Community Experience Distilled

Gradle Effective Implementations Guide

Second Edition

A comprehensive guide to get up and running with build automation using Gradle

Hubert Klein Ikkink

[PACKT] open source* PUBLISHING community experience distilled

Gradle Effective Implementations Guide

Second Edition

A comprehensive guide to get up and running with build automation using Gradle

Hubert Klein Ikkink

BIRMINGHAM - MUMBAI

Gradle Effective Implementations Guide

Second Edition

First published: October 2012

Second edition: May 2016

Production reference: 1250516

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78439-497-4

www.packtpub.com

Credits

Author

Hubert Klein Ikkink

Reviewer

Izzet Mustafaiev

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Manish Nainani

Content Development Editor

Shweta Pant

Technical Editor

Suwarna Patil

Copy Editor

Vibha Shukla

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Disha Haria

Production Coordinator

Nilesh Mohite

About the Author

Hubert Klein Ikkink was born in 1973 and currently lives in Tilburg, Netherlands, with his beautiful wife and gorgeous children. He is also known as mrhaki, which is simply the first letters of his name prepended with mr. He studied information systems and management at Tilburg University. After finishing his studies, he started working at a company specialized in knowledge-based software. There he started writing his first Java software (yes, an applet!) in 1996. During these years, his focus switched from applets to servlets to Java Enterprise Edition applications to Spring-based software.

In 2008, he wanted to have fun when writing software. The larger projects he was working on were more about writing configuration XML files and tuning performance and less about real development in his eyes, so he started to look around and noticed that Groovy was a very good language to learn. He could still use the existing Java code and libraries and use his Groovy classes in Java. The learning curve isn't steep and to support his learning phase, he wrote interesting Groovy facts on his blog with the title Groovy Goodness. He posts small articles with a lot of code samples to understand how to use Groovy. Since November 2011, he is also a DZone Most Valuable Blogger (MVB), where DZone post his blog items on their site. During these years, he also wrote about other subjects such as Grails, Gradle, Spock, Asciidoctor, and Ratpack.

Hubert was invited to speak at conferences such as Gr8Conf in Copenhagen, Minneapolis, and Greach, Madrid. Also, he gave presentations at Java conferences such as JFall in Netherlands and Javaland in Germany.

Hubert works for a company called JDriven in Netherlands. JDriven focuses on technologies that simplify and improve development of enterprise applications. Employees of JDriven have years of experience with Java and related technologies and are all eager to learn about new technologies. Hubert works on projects using Grails and Java combined with Groovy and Gradle.

It was a great honor to be asked by Packt Publishing to write this book. I knew it beforehand that it would be a lot of work and somehow needed to be combined with my daytime job. I couldn't have written the book without the help of a lot of people and I want to thank all these people.

First of all, I would like to thank my family for supporting me while writing the book. They gave me space and time to write the book. Thank you for your patience and a big kiss to Kim, Britt, Liam, and Chloë, I love you. Of course, I would also like to thank all the people at Gradle Inc. for making Gradle such a great build tool.

Finally, I would like to thank the great staff at Packt Publishing. Shweta Pant kept me on schedule and made sure everything was submitted on time. I would also like to thank all the editors for reviewing the book. They really helped me to keep focus and be concise with the text.

About the Reviewer

Izzet Mustafaiev is a family guy with a tendency of having BBQ parties and traveling.

Professionally, he's a software engineer, working with EPAM Systems with primary language—Java, hands on Groovy/Ruby, and exploring FP with Erlang/Elixir. Izzet has participated in different projects as a developer and architect. He advocates XP, clean code, and DevOps habits and practices, speaking at engineering conferences.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Gradle is the next generation in build automation. Gradle uses convention-over-configuration to provide good defaults, but it is also flexible to be used in every situation you encounter in daily development. Build logic is described with a powerful DSL and empowers developers to create reusable and maintainable build logic.

We will see more about Gradle in this book. We will discuss Gradle's features with code samples throughout the book. We will also discuss how to write tasks, work with files, and write build scripts using the Groovy DSL. Next, we will discuss how to use Gradle in projects to compile, package, test, check code quality, and deploy applications. Finally we will see how to integrate Gradle with continuous integration servers and development environments (IDEs).

After reading this book, you will know how to use Gradle in your daily development. We can write tasks, apply plugins, and write build logic using the Gradle build language.

What this book covers

`Chapter 1`, *Starting with Gradle*, introduces Gradle and how to install it. We will write our first Gradle script and discuss command-line and GUI features of Gradle.

`Chapter 2`, *Creating Gradle Build Scripts*, looks at tasks as part of the Gradle build scripts. We will see how to define tasks and how to use task dependencies to describe the build logic.

`Chapter 3`, *Working with Gradle Build Scripts*, covers more functionalities that we can apply in Gradle scripts. We will discuss how to work with files and directories, apply logging to our build scripts, and use properties to parameterize our build scripts.

`Chapter 4`, Using Gradle for Java Projects, discusses all about using the Java plugin for Gradle projects. Gradle offers several tasks and configuration convention that makes working with Java project very easy. We see how to customize the configuration for project that cannot follow the conventions.

`Chapter 5`, *Dependency Management*, covers the support for dependencies by Gradle. We will discuss how to use configurations to organize dependencies. We will also see how to use repositories with dependencies in our build scripts.

`Chapter 6`, *Testing, Building, and Publishing Artifacts*, introduces the support of Gradle to run tests from the build script. We will discuss how to build several artifacts for a project and how to publish the artifacts to a repository so that other developers can reuse our code.

`Chapter 7`, *Multi-project Builds*, covers Gradle's support for multi-project builds. With Gradle, we can configure multiple projects that can be related to each other easily. We will also see how Gradle can automatically build related or dependent projects if necessary.

`Chapter 8`, *Mixed Languages*, explains the Scala and Groovy plugin that is included with Gradle to work with projects that have Scala or Groovy code.

`Chapter 9`, *Maintaining Code Quality*, introduces the code quality plugins of Gradle. We will see how to use and configure the plugins to include code analysis in our build process.

`Chapter 10`, *Writing Custom Tasks and Plugins*, introduces what we need to do to write our own custom task and plugins. We will see how to decouple the definition and usage of a custom task and plugin in separate source files. We will also discuss how to reuse our custom task and plugin in other projects.

`Chapter 11`, *Gradle in the Enterprise*, introduces the support of several continuous integration tools for Gradle. We will discuss how to configure a continuous integration server to automatically invoke our Gradle build scripts.

`Chapter 12`, *IDE Support*, looks at how Gradle can generate project files for Eclipse and IntelliJ IDEA. We will also see how the IDEs support Gradle from within the IDE in order to run, for example, tasks and keep track of dependencies defined in Gradle scripts.

What you need for this book

In order to work with Gradle and the code samples in the book, we need at least a Java Development Toolkit (JDK, 1.6 or higher version), Gradle, and a good text editor. In the first chapter, we will see how to install Gradle on our computer.

Who this book is for

You are working on Java (Scala and Groovy) applications and want to use build automation to automatically compile, package, and deploy your application. You might have worked with other build automation tools, such as Maven or ANT, but this is not necessary to understand the topics in this book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the include directive."

