TO-DATE`, as shown in the following command-line output:

```
$ gradle dist
:init
:compile UP-TO-DATE
:sourcesJar UP-TO-DATE
:dist
```

Gradle marks compilation task UP-TO-DATE and doesn't execute it.

Being able to add incremental build functionality to an imported Ant target is a big win for Ant users coming to Gradle. It proves to be a huge timesaver, especially in enterprise builds with many dependencies and source files.

Even if you don't import an existing Ant build script, Gradle allows for executing Ant tasks directly from your Gradle build script. In this book, you've already seen some examples. To round out your use case, let's discuss how to incorporate one of Ant's standard tasks into the build.

9.1.2 Using standard Ant tasks from Gradle

Gradle's `AntBuilder` provides direct access to all standard Ant tasks within your build script—no additional configuration needed. At runtime, Gradle checks the bundled Ant JAR files available on its classpath for the respective Ant task. Figure 9.3 illustrates the interaction between the use of an Ant task within the Gradle build script, the Gradle runtime, and its included Ant tasks. Using standard Ant tasks in Gradle comes in handy if you don't want to import an existing Ant script, or if you feel more comfortable with the Ant syntax.

Using an Ant task in a Gradle build script isn't hard, as long as you remember some basic rules:

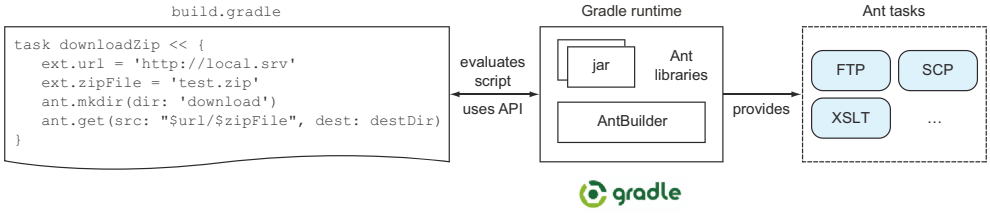

Figure 9.3 Using Ant tasks from Gradle

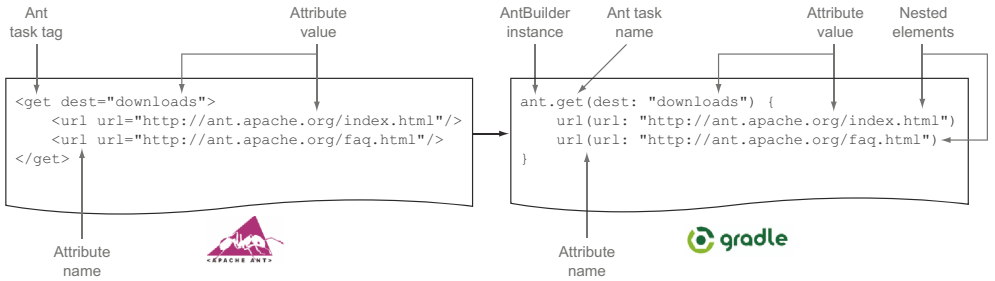

Figure 9.4 Relevant Ant task elements in Gradle

- Use the implicit `AntBuilder` variable `ant` to define an Ant task.
- The Ant task name you use with the `AntBuilder` instance is the same as the tag name in Ant.
- Task attributes are wrapped in parentheses.
- Define a task attribute name and value with the following pattern: `<name>: <value>`. Alternatively, task attributes can be provided as `Map`; for example `[<name1>: <value1>, <name2>: <value2>]`.
- Nested task elements don't require the use of the implicit `ant` variable. The parent task element wraps them with curly braces.

Let's look at a concrete example: the Ant Get task. The purpose of the task is to download remote files to your local disk via HTTP(S). You can find its documentation in the Ant online manual at https://ant.apache.org/manual/Tasks/get.html. Assume that you want to download two files to the destination directory `downloads`. Figure 9.4 shows how to express this logic using the `AntBuilder` DSL.

If you compare the task definition in Ant and Gradle, there are a few differences. You got rid of the pointy brackets and ended up with a more readable task definition. The important point is that you don't have to rewrite existing Ant functionality, and its integration into Gradle is seamless.

Next, you'll use the Get task in the context of your previous code examples. As part of your distribution, you want to bundle a project description file and the release notes. Each file resides on a different server and isn't part of the source code in version control. This is a perfect use case for the Get Ant task. The following listing shows how to apply the Get Ant task to download both files and make them part of the generated JAR file.

