SAMPLE CHAPTER

Gradle IN ACTION

Benjamin Muschko

FOREWORD BY Hans Dockter

MANNING

Gradle in Action

by Benjamin Muschko

Chapter 2

brief contents

2 Next-generation builds with Gradle

This chapter covers

- Understanding how Gradle compares to other build tools
- Describing Gradle's compelling feature set
- Installing Gradle
- Writing and executing a simple Gradle script
- Running Gradle on the command line

For years, builds had the simple requirements of compiling and packaging software. But the landscape of modern software development has changed, and so have the needs for build automation.

Today, projects involve large and diverse software stacks, incorporate multiple programming languages, and apply a broad spectrum of testing strategies. With the rise of agile practices, builds have to support early integration of code as well as frequent and easy delivery to test and production environments.

Established build tools continuously fall short in meeting these goals in a simple but customizable fashion. How many times have your eyes glazed over while looking at XML to figure out how a build works? And why can't it be easier to add custom logic to your build? All too often, when adding on to a build script, you can't shake the feeling of implementing a workaround or hack. I feel your pain. There has to be a better way of doing these things in an expressive and maintainable way. There is—it's called Gradle.

Gradle is the next evolutionary step in JVM-based build tools. It draws on lessons learned from established tools like Ant and Maven and takes their best ideas to the next level. Following a build-by-convention approach, Gradle allows for declaratively modeling your problem domain using a powerful and expressive domain-specific language (DSL) implemented in Groovy instead of XML. Because Gradle is a JVM native, it allows you to write custom logic in the language you're most comfortable with, be it Java or Groovy.

In the Java world, an unbelievably large number of libraries and frameworks are available. Dependency management is used to automatically download these artifacts from a repository and make them available to your application code. Having learned from the shortcomings of existing dependency management solutions, Gradle provides its own implementation. Not only is it highly configurable, it also strives to be as compatible as possible with existing dependency management infrastructures (like Maven and Ivy). Gradle's ability to manage dependencies isn't limited to external libraries. As your project grows in size and complexity, you'll want to organize the code into modules with clearly defined responsibilities. Gradle provides powerful support for defining and organizing multiproject builds, as well as modeling dependencies between projects.

I know, all of this sounds promising, but you're still stuck with your legacy build. Gradle doesn't leave you in the dust, but makes migration easy. Ant gets shipped with the runtime and therefore doesn't require any additional setup. Gradle provides teams with the ability to apply their accumulated Ant knowledge and investment in build infrastructure. Imagine the possibilities of using existing Ant tasks and scripts directly in your Gradle build scripts. Legacy build logic can be reused or migrated gradually. Gradle does the heavy lifting for you.

To get started with Gradle, all you need to bring to the table is a good understanding of the Java programming language. If you're new to project automation or haven't used a build tool before, chapter 1 is a good place to start. This book will teach you how to effectively use Gradle to build and deliver real-world projects.

In this chapter, we'll compare existing JVM-language build tools with the features Gradle has to offer. Later, you'll learn how Gradle can help you automate your software delivery process in the context of a continuous delivery deployment pipeline. To get a first taste of what it's like to use Gradle, you'll install the runtime, write a simple build script, and run it on the command line. Join me on an exciting journey as we explore the world of Gradle.

2.1 Why Gradle? Why now?

If you've ever dealt with build systems, frustration may be one of the feelings that comes up when thinking about the challenges you've faced. Shouldn't the build tool naturally help you accomplish the goal of automating your project? Instead, you had to compromise on maintainability, usability, flexibility, extendibility, or performance.

Let's say you want to copy a file to a specific location under the condition that you're building the release version of your project. To identify the version, you check a string in the metadata describing your project. If it matches a specific numbering scheme (for example, 1.0-RELEASE), you copy the file from point A to point B. From an outside perspective, this may sound like a trivial task. If you have to rely on XML, the build language of many traditional tools, expressing this simple logic becomes a nightmare. The build tool's response is to add scripting functionality through nonstandard extension mechanisms. You end up mixing scripting code with XML or invoking external scripts from your build logic. It's easy to imagine that you'll need to add more and more custom code over time. As a result, you inevitably introduce accidental complexity, and maintainability goes out the window. Wouldn't it make sense to use an expressive language to define your build logic in the first place?

Here's another example. Maven follows the paradigm of convention over configuration by introducing a standardized project layout and build lifecycle for Java projects. That's a great approach if you want to ensure a unified application structure for a greenfield project—a project that lacks any constraints imposed by prior work. However, you may be the lucky one who needs to work on one of the many legacy projects that follow different conventions. One of the conventions Maven is very strict about is that one project needs to produce one artifact, such as a JAR file. But how do you create two different JAR files from one source tree without having to change your project structure? Just for this purpose, you'd have to create two separate projects. Again, even though you can make this happen with a workaround, you can't shake off the feeling that your build process will need to adapt to the tool, not the tool to your build process.

These are only some of the issues you may have encountered with existing solutions. Often you've had to sacrifice nonfunctional requirements to model your enterprise's automation domain. But enough with the negativity—let's see how Gradle fits into the build tool landscape.

2.1.1 Evolution of Java build tools

Let's look at how build tools have evolved over the years. As I discussed in chapter 1, two tools have dominated building Java projects: Ant and Maven. Over the course of years, both tools significantly improved and extended their feature set. But even though both are highly popular and have become industry standards, they have one weak point: build logic has to be described in XML. XML is great for describing hierarchical data, but falls short on expressing program flow and conditional logic. As a build script grows in complexity, maintaining the build code becomes a nightmare.

Ant's first official version was released in 2000. Each element of work (a *target* in Ant's lingo) can be combined and reused. Multiple targets can be chained to combine single units of work into full workflows. For example, you might have one target for compiling Java source code and another one for creating a JAR file that packages the class files. Building a JAR file only makes sense if you first compiled the source code. In Ant, you make the JAR target depend on the compile target. Ant doesn't give any guidance on how to structure your project. Though it allows for maximum flexibility, Ant makes each build script unique and hard to understand. External libraries required by your project were usually checked into version control, because there was no sophisticated mechanism to automatically pull them from a central location. Early versions of Ant required a lot of discipline to avoid repetitive code. Its extension mechanism was simply too weak. As a result, the bad coding practice of copying and pasting code was the only viable option. To unify project layouts, enterprises needed to impose standards.

Maven 1, released in July 2004, tried to ease that process. It provided a standardized project and directory structure, as well as dependency management. Unfortunately, custom logic is hard to implement. If you want to break out of Maven's conventions, writing a plugin, called a *Mojo*, is usually the only solution. The name Mojo might imply a straightforward, easy, and sexy way to extend Maven; in reality, writing a plugin in Maven is cumbersome and overly complex.

Later, Ant caught up with Maven by introducing dependency management through the Apache library Ivy, which can be fully integrated with Ant to declaratively specify dependencies needed for your project's compilation and packaging process. Maven's dependency manager, as well as Ivy, support resolving transitive dependencies. When I speak of transitive dependencies, I mean the graph of libraries required by your specified dependencies. A typical example of a transitive dependency would be the XML parser library Xerces that requires the XML APIs library to function correctly. Maven 2, released in October 2005, took the idea of convention over configuration even further. Projects consisting of multiple modules could define their dependencies on each other.

These days a lot of people are looking for alternatives to established build tools. We see a shift from using XML to a more expressive and readable language to define builds. A build tool that carries on this idea is Gant, a DSL on top of Ant written in Groovy. Using Gant, users can now combine Groovy's language features with their existing knowledge of Ant without having to write XML. Even though it wasn't part of the core Maven project, a similar approach was proposed by the project Maven Polyglot that allows you to write your build definition logic, which is the project object model (POM) file, in Groovy, Ruby, Scala, or Clojure.

