



C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# Getting Started with SBT for Scala

Equip yourself with a high-productivity work environment using SBT, a build tool for Scala

**Shiti Saxena**

**[PACKT]** open source\*  
PUBLISHING community experience distilled

# Getting Started with SBT for Scala

Equip yourself with a high-productivity work environment using SBT, a build tool for Scala

**Shiti Saxena**



BIRMINGHAM - MUMBAI

# Getting Started with SBT for Scala

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1040913

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-78328-267-8

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Aniket Sawant ([aniket\\_sawant\\_photography@hotmail.com](mailto:aniket_sawant_photography@hotmail.com))

# Credits

**Author**

Shiti Saxena

**Project Coordinator**

Joel Goveya

**Reviewers**

Aaruna Godthi

Rohit Rai

Prashant Sharma

**Proofreaders**

Maria Gould

Paul Hindle

**Acquisition Editor**

Edward Gordon

**Indexer**

Monica Ajmera Mehta

**Commissioning Editor**

Mohammed Fahad

**Graphics**

Abhinash Sahu

**Technical Editor**

Jalasha D'costa

**Production Coordinator**

Conidon Miranda

**Cover Work**

Conidon Miranda

# About the Author

**Shiti Saxena** is a software engineer with around three years of work experience. She is currently working with Imaginea (a business unit of Pramati). She has also previously worked with Tata Consultancy Services Ltd. and Genpact.

A true polyglot, she has experience with various languages including Scala, JavaScript, Java, Python, Perl, and C. She also likes to work with Play Scala and AngularJS.

She blogs at <http://eraoferrors.blogspot.in> and maintains open source projects at GitHub.

In her spare time, she loves to travel, watch movies, and likes to spend time playing her piano.

# Acknowledgments

I am indebted to my mother, Nithi, and my sisters, Shaila, Anshu, and Aastha, for their constant support.

A special thanks to my cousin, Rohit, for guiding, mentoring, understanding, and pampering me. He made me believe that I could write this book, and supported me when I was finding it difficult to finish. He has always been there for me, irrespective of the time or busy schedule.

I would like to thank Jay and Vijay Pullur for starting Pramati, and everyone at Pramati for making it a great place to work.

I would like to thank Kalyan and Apurba for their support and guidance. I wouldn't have learned many a thing if it wasn't for them.

I would like to thank Mark Harrah and Typesafe, without whose efforts, this technology, and eventually this book, wouldn't have been possible.

I thank my friends for being there when I need them.

I thank the editors at Packt, Meeta, Joel, and Mohammed Fahad, for being good to me and understanding the difficulties I faced while writing the book. They showed great patience when I was unable to meet the deadlines. I am also thankful to the reviewers, Rohit, Prashant, and Aaruna, for their valuable feedback that helped improve the book.

I am grateful to everyone who has in some way or the other helped me reach this stage in my life. Thanks!

# About the Reviewers

Coming from a Statistics background and with a Master's degree in Information Technology, **Aaruna Godthi** likes taking up challenges. He is a polyglot in the field of programming.

He currently works at Pramati Technologies. He has worked a bit on Hadoop and Sentiment analysis (entity extraction and association of sentiment). Currently, he is exploring various JavaScript frameworks including Dojo, Ember, and AngularJS. Other than this, he likes exploring offbeat places.

**Rohit Rai** is a software professional and founder of Tuplejump. With over 10 years of experience in the industry, he has been exposed to a wide variety of languages and platforms. He has been working with Scala for over three years, and he decided to take a whole-hearted dive when Scala became the primary development language for the Tuplejump Data Engineering Platform.

He is an active contributor to the open source community and a developer and maintainer of various projects around Scala and SBT including the play-yoman integration, play socket.io plugin, Calliope – the cassandra+spark bridge, and so on.

He has also authored the book *Socket.IO Real-time Web Application Development*, Packt Publishing.

---

First of all, I would like to thank the author and publishers for choosing me to review the book. I would to thank my family, friends, and colleagues for understanding that I have been stealing out time from various activities to meet the book deadlines.

---

**Prashant Sharma** works for Pramati Technologies, which is an umbrella company under which he is part of a development team at Imaginea. He is an open source contributor at Spark, and has been working on projects based on Scala as both a hobby and as his profession. He has also worked on distributed systems, such as ejabberd in erlang, and also other Java-based RESTful and soap-based web services to build multitenant systems. Academically, he is interested in exploring both functional programming and distributed computing.