Listing 9.5 Using the standard Get Ant task

```
task downloadReleaseDocumentation {
   logging.level = LogLevel.INFO
   ext.repoUrl = 'https://repository-gradle-in-action.forge.cloudbees.com/
                 ➥ release'

   doLast {
      ant.get(dest: ant.properties.build) {
         url(url: "$repoUrl/files/README.txt")
         url(url: "$repoUrl/files/RELEASE_NOTES.txt")
```

Uses implicit AntBuilder variable to access Get Ant task

Declares nested URL elements

```
        }
    }
}

dist.dependsOn downloadReleaseDocumentation
```

All standard Ant tasks can be used with this technique because they're bundled with the Gradle runtime. Make sure to keep the Ant documentation handy when writing your logic. Optional or third-party Ant tasks usually require you to add another JAR file to the build script's classpath. You already learned how to do that in chapter 5 when you used the external Cargo Ant tasks to deploy your To Do application to a web container. Please refer to the code examples in chapter 5 for more information on how to use optional Ant tasks.

So far, you've learned many ways to interact with existing Ant build scripts or tasks from Gradle. But what if you're planning to move to Gradle long term? How do you approach a step-by-step migration?

9.1.3 Migration strategies

Gradle doesn't force you to fully migrate an existing Ant script in one go. A good place to start is to import the existing Ant build script and get familiar with Gradle while using existing logic. In this first step, you only have to invest minimal effort. Let's look at some other measures you may want to take.

Migrating individual Ant targets to Gradle tasks

Later, you'll translate the logic of Ant targets into Gradle tasks, but you'll start small by picking targets with simple logic. Try to implement the logic of the targets "the Gradle way" instead of falling back to an implementation backed by the `AntBuilder`. Let's discuss this with the help of an example. Assume you have the following Ant target:

```
<target name="create-manual">
   <zip destfile="dist/manual.zip">
      <fileset dir="docs/manual"/>
      <fileset dir="." includes="README.txt"/>
   </zip>
</target>
```

In Gradle, it's beneficial to implement the same logic with the help of an enhanced task of type `org.gradle.api.tasks.bundling.Zip`, as shown in the following code snippet:

```
task createManual(type: Zip) {
   baseName = 'manual'
   destinationDir = file('dist')
   from 'docs/manual'
   from('.') {
      include 'README.txt'
   }
}
```

This approach automatically buys you incremental build functionality without actually having to explicitly declare inputs and outputs. If there's no direct Gradle task type for

the logic you want to transfer, you can still fall back to the `AntBuilder`. Over time, you'll see that your Ant build script will get smaller and smaller while the logic in your Gradle build will grow.

INTRODUCING DEPENDENCY MANAGEMENT

One of Gradle's many useful features is dependency management. If you're an Ant user and aren't already using Ivy's dependency management, it will relieve you of the burden of having to manually manage external libraries. When migrating to Gradle, dependency management can be used even if you're compiling your sources within an Ant target. All you need to do is move the dependency declaration into your Gradle build script and provide a new property to the Ant script. The next listing demonstrates how to do this for a simple code example.

Listing 9.6 Declaring compilation dependencies

```
configurations {
   antCompile
}

repositories {
   mavenCentral()
}

dependencies {
   antCompile 'org.apache.commons:commons-lang3:3.1'
}

ant.properties.antCompileClasspath = configurations.antCompile.asPath
```

Custom configuration for Ant compilation dependencies

Definition of Ant dependencies assigned to custom configuration

Sets a new Ant property to be used for compilation in Ant build script

With this code in place, you can use the provided Ant property named `antCompileClasspath` for setting the classpath in the Ant build script:

```
<target name="compile" depends="init" description="compile the source">
   <javac srcdir="${src}" destdir="${build}"
          classpath="${antCompileClasspath}" includeantruntime="false"/>
</target>
```

Using the compile classpath set from Gradle build script

The change to the Ant build script was minimal. You can now also get rid of the `lib` directory in your Ant build, because Gradle's dependency manager automatically downloads the dependencies. Of course, you could also move the Javac task to Gradle. But why do all of this work when you can simply use the Gradle Java plugin? Introducing the plugin would eliminate the need for the Ant build logic.