We're on the cusp of a new era of application development: polyglot programming. Many applications today incorporate multiple programming languages, each of which is best suited to implement a specific problem domain. It's not uncommon to face projects that use client-side languages like JavaScript that communicate with a mixed, multilingual backend like Java, Groovy, and Scala, which in turn calls off to a C++ legacy application. It's all about the right tool for the job. Despite the benefits

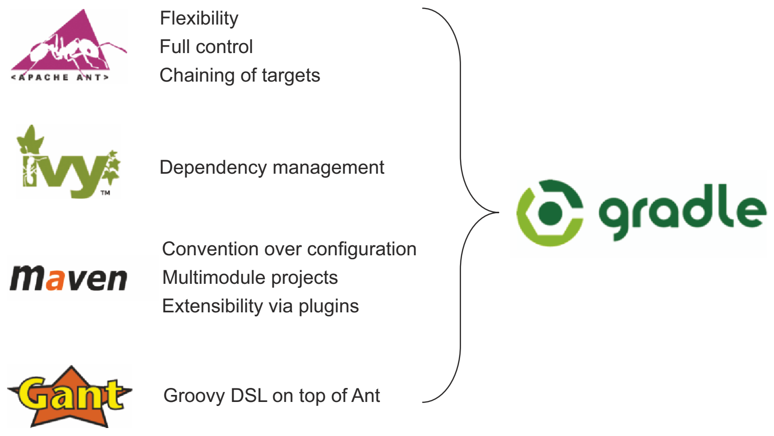


Figure 2.1 Gradle combines the best features from other build tools.

of combining multiple programming languages, your build tool needs to fluently support this infrastructure as well. JavaScript needs to be merged, minified, and zipped, and your server-side and legacy code needs to be compiled, packaged, and deployed.

Gradle fits right into that generation of build tools and satisfies many requirements of modern build tools (figure 2.1). It provides an expressive DSL, a convention over configuration approach, and powerful dependency management. It makes the right move to abandon XML and introduce the dynamic language Groovy to define your build logic. Sounds compelling, doesn't it? Keep reading to learn about Gradle's feature set and how to get your boss on board.

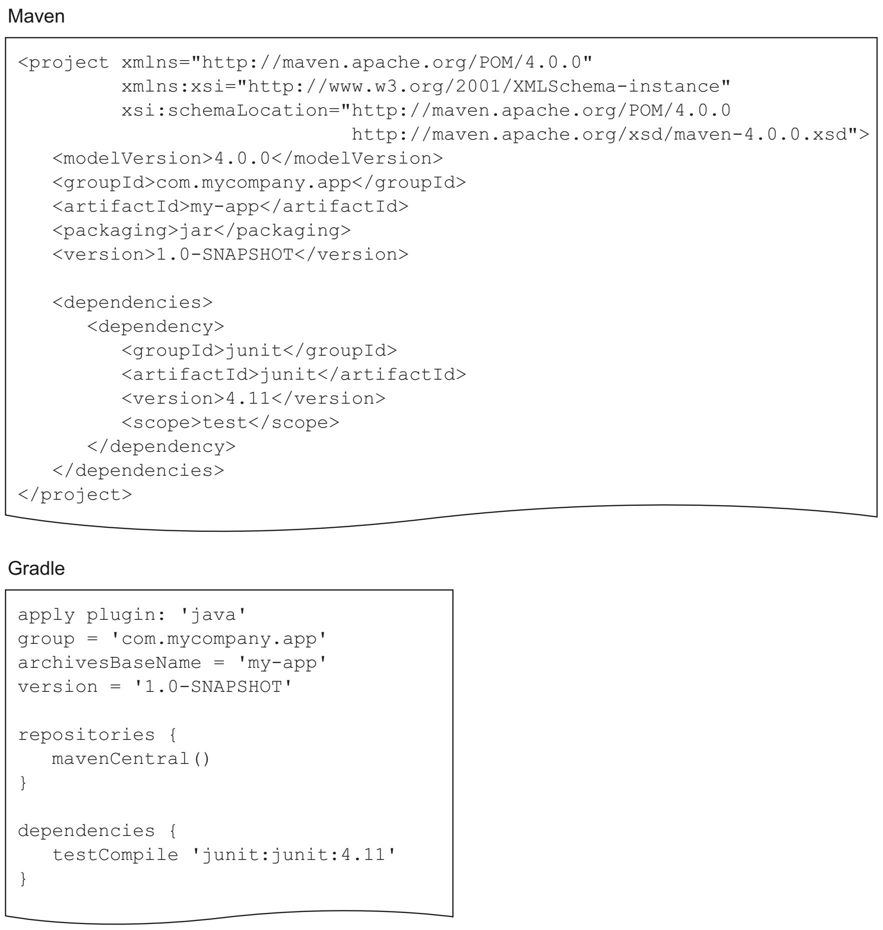
2.1.2 Why you should choose Gradle

If you're a developer, automating your project is part of your day-to-day business. Don't you want to treat your build code like any other piece of software that can be extended, tested, and maintained? Let's put software engineering back into the build. Gradle build scripts are declarative, readable, and clearly express their intention. Writing code in Groovy instead of XML, sprinkled with Gradle's build-by-convention philosophy, significantly cuts down the size of a build script and is far more readable (see figure 2.2).

It's impressive to see how much less code you need to write in Gradle to achieve the same goal. With Gradle you don't have to make compromises. Where other build tools like Maven propose project layouts that are "my way or the highway," Gradle's DSL allows for flexibility by adapting to nonconventional project structures.

Gradle's motto

"Make the impossible possible, make the possible easy, and make the easy elegant" (adapted quote from Moshé Feldenkrais).



Maven

```
<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-SNAPSHOT</version>

   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.11</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

Gradle

```
apply plugin: 'java'
group = 'com.mycompany.app'
archivesBaseName = 'my-app'
version = '1.0-SNAPSHOT'