You can find Prashant's work at GitHub (<https://github.com/ScrapCodes>) and follow him on Twitter (@ScrapCodes).

**More on Imaginea** (<http://imaginea.com/>)

Imaginea has been at the forefront of technological evolution by developing sophisticated software solutions for both its own customers and the open source community.

GitHub: <https://github.com/imaginea>



# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

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](http://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](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://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.



<http://PacktLib.PacktPub.com>

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

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Hello World with SBT</b>	<b>5</b>
<b>Why SBT?</b>	<b>6</b>
<b>Installing SBT</b>	<b>7</b>
Installing from a package	8
Installing on Mac	8
Installing SBT manually	8
<b>Creating a new project</b>	<b>9</b>
<b>Compiling, testing, and running your project</b>	<b>11</b>
<b>Going interactive with the SBT shell</b>	<b>12</b>
Triggering SBT commands on saves	13
<b>Summary</b>	<b>14</b>
<b>Chapter 2: .sbt Build Definitions</b>	<b>15</b>
<b>The theory of .sbt</b>	<b>15</b>
<b>The .sbt syntax</b>	<b>16</b>
<b>Understanding keys</b>	<b>18</b>
Keys	20
Setting keys	20
Task keys	21
Input keys	22
<b>Working with scopes</b>	<b>22</b>
<b>Summary</b>	<b>24</b>
<b>Chapter 3: Dependency Management</b>	<b>25</b>
<b>Quick introduction to Maven or Ivy dependency management</b>	<b>26</b>
How Ivy works	26
Resolve	27
Retrieve	28
Publish	28

<b>Dependency management in SBT</b>	<b>28</b>
Automatic dependency management	28
Declaring dependencies in the build definition	29
Dependencies using Maven files	33
Dependencies using Ivy files or Ivy XML	33
Adding JAR files manually	34
<b>Resolvers</b>	<b>35</b>
<b>Summary</b>	<b>38</b>
<b>Chapter 4: Full Build Definitions</b>	<b>39</b>
Build definition project	40
.sbt and .scala	41
Working with full build definitions	42
Multiproject builds	43
Summary	47
<b>Chapter 5: Compile, Test, and Run</b>	<b>49</b>
Commands	49
Logging	53
Forking the JVM	56
Parallel execution	58
SBT scripts and REPL	60
Classpath, sources, and resources	62
Test	64
Summary	68
<b>Index</b>	<b>69</b>

---

# Preface

Simple Build Tool, abbreviated to SBT, is a build tool for Scala and Java projects. SBT simplifies the process of developing, building, and maintaining a project. This book will show you how to build a Scala project using SBT Version 0.12.3, and the different ways in which SBT can be set up.

## What this book covers

*Chapter 1, Hello World with SBT*, introduces SBT, and how to get it up and running with an example.

*Chapter 2, .sbt Build Definitions*, explains the components of a simple `.sbt` build file.

*Chapter 3, Dependency Management*, explains how project dependencies can be configured in SBT.

*Chapter 4, Full Build Definitions*, introduces the `.scala` build definition and the scenarios in which it can be extremely useful.

*Chapter 5, Compile, Test, and Run*, explains SBT commands and some configurations which can come in handy when developing an application.

## What you need for this book

All that you need to use this book is a PC with Linux or Windows OS or a Mac, a text editor, and Java 1.6 or later. There are SBT plugins available for various IDEs, but setting it up is completely optional.

## Who this book is for

This book is aimed at developers who want to use SBT to build Scala projects. The readers are expected to know programming in Scala.

## Conventions

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

Code words in text are shown as follows: "Create an `sbt.bat` file."