TACKLING TASK NAME CLASHES

Sooner or later in your migration, you'll come to a point where you'll want to pull in one or more Gradle plugins. Your example Ant project resembles the typical tasks needed in a Java project: compile the source code, assemble a JAR file, and clean up existing artifacts. Applying the Java plugin works like a charm, as long as your Ant script doesn't define any targets that have the same name as any of the tasks exposed by the plugin. Give it a shot by modifying your Gradle build to have the following content:

```
ant.importBuild '../ant/build.xml'
apply plugin: 'java'
```

Executing any Gradle task will indicate that you have a task namespace clash, as shown in the following command-line output:

```
$ gradle tasks

FAILURE: Build failed with an exception.

* Where:
Build file '/Users/Ben/Dev/books/gradle-in-action/code/migrating-ant-
➥ build/gradle/build.gradle' line: 2

* What went wrong:
A problem occurred evaluating root project 'gradle'.
> Cannot add task ':clean' as a task with that name already exists.
```

You have two choices in this situation. Either exclude the existing Ant target, or wrap the imported Ant target with a Gradle task with a different name. The approach you take depends on your specific use case. The following code snippet demonstrates how to trick `AntBuilder` into thinking that the task already exists:

```
ant.project.addTarget('clean', new org.apache.tools.ant.Target())
ant.importBuild '../ant/build.xml'
apply plugin: 'java'
```

Excludes Ant target with name clean

As a result, the original Ant target is excluded; the `clean` task provided by the Java plugin is used instead.

Excluding some of the less complex Ant targets may work for you, but sometimes you want to preserve existing logic because it would require a significant amount of time to rewrite it. In those cases, you can build in another level of indirection, as shown in figure 9.5.

The import of the Ant build script can happen in a second Gradle build script named `importedAntBuild.gradle`:

```
ant.importBuild '../ant/build.xml'
```

The consuming Gradle build script declares an enhanced task of type `GradleBuild` that defines the Ant target you want to use with a new name. You can think of this technique as renaming an existing Ant target. The following code snippet demonstrates its use:

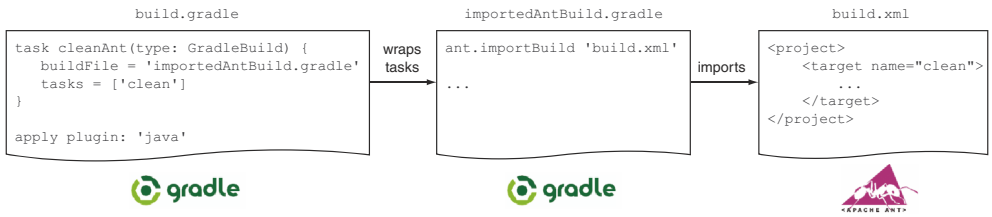

Figure 9.5 Wrapping an imported Ant target by exposing a Gradle task with a new name

```
task cleanAnt(type: GradleBuild) {
   buildFile = 'importedAntBuild.gradle'
   tasks = ['clean']
}

apply plugin: 'java'
```

Preferred task name of type GradleBuild

Originating Gradle build file

Tasks from originating build script to be invoked when wrapped task is executed

With this code in place, the exposed Gradle task name is `cleanAnt`:

```
$ gradle tasks
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

...

Other tasks
-----------
cleanAnt

...
```

Wrapping task with name cleanAnt that directs the call to imported clean Ant target

If you want to let the standard Gradle `clean` task depend on the Ant clean-up logic, you can define a task dependency between them:

```
clean.dependsOn cleanAnt
```

We discussed how to approach a migration from Ant to Gradle step by step without completely blocking the build or delivery process. In the next section, we'll compare commonalities and differences between Maven and Gradle concepts. As we did in this section, we'll also talk about build migration strategies.

9.2 Maven and Gradle

Gradle's integration with Ant is superb. It allows for importing existing Ant build scripts that translate targets in Gradle tasks, enable executing them transparently from Gradle, and provide the ability to enhance targets with additional Gradle functionality (for example, incremental build support). With these features in place, you can approach migrating from Ant to Gradle through various strategies.

Unfortunately, the same cannot be said about the current Maven integration support. At the time of writing, Gradle doesn't provide any deep imports of existing Maven builds. This means that you can't just point your Gradle build to an existing POM file to derive metadata at runtime and to execute Maven goals. But there are some counter strategies to deal with this situation, and we'll discuss them in this section. Before we dive into migrating from Maven to Gradle, let's compare commonalities and differences between both systems. Then, we'll map some of Maven's core concepts to Gradle functionality.

9.2.1 Commonalities and differences

When directly comparing Maven and Gradle, you can find many commonalities. Both build tools share the same concepts (for example, dependency management and

convention over configuration) even though they may be implemented differently, use diverging vocabulary, or require specific usage. Let's discuss some important differences frequently asked about on the Gradle forum.