repositories {
   mavenCentral()
}

dependencies {
   testCompile 'junit:junit:4.11'
}
```

Figure 2.2 Comparing build script size and readability between Maven and Gradle

Never change a running system, you say? Your team already spent a lot of time on establishing your project's build code infrastructure. Gradle doesn't force you to fully migrate all of your existing build logic. Good integration with other tools like Ant and Maven is at the top of Gradle's priority list. We'll take a deeper look at Gradle's integration features and potential migration strategies in chapter 9.

The market seems to be taking notice of Gradle. In spring 2010, Gradle was awarded the Springy award for the most innovative open source project (http://www.springsource.org/node/2871). ThoughtWorks, a highly regarded software development consultancy, periodically publishes a report on emerging technologies, languages, and tools—their so-called technology radar. The goal of the technology radar is to help decision makers in the software industry understand trends and their effect on the market. In their latest edition of the report from May 2013 (http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf), Gradle was rated with the status Adopt, indicating a technology that should be adopted by the industry.

Recognition by ThoughtWorks

"Two things have caused fatigue with XML-based build tools like Ant and Maven: too many angry pointy braces and the coarseness of plug-in architectures. While syntax issues can be dealt with through generation, plug-in architectures severely limit the ability for build tools to grow gracefully as projects become more complex. We have come to feel that plug-ins are the wrong level of abstraction, and prefer language-based tools like Gradle and Rake instead, because they offer finer-grained abstractions and more flexibility long term."

Gradle found adopters early on, even before a 1.0 version was released. Popular open source projects like Groovy and Hibernate completely switched to Gradle as the backbone for their builds. Every Android project ships with Gradle as the default build system. Gradle also had an impact on the commercial market. Companies like Orbitz, EADS, and Software AG embraced Gradle as well, to name just a few. VMware, the company behind Spring and Grails, made significant investments in choosing Gradle. Many of their software products, such as the Spring framework and Grails, are literally built on the trust that Gradle can deliver.

2.2 Gradle's compelling feature set

Let's take a closer look at what sets Gradle apart from its competitors: its compelling feature set (see figure 2.3). To summarize, Gradle is an enterprise-ready build system, powered by a declarative and expressive Groovy DSL. It combines flexibility and effortless extendibility with the idea of convention over configuration and support for traditional dependency management. Backed by a professional services company

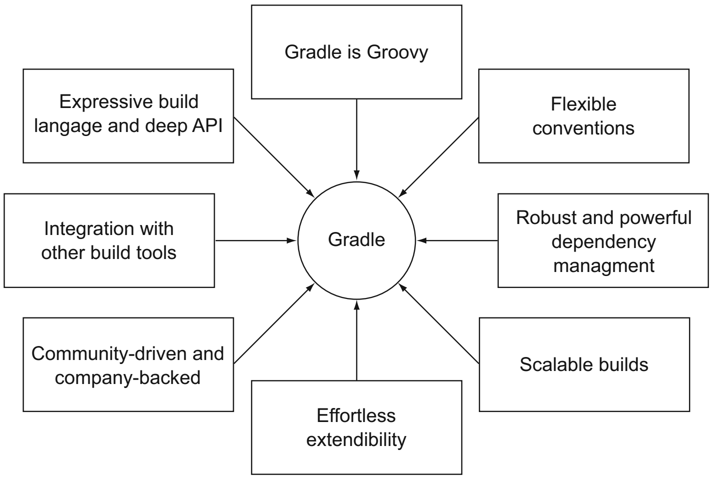

Gradle is Groovy
Expressive build langage and deep API
Flexible conventions
Integration with other build tools
Gradle
Robust and powerful dependency managment
Community-driven and company-backed
Scalable builds
Effortless extendibility

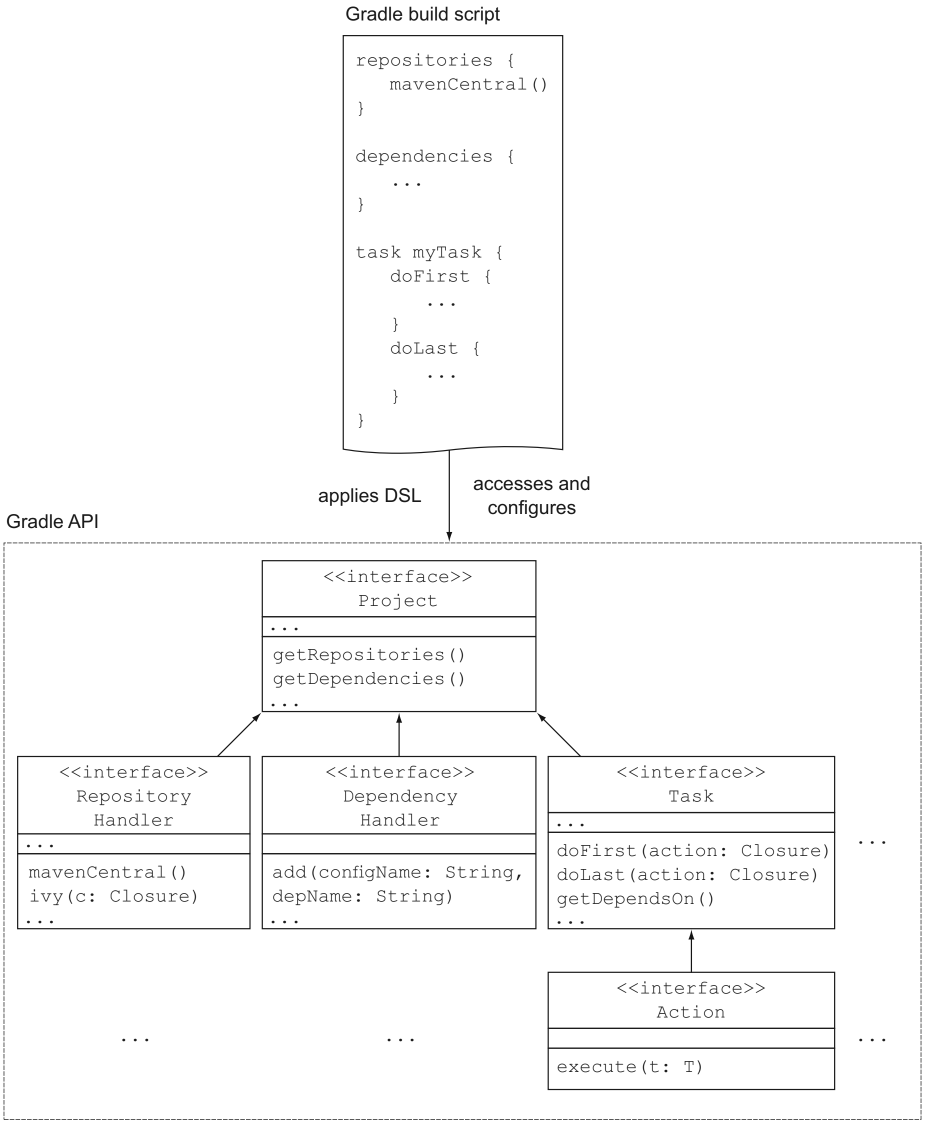
Figure 2.3 Gradle's compelling feature set

(Gradleware) and strong community involvement, Gradle is becoming the number-one choice build solution for many open source projects and enterprises.

2.2.1 Expressive build language and deep API

The key to unlocking Gradle's power features within your build script lies in discovering and applying its domain model, as shown in figure 2.4.

As you can see in the figure, a build script directly maps to an instance of type `Project` in Gradle's API. In turn, the `dependencies` configuration block in the build