A block of code is set as follows:

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

Any command-line input or output is written as follows:

```
$ gradle -v
------------------------------------------------------------
Gradle 2.12
------------------------------------------------------------
Build time: 2016-03-14 08:32:03 UTC
Chapter 1
[ 5 ]
 Build number: none
 Revision: b29fbb64ad6b068cb3f05f7e40dc670472129bc0
 Groovy: 2.4.4
 Ant: Apache Ant(TM) version 1.9.3 compiled on
 December23 2013
 JVM: 1.8.0_66 (Oracle Corporation 25.66-b17)
 OS: Mac OS X 10.11.3 x86_64
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "With the **Gradle GUI**, we have a graphical overview of the tasks in a project and we can execute them by simply clicking on the mouse."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Gradle-Effective-Implementations-Guide-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the **Errata** section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Starting with Gradle

When we develop a software, we write, compile, test, package, and finally, distribute the code. We can automate these steps by using a build system. The big advantage is that we have a repeatable sequence of steps. The build system will always follow the steps that we have defined, so we can concentrate on writing the actual code and don't have to worry about the other steps.

Gradle is one such build system.

In this chapter, we will cover the following topics:

- Getting started with Gradle
- Writing our first build script
- Looking at default tasks
- Learning about command-line options
- Discussing the Gradle graphical user interface

Gradle is a tool for build automation. With Gradle, we can automate compiling, testing, packaging, and deployment of our software or any other types of projects. Gradle is flexible, but has sensible defaults for most projects. This means that we can rely on the defaults if we don't want something special, but we can still can use the flexibility to adapt a build to certain custom needs.

Gradle is already used by large open source projects such as Spring, Hibernate, and Grails. Enterprise companies such as LinkedIn and Netflix also use Gradle.

In this chapter, we will explain what Gradle is and how to use it in our development projects.

Let's take a look at some of Gradle's features.

Declarative builds and convention over configuration

Gradle uses a **Domain Specific Language** (**DSL**) based on Groovy to declare builds. The DSL provides a flexible language that can be extended by us. As the DSL is based on Groovy, we can write Groovy code to describe a build and use the power and expressiveness of the Groovy language. Groovy is a language for the **Java Virtual Machine** (**JVM**), such as Java and Scala. Groovy makes it easy to work with collections, has closures, and a lot of useful features. The syntax is closely related to the Java syntax. In fact, we could write a Groovy class file with Java syntax and it will compile. However, using the Groovy syntax makes it easier to express the code intent and we need less boilerplate code than with Java. To get the most out of Gradle, it is best to also learn the basics of the Groovy language, but it is not necessary to start writing Gradle scripts.

Gradle is designed to be a build language and not a rigid framework. The Gradle core itself is written in Java and Groovy. To extend Gradle, we can use Java and Groovy to write our custom code. We can even write our custom code in Scala if we want to.

Gradle provides support for Java, Groovy, Scala, web, and OSGi projects out of the box. These projects have sensible convention-over-configuration settings that we probably already use ourselves. However, we have the flexibility to change these configuration settings if required for our projects.

Support for Ant Tasks and Maven repositories

Gradle supports **Ant Tasks** and projects. We can import an Ant build and reuse all the tasks. However, we can also write Gradle tasks dependent on Ant Tasks. The integration also applies for properties, paths, and so on.

Maven and Ivy repositories are supported to publish or fetch dependencies. So, we can continue to use any repository infrastructure that we already have.

Incremental builds

With Gradle, we have incremental builds. This means the tasks in a build are only executed if necessary. For example, a task to compile source code will first check whether the sources have changed since the last execution of the task. If the sources have changed, the task is executed; but if the sources haven't changed, the execution of the task is skipped and the task is marked as being up to date.

Gradle supports this mechanism for a lot of provided tasks. However, we can also use this for tasks that we write ourselves.

Multi-project builds

Gradle has great support for multi-project builds. A project can simply be dependent on other projects or be a dependency of other projects. We can define a graph of dependencies among projects, and Gradle can resolve these dependencies for us. We have the flexibility to define our project layout as we want.

Gradle has support for partial builds. This means that Gradle will figure out whether a project, which our project depends on, needs to be rebuild or not. If the project needs rebuilding, Gradle will do this before building our own project.

Gradle Wrapper

The **Gradle Wrapper** allows us to execute Gradle builds even if Gradle is not installed on a computer. This is a great way to distribute source code and provide the build system with it so that the source code can be built.

Also in an enterprise environment, we can have a zero-administration way for client computers to build the software. We can use the wrapper to enforce a certain Gradle version to be used so that the whole team is using the same version. We can also update the Gradle version for the wrapper, and the whole team will use the newer version as the wrapper code is checked in to version control.

Free and open source

Gradle is an open source project and it is licensed under the **Apache License** (**ASL**).

Getting started

In this section, we will download and install Gradle before writing our first Gradle build script.

Before we install Gradle, we must make sure that we have a **Java Development SE Kit (JDK)** installed on our computer. Gradle requires JDK 6 or higher. Gradle will use the JDK found on the path of our computer. We can check this by running the following command on the command line:

```
$ java –version
```

Although Gradle uses Groovy, we don't have to install Groovy ourselves. Gradle bundles the Groovy libraries with the distribution and will ignore a Groovy installation that is already available on our computer.

Gradle is available on the Gradle website at `http://www.gradle.org/downloads`. From this page, we can download the latest release of Gradle. We can also download an older version if we want. We can choose among three different distributions to download. We can download the complete Gradle distribution with binaries, sources, and documentation; or we can only download the binaries; or we can only download the sources.

To get started with Gradle, we will download the standard distribution with the binaries, sources, and documentation. At the time of writing this book, the current release is 2.12.

Installing Gradle

Gradle is packaged as a ZIP file for one of the three distributions. So when we have downloaded the Gradle full-distribution ZIP file, we must unzip the file. After unpacking the ZIP file we have:

- Binaries in the `bin` directory
- Documentation with the user guide, Groovy DSL, and API documentation in the `doc` directory
- A lot of samples in the `samples` directory
- Source code of Gradle in the `src` directory
- Supporting libraries for Gradle in the `lib` directory
- A directory named `init.d`, where we can store Gradle scripts that need to be executed each time we run Gradle

Once we have unpacked the Gradle distribution to a directory, we can open a command prompt. We go to the directory where we have installed Gradle. To check our installation, we run `gradle -v` and get an output with the used JDK and library versions of Gradle, as follows:

```
$ gradle -v
------------------------------------------------------------
Gradle 2.12
------------------------------------------------------------
Build time:   2016-03-14 08:32:03 UTC
Build number: none
Revision:     b29fbb64ad6b068cb3f05f7e40dc670472129bc0
Groovy:       2.4.4
Ant:          Apache Ant(TM) version 1.9.3 compiled on
 December23 2013
JVM:          1.8.0_66 (Oracle Corporation 25.66-b17)
OS:           Mac OS X 10.11.3 x86_64
```

Here, we can check whether the displayed version is the same as the distribution version that we have downloaded from the Gradle website.

To run Gradle on our computer, we have to only add `$GRADLE_HOME/bin` to our `PATH` environment variable. Once we have done that, we can run the `gradle` command from every directory on our computer.

If we want to add **JVM** options to Gradle, we can use the `JAVA_OPTS` and `GRADLE_OPTS` environment variables. `JAVA_OPTS` is a commonly used environment variable name to pass extra parameters to a Java application. Gradle also uses the `GRADLE_OPTS` environment variable to pass extra arguments to Gradle. Both environment variables are used, so we can even set them both with different values. This is mostly used to set, for example, an HTTP proxy or extra memory options.

Installing with SKDMAN!

Software Development Kit Manager (**SDKMAN**!) is a tool to manage versions of software development kits such as Gradle. Once we have installed SKDMAN!, we can simply use the `install` command and SDKMAN! downloads Gradle and makes sure that it is added to our `$PATH` variable. SDKMAN! is available for Unix-like systems, such as Linux, Mac OSX, and Cygwin (on Windows).

First, we need to install SDKMAN! with the following command in our shell:

```
$ curl -s get.sdkman.io | bash
```

Next, we can install Gradle with the `install` command:

```
$ sdk install gradle
Downloading: gradle 2.12
% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
      0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:-
-     0
      0   354    0     0    0     0      0      0 --:--:-- --:--:-- --:--:-
-     0
    100 42.6M  100 42.6M    0     0  1982k      0  0:00:22  0:00:22 --:--:-
- 3872k
Installing: gradle 2.12
Done installing!
Do you want gradle 2.12 to be set as default? (Y/n): Y
Setting gradle 2.12 as default.
```

If we have multiple versions of Gradle, it is very easy to switch between versions with the `use` command:

```
$ sdk use gradle 2.12
 Using gradle version 2.12 in this shell.
```

Writing our first build script

We now have a running Gradle installation. It is time to create our first Gradle build script. Gradle uses the concept of projects to define a related set of tasks. A Gradle build can have one or more projects. A project is a very broad concept in Gradle, but it is mostly a set of components that we want to build for our application.

A project has one or more tasks. Tasks are a unit of work that need to be executed by the build. Examples of tasks are compiling source code, packaging class files into a JAR file, running tests, and deploying the application.

We now know a task is a part of a project, so to create our first task, we also create our first Gradle project. We use the `gradle` command to run a build. Gradle will look for a file named `build.gradle` in the current directory. This file is the build script for our project. We define our tasks that need to be executed in this build script file.

We create a new `build.gradle` file and open this in a text editor. We type the following code to define our first Gradle task:

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

With this code, we will define a `helloWorld` task. The task will print the words `Hello world.` to the console. The `println` is a **Groovy** method to print text to the console and is basically a shorthand version of the `System.out.println` Java method.

The code between the brackets is a **closure**. A closure is a code block that can be assigned to a variable or passed to a method. Java doesn't support closures, but Groovy does. As Gradle uses Groovy to define the build scripts, we can use closures in our build scripts.

The `<<` syntax is, technically speaking, an operator shorthand for the `leftShift()` method, which actually means **add to**. Therefore, here we are defining that we want to add the closure (with the `println 'Hello world'` statement) to our task with the `helloWorld` name.

First, we save `build.gradle`, and with the `gradle helloWorld` command, we execute our build:

```
$ gradle helloWorld
:helloWorld
Hello world.
BUILD SUCCESSFUL
Total time: 2.384 secs
This build could be faster, please consider using the Gradle  Daemon:
https://docs.gradle.org/2.12/userguide/gradle_daemon.html
```

The first line of output shows our line `Hello world`. Gradle adds some more output such as the fact that the build was successful and the total time of the build. As Gradle runs in the JVM, every time we run a Gradle build, the JVM must be also started. The last line of the output shows a tip that we can use the Gradle daemon to run our builds. We will discuss more about the Gradle daemon later, but it essentially keeps Gradle running in memory so that we don't get the penalty of starting the JVM each time we run Gradle. This drastically speeds up the execution of tasks.

We can run the same build again, but only with the output of our task using the Gradle `--quiet` or `-q` command-line option. Gradle will suppress all messages except error messages. When we use the `--quiet` (or `-q`) option, we get the following output:

```
$ gradle --quiet helloWorld
Hello world.
```

Default Gradle tasks

We can create our simple build script with one task. We can ask Gradle to show us the available tasks for our project. Gradle has several built-in tasks that we can execute. We type `gradle -q` tasks to see the tasks for our project:

```
$ gradle -q tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------
Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]
Help tasks
----------
components - Displays the components produced by root project
'hello-world'. [incubating]
dependencies - Displays all dependencies declared in root project  'hello-
world'.
dependencyInsight - Displays the insight into a specific n      dependency
in root project 'hello-world'.
help - Displays a help message.
model - Displays the configuration model of root project   'hello-world'.
[incubating]
projects - Displays the sub-projects of root project 'hello-world'.
properties - Displays the properties of root project 'hello-world'.
tasks - Displays the tasks runnable from root project 'hello-world'.
Other tasks
-----------
helloWorld
To see all tasks and more detail, run gradle tasks --all
To see more detail about a task, run gradle help --task <task>
```

Here, we see our `helloWorld` task in the `Other tasks` section. The Gradle built-in tasks are displayed in the `Help tasks` section. For example, to get some general help information, we execute the `help` task:

```
$ gradle -q help
Welcome to Gradle 2.12.
To run a build, run gradle <task> ...
To see a list of available tasks, run gradle tasks
To see a list of command-line options, run gradle --help
To see more detail about a task, run gradle help --task <task>
```

The `properties` task is very useful to see the properties available for our project. We haven't defined any property ourselves in the build script, but Gradle provides a lot of built-in properties. The following output shows some of the properties:

```
$ gradle -q properties
------------------------------------------------------------
Root project
------------------------------------------------------------
allprojects: [root project 'hello-world']
ant: org.gradle.api.internal.project.DefaultAntBuilder@3bd3d05e
antBuilderFactory:
org.gradle.api.internal.project.DefaultAntBuilderFactory@6aba5d30
artifacts:
org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@61d3
4b4
asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@588307f7
baseClassLoaderScope:
org.gradle.api.internal.initialization.DefaultClassLoaderScope@7df76d99
buildDir: /Users/mrhaki/Projects/gradle-effective-implementation-
guide-2/gradle-impl-guide-2/src/docs/asciidoc/Chapter1/Code_Files/hello-
world/build
buildFile: /Users/mrhaki/Projects/gradle-effective-implementation-
guide-2/gradle-impl-guide-2/src/docs/asciidoc/Chapter1/Code_Files/hello-
world/build.gradle
buildScriptSource: org.gradle.groovy.scripts.UriScriptSource@459cfcca
buildscript:
org.gradle.api.internal.initialization.DefaultScriptHandler@2acbc859
childProjects: {}
class: class org.gradle.api.internal.project.DefaultProject_Decorated
classLoaderScope:
org.gradle.api.internal.initialization.DefaultClassLoaderScope@6ab7ce48
components: []
configurationActions:
org.gradle.configuration.project.DefaultProjectConfigurationActionContainer
@2c6aed22 ...
```

The `dependencies` task will show dependencies (if any) for our project. Our first project doesn't have any dependencies, as the output shows when we run the task:

```
$ gradle -q dependencies
------------------------------------------------------------
Root project
------------------------------------------------------------
No configurations
```

The `projects` tasks will display subprojects (if any) for a root project. Our project doesn't have any subprojects. Therefore, when we run the `projects` task, the output shows us our project has no subprojects:

```
$ gradle -q projects
------------------------------------------------------------
Root project
------------------------------------------------------------
Root project 'hello-world'
No sub-projects
To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :tasks
```

The `model` tasks displays information about the model that Gradle builds internally from our project build file. This feature is incubating, which means that the functionality can change in future versions of Gradle:

```
$ gradle -q model
------------------------------------------------------------
Root project
------------------------------------------------------------
 + model
    + tasks
         | Type:       org.gradle.model.ModelMap<org.gradle.api.Task>
           | Creator:     Project.<init>.tasks()
           + components
                 | Type:
org.gradle.api.reporting.components.ComponentReport
                 | Value:      task ':components'
                 | Creator:     tasks.addPlaceholderAction(components)
                 | Rules:
                   copyToTaskContainer
           + dependencies
                 | Type:
org.gradle.api.tasks.diagnostics.DependencyReportTask
                 | Value:      task ':dependencies'
                 | Creator:     tasks.addPlaceholderAction(dependencies)
                 | Rules:
    ...
```

We will discuss more about these and the other tasks in this book.

Task name abbreviation

Before we look at more Gradle command-line options, it is good to discuss a real-time save feature of Gradle: task name abbreviation. With task name abbreviation, we don't have to type the complete task name on the command line. We only have to type enough of the name to make it unique within the build.

In our first build, we only have one task, so the `gradle h` command should work just fine. However, we didn't take the built-in `help` task into account. So, to uniquely identify our `helloWorld` task, we use the `hello` abbreviation:

```
$ gradle -q hello
Hello world.
```

We can also abbreviate each word in a CamelCase task name. For example, our `helloWorld` task name can be abbreviated to `hW`:

```
$ gradle -q hW
Hello world.
```

This feature saves us the time spent in typing the complete task name and can speed up the execution of our tasks.

Executing multiple tasks

With just a simple build script, we already discussed that we have a couple of default tasks besides our own task that we can execute. To execute multiple tasks, we only have to add each task name to the command line. Let's execute our `helloWorld` custom task and built-in `tasks` task:

```
$ gradle helloWorld tasks
:helloWorld
Hello world.
:tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------
Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]
Help tasks
----------
components - Displays the components produced by root project 'hello-
```

```
world'. [incubating]
 dependencies - Displays all dependencies declared in root project 'hello-
world'.
    dependencyInsight - Displays the insight into a specific  dependency in
root project 'hello-world'.
    help - Displays a help message.
    model - Displays the configuration model of root project 'hello-world'.
[incubating]
    projects - Displays the sub-projects of root project 'hello-world'.
    properties - Displays the properties of root project 'hello-world'.
    tasks - Displays the tasks runnable from root project 'hello-world'.
    Other tasks
    -----------
    helloWorld
    To see all tasks and more detail, run gradle tasks --all
    To see more detail about a task, run gradle help --task <task>
    BUILD SUCCESSFUL
    Total time: 2.028 secs
    This build could be faster, please consider using the Gradle Daemon:
https://docs.gradle.org/2.12/userguide/gradle_daemon.html
```

We see the output of both tasks. First, `helloWorld` is executed, followed by `tasks`. In the output, we see the task names prepended with a colon (`:`) and the output is in the next lines.

Gradle executes the tasks in the same order as they are defined in the command line. Gradle will only execute a task once during the build. So even if we define the same task multiple times, it will only be executed once. This rule also applies when tasks have dependencies on other tasks. Gradle will optimize the task execution for us and we don't have to worry about that.

Command-line options

The `gradle` command is used to execute a build. This command accepts several command-line options. We know the `--quiet` (or `-q`) option to reduce the output of a build. If we use the `--help` (or `-h` or `-?`) option, we see the complete list of options, as follows:

```
$ gradle --help
USAGE: gradle [option...] [task...]
-?, -h, --help          Shows this help message.
-a, --no-rebuild        Do not rebuild project dependencies.
-b, --build-file        Specifies the build file.
-c, --settings-file     Specifies the settings file.
--configure-on-demand   Only relevant projects are configured in this build
```

```
run. This means faster build for large multi-project builds. [incubating]
    --console               Specifies which type of console output to
generate. Values are 'plain', 'auto' (default) or 'rich'.
    --continue              Continues task execution after a task failure.
    -D, --system-prop       Set system property of the JVM (e.g. -
Dmyprop=myvalue).
    -d, --debug             Log in debug mode (includes normal stacktrace).
    --daemon                Uses the Gradle daemon to run the build. Starts
the daemon if not running.
    --foreground            Starts the Gradle daemon in the foreground.
[incubating]
    -g, --gradle-user-home  Specifies the gradle user home directory.
    --gui                   Launches the Gradle GUI.
    -I, --init-script       Specifies an initialization script.
    -i, --info              Set log level to info.
    -m, --dry-run           Runs the builds with all task actions disabled.
    --max-workers           Configure the number of concurrent workers
Gradle is allowed to use. [incubating]
    --no-color              Do not use color in the console output.
[deprecated - use --console=plain instead]
    --no-daemon             Do not use the Gradle daemon to run the build.
    --offline               The build should operate without accessing
network resources.
    -P, --project-prop      Set project property for the build script (e.g.
-Pmyprop=myvalue).
    -p, --project-dir       Specifies the start directory for Gradle.
Defaults to current directory.
    --parallel              Build projects in parallel. Gradle will attempt
to determine the optimal number of executor threads to use. [incubating]
    --parallel-threads      Build projects in parallel, using the specified
number of executor threads. [deprecated - Please use --parallel, optionally
in conjunction with --max-workers.] [incubating]
    --profile               Profiles build execution time and generates a
report in the <build_dir>/reports/profile directory.
    --project-cache-dir     Specifies the project-specific cache directory.
Defaults to .gradle in the root project directory.
    -q, --quiet             Log errors only.
    --recompile-scripts     Force build script recompiling.
    --refresh-dependencies  Refresh the state of dependencies.
    --rerun-tasks           Ignore previously cached task results.
    -S, --full-stacktrace   Print out the full (very verbose) stacktrace
for all exceptions.
    -s, --stacktrace        Print out the stacktrace for all exceptions.
    --stop                  Stops the Gradle daemon if it is running.
    -t, --continuous        Enables continuous build. Gradle does not exit
and will re-execute tasks when task file inputs change. [incubating]
    -u, --no-search-upward  Don't search in parent folders for a
settings.gradle file.
```