PROVIDED SCOPE

Maven users will feel right at home when it comes to declaring dependencies with Gradle. Many of the Maven dependency scopes have a direct equivalent to a Gradle configuration provided by the Java or War plugin. Please refer to table 9.1 for a quick refresher.

Table 9.1 Maven dependency scopes and their Gradle configuration representation

Maven Scope	Gradle Java Plugin Configuration	Gradle War Plugin Configuration
`compile`	`compile`	N/A
`provided`	N/A	`providedCompile`, `providedRuntime`
`runtime`	`runtime`	N/A
`test`	`testCompile`, `testRuntime`	N/A

There's one Maven scope that only finds a representation with the War plugin: `provided`. Dependencies defined with the `provided` scope are needed for compilation but aren't exported (that is, bundled with the runtime distribution). The scope assumes that the runtime environment provides the dependency. One typical example is the Servlet API library. If you aren't building a web application, you won't have an equivalent configuration available in Gradle. This is easily fixable—you can define the behavior of the scope as a custom configuration in your build script. The following listing shows how to create a `provided` scope for compilation purposes and one for exclusive use with unit tests.

Listing 9.7 Custom `provided` configuration

```
configurations {
   provided                                          // Declares provided configurations
   testProvided.extendsFrom provided
}

sourceSets {
   main {
      compileClasspath += configurations.provided    // Added provided configurations to compilation classpath
   }

   test {
      compileClasspath += configurations.testProvided
   }
}
```

DEPLOYING ARTIFACTS TO MAVEN REPOSITORIES

Maven repositories are omnipresent sources for external libraries in the build tool landscape. This is particularly true for Maven Central, the go-to location on the web for retrieving open source libraries.

So far, you've learned how to consume libraries from Maven repositories. Being able to publish an artifact to a Maven repository is equally important because in an enterprise setting a Maven repository may be used to share a reusable library across teams or departments. Maven ships with support for deploying an artifact to a local or remote Maven repository. As part of this process, a `pom.xml` file is generated containing the meta-information about the artifact.

Gradle provides a 100% compatible plugin that resembles Maven's functionality of uploading artifacts to repositories: the Gradle Maven Publishing plugin. We won't discuss the plugin any further in this chapter. If you're eager to learn more about it, jump directly to chapter 14.

SUPPORT FOR MAVEN PROFILES

Maven 2.0 introduces the concept of a build *profile*. A profile defines a set of environment-specific parameters through a subset of elements in a POM (the project `pom.xml`, `settings.xml`, or `profiles.xml` file). A typical use case for a profile is to define properties for use in a specific deployment environment. If you're coming to Gradle, you'll want to either reuse an existing profile definition or to emulate this functionality.

I'm sorry to disappoint you, but Gradle doesn't support the concept of profiles. Don't let this be a downer. There are two ways to deal with this situation. Let's first look at how you can read an existing profile file. Assume that you have the `settings.xml` file shown in the following listing located in your Maven home directory (`~/.m2`).

Listing 9.8 Maven profile file defining application server home directories

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
   <profiles>
      <profile>
         <id>appserverConfig-dev</id>
         <activation>
            <property>
               <name>env</name>
               <value>dev</value>
            </property>
         </activation>
         <properties>
            <appserver.home>/path/to/dev/appserver</appserver.home>
         </properties>
      </profile>
      <profile>
         <id>appserverConfig-test</id>
         <activation>
            <property>
               <name>env</name>
               <value>test</value>
            </property>
         </activation>
         <properties>
```

Development environment property

Application server home directory for development environment

Test environment property

```
            <appserver.home>/path/to/test/appserver</appserver.home>
          </properties>
        </profile>
    </profiles>
    ...
</settings>
```

Application server home directory for test environment

The settings file declares two profiles for determining the application server home directory for deployment purposes. Based on the provided environment value with the key env, Maven will pick the appropriate profile.

It's easy to implement the same functionality with Gradle. The next listing shows how to read the settings file, traverse the XML elements, and select the requested profile based on the provided property env.

Listing 9.9 Reading environment-specific property from settings file