```
repositories {
   mavenCentral()
}

dependencies {
   ...
}

task myTask {
   doFirst {
      ...
   }
   doLast {
      ...
   }
}
```

Figure 2.4 Build scripts apply the Gradle DSL and have access to its deep API.

script invokes the method `dependencies()` of the project instance. Like most APIs in the Java world, it's available as HTML Javadoc documentation on Gradle's website at http://www.gradle.org/docs/current/javadoc/index.html. Who would have known? You're actually dealing with code. Without knowing it, you generate an object representation of your build logic in memory. In chapter 4, we'll explore many of Gradle's API classes and how they're represented in your build script.

Each element in a Gradle script has a one-to-one representation with a Java class; however, some of the elements have been sugarcoated with a sprinkle of Groovy syntax. Having a Groovy-fied version of a class in many cases makes the code more compact than its Java counterpart and allows for using new language features like closures.

Gradle can't know all the requirements specific to your enterprise build. By exposing hooks into lifecycle phases, Gradle allows for monitoring and configuring your build script's execution behavior. Let's assume you have the very unique requirement of sending out an email to the development team whenever a unit test failure occurs. The way you want to send an email (for example, via SMTP or a third-party email service provider) and the list of recipients are very specific to your build. Other builds using Gradle may not be interested in this feature at all. By writing a custom test listener that's notified after the test execution lifecycle event, you can easily incorporate this feature for your build.

Gradle establishes a vocabulary for its model by exposing a DSL implemented in Groovy. When dealing with a complex problem domain, in this case the task of building software, being able to use a common language to express your logic can be a powerful tool. Let's look at some examples. Most common to builds is the notation of a unit of work that you want to get executed. Gradle describes this unit of work as a task. Part of Gradle's standard DSL is the ability to define tasks very specific to compiling and packaging Java source code. It's a language for building Java projects with its own vocabulary that doesn't need to be relevant to other contexts.

Another example is the way you can express dependencies to external libraries, a very common problem solved by build tools. Out-of-the-box Gradle provides you with two configuration blocks for your build script that allow you to define the dependencies and repositories that you want to retrieve them from. If the standard DSL elements don't fit your needs, you can even introduce your own vocabulary through Gradle's extension mechanism.

This may sound a little nebulous at first, but once you're past the initial hurdle of learning the build language, creating maintainable and declarative builds comes easy. A good place to start is the Gradle Build Language Reference Guide at http://www.gradle.org/docs/current/dsl/index.html. Gradle's DSL can be extended. You may want to change the behavior of an existing task or add your own idioms for describing your business domain. Gradle offers you plenty of options to do so.

2.2.2 Gradle is Groovy

Prominent build tools like Ant and Maven define their build logic through XML. As we all know, XML is easy to read and write, but can become a maintenance nightmare if used in large quantities. XML isn't very expressive. It makes it hard to define complex custom logic. Gradle takes a different approach. Under the hood, Gradle's DSL is written with Groovy providing syntactic sugar on top of Java. The result is a readable and expressive build language. All your scripts are written in Groovy as well. Being able to use a programming language to express your build needs is a major plus. You don't have to be a Groovy expert to get started. Because Groovy is written on top of Java, you can migrate gradually by trying out its language features. You could even write your custom logic in plain Java—Gradle couldn't care less. Battle-scarred Groovy veterans will assure you that using Groovy instead of Java will boost your productivity by orders of magnitude. A great reference guide is the book *Groovy in Action, Second Edition* by Dirk Konig et al. (Manning, 2009) For a primer on Groovy, see appendix B.

2.2.3 Flexible conventions

One of Gradle's big ideas is to give you guidelines and sensible defaults for your projects. Every Java project in Gradle knows exactly where source and test class file are supposed to live, and how to compile your code, run unit tests, generate Javadoc reports, and create a distribution of your code. All of these tasks are fully integrated into the build lifecycle. If you stick to the convention, there's only minimal configuration effort on your part. In fact, your build script is a one-liner. Seriously! Do you want to learn more about building a Java project with Gradle? Well, you can—we'll cover it in chapter 3. Figure 2.5 illustrates how Gradle introduces conventions and lifecycle tasks for Java projects.

Default tasks are provided that make sense in the context of a Java project. For example, you can compile your Java production source code, run tests, and assemble a JAR file. Every Java project starts with a standard directory layout. It defines where to find production source code, resource files, and test code. Convention properties are used to change the defaults.

The same concept applies to other project archetypes like Scala, Groovy, web projects, and many more. Gradle calls this concept *build by convention*. The build script developer doesn't need to know how this is working under the hood. Instead, you can concentrate on what needs to be configured. Gradle's conventions are similar to the ones provided by Maven, but they don't leave you feeling boxed in. Maven is very opinionated; it proposes that a project only contains one Java source directory and only produces one single JAR file. This is not necessarily reality for many enterprise projects. Gradle allows you to easily break out of the conventions. On the opposite side of the spectrum, Ant never gave you a lot of guidance on how to structure your build script, allowing for a maximum level of flexibility. Gradle takes the middle ground by offering conventions combined with the ability to easily change them. Szczepan Faber, one of Gradle's core engineers, put it this way on his blog: "Gradle is an opinionated framework on top of an unopinionated toolkit." (*Monkey Island*, "opinionated or not," June 2, 2012, http://monkeyisland.pl/2012/06/02/opinionated-or-not/.)

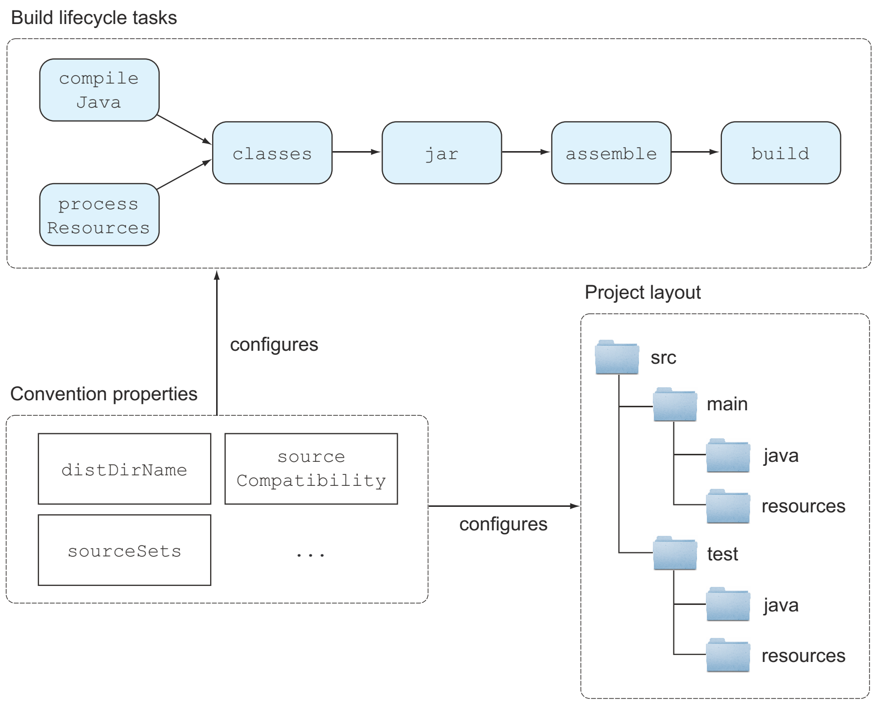


Figure 2.5 In Gradle, Java projects are build by convention with sensible defaults. Changing the defaults is easy and achieved through convention properties.

2.2.4 Robust and powerful dependency management

Software projects are usually not self-contained. All too often, your application code uses a third-party library providing existing functionality to solve a specific problem. Why would you want to reinvent the wheel by implementing a persistence framework if Hibernate already exists? Within an organization, you may be the consumer of a component or module implemented by a different team. External dependencies are accessible through repositories, and the type of repository is highly dependent on what your company prefers. Options range from a plain file system to a full-fledged enterprise repository. External dependencies may have a reference to other libraries or resources. We call these *transitive dependencies.*

Gradle provides an infrastructure to manage the complexity of resolving, retrieving, and storing dependencies. Once they're downloaded and put in your local cache, they're made available to your project. A key requirement of enterprise builds is reproducibility. Recall the story of Tom and Joe from chapter 1. Do you remember the last time your coworker said, "But it works on my box"? Builds have to produce the same result on different machines, independent of the contents of your local cache. Dependency managers like Ivy and Maven in their current implementation cannot fully guarantee reproducibility. Why is that? Whenever a dependency is downloaded and stored in the local cache, it doesn't take into account the artifact's origin. In situations where the repository is changed for a project, the cached dependency is considered resolved, even though the artifact's content may be slightly different. At worst, this will cause a failing build that's extremely hard to debug. Another common complaint specific to Ivy is the fact that dependency snapshot versions, artifacts currently under development with the naming convention `-SNAPSHOT`, aren't updated correctly in the local cache, even though it changed on the repository and is marked as changing. There are many more scenarios where current solutions fall short. Gradle provides its own configurable, reliable, and efficient dependency management solution. We'll have a closer look at its features in chapter 5.

Large enterprise projects usually consist of multiple modules to separate functionality. In the Gradle world, each of the submodules is considered a project that can define dependencies to external libraries or other modules. Additionally, each subproject can be run individually. Gradle figures out for you which of the subproject dependencies need to be rebuilt, without having to store a subproject's artifact in the local cache.

2.2.5 Scalable builds

For some companies, a large project with hundreds of modules is reality. Building and testing minor code changes can consume a lot of time. You may know from personal experience that deleting old classes and resources by running a cleanup task is a natural reflex. All too often, you get burned by your build tool not picking up the changes and their dependencies. What you need is a tool that's smart enough to only rebuild the parts of your software that actually changed. Gradle supports incremental builds by specifying task inputs and outputs. It reliably figures out for you which tasks need to be skipped, built, or partially rebuilt. The same concept translates to multimodule projects, called partial builds. Because your build clearly defines the dependencies between submodules, Gradle takes care of rebuilding only the necessary parts. No more running `clean` by default!

Automated unit, integration, and functional tests are part of the build process. It makes sense to separate short-running types of tests from the ones that require setting up resources or external dependencies to be run. Gradle supports parallel test execution. This feature is fully configurable and ensures that you're actually taking advantage of your processor's cores. The buck doesn't stop here. Gradle is going to support distributing test execution to multiple machines in a future version. I'm sorry to tell you, but the days of reading your Twitter feed between long builds are gone.

Developers run builds many times during development. That means starting a new Gradle process each time, loading all its internal dependencies, and running the build logic. You'll notice that it usually takes a couple of seconds before your script actually starts to execute. To improve the startup performance, Gradle can be run in daemon mode. In practice, the Gradle command forks a daemon process, which not only executes your build, but also keeps running in the background. Subsequent build invocations will piggyback on the existing daemon process to avoid the startup costs. As a result, you'll notice a far snappier initial build execution.

2.2.6 Effortless extendibility