```
    -v, --version           Print version info.
    -x, --exclude-task      Specify a task to be excluded from execution.
```

Logging options

Let's look at some of the options in more detail. The `--quiet` (or `-q`), `--debug` (or `-d`), `--info` (or `-i`), `--stacktrace` (or`-s`), and `--full-stacktrace` (or`-S`) options control how much output we see when we execute tasks. To get the most detailed output, we use the `--debug` (or `-d`) option. This option provides a lot of output with information about the steps and classes used to run the build. The output is very verbose, therefore, we will not use it much.

To get a better insight on the steps that are executed for our task, we can use the `--info` (or `-i`) option. The output is not as verbose as with `--debug`, but it can provide a better understanding of the build steps:

```
 $ gradle --info helloworld
 Starting Build
 Settings evaluated using settings file '/master/settings.gradle'.
    Projects loaded. Root project using build file
'/Users/mrhaki/Projects/gradle-effective-implementation-guide-2/gradle-
impl-guide-2/src/docs/asciidoc/Chapter1/Code_Files/hello-
world/build.gradle'.
    Included projects: [root project 'hello-world']
    Evaluating root project 'hello-world' using build file
'/Users/mrhaki/Projects/gradle-effective-implementation-guide-2/gradle-
impl-guide-2/src/docs/asciidoc/Chapter1/Code_Files/hello-
world/build.gradle'.
    All projects evaluated.
    Selected primary task 'helloWorld' from project :
    Tasks to be executed: [task ':helloWorld']
    :helloWorld (Thread[main,5,main]) started.
    :helloWorld
    Executing task ':helloWorld' (up-to-date check took 0.001 secs) due to:
      Task has not declared any outputs.
    Hello world.
    :helloWorld (Thread[main,5,main]) completed. Took 0.021 secs.
    BUILD SUCCESSFUL
    Total time: 1.325 secs
    This build could be faster, please consider using the Gradle Daemon:
https://docs.gradle.org/2.12/userguide/gradle_daemon.html
```

If our build throws exceptions, we can see the stack trace information with the `--stacktrace` (or `-s`) and `--full-stacktrace` (or `-S`) options. The latter option will output the most information and is the most verbose. The `--stacktrace` and `--full-stacktrace` options can be combined with the other logging options.

Changing the build file and directory

We created our build file with the `build.gradle` name. This is the default name for a build file. Gradle will look for a file with this name in the current directory to execute the build. However, we can change this with the `--build-file` (or `-b`) and `--project-dir` (or `-p`) command-line options.

Let's run the `gradle` command from the parent directory of our current directory:

```
$ cd ..
$ gradle --project-dir hello-world -q helloWorld
Hello world.
```

We can also rename our `build.gradle` to, for example, `hello.build` and still execute our build:

```
$ mv build.gradle hello.build
$ gradle --build-file hello.build -q helloWorld
Hello world.
```

Running tasks without execution

With the `--dry-run` (or `-m`) option, we can run all tasks without really executing them. When we use the `dry-run` option, we can see the tasks that are executed, so we get an insight on the tasks that are involved in a certain build scenario. We don't even have to worry whether the tasks are actually executed. Gradle builds up a **Directed Acyclic Graph** (**DAG**) with all tasks before any task is executed. The DAG is build so that the tasks will be executed in order of dependencies, and a task is only executed once:

```
$ gradle --dry-run helloWorld
:helloWorld SKIPPED
BUILD SUCCESSFUL
Total time: 1.307 secs
This build could be faster, please consider using the Gradle Daemon:
https://docs.gradle.org/2.12/userguide/gradle_daemon.html
```

Gradle daemon

We already discussed that Gradle executes in a JVM, and each time we invoke the `gradle` command, a new JVM is started, the Gradle classes and libraries are loaded, and the build is executed. We can reduce the build execution time if we don't have to load JVM and Gradle classes and libraries each time we execute a build. The `--daemon` command-line option starts a new Java process that will have all Gradle classes and libraries already loaded and then execute the build. Next time when we run Gradle with the `--daemon` option, only the build is executed as the JVM with the required Gradle classes and libraries is already running.

The first time we execute Gradle with the `--daemon` option, the execution speed will not have improved as the Java background process was not started yet. However, the next time, we can see a major improvement:

```
$ gradle --daemon helloWorld
Starting a new Gradle Daemon for this build (subsequent builds will be
faster).
:helloWorld
Hello world.
BUILD SUCCESSFUL
Total time: 2.136 secs
$ gradle helloWorld
:helloWorld
 Hello world.
BUILD SUCCESSFUL
Total time: 0.594 secs
```

Even though the daemon process is started, we can still run Gradle tasks without using the daemon. We use the `--no-daemon` command-line option to run a Gradle build, and then the daemon is not used:

```
$ gradle --no-daemon helloWorld
:helloWorld
Hello world.
BUILD SUCCESSFUL
Total time: 1.325 secs
```

To stop the daemon process, we use the `--stop` command-line option:

```
$ gradle --stop
Stopping daemon(s).
Gradle daemon stopped.
```

This will stop the Java background process completely.

To always use the `--daemon` command-line option, but we don't want to type it every time we run the `gradle` command, we can create an alias if our operating system supports aliases. For example, on a Unix-based system, we can create an alias and then use the alias to run the Gradle build:

```
$ alias gradled='gradle --daemon'
$ gradled helloWorld
:helloWorld
Hello world.
BUILD SUCCESSFUL
Total time: 0.572 secs
```

Instead of using the `--daemon` command-line option, we can use the `org.gradle.daemon` Java system property to enable the daemon. We can add this property to the `GRADLE_OPTS` environment variable so that it is always used when we run a Gradle build:

```
$ export GRADLE_OPTS="-Dorg.gradle.daemon=true"
$ gradle helloWorld
:helloWorld
Hello world.
BUILD SUCCESSFUL
Total time: 0.575 secs
```

Finally, we can add a `gradle.properties` file to the root of our project directory. In the file, we can define a `org.gradle.daemon` property and assign the `true` value to enable the Gradle daemon for all builds that are executed from this directory.

Let's create a `gradle.properties` file with the following contents:

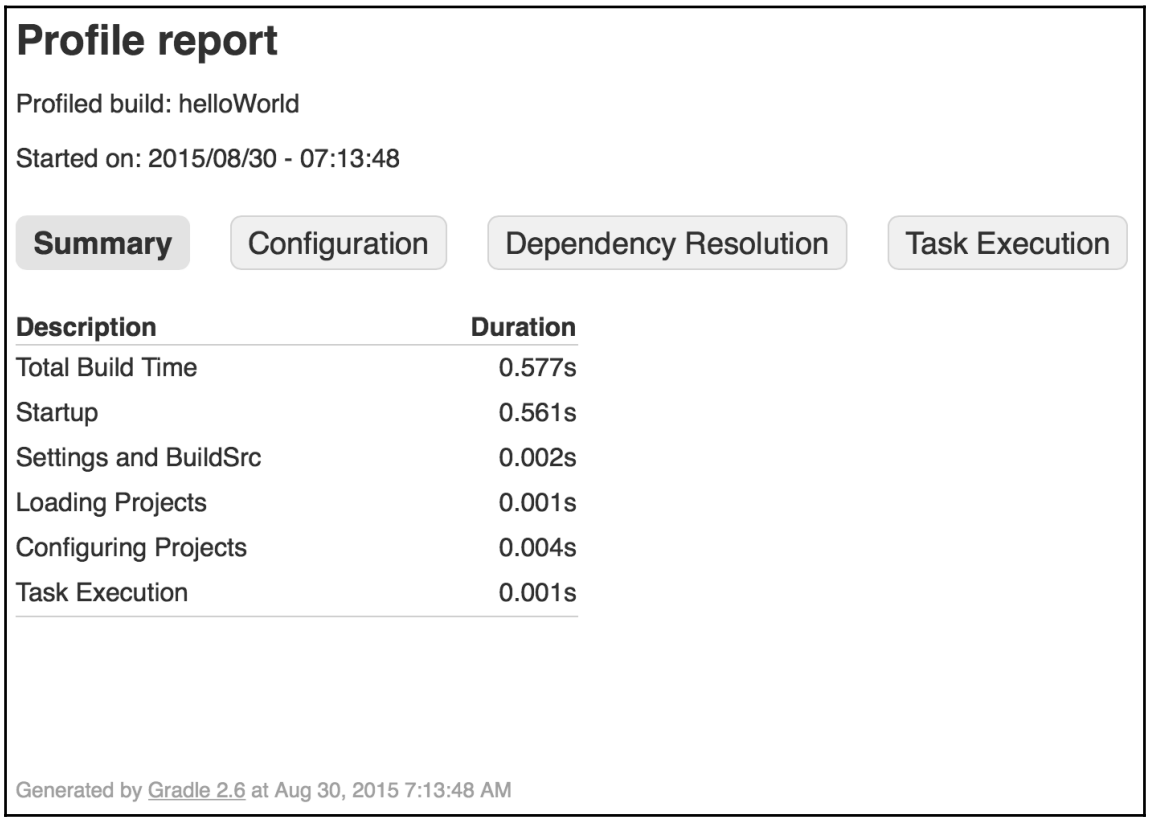
```
org.gradle.daemon=true
```

We can run our example task, `helloWorld`, and the build will use the Gradle daemon:

```
$ gradle helloWorld
:helloWorld
Hello world.
BUILD SUCCESSFUL
Total time: 0.58 secs
```

Profiling

Gradle also provides the `--profile` command-line option. This option records the time that certain tasks take to complete. The data is saved in an HTML file in the `build/reports/profile` directory. We can open this file in a web browser and check the time taken for several phases in the build process. The following image shows the HTML contents of the profile report:



HTML page with profiling information

Offline usage

If we don't have access to a network at some location, we might get errors from our Gradle build, when a task needs to download something from the Internet, for example. We can use the `--offline` command-line option to instruct Gradle to not access any network during the build. This way we can still execute the build if all necessary files are already available offline and we don't get an error.

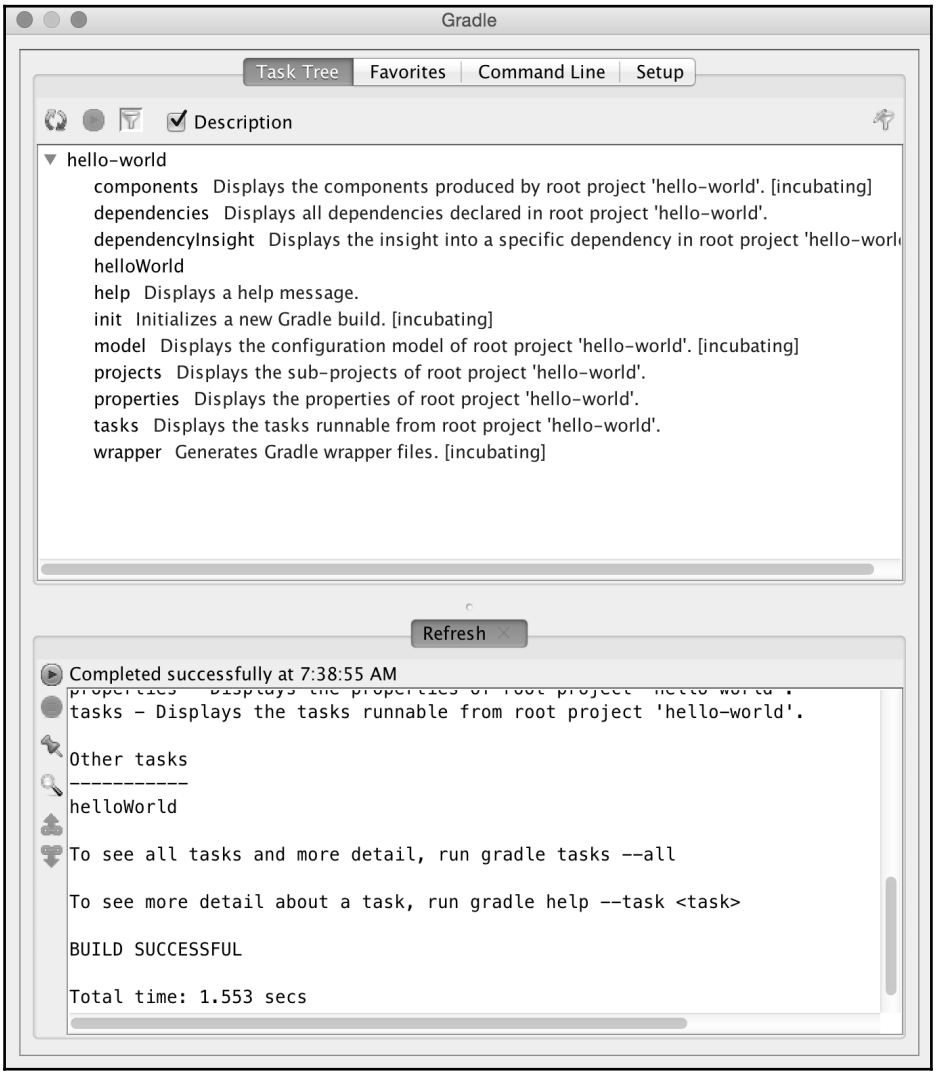
Understanding the Gradle graphical user interface

Finally, we take a look at the `--gui` command-line option. With this option, we start a graphical shell for our Gradle builds. Until now, we used the command line to start a task. With the **Gradle GUI**, we have a graphical overview of the tasks in a project and we can execute them by simply clicking on the mouse.

To start the GUI, we invoke the following command:

```
$ gradle --gui
```

A window is opened with a graphical overview of our task tree. We only have one task that one is shown in the task tree, as we can seen in the following screenshot:



Overview of tasks in the Gradle GUI

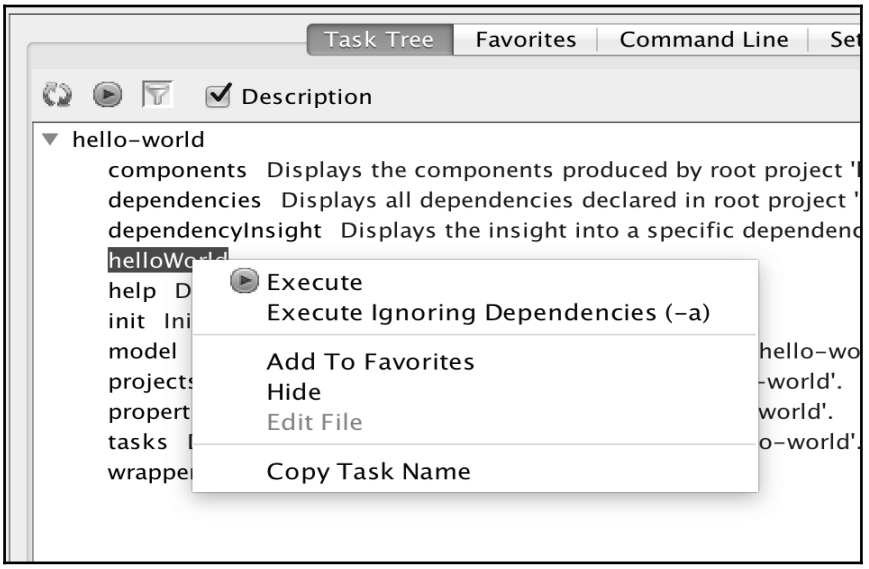
The output of running a task is shown at the bottom of the window. When we start the GUI for the first time, the `tasks` task is executed and we see the output in the window.

Task tree

The Task Tree tab shows projects and tasks found in our build project. We can execute a task by double-clicking on the task name.

By default, all the tasks are shown, but we can apply a filter to show or hide certain projects and tasks. The **Edit filter** button opens a new dialog window where we can define the tasks and properties that are a part of the filter. The **Toggle filter** button makes the filter active or inactive.

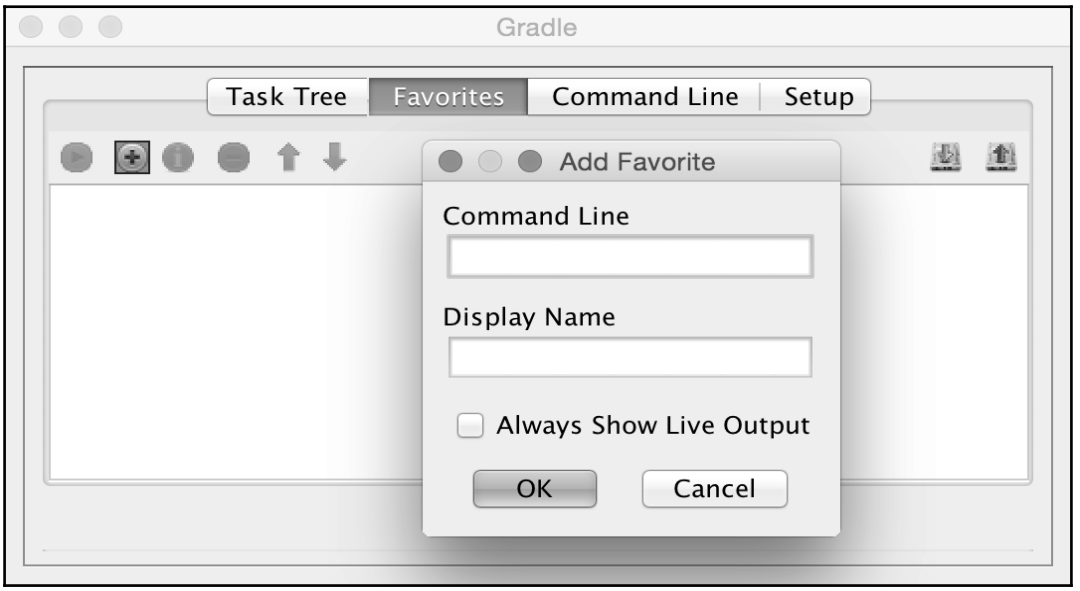
We can also right-click on the project and task names. This opens a context menu where we can choose to execute the task, add it to the favorites, hide it (adds it to the filter), or edit the build file. If we have associated the `.gradle` extension to a text editor in our operating system, then the editor is opened with the content of the build script. These options can be seen in the following screenshot:



Available actions for a task in the Gradle GUI

Favorites

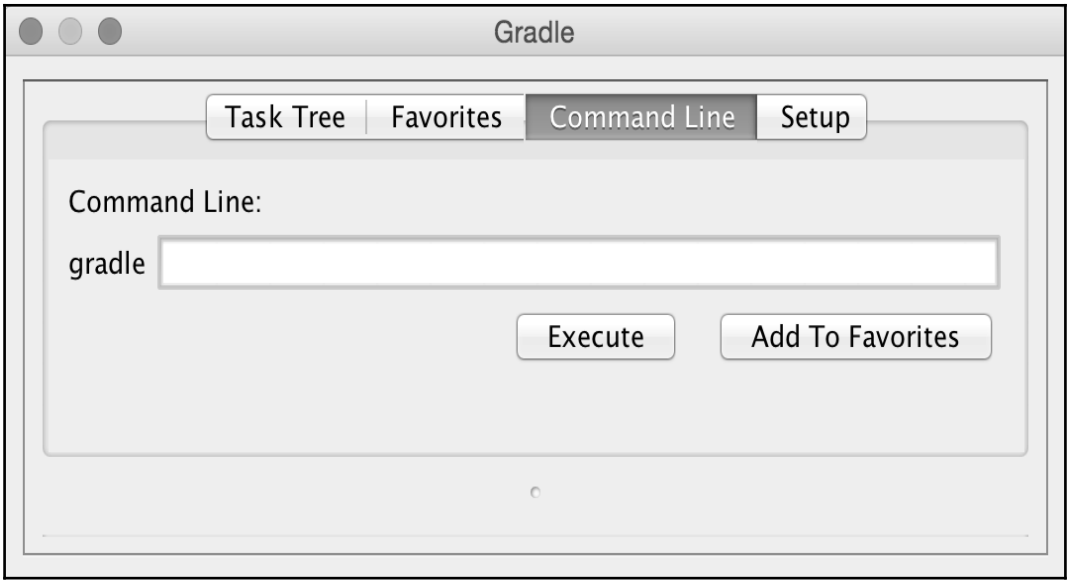
The Favorites tab stores tasks we want to execute regularly. We can add a task by right-clicking on the task in the **Task Tree** tab and selecting the **Add To Favorites** menu option, or if we have opened the **Favorites** tab, we can select the **Add** button and manually enter the project and task name that we want to add to our favorites list. We can see the **Add Favorite** dialog window in the following screenshot:



Add favorites in the Gradle GUI

Command line

On the Command Line tab, we can enter any Gradle command that we normally would enter on the command prompt. The command can be added to **Favorites** as well. We see the **Command Line** tab contents in the following image:



Execute task with command-line optins

Setup

The last tab is the Setup tab. Here, we can change the project directory, which is set to the current directory by default.