```
def getMavenSettingsCredentials = {
    String userHome = System.getProperty('user.home')
    File mavenSettings = new File(userHome, '.m2/settings.xml')
    XmlSlurper xmlSlurper = new XmlSlurper()
    xmlSlurper.parse(mavenSettings)
}

task printAppServerHome << {
    def env = project.hasProperty('env') ? project.getProperty('env')
              ➥ : 'dev'
    logger.quiet "Using environment '$env'"
    def settings = getMavenSettingsCredentials()
    def allProfiles = settings.profiles.profile
    def profile = allProfiles.find {
                      it.activation.property.name == 'env' &&
                      ➥ it.activation.property.value == env }
    def appServerHome = profile.properties.'appserver.home'
    println "The $env server's home directory: $appServerHome"
}
```

Parse settings file with Groovy's XmlSlurper

Parse provided environment property

Traverse XML elements to find the application server property value

To run this example, call the task name and provide the property as a command-line parameter:

```
$ gradle printAppServerHome -Penv=test
:printAppServerHome
Using environment 'test'
The test server's home directory: /path/to/test/appserver
```

This works great if you want to stick with the settings file. However, at some point you may want to get rid of this artifact to cut off any dependency on Maven concepts. You have many options for defining values for a specific profile:

- Put them into the `gradle.properties` file. Unfortunately, property names are flat by nature, so you'll need to come up with a naming scheme; for example, `env.test.app.server.path`.
- Define them directly in your build script. The benefit of this approach is that you're able to use any data type available to Groovy to declare properties.

- Use Groovy's `ConfigSlurper` utility class for reading configuration files in the form of Groovy scripts. Chapter 14 uses this method and provides a full-fledged example.

GENERATING A SITE FOR THE PROJECT

A frequently requested feature from Maven users interested in using Gradle is the ability to generate a site for their projects. Maven's site plugin allows for creating a set of HTML files by running a single goal. A site usually exposes a unified view on general information about the project extracted from the POM, as well as aggregated reporting on test and static code analysis results.

At the time of writing, this functionality isn't available to Gradle users. A standard Gradle plugin that started down that road is `build-dashboard`. The plugin was introduced with Gradle 1.5 and exposes a task for building a dashboard HTML report that contains references to all reports generated during the build. Make sure to check the Gradle online documentation for more information.

9.2.2 Migration strategies

In the previous section, we discussed major differences between the build tools Maven and Gradle. The illustrated approaches for bridging the gap between certain functionality will give you a head start on a successful migration. Because Gradle doesn't offer importing Maven goals into the Gradle model at runtime, a migration will have to take place in parallel to the existing Maven setup. This procedure ensures a smooth migration without disrupting the build and delivery process.

Let's see how this can be done with the sample Maven POM file from chapter 1. For a quick refresher on the code, look at the following listing.

Listing 9.10 Original Maven POM file

```
<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>
   <groupId>com.mycompany.app</groupId>
   <artifactId>my-app</artifactId>
   <packaging>jar</packaging>
   <version>1.0</version>

   <dependencies>
      <dependency>
         <groupId>org.apache.commons</groupId>
         <artifactId>commons-lang3</artifactId>
         <version>3.1</version>
         <scope>compile</scope>
      </dependency>
   </dependencies>
</project>
```

Maven's artifactId element value maps to a Gradle project name

Thankfully, you don't have to manually convert the Maven build logic to a Gradle script. If you're a user of Gradle ≥ 1.6, you're offered a little bit of help: the build

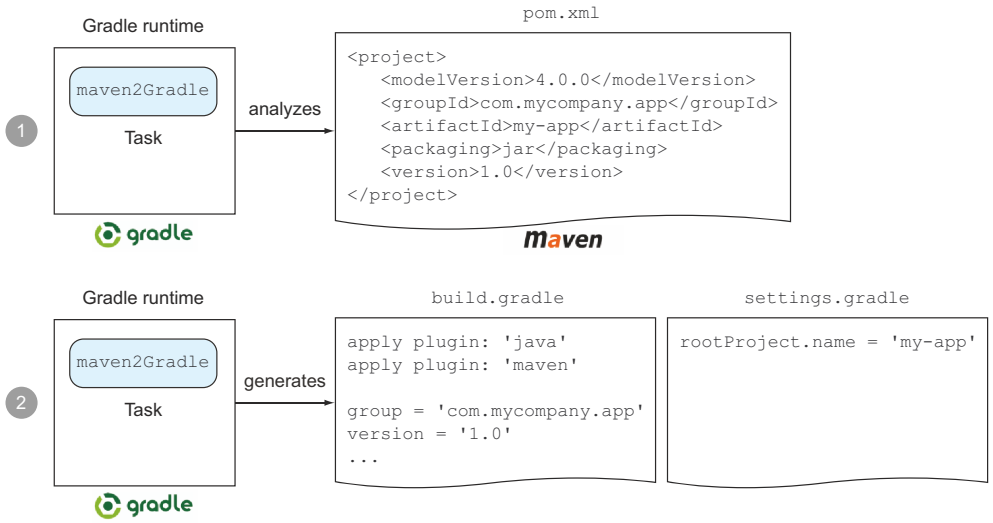

Figure 9.6 Generating Gradle build scripts from a Maven POM

setup plugin. The plugin supports generating `build.gradle` and `settings.gradle` files by analyzing a Maven `pom.xml` file, as shown in figure 9.6.

Let's take a closer look at how to use the plugin.

USING THE MAVEN2GRADLE TASK

The build setup plugin is automatically available to all Gradle builds independent from the build script's configuration. You learned in chapters 2 and 3 that the plugin can be used to generate a new Gradle build script or to add the wrapper files to your project. There's another task provided by the plugin that we haven't looked at yet: `maven2Gradle`. This task is only presented if your current directory contains a `pom.xml` file. Create a new directory, create a `pom.xml` file, and copy the contents of listing 9.11 into it. The directory should look as follows:

```
.
└── pom.xml
```

Now, navigate to this directory on the command line and list the available Gradle tasks:

```
$ gradle tasks --all
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Build Setup tasks
-----------------
setupBuild - Initializes a new Gradle build. [incubating]
maven2Gradle - Generates a Gradle build from a Maven POM.
➥ [incubating]
setupWrapper - Generates Gradle wrapper files. [incubating]

...
```

Maven POM converter task

Executing the `maven2Gradle` task will analyze the effective Maven POM configuration. When I talk about the effective POM, I mean the interpolated configuration of the `pom.xml` file, any parent POM, and any settings provided by active profiles. Even though the task is at an early stage of development, you can expect the following major conversion features:

- Conversion of single-module and multimodule Maven projects
- Translation from declared Maven dependencies and repositories
- Support for converting Maven projects for building plain Java projects as well as web projects
- Analyzing project metadata like ID, description, version, and compiler settings, and translating it into a Gradle configuration

In most cases, this converter task does a good job of translating the build logic from Maven to Gradle. One thing to remember is that the converter doesn't understand any third-party plugin configuration, and therefore can't create any Gradle code for it. This logic has to be implemented manually. You'll execute the task on your `pom.xml` file:

```
$ gradle maven2Gradle
:maven2Gradle
Maven to Gradle conversion is an incubating feature. Enjoy it and let
➥ us know how it works for you.
Working path:/Users/Ben/Dev/books/gradle-in-action/code/maven2gradle

This is single module project.
Configuring Maven repositories... Done.
Configuring Dependencies... Done.
Adding tests packaging...Generating settings.gradle if needed...
Done.
Generating main build.gradle... Done.
```

After executing the task, you'll find the expected Gradle files in the same directory as the POM file:

```
.
├── build.gradle
├── pom.xml
└── settings.gradle
```

Generated Gradle files

Let's take a closer look at the generated `build.gradle` file, as shown in the next listing. The file contains all necessary DSL elements you'd expect: plugins, project metadata, repositories, and dependencies.

Listing 9.11 Generated Gradle build script

```
apply plugin: 'java'
apply plugin: 'maven'

group = 'com.mycompany.app'
version = '1.0'

description = """"""
```

```
sourceCompatibility = 1.5
targetCompatibility = 1.5

repositories {
   mavenRepo url: "http://repo.maven.apache.org/maven2" <-- Generates URL for Maven Central instead of mavenCentral() shortcut
}

dependencies {
   compile group: 'org.apache.commons', name: 'commons-lang3', version:'3.1'
}
```

The project name can only be set during Gradle's initialization phase. For that reason, the `maven2Gradle` task also generates the settings file shown in the following listing.

Listing 9.12 Generated Gradle settings file