Most enterprise builds are not alike, nor do they solve the same problems. Once you're past the initial phase of setting up your basic scripts, you'll want to implement custom logic. Gradle is not opinionated about the way you implement that code. Instead, it gives you various choices to pick from, depending on your specific use case. The easiest way to implement custom logic is by writing a task. Tasks can be defined directly in your build script without special ceremony. If you feel like complexity takes over, you may want to explore the option of a custom task that allows for writing your logic within a class definition, making structuring your code easy and maintainable. If you want to share reusable code among builds and projects, plugins are your best friend. Representing Gradle's most powerful extension mechanism, plugins give you full access to Gradle's API and can be written, tested, and distributed like any other piece of software. Writing a plugin is surprisingly easy and doesn't require a lot of additional descriptors.

2.2.7 Integration with other build tools

Wouldn't it be a huge timesaver to be able to integrate with existing build tools? Gradle plays well with its predecessors Ant, Maven, and Ivy, as shown in figure 2.6.

If you're coming from Ant, Gradle doesn't force you to fully migrate your build infrastructure. Instead, it allows you to import existing build logic and reuse standard Ant tasks. Gradle builds are 100% compatible with Maven and Ivy repositories. You can retrieve dependencies and publish your own artifacts. Gradle provides a converter for existing Maven builds that can translate the build logic into a Gradle build script.

Existing Ant scripts can be imported into your Gradle build seamlessly and used as you'd use any other external Gradle script. Ant targets directly map to Gradle tasks at runtime. Gradle ships with the Ant libraries and exposes a helper class to your scripts

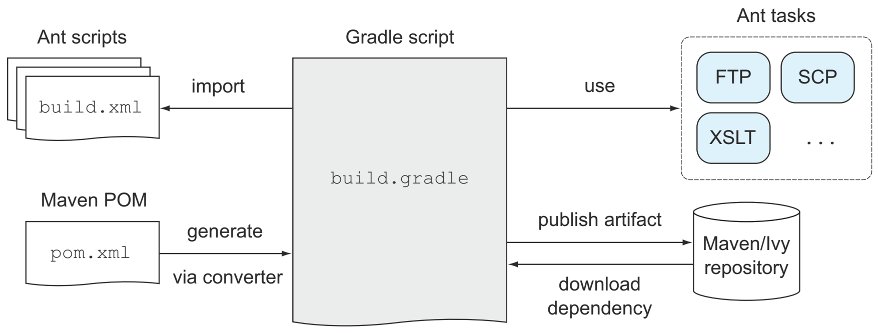


Figure 2.6 Gradle provides deep integration with other build tools and opens the door to gradually migrate your existing Ant or Maven build.

called `AntBuilder`, which fully blends into Gradle's DSL. It still looks and feels like Ant's XML, but without the pointy brackets. Ant users will feel right at home, because they don't have to transition to Gradle syntax right away. Migrating from Ant to Gradle is also a no-brainer. You can take baby steps by reusing your existing Ant logic while using Gradle's benefits at the same time.

Gradle aims to reach a similar depth of integration with Maven. At the time of writing, this hasn't been realized yet. In the long run, Maven POMs and plugins will be treated as Gradle natives. Maven and Ivy repositories have become an important part of today's build infrastructure. Imagine a world without Maven Central to help access specific versions of your favorite project dependencies. Retrieving dependencies from a repository is only one part of the story; publishing to them is just as important. With a little configuration, Gradle can upload your project's artifact for companywide or public consumption.

2.2.8 Community-driven and company-backed

Gradle is free to use and ships with the Apache License 2.0. After its first release in April 2008, a vibrant community quickly started to form around it. Over the past five years, open source developers have made major contributions to Gradle's core code base. Being hosted on GitHub turned out to be very beneficial to Gradle. Code changes can be submitted as pull requests and undergo a close review process by the core committers before making it into the code base. If you're coming from other build tools like Maven, you may be used to a wide range of reusable plugins. Apart from the standard plugins shipped with the runtime, the Gradle community releases new functionality almost daily. Throughout the book, you'll use many of the standard plugins shipped with Gradle. Appendix A gives a broader spectrum on standard as well as third-party plugins. Every community-driven software project needs a forum to get immediate questions answered. Gradle connects with the community through the Gradle forum at http://forums.gradle.org/gradle. You can be sure you'll get helpful responses to your questions on the same day.

Gradleware is the technical service and support company behind Gradle. Not only does it provide professional advice for Gradle itself, it aims for a wide range of enterprise automation consulting. The company is backed by high-caliber engineers very experienced in the domain. Recently, Gradleware started to air free webinars to spark interest for newcomers and deepen knowledge for experienced Gradle users.

2.2.9 Icing on the cake: additional features

Don't you hate having to install a new runtime for different projects? Gradle Wrapper to the rescue! It allows for downloading and installing a fresh copy of the Gradle runtime from a specified repository on any machine you run the build on. This process is automatically triggered on the first execution of the build. The Wrapper is especially useful for sharing your builds with a distributed team or running them on a CI platform.

Gradle is also equipped with a rich command-line interface. Using command-line options, you can control everything from specifying the log level, to excluding tests, to displaying help messages. This is nothing special; other tools provide that, too. Some of the features stand out, though. Gradle allows for running commands in an abbreviated, camel-cased form. In practice, a command named `runMyAwesomeTask` would be callable with the abbreviation `rMAT`. Handy, isn't it? Even though this book presents most of its examples by running commands in a shell, bear in mind that Gradle provides an out-of-the-box graphical user interface.

2.3 The bigger picture: continuous delivery

Being able to build your source code is only one aspect of the software delivery process. More importantly, you want to release your product to a production environment to deliver business value. Along the way, you want to run tests, build the distribution, analyze the code for quality-control purposes, potentially provision a target environment, and deploy to it.

There are many benefits to automating the whole process. First and foremost, delivering software manually is slow, error-prone, and nerve-wracking. I'm sure every one of us hates the long nights due to a deployment gone wrong. With the rise of agile methodologies, development teams are able to deliver software faster. Release cycles of two or three weeks have become the norm. Some organizations like Etsy and Flickr even ship code to production several times a day! Optimally, you want to be able to release software by selecting the target environment simply by pressing a button. Practices like automated testing, CI, and deployment feed into the general concept of continuous delivery.

In this book, we'll look at how Gradle can help get your project from build to deployment. It'll enable you to automate many of the tasks required to implement continuous delivery, be they compiling your source code, deploying a deliverable, or calling external tools that help you with implementing the process. For a deep dive on continuous delivery and all of its aspects, I recommend *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble and David Farley (Addison Wesley, 2010).

2.3.1 Automating your project from build to deployment

Continuous delivery introduces the concept of a deployment pipeline, also referred to as the build pipeline. A deployment pipeline represents the technical implementation of the process for getting software from version control into your production environment. The process consists of multiple stages, as shown in figure 2.7.

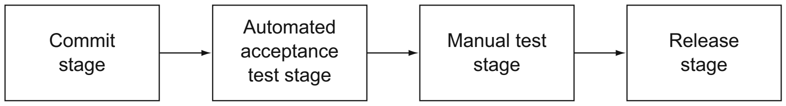


Figure 2.7 Stages of a deployment pipeline

- *Commit stage*: Reports on the technical health level of your project. The main stakeholder of this phase is the development team as it provides feedback about broken code and finds "code smells." The job of this stage is to compile the code, run tests, perform code analysis, and prepare the distribution.
- *Automated acceptance test stage*: Asserts that functional and nonfunctional requirements are met by running automated tests.
- *Manual test stage*: Verifies that the system is actually usable in a test environment. Usually, this stage involves QA personnel to verify requirements on the level of user stories or use cases.
- *Release stage*: Either delivers the software to the end user as a packaged distribution or deploys it to the production environment.

Let's see what stages of the deployment pipeline can benefit from project automation. It's obvious that the manual test stage can be excluded from further discussion, because it only involves manual tasks. This book mainly focuses on using Gradle in the commit and automated acceptance test stages. The concrete tasks we're going to look at are

- Compiling the code
- Running unit and integration tests
- Performing static code analysis and generating test coverage
- Creating the distribution
- Provisioning the target environment
- Deploying the deliverable
- Performing smoke and automated functional tests

Figure 2.8 shows the order of tasks within each of the stages. While there are no hard rules that prevent you from skipping specific tasks, it's recommended that you follow the order. For example, you could decide to compile your code, create the distribution, and deploy it to your target environment without running any tests or static code analysis. However, doing so increases the risk of undetected code defects and poor code quality.

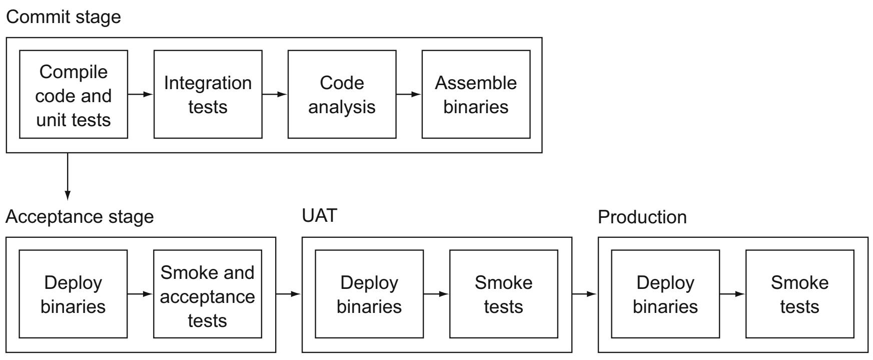


Figure 2.8 Tasks performed in stages of build pipeline

Topics like infrastructure provisioning, automated deployment, and smoke testing can also be applied to the release stage. In practice, applying these techniques to a production environment is more complex than in a controlled test environment. In a production environment, you may have to deal with clustered and distributed server infrastructures, zero-downtime release rollouts, and automated rollbacks to the previous release.

Covering these advanced topics would go beyond the scope of this book. However, there are some great examples of deployment management tools in the wild that you may want to check out, such as Asgard, a web-based cloud management and deployment tool built and used by Netflix (https://github.com/Netflix/asgard). But enough pure theory—let's get your feet wet by installing Gradle on your machine and building your first project. In chapter 3, we'll go even further by exploring how to implement and run a complex Java project using Gradle.

2.4 Installing Gradle

As a prerequisite, make sure you've already installed the JDK with a version of 1.5 or higher. Even though some operating systems provide you with an out-of-the-box Java installation, make sure you have a valid version installed on your system. To check the JDK version, run `java -version`.

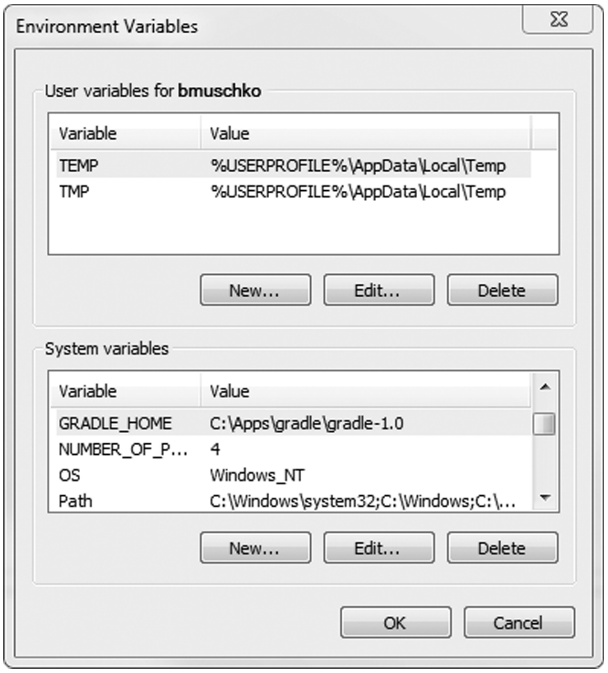
Getting started with Gradle is easy. You can download the distribution directly from the Gradle homepage at http://gradle.org/downloads. As a beginner to the tool, it makes sense to choose the ZIP file that includes the documentation and a wide range of source code examples to explore. Unzip the downloaded file to a directory of your choice. To reference your Gradle runtime in the shell, you'll need to create the environment variable `GRADLE_HOME` and add the binaries to your shell's execution path:

- *Mac OS X and *nix*: To make Gradle available in your shell, add the following two lines to your initialization script (for example, `~/.profile`). These instructions assume that you installed Gradle in the directory `/opt/gradle`:

  ```
  export GRADLE_HOME=/opt/gradle
  export PATH=$PATH:$GRADLE_HOME/bin
  ```

- *Windows*: In the dialog environment variable, define the variable `GRADLE_HOME` and update your path settings (figure 2.9).

You'll verify that Gradle has been installed correctly and is ready to go. To check the version of the installed runtime, issue the command `gradle -v` in your shell. You should see meta-information about the installation, your JVM, and the operating system. The following example shows the version output of a successful Gradle 1.7 installation.


Environment Variables

User variables for **bmuschko**

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
GRADLE_HOME	C:\Apps\gradle\gradle-1.0
NUMBER_OF_P...	4
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\...

Figure 2.9 Updating variable settings on Windows

```
$ gradle -v

------------------------------------------------------------
Gradle 1.7
------------------------------------------------------------

Build time:   2013-08-06 11:19:56 UTC
Build number: none
Revision:     9a7199efaf72c620b33f9767874f0ebced135d83

Groovy:       1.8.6
Ant:          Apache Ant(TM) version 1.8.4 compiled on May 22 2012
Ivy:          2.2.0
JVM:          1.6.0_51 (Apple Inc. 20.51-b01-457)
OS:           Mac OS X 10.8.4 x86_64
```

Setting Gradle's JVM options

Like every other Java application, Gradle shares the same JVM options set by the environment variable `JAVA_OPTS`. If you want to pass arguments specifically to the Gradle runtime, use the environment variable `GRADLE_OPTS`. Let's say you want to increase the default maximum heap size to 1 GB. You could set it like this:

```
GRADLE_OPTS="-Xmx1024m"
```

The preferred way to do that is to add the variable to the Gradle startup script under `$GRADLE_HOME/bin`.

Now that you're all set, you'll implement a simple build script with Gradle. Even though most of the popular IDEs provide a Gradle plugin, all you need now is your favorite editor. Chapter 10 will discuss Gradle plugin support for IDEs like IntelliJ, Eclipse, and NetBeans.

2.5 Getting started with Gradle

Every Gradle build starts with a script. The default naming convention for a Gradle build script is build.gradle. When executing the command gradle in a shell, Gradle looks for a file with that exact name. If it can't be located, the runtime will display a help message.

Let's set the lofty goal of creating the typical "Hello world!" example in Gradle. First you'll create a file called build.gradle. Within that script, define a single atomic piece of work. In Gradle's vocabulary, this is called a task. In this example, the task is called helloWorld. To print the message "Hello world!" make use of Gradle's lingua franca, Groovy, by adding the println command to the task's action doLast. The method println is Groovy's shorter equivalent to Java's System.out.println:

```
task helloWorld {
   doLast {
      println 'Hello world!'
   }
}
```

Give it a spin:

```
$ gradle -q helloWorld
Hello world!
```

As expected, you see the output "Hello world!" when running the script. By defining the optional command-line option quiet with -q, you tell Gradle to only output the task's output.

Without knowing it, you already used Gradle's DSL. Tasks and actions are important elements of the language. An action named doLast is almost self-expressive. It's the last action that's executed for a task. Gradle allows for specifying the same logic in a more concise way. The left shift operator << is a shortcut for the action doLast. The following snippet shows a modified version of the first example:

```
task helloWorld << {
   println 'Hello world!'
}
```

Printing "Hello world!" only goes so far. I'll give you a taste of more advanced features in the example build script shown in the following listing. Let's strengthen our belief in Gradle by exercising a little group therapy session. Repeat after me: Gradle rocks!

Listing 2.1 Dynamic task definition and task chaining

```
task startSession << {
   chant()
}
```

```
def chant() {
   ant.echo(message: 'Repeat after me...')        ❶ Implicit Ant task usage
}

3.times {
   task "yayGradle$it" << {                       ❷ Dynamic task definition
       println 'Gradle rocks'
   }
}

yayGradle0.dependsOn startSession                 ❸ Task dependencies
yayGradle2.dependsOn yayGradle1, yayGradle0
task groupTherapy(dependsOn: yayGradle2)
```

You may not notice it at first, but there's a lot going on in this listing. You introduced the keyword `dependsOn` to indicate dependencies between tasks ❸. Gradle makes sure that the depended-on task will always be executed before the task that defines the dependency. Under the hood, `dependsOn` is actually a method of a task. Chapter 4 will cover the internals of tasks, so we won't dive into too much detail here.

A feature we've talked about before is Gradle's tight integration with Ant ❶. Because you have full access to Groovy's language features, you can also print your message in a method named `chant()`. This method can easily be called from your task. Every script is equipped with a property called `ant` that grants direct access to Ant tasks. In this example, you print out the message "Repeat after me" using the Ant task `echo` to start the therapy session.

A nifty feature Gradle provides is the definition of dynamic tasks, which specify their name at runtime. Your script creates three new tasks within a loop ❷ using Groovy's `times` method extension on `java.lang.Number`. Groovy automatically exposes an implicit variable named `it` to indicate the loop iteration index. You're using this counter to build the task name. For the first iteration, the task would be called `yayGradle0`.

Now running `gradle groupTherapy` results in the following output:

```
$ gradle groupTherapy
:startSession
[ant:echo] Repeat after me...
:yayGradle0
Gradle rocks
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

As shown in figure 2.10 Gradle executed the tasks in the correct order. You may have noticed that the example omitted the `quiet` command-line option, which gives more information on the tasks run.

Thanks to your group therapy, you got rid of your deepest fears that Gradle will be just another build tool that can't deliver. In the next chapter, you'll stand up a

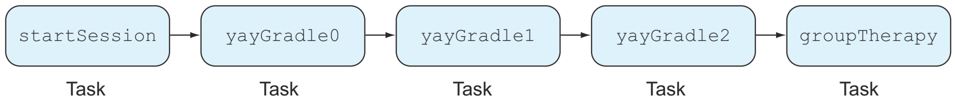


Figure 2.10 Task dependency graph

full-fledged Java application covering a broad range of Gradle's core concepts. For now, let's get more accustomed to Gradle's command line.

2.6 Using the Command line

In the previous sections, you executed the tasks `helloWorld` and `groupTherapy` on the command line, which is going to be your tool of choice for running most examples throughout this book. Even though using an IDE may seem more convenient to newcomers, a deep understanding of Gradle's command-line options and helper tasks will make you more efficient and productive in the long run.

2.6.1 Listing available tasks of a project

In the last section I showed you how to run a specific task using the `gradle` command. Running a task requires you to know the exact name. Wouldn't it be great if Gradle could tell you which tasks are available without you having to look at the source code? Gradle provides a helper task named `tasks` to introspect your build script and display each available task, including a descriptive message of its purpose. Running `gradle tasks` in quiet mode produces the following output:

```
$ gradle -q tasks

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

Build Setup tasks
-----------------
setupBuild - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
dependencies - Displays the dependencies of root project 'grouptherapy'.
dependencyInsight - Displays the insight into a specific dependency in root
➥ project 'grouptherapy'.
help - Displays a help message
projects - Displays the sub-projects of root project 'grouptherapy'.
properties - Displays the properties of root project 'grouptherapy'.
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
➥ the displayed tasks may belong to subprojects).

Other tasks
-----------
groupTherapy

To see all tasks and more detail, run with --all.
```

Build setup tasks help you initialize the Gradle build (for example, generate the build.gradle file)

1 Help task group listing task names and their descriptions separated by hyphen

2 Uncategorized tasks that are not assigned to a task group

3 Tasks without descriptions aren't self-expressive; in chapter 3 you'll learn how to add an appropriate task description.

There are some things to note about the output. Gradle provides the concept of a task group, which can be seen as a cluster of tasks assigned to that group. Out of the box, each build script exposes the task group `Help tasks` ❶ without any additional work from the developer. If a task doesn't belong to a task group, it's displayed under `Other tasks` ❷. This is where you find the task `groupTherapy` ❸. We'll look at how to add a task to a task group in chapter 4.

You may wonder what happened to the other tasks that you defined in your build script. On the bottom of the output, you'll find a note that you can get even more details about your project's tasks by using the `--all` option. Run it to get more information on them:

```
$ gradle -q tasks --all

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

Build Setup tasks
-----------------
setupBuild - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
dependencies - Displays the dependencies of root project 'grouptherapy'.
help - Displays a help message
projects - Displays the sub-projects of root project 'grouptherapy'.
properties - Displays the properties of root project 'grouptherapy'.
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
➥ the displayed tasks may belong to subprojects).

Other tasks
-----------
groupTherapy
    startSession
    yayGradle0
    yayGradle1
    yayGradle2
```

❶ **Root task of dependency graph**

Indented names of dependent tasks listed in order of execution

The `--all` option is a great way to determine the execution order of a task graph before actually executing it. To reduce the noise, Gradle is smart enough to hide tasks that act as dependencies to a root task ❶. For better readability, dependent tasks are displayed indented and ordered underneath the root task.