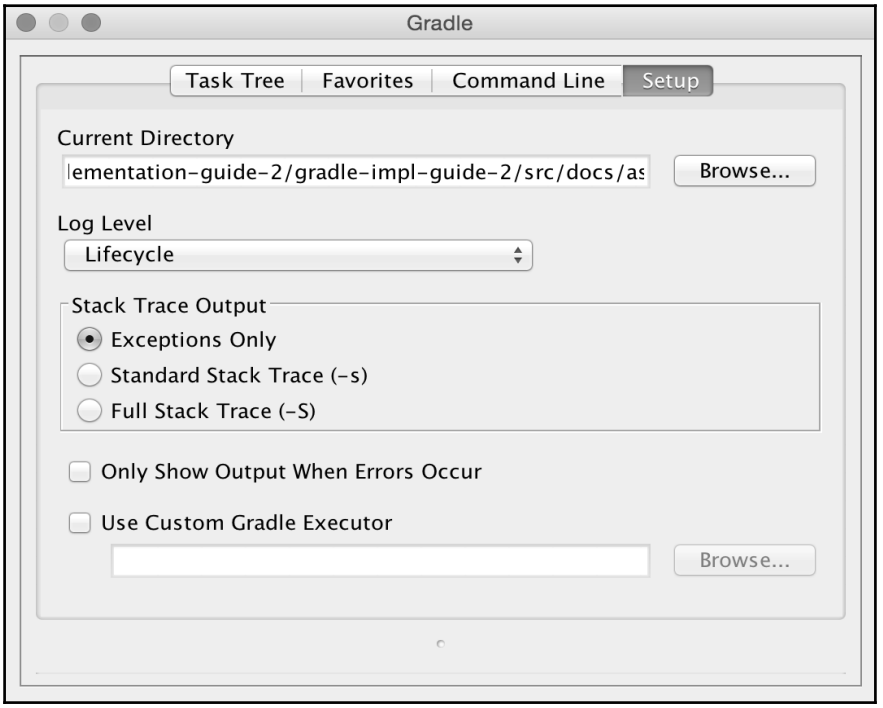
We discussed about the different logging levels as command-line options previously in this chapter. With the GUI, we can select the logging level from the **Log Level** select box with the different log levels. We can choose **debug**, **info**, **Lifecycle**, and **error** as log levels. The error log level only shows errors and is the least verbose, while debug is the most verbose log level. The lifecycle log level is the default log level.

Here, we also can set how detailed the exeption stack trace information should be. In the **Stack Trace Output** section, we can choose from the following three options:

- **Exceptions Only**: This is for only showing the exceptions when they occur, which is the default value
- **Standard Stack Trace (-s)**: This is for showing more stack trace information for the exceptions
- **-S)**: This is for the most verbose stack trace information for exceptions

If we enable the **Only Show Output When Error Occurs** option, then we only get output from the build process if the build fails. Otherwise, we don't get any output.

Finally, we can define a different way to start Gradle for the build with the **Use Custom Gradle Executor** option. For example, we can define a different batch or script file with extra setup information to run the build process. The following screenshot shows the **Setup** tab page and all the options that we can set:



Setup Gradle options in the Gradle GUI

Summary

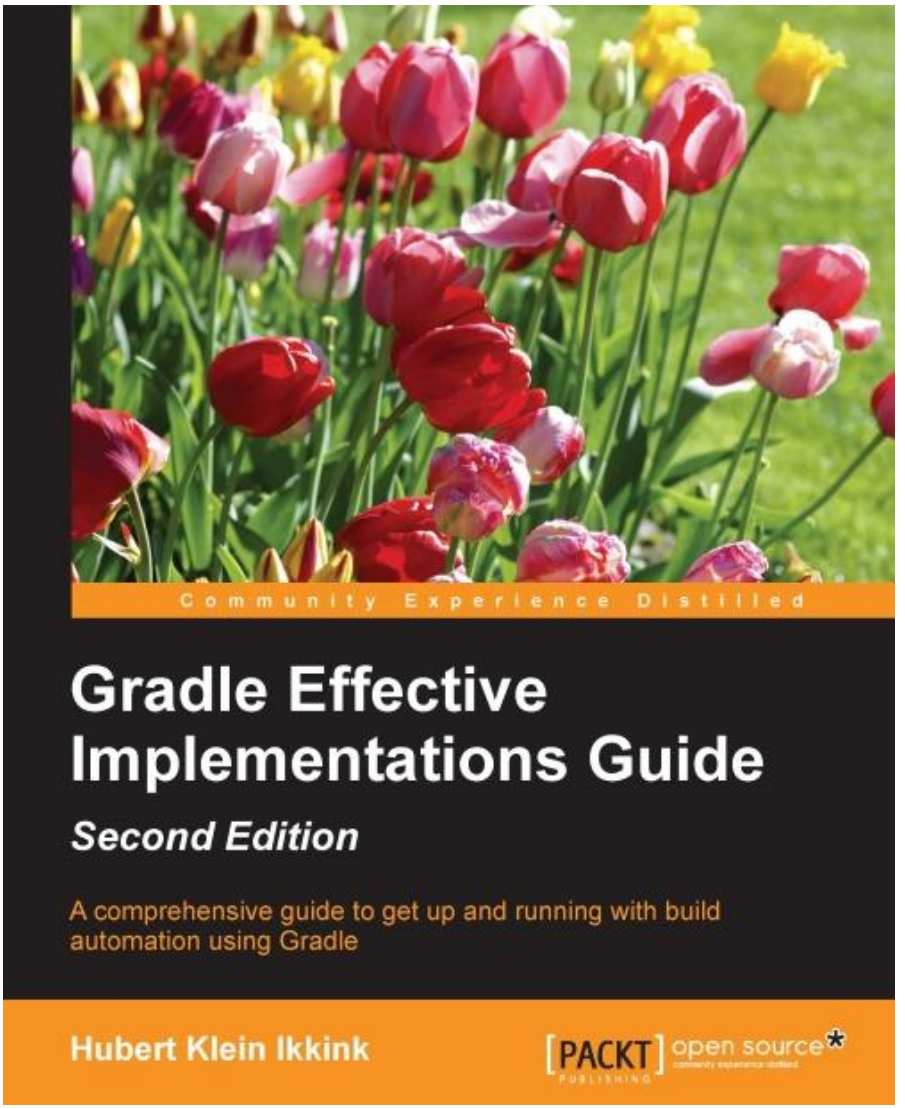
So, now we have discussed how to install Gradle on our computers. We have written our first Gradle build script with a simple task.

We have also seen how we use the built-in tasks of Gradle to get more information about a project. We discussed how to use the command-line options to help us run the build scripts. We have looked at the Gradle GUI and how we can use it to run Gradle build scripts.

In the next chapter, we will take a further look at tasks. We will discuss how to add actions to a task. We write more complex tasks, where the tasks will depend on other tasks. We will also discuss how Gradle builds up a task graph internally and how to use this in our projects.

Purchase the full book

Get 50% discount on the eBook format using coupon code **GRADLE50**



Packt> Buy Now