```
rootProject.name = 'my-app'
```

The generated Gradle files will give you a good start in migrating your complete Maven. Without compromising your existing build, you can now add on to the generated Gradle logic until you've fully transferred all functionality. At that point, you can flip the switch and continue to use Gradle as the primary build tool and start building confidence. Should you encounter any impediments, you can always fall back to your Maven build.

Wouldn't it be great if you could automatically determine that the build artifacts between the Maven and Gradle build are the same? That's the primary goal of the Gradle build comparison plugin.

9.3 Comparing builds

The build comparison plugin was introduced with Gradle 1.2. It has the high goal of comparing the outcomes of two builds. When I speak of an outcome, I mean the binary artifact produced by a build—for example, a JAR, WAR, or EAR file. The plugin aims to support the following comparisons:

- Gradle build compared to an Ant or Maven build in the case of a build tool migration
- Comparing the same Gradle build with two different versions in the case of an upgrade
- Comparing a Gradle build with the same version after changing build logic

I know you're excited because this plugin could be extremely helpful in comparing the build outcomes after migrating from Ant or Maven. Unfortunately, I have to disappoint you. At the time of writing, this functionality hasn't been implemented. What you can do, though, is compare a Gradle build after upgrading the version. Figure 9.7 demonstrates such a use case in the context of your To Do application.

Your sample project creates three binary artifacts: two JAR files produced by the projects `model` and `repository`, and one WAR file produced by the `web` project. The WAR file includes the two other JAR files. A comparison between two builds would

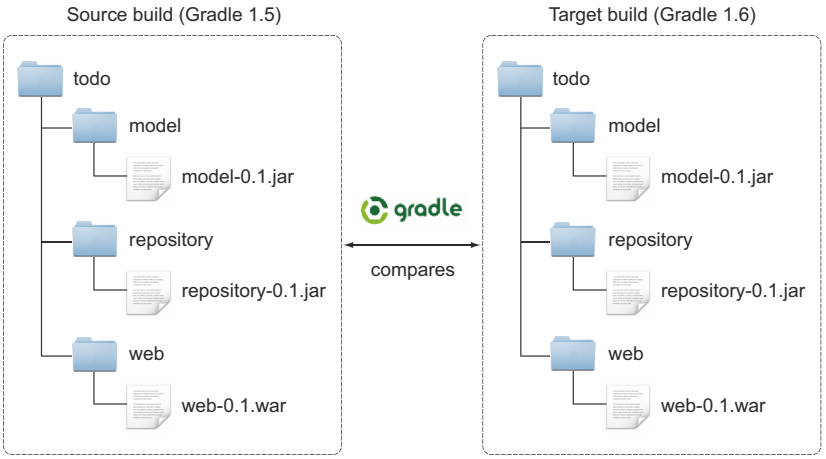

Figure 9.7 Comparing build outcomes of two different Gradle versions

have to take into account these files. Let's say you want to upgrade your project from Gradle 1.5 to 1.6. The following listing shows the necessary setup required to compare the builds.

Listing 9.13 Upgrading the Gradle runtime

```
apply plugin: 'compare-gradle-builds'

compareGradleBuilds {
   sourceBuild {
      projectDir = rootProject.projectDir
      gradleVersion = '1.5'
   }

   targetBuild {
      projectDir = sourceBuild.projectDir
      gradleVersion = '1.6'
   }
}
```

Source build definition pointing to root project

Target build definition pointing to root project of source build definition

If you want to initiate a build comparison, all you need to do is execute the provided task `compareGradleBuilds`. The task will fail if the outcome of the compared build is different. Because the command-line output of this task is lengthy, I won't show it here. What's more interesting is the reporting structure produced by the task. The following directory tree shows the build outcome files used to compare the build and HTML report:

```
.
└── build
    └── reports
        └── compareGradleBuilds
            ├── files
            │   ├── source
```

Build outcome files generated with Gradle 1.5