2.6.2 Task execution

In the previous examples, you told Gradle to execute one specific task by adding it as an argument to the command `gradle`. Gradle's command-line implementation will in turn make sure that the task and all its dependencies are executed. You can also execute multiple tasks in a single build run by defining them as command-line parameters. Running `gradle yayGradle0 groupTherapy` would execute the task `yayGradle0` first and the task `groupTherapy` second.

Tasks are always executed just once, no matter whether they're specified on the command line or act as a dependency for another task. Let's see what the output looks like:

```
$ gradle yayGradle0 groupTherapy
:startSession
[ant:echo] Repeat after me...
:yayGradle0
Gradle rocks
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

No surprises here. You see the same output as if you'd just run `gradle groupTherapy`. The correct order was preserved and each of the tasks was only executed once.

TASK NAME ABBREVIATION

One of Gradle's productivity tools is the ability to abbreviate camel-cased task names on the command line. If you wanted to run the previous example in the abbreviated form, you'd just have to type `gradle yG0 gT`. This is especially useful if you're dealing with very long task names or multiple task arguments. Keep in mind that the task name abbreviation has to be unique to enable Gradle to identify the corresponding task. Consider the following scenario:

```
task groupTherapy << {
   ...
}

task generateTests << {
   ...
}
```

Using the abbreviation `gT` in a build that defines the tasks `groupTherapy` and `generate-Tests` causes Gradle to display an error:

```
$ gradle yG0 gT

FAILURE: Could not determine which tasks to execute.

* What went wrong:
Task 'gT' is ambiguous in root project 'grouptherapy'. Candidates are:
➥ 'generateTests', 'groupTherapy'.

* Try:
Run gradle tasks to get a list of available tasks.

BUILD FAILED
```

EXCLUDING A TASK FROM EXECUTION

Sometimes you want to exclude a specific task from your build run. Gradle provides the command-line option `-x` to achieve that. Let's say you want to exclude the task `yayGradle0`:

```
$ gradle groupTherapy -x yayGradle0
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

Gradle excluded the task `yayGradle0` and its dependent task `startSession`, a concept Gradle calls *smart exclusion*. Now that you're becoming familiar with the command line, let's explore some more helpful functions.

2.6.3 Command-line options

In this section, we explore the most important general-purpose options, flags to control your build script's logging level, and ways to provide properties to your project. The `gradle` command allows you to define one or more options at the same time. Let's say you want to change the log level to INFO using the `-i` option *and* print out any stack trace if an error occurs during execution with the option `-s`. To do so, execute the task `groupTherapy` command like this: `gradle groupTherapy -is` or `gradle groupTherapy -i -s`. As you can see, it's very easy to combine multiple options. To discover the full set, run your build with the `-h` argument or see appendix A of this book. I won't go over all the available options, but the most important ones are as follows:

- *`-?, -h, --help`*: Prints out all available command-line options including a descriptive message.
- *`-b, --build-file`*: The default naming convention for Gradle build script is `build.gradle`. Use this option to execute a build script with a different name (for example, `gradle -b test.gradle`).
- *`--offline`*: Often your build declares dependencies on libraries only available in repositories outside of your network. If these dependencies were not stored in your local cache yet, running a build without a network connection to these repositories would result in a failed build. Use this option to run your build in offline mode and only check the local dependency cache for dependencies.

Property options

- *`-D, --system-prop`*: Gradle runs as a JVM process. As with all Java processes, you can provide a system property like this: `-Dmyprop=myvalue`.
- *`-P, --project-prop`*: Project properties are variables available in your build script. You can use this option to pass a property to the build script directly from the command line (for example, `-Pmyprop=myvalue`).

Logging options

- *`-i, --info`*: In the default settings, a Gradle build doesn't output a lot of information. Use this option to get more informative messages by changing Gradle's logger to INFO log level. This is helpful if you want to get more information on what's happening under the hood.

- `-s, --stacktrace`: If you run into errors in your build, you'll want to know where they stem from. The option `-s` prints out an abbreviated stack trace if an exception is thrown, making it perfect for debugging broken builds.
- `-q, --quiet`: Reduces the log messages of a build run to error messages only.

HELP TASKS

- `tasks`: Displays all runnable tasks of your project including their descriptions. Plugins applied to your project may provide additional tasks.
- `properties`: Emits a list of all available properties in your project. Some of these properties are provided by Gradle's project object, the build's internal representation. Other properties are user-defined properties originating from a property file or property command-line option, or directly declared in your build script.

2.6.4 Gradle daemon

When using Gradle on a day-to-day basis, you'll find yourself having to run your build repetitively. This is especially true if you're working on a web application. You change a class, rebuild the web application archive, bring up the server, and reload the URL in the browser to see your changes being reflected. Many developers prefer test-driven development to implement their application. For continuous feedback on their code quality, they run their unit tests over and over again to find code defects early on. In both cases, you'll notice a significant productivity hit. Each time you initiate a build, the JVM has to be started, Gradle's dependencies have to be loaded into the class loader, and the project object model has to be constructed. This procedure usually takes a couple of seconds. Gradle daemon to the rescue!

The daemon runs Gradle as a background process. Once started, the `gradle` command will reuse the forked daemon process for subsequent builds, avoiding the startup costs altogether. Let's come back to the previous build script example. On my machine, it takes about three seconds to successfully complete running the task `groupTherapy`. Hopefully, we can improve the startup and execution time. It's easy to start the Gradle daemon on the command line: simply add the option `--daemon` to your `gradle` command. You may notice that we add a little extra time for starting up the daemon as well. To verify that the daemon process is running, you can check the process list on your operating system:

- *Mac OS X and *nix*: In a shell run the command `ps | grep gradle` to list the processes that contain the name gradle.
- *Windows*: Open the task manager with the keyboard shortcut Ctrl+Shift+Esc and click the Processes tab.

Subsequent invocations of the `gradle` command will now reuse the daemon process. Give it a shot and try running `gradle groupTherapy --daemon`. Wow, you got your startup and execution time down to about one second! Keep in mind that a daemon process will only be forked once even though you add the command-line option `--daemon`. The daemon process will automatically expire after a three-hour idle time. At any time you can choose to execute your build without using the daemon by adding the command-line option `--no-daemon`. To stop the daemon process, manually run `gradle --stop`. That's the Gradle daemon in a nutshell. For a deep dive into all configuration options and intricacies, please refer to the Gradle online documentation at http://gradle.org/docs/current/userguide/gradle_daemon.html.

2.7 Summary

Existing tools can't meet the build needs of today's industry. Improving on the best ideas of its competitors, Gradle provides a build-by-convention approach, reliable dependency management, and support for multiproject builds without having to sacrifice the flexibility and descriptiveness of your build.

In this chapter, we explored how Gradle can be used to deliver in each of the phases of a deployment pipeline in the context of continuous delivery. Throughout the book, we'll pick up on each of the phases by providing practical examples.

Next, you got a first taste of Gradle's powerful features in action. You installed the runtime, wrote a first simple build script, and executed it. By implementing a more complex build script, you found out how easy it is to define task dependencies using Gradle's DSL. Knowing the mechanics of Gradle's command line and its options is key to becoming highly productive. Gradle offers a wide variety of command-line switches for changing runtime behavior, passing properties to your project, and changing the logging level. We explored how running Gradle can be a huge timesaver if you have to continuously execute tasks, such as during test-driven development.

In chapter 3, I'll show how to build a full-fledged, web-enabled application with Gradle. Starting out with a simple, standalone Java application, you'll extend the code base by adding a web component and use Gradle's in-container web development support to efficiently implement the solution. We won't stop there. I'm going to show how to enhance your web archive to make it enterprise-ready and make the build transferable across machines without having to install the Gradle runtime.

JAVA/PROGRAMMING

Gradle IN ACTION

Benjamin Muschko

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

Free eBook
SEE INSERT

"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]