```
│   │   ├── _model_jar
│   │   │   └── model_0.1.jar
│   │   ├── _repository_jar
│   │   │   └── repository_0.1.jar
│   │   └── _web_war
│   │       └── web_0.1.war
│   └── target
│       ├── _model_jar
│       │   └── model_0.1.jar
│       ├── _repository_jar
│       │   └── repository_0.1.jar
│       └── _web_war
│           └── web_0.1.war
└── index.html
```

Build outcome files generated with Gradle 1.6

Build comparison HTML report

The report file `index.html` gives detailed information about the compared builds, their versions, the involved binary artifacts, and the result of the comparison. If Gradle determines that the compared builds aren't identical, it'll be reflected in the reports, as shown in figure 9.8.

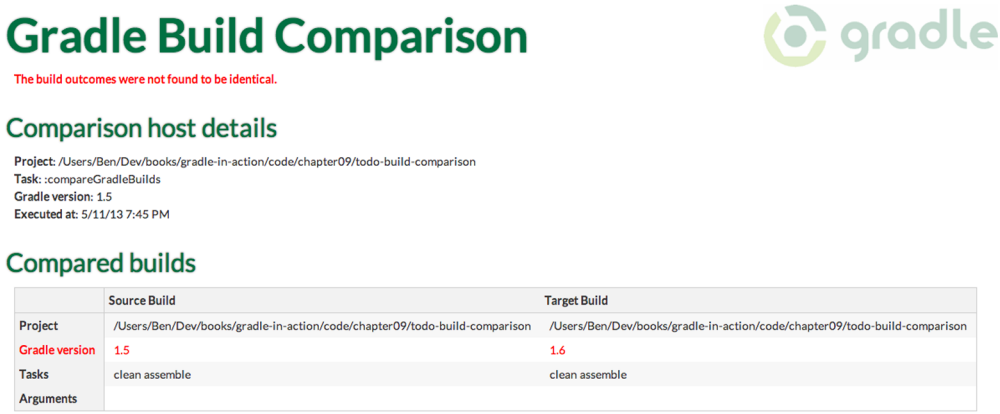

Figure 9.8 Sample build comparison HTML report for upgrading a Gradle version

I think it becomes apparent how helpful the functionality of this plugin can be in making a determination whether an upgrade can be performed without side effects. The plugin has a lot of future potential, especially if you're interested in comparing existing builds backed by other build tools within the scope of a migration.

9.4 Summary

In this chapter, we discussed how the traditional Java-based build tools Ant and Maven fit into the picture of integration and migration. As an Ant user, you have the most options and the most powerful tooling. Gradle allows for deep imports of an Ant build script by turning Ant targets into Gradle tasks. Even if you don't import existing Ant builds, you can benefit from reusing standard and third-party Ant tasks. You learned

that migrating from Ant to Gradle can be done in baby steps: first import the existing build, then introduce dependency management, and then translate targets into tasks using Gradle's API. Finally, make good use of Gradle plugins.

Maven and Gradle share similar concepts and conventions. If you're coming from Maven, the basic project layout and dependency management usage patterns should look strikingly familiar. We discussed some Maven features that are missing from Gradle, such as the `provided` scope and the concept of profiles. You saw that Gradle (and ultimately its underlying language Groovy) is flexible enough to find solutions to bridge the gap. Unfortunately, Gradle doesn't support importing Maven goals from a POM at runtime, which makes migration less smooth than for Ant users. With Gradle's `maven2Gradle` conversion task, you can get a head start on a successful migration by generating a `build.gradle` file from an effective POM.

Upgrading a Gradle build from one version to another shouldn't cause any side effects or even influence your ability to compile, assemble, and deploy your application code. The build comparison plugin automates upgrade testing by comparing the outcomes of a build with two different versions. With this information in hand, you can mitigate the risk of a failed upgrade.

Congratulations, you got to know Gradle's most essential features! This chapter concludes part 2 of the book. In part 3, we'll shift our focus to using Gradle in the context of a continuous delivery process. First, let's discuss how to use Gradle within popular IDEs.

JAVA/PROGRAMMING

Gradle IN ACTION

Benjamin Muschko

Free eBook
SEE INSERT

Gradle is a general-purpose build automation tool. It extends the usage patterns established by its forerunners Ant and Maven and allows builds that are expressive, maintainable, and easy to understand. Using a flexible Groovy-based DSL, Gradle provides declarative and extendable language elements that let you model your project's needs the way you want.

Gradle in Action is a comprehensive guide to end-to-end project automation with Gradle. Starting with the basics, this practical, easy-to-read book discusses how to establish an effective build process for a full-fledged, real-world project. Along the way, it covers advanced topics like testing, continuous integration, and monitoring code quality. You'll also explore tasks like setting up your target environment and deploying your software.

What's Inside

- A comprehensive guide to Gradle
- Practical, real-world examples
- Transitioning from Ant and Maven
- In-depth plugin development
- Continuous delivery with Gradle

The book assumes a basic background in Java but no knowledge of Groovy.

Benjamin Muschko is a member of the Gradleware engineering team and the author of several popular Gradle plugins.

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

"The authoritative guide."
—From the Foreword by Hans Dockter, Founder of Gradle and Gradleware

"A new way to automate your builds. You'll never miss the old one."
—Nacho Ormeño, startupXplore

"Required reading for the polyglot programmer!"
—Rob Bugh, ReachForce

"The best Gradle reference ever! Full of real-world examples."
—Wellington R. Pinheiro
Walmart eCommerce Brazil

"The missing book to help make Gradle accessible to any developer."
—Samuel Brown, Blackboard, Inc.

ISBN 13: 978-1-617291-30-2
ISBN 10: 1-617291-30-7
54499
9 781617 291302

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]