A block of code is set as follows:

```
class IntroSpec extends Specification{
  "The phrase 'Never odd or even'" should{
    "be a palindrome" in {
      Introduction.isPalindrome("Never odd or even") must beTrue
    }
  }
}
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
class IntroSpec extends Specification{
  "The phrase 'Never odd or even'" should{
    "be a palindrome" in {
      Introduction.isPalindrome("Never odd or even") must beTrue
    }
  }
}
```

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

```
>compile
[success] Total time: 0 s, completed
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."

[  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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via 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 on [www.packtpub.com/authors](http://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 all Packt books you have purchased 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.

## 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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## Piracy

Piracy of copyright 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](mailto: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

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1

## Hello World with SBT

In the stone age of programming, a project team comprised several programmers and a person whose job was to build and deploy the software developed by others. To compile the project, this person would have to compile each file individually. If compilation for all of the files was successful, they went on to deploy the project. There were no means to check for errors while integration. One had to come up with a method to have logs for each process. Deploying was equally complex as it required modules to be placed manually in the right location.

Thankfully, this is not the case anymore.

Building projects involves compiling your code and assembling the binaries with the required dependencies and resources. Build tools are programs that assist in managing and maintaining programs. Initial build tools had support for compiling and linking modules in the required order. Gradually, other features were included.

Some key advantages of automating the build process are as follows:

- Reduces time taken for compiling and linking
- Reduces scope of errors by humans
- Consistency
- Cost efficient

**Make** was one of the earliest build tools. It is also the most widespread tool. Make was originally created by *Stuart Feldman* in 1977 at Bell Labs to ease the process of building and deploying. Prior to Make, the program source was accompanied by OS dependent "make" and "install" shell scripts in the Unix build system. With Make, it became possible to combine the commands for different targets into a single file and abstract out dependency tracking and archive handling. Hence, Make was an important milestone on the path to modern build environments.



In the late 1990s, James Duncan Davidson, a software architect at Sun Microsystems, conceived the idea of **Another Neat Tool (ANT)** while turning Tomcat into an open source product. ANT is similar to Make, but it is implemented using Java. ANT uses an XML file to describe the build process and its dependencies. Today, ANT is the most widely used build tool for Java development projects.

In 2002, Jason van Zyl felt the need for a standardized project structure, and it led to the creation of Maven. Maven projects are configured using a Project Object Model in an XML file by describing the software project being built, its dependencies on other external modules and components, the build order, directories, and the required plugins.

In September 2004, Jayasoft came up with a transitive relation dependency manager called **Ivy**. It provides a way to resolve project dependencies in ANT projects.

In 2009, Gradleware built another build tool which used a Groovy-based DSL. Gradle utilizes a directed acyclic graph to determine the order in which tasks can be run.

Leiningen started off as a project with the aim of simplifying Apache Maven for Clojure projects by using Clojure to describe the build requirements. It uses a Clojure-based DSL for writing the build scripts.

Inspired by its predecessors, **Simple Build Tool (SBT)** was created by Typesafe Engineer, Mark Harrah. It is written in Scala and can be used to compile both Java and Scala code. SBT is not just a build tool, but also provides the basic framework for a development environment. Its integrated Scala console, the continuous compile mechanism, and the fast compile server, make it a necessary tool in any Scala developer's toolkit.

## Why SBT?

Apart from being the build tool of choice for most Scala projects, SBT provides some very interesting features besides the ones provided by a conventional build tool.

To begin with, SBT can be configured using a simple Scala-based DSL and extended to use a full-fledged configuration with Scala as the project demands.

SBT uses incremental recompilation to reduce the compilation time of the Scala code. An incremental compiler will only compile what needs to be recompiled after changes. It also keeps the compiler's JVM up and running with the project's compiled code to prevent the load time.

SBT commands can be run in a triggered execution mode. That is, if opted by the user, specific tasks will be run whenever the user makes a change in any of the source code. This was originally meant for continuous compilation, but has now been extended for various other tasks. It is also possible to run multiple commands in this mode.

SBT provides support for testing with ScalaCheck, specs, and ScalaTest. Additionally, JUnit tests can also be run by using a plugin (**junit-interface**). SBT allows the user to run tests selectively or run all the tests. It also allows running only those tests that failed earlier or were excluded earlier or have a dependency on the recompiled code that is run.

SBT can start the Scala REPL with the project loaded in the classpath. One could call the methods defined in the project code within the REPL.

SBT can be configured to have subprojects. This provides better means for achieving modularity. Dependent modules can be clubbed using a single build and desired modules can be made completely independent of others.

SBT allows inclusion of external projects using their path or URL. This means it's also possible to include Git repositories as a dependency for the project. This makes the developer's life a lot simpler as he/she can easily have a project that depends on another project.

SBT runs tasks in parallel. Task parallelism refers to running one or more independent tasks concurrently. Task parallelism ensures efficient and scaleable use of resource systems. This is really useful for running tests in parallel.

SBT supports the inclusion of libraries as unmanaged or managed dependencies; that is, libraries can simply be included by adding the corresponding JAR in the `lib` directory or by specifying the dependencies in the build definition. SBT uses Apache Ivy to implement managed dependencies.

There are plugins to support SBT in Sublime Text, IntelliJ Idea, Eclipse, and Netbeans IDE. This makes it easier and simpler to adopt SBT.

## Installing SBT

SBT can either be installed from a package or manually.

## Installing from a package

SBT can be installed directly using MSI for Windows, RPM, or DEB packages downloaded from the SBT website.

You could also use the zip/tgz package, but after extracting from the zip/tgz package, you need to add the `sbt/bin` folder to `PATH`.

## Installing on Mac

SBT can be installed on Mac using either Macports, as follows:

```
$ port install sbt
```

Or, by using Homebrew, as follows:

```
$ brew install sbt
```

## Installing SBT manually

Download `sbt-launch.jar` (available at <http://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/sbt-launch/0.12.3/sbt-launch.jar>) and write a script. The script and the JAR file should be in the same folder.

On Windows, create an `sbt.bat` file and add it to `PATH`. The contents of the file should be as follows:

```
$ set SCRIPT_DIR=%~dp0
$ java -Xmx512M -jar "%SCRIPT_DIR%sbt-launch.jar" %*
```

On Unix, save the JAR file and script in your `~/bin` directory. Name the script `sbt`. The contents of the script should be as follows:

```
java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled
-XX:MaxPermSize=384M -jar `dirname $0`/sbt-launch.jar "$@"
```

Make the script executable:

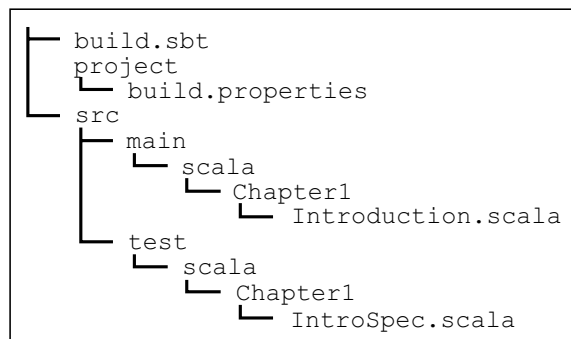
```
$ chmod u+x ~/bin/sbt
```

To check if the installation was successful, use the `about` or the `version` command:

```
$ sbt --version
sbt launcher version 0.12.1
```

## Creating a new project

The following diagram shows the basic directory structure of a Scala SBT:



SBT recommends, and uses, the standard Maven project directory structure by default.

The files generated by SBT are written in the target directory within the project folder by default.

Let's write a program that checks for palindromes. Create `Introduction.scala`:

```
package Chapter1

object Introduction extends App {

  def isPalindrome(word: String) = {
    val modifiedWord = word.toLowerCase.replaceAll("[^a-z0-9]", "")
    val reversed = modifiedWord.reverse
    modifiedWord == reversed
  }

  println("Is 'Herculaneum' a palindrome? " +
    isPalindrome("Herculaneum"))
}
```

The `IntroSpec.scala` file is as follows:

```
import Chapter1.Introduction
import org.specs2.mutable._
```

```
class IntroSpec extends Specification{
  "The phrase 'Never odd or even'" should{
    "be a palindrome" in {
      Introduction.isPalindrome("Never odd or even") must beTrue
    }
  }

  "The phrase 'Mr. Owl ate my metal worm'" should{
    "be a palindrome" in {
      Introduction.isPalindrome("Mr. Owl ate my metal worm") must
beTrue
    }
  }

  "The date '20:02 02/20 2002'" should{
    "be a palindrome" in{
      Introduction.isPalindrome("20:02 02/20 2002") must beTrue
    }
  }
}
```

The build file is used to specify the basic build settings and manage dependencies. We will see the build file in detail in the next chapter. It should be similar to the following:

```
name := "Introduction"

version := "1.0"

scalaVersion := "2.9.1"

resolvers += Seq("snapshots" at "http://oss.sonatype.org/content/
repositories/snapshots",
  "releases" at "http://oss.sonatype.org/content/repositories/
releases")

libraryDependencies += Seq( "org.specs2" %% "specs2" % "1.12.3" %
"test")
```



The blank lines between each item are required to separate them and are not there just for clarity in reading.

## Compiling, testing, and running your project

From the terminal/command prompt, go to the project directory:

```
$ cd introduction
```

To compile the project, use the following command:

```
$ sbt compile
```

The result will be as follows:

```
[info] Loading project definition from /home/introduction/project
[info] Set current project to Introduction (in build file:/home/
introduction/)
[success] Total time: 0 s, completed
```

To run the tests, use the following command:

```
$ sbt test
```

The result will be as follows:

```
[info] Loading project definition from /home/introduction/project
[info] Set current project to Introduction (in build file:/home/
introduction/)
[info] IntroSpec
...
[info] Passed: : Total 3, Failed 0, Errors 0, Passed 3, Skipped 0
[success] Total time: 5 s, completed
```

To run the project, use the following command:

```
$ sbt run
```

The result will be as follows:

```
[info] Loading project definition from /home/introduction/project
[info] Set current project to Introduction (in build file:/home/
introduction/)
[info] Running Chapter1.Introduction
Is 'Herculaneum' a palindrome? false
[success] Total time: 0 s, completed
```

It is also possible to run `sbt` in batch mode, that is, multiple commands separated by spaces. Execution will happen in the given order. Commands that take arguments must be enclosed in quotes.

The `clean` command is used to delete all the generated files in the target folder:

```
$ sbt clean compile test
```

The result will be as follows:

```
[info] Loading project definition from /home/introduction/project
...
[info] Passed: : Total 3, Failed 0, Errors 0, Passed 3, Skipped 0
[success] Total time: 7 s, completed
```

## Going interactive with the SBT shell

While you are still in your project folder, type `sbt` in the terminal/command prompt:

```
$ sbt
```

It will load the required JAR files, and finally, result in an SBT shell that looks like the following:

```
[info] Loading project definition from /home/introduction/project
[info] Set current project to Introduction (in build file:/home/
introduction/)
>
```

Once the shell has started, you can compile, test, or run the project with `compile`, `test`, and `run` commands respectively:

```
> compile
[success] Total time: 0 s, completed

> test
[info] IntroSpec
.....
[info] Passed: : Total 3, Failed 0, Errors 0, Passed 3, Skipped 0
[success] Total time: 2 s, completed
```

```
>run
[info] Running Chapter1.Introduction
Is 'Herculaneum' a palindrome? false
[success] Total time: 0 s, completed
```

Typing `console` at the command prompt will open Scala REPL with the project's classpath, where you can try out live Scala examples based on your project's code. For example, consider the following:

```
> console
[info] Starting scala interpreter...
...
scala>
```

Let's run our palindrome function here:

```
scala> Chapter1.Introduction.isPalindrome("hello")
res0: Boolean = false
```

You could try running other functions as an exercise.

## Triggering SBT commands on saves

Passing a `~` character before a command tells SBT to keep looking for changes and run the same command if there are any changes in any of the source files.

This is possible in both interactive and batch modes.

```
> ~run
[info] Running Chapter1.Introduction
Is 'Herculaneum' a palindrome? false
[success] Total time: 0 s, completed Apr 11, 2013 4:04:08 PM
1. Waiting for source changes... (press enter to interrupt)
```

After changes are saved, it resumes on its own.

```
[info] Compiling 1 Scala source to /home/introduction/target/scala-2.9.1/
classes...
[info] Running Chapter1.Introduction
Is 'HelloWorld' a palindrome? false
```



```
[success] Total time: 1 s, completed Apr 11, 2013 4:04:51 PM
2. Waiting for source changes... (press enter to interrupt)
[info] Compiling 1 Scala source to /home/introduction/target/scala-2.9.1/
classes...
[info] Running Chapter1.Introduction
Is 'Herculaneum' a palindrome? false
[success] Total time: 1 s, completed Apr 11, 2013 4:05:11 PM
3. Waiting for source changes... (press enter to interrupt)
```

## Summary

In this chapter, we saw how to build a simple Scala project in SBT and execute basic operations, such as compile, test, and run. In the following chapters, we will see how we can customize the build for our project based on various settings.

# 2

## .sbt Build Definitions

In the previous chapter, we saw how to set up a simple Scala project in SBT, but generally, one would be working on much more complex projects. In order to use SBT efficiently, it is important to understand how the build definition works. In this chapter, we will see how to write a build definition and the options that can be configured through it.

### The theory of .sbt

In SBT, project-specific properties, such as library dependencies, Scala version, and so on, which are required for a successful build, are declared in the build definition.

The build definition can be a `.sbt` file or a `.scala` file or even a combination of the two. The `.sbt` file should be located in the base directory and is generally named `build.sbt`. The `.scala` file should be located in the project subdirectory of the base directory.

The build definition has keys and the transformation to be applied on the associated values. The type of value of a key is predefined and cannot be modified. In the following line of code, `name` is the key, `:=` is the transformation, and `Introduction` is the value:

```
name := "Introduction"
```

The value type for the key `name` is `String`. We will discuss keys in detail in the following sections.

Each key-value pair is a build property. So, we could say that a build definition is a list of properties.

SBT examines the project and generates an immutable map describing the build. If a build definition is found, it uses those values for the keys. When SBT comes across a build definition, it sorts the list of custom properties. The sorting is done such that all the same key transforms are together and dependent keys come after the key they depend on. Once sorted, the transformations are applied on the default map one by one.

## The .sbt syntax

The build definition is essentially a list of Scala expressions that are split and passed to the compiler individually. Blank lines act as a delimiter in a .sbt file. Removing the blank lines after each expression in the build.sbt file and compiling it will result in the following:

```
[info] Loading project definition from /home/introduction/project
/home/introduction/build.sbt:2: error: eof expected but ';' found.
version := "1.0"
^

[error] Error parsing expression. Ensure that settings are separated by
blank lines.
```

This is because without a blank line, SBT is clueless about the beginning and end of an expression.

Each expression consists of a key (the left operand), an operator, and a value (the right operand). For example, in the following expression, `version` is the key, `:=` is the operator, and `1.0` is the value:

```
version := "1.0"
```

`build.sbt` imports `sbt.Keys._` implicitly. Hence, we can directly refer to `sbt.Keys.version` with `version`.

SBT is typesafe or typeaware. To understand this, change the preceding expression in `build.sbt` to the following and try to compile the project:

```
version:=2
```

You will get the following error:

```
[info] Loading project definition from /home/introduction/project
/home/introduction/build.sbt:3: error: type mismatch;
 found   : Int(2)
 required: String
version := 2
      ^

[error] Type error in expression
```

This is because each key expects a specific type of value. If the given value's type does not match with the predefined type, there will be an error while compiling the project.

Each operator is a method defined for the key. The previous expression is shorthand for this:

```
version.:=("1.0")
```

The different kinds of operators are listed next with examples. Add or replace the example snippet in your `build.sbt` file to see the changes.

- `:=`: This is used to add a setting or replace the value of a setting with a new one. We used this in the build file for `name`, `version`, and `scalaVersion`.
- `++=`: This works only for values with the type `Seq[T]`. It concatenates the specified sequence to the actual value. We used this in the `build.sbt` file for `resolvers` and `libraryDependencies`.
- `+=`: This can be used when a single value is to be appended to the default value. So, modifying the name expression will append `Introduction` to the default value (the value of `name` became `default-b65a81Introduction` in my case).

```
name += "Introduction"
```

- `~=`: This can be used when we intend to modify the previous value of a key:

```
name ~= {n => "project"+n}
```

This will add `project` before the default name. (The value of `name` became `projectdefault-b65a81` in my case.)

- `<=<`: This is another special operator. It can be used to define a key's value based on other key(s). The key assigned in this manner is said to have a dependency on another key. For example:

```
version <=<= name {n => n + "-1.0"}
```

This will result in the version being `Introduction-1.0`.

Here, the `version` key has a dependency on `name`. If you do not wish to modify the other key's value, the function is not required.

```
organization <<= name
```

This is also a valid definition where we assign the value of a name to the organization.

An example of using multiple keys is as follows:

```
version <=<= (name,organization) {(n,o) => o + "-" + n + "-1.0"}
```

Here, the organization is assigned as:

```
organization := "Packt".
```

This will result in the version being `Packt-Introduction-1.0`.

It is possible to have at max a tuple of 9 keys. In cases where a key whose value is not defined is used for dependency, SBT will give an error.

The following table summarizes the transformations:

Transformation operator	Purpose
<code>:=</code>	Set a new value or replace existing value
<code>++=</code>	Append a sequence of values to the existing value
<code>+=</code>	Append to the existing value
<code>~=</code>	Modify the previously set value
<code>&lt;=&amp;, &lt;&lt;</code>	Set value based on another key's value

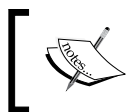
## Understanding keys

In SBT, keys are defined for different purposes, ranging from simple ones, such as log-related, to complex ones, such as dependency-management-related. Each key has a rank associated with it which is used to prioritize the information to be displayed. A key can be categorized into one of the following categories:

- **Setting keys:** This category contains keys whose values are computed on loading the project, and are stored for further use
- **Task keys:** This category contains keys whose values are recomputed each time it is executed
- **Input keys:** This category contains keys like the Task keys, but these keys have command-line arguments as input

Since a Task key is computed on each execution, a Setting key cannot depend on a Task key. Trying to do so will throw an error.

The value of a Setting key can be found by typing the key name in the interactive shell or the SBT key name in the command prompt. Typing `Task Key` or `Input Key` instead would result in execution of the associated task.



The CamelCase key names in the build definition change to hyphen separated ones in the prompt or interactive shell. So, `scalaVersion` used in the build definition is referred to as `scala-version`. This is common to all the keys.

Before going through the examples, let's modify the `Introduction.scala` file used in *Chapter 1, Hello World with SBT*.

```
package Chapter2

object Introduction extends App {

  def isPalindrome(word: String) = {
    val modifiedWord = word.toLowerCase.replaceAll("[^a-z0-9]", "")
    val reversed = modifiedWord.reverse
    modifiedWord == reversed
  }

  val givenString = args.mkString
  println("Is "+givenString+" a palindrome? " +
    isPalindrome(givenString))
}
```

And now the examples for the different types of keys are:

```
> name
[info] Introduction

> compile
[success] Total time: 0 s, completed

> run "madam im adam"
[info] Running Chapter2.Introduction madam im adam
Is madam im adam a palindrome? True
```

Here, `name` is the Setting key, `compile` is the Task key, and `run` is the Input key. You must be thinking, how can `run` be an Input key? It's got to be a Task key, since earlier we executed it without any arguments. Well, `run` takes a blank string as the default argument when no arguments are passed. Thus, earlier, when we simply executed `run` without any command line arguments, the execution was successful.

Even now, we could execute `run` without any arguments as follows:

```
> run
[info] Running Chapter2.Introduction
Is a palindrome? true
[success]
```

## Keys

Let's look at the commonly used keys and their purposes.

## Setting keys


The following is a list of setting keys mentioned:

- `name`: This represents the project name. Its value type is `String`.
- `organization`: This represents the organization/group ID. Its value type is `String`.
- `version`: This represents the version/revision of the current module. Its value type is `String`.
- `scalaVersion`: This represents the version of Scala to be used for the build. Its value type is `String`.
- `isSnapshot`: This represents whether the version is a snapshot version. Its value type is `Boolean`. By default it is set to `false`.
- `offline`: This is used to configure SBT to work without network connection where ever possible. Its value type is `Boolean`. By default it is set to `false`.
- `resolvers`: This is used to provide additional resource URIs for automatically managed dependencies. Its value type is `Seq[Resolver]`.
- `libraryDependencies`: This is used to declare managed dependencies. Its value type is a sequence of Ivy module IDs. We will see more on this in the next chapter.
- `parallelExecution`: This is used to configure whether tasks should be executed in parallel or not. Its value type is `Boolean` and the default value is `true`.
- `publishTo`: This represents `resolver` to which the project should be published.
- `publishMavenStyle`: This is used to configure whether to generate and publish a POM or not. Its value type is `Boolean`. If set to `true`, POM is published, otherwise an Ivy file is published.
- `PollInterval`: This represents the interval between checks for modified sources when running in triggered execution mode. Its value type is `Int` and the default value is 500.

## Path-related setting keys

The following is a list of the path-related setting keys mentioned:

- `scalaSource`: This represents the default Scala source directory. Its value type is a `File`.
- `javaSource`: This represents the default Scala source directory. Its value type is a `File`.
- `sourceDirectories`: This represents the list of all source directories both managed and unmanaged. Its value type is `Seq[File]`.

 Modify these only if your project structure is not similar to the standard structure.

You could refer to `basicConfig.sbt` while writing the build definition for a new project.

## Task keys

The following is a list of Task keys mentioned:

- `clean`: When executed, this key deletes the files produced by the build, such as generated sources, compiled classes, and task caches
- `console`: This starts the Scala interpreter with the project classes on the classpath
- `compile`: This command compiles the sources
- `doc`: This generates API documentation for the project
- `packageBin`: This produces a main artifact (binary JAR)
- `packageSrc`: This produces a source artifact (JAR containing sources and resources)
- `test`: This runs all the tests
- `update`: The execution of this command resolves and retrieves dependencies if required
- `publish`: This publishes artifacts to a repository
- `publishLocal`: This publishes artifacts to the local repository

We have already seen how tasks are executed, so let's skip ahead to see how Input keys work.



## Input keys

There are only four Input keys. They are as follows:

- `run`: This is used to run a main class with the command-line arguments. If no arguments are provided, a blank string is used.
- `runMain`: This is used to run a specific main class. It expects the class name followed by arguments. Simply trying to execute `runMain` without any arguments will result in an error.

The following is an example for this:

```
> run-main Chapter2.Introduction "A man,a plan,a canal:Panama"
[info] Running Chapter2.Introduction A man,a plan,a canal:Panama
Is A man,a plan,a canal:Panama a palindrome? True
```

- `testOnly`: This executes the test provided as arguments, or all tests if no argument is provided. It supports wildcards as well.
- `testQuick`: This also uses the filter approach such as `testOnly`, but executes the test only if the test had failed earlier, or was not run earlier, or the transitive dependencies of the test changed.



The quotes for the parameters are completely optional.

## Working with scopes

A context is called a **scope** in SBT. Each key can have more than one value, but in different scopes. In a given scope, a key has only one value. We didn't specify any scope for the keys in the build definition because in most cases the scope is implied. To set a scope specific value for the key, we need to mention the scope.

Scoping can be done at various levels where each level has its own scope. A scope can be defined at any of the following levels:

- **Project**: When using a single build for multiple projects, separate settings may be required for each project. In such scenarios, it will be useful to define keys at a project level. We will see more about the multiple project build definition in *Chapter 4, Full Build Definitions*.

- **Configuration:** A configuration is a module into which process-related sources, packages, and so on are grouped. The built-in configurations that SBT has are inspired by Ivy and Maven dependency scopes. Some common SBT configurations are as follows:
  - **Compile:** Paths for compile tasks
  - **Test:** Paths for testing-related tasks
  - **Runtime:** Paths for the run task

All the keys associated with compiling, packaging, and running are scoped by their respective configuration and may work differently in each configuration. For example, Scala/Java options or the sources can be set at configuration level.

- **Task:** Scoping at Task level means that the keys be scoped by a Task key; that is, the same setting can have different values for different tasks.

SBT has a special value, `Global` that can be used to specify scoping at the global level. When no scope is specified, the key will be scoped to the current project, Global configuration and Global task.

Setting a value for a scoped key in the SBT build definition is very simple. SBT provides an operator called `in`, which is used to specify the scope of the value being set. Let's look at the following examples:

```
parallelExecution in Test := false
```

If we include this in the build definition, tests will not be executed in parallel. To disable parallel execution for the project, we could set:

```
parallelExecution in Global := false
```

To set a value for a specific task, use this:

```
parallelExecution in testOnly:=false
```

To set it for multiple scopes, use this:

```
parallelExecution in (Test, RunTime) := false
```

To check the scope-specific value of a key in the interactive shell for configuration level (`scope:key-name`), use this:

```
> test:parallel-execution
[info] false
```

And for the task level (`task-name::key-name`), use this:

```
> test-only::parallel-execution  
[info] false
```

When working at the project level, each project should have its own `.sbt` file with the settings. The settings so configured are specific to the respective project. If no value is associated with a scoped key, then that key is undefined in that scope. In such cases, SBT tries to get the value associated with a more general scope. This allows the user to set keys whose values are common for various scopes at a general level and override wherever required.

## Summary

In this chapter, we learned about the different types of keys SBT provides in a build and the purpose of each key. We also saw the different syntaxes and ways in which a value can be mapped to a key, and the various scopes at which SBT supports configuration of the build. For more details, you could refer to the SBT documentation at <http://www.scala-sbt.org/release/docs/Getting-Started/More-About-Settings.html>.

# 3

## Dependency Management

Software development never happens in isolation. Most of the time, for complex projects, you will not work alone, nor will you develop all the components in the product from scratch. Products are generally created in teams, and they generally rely on external libraries and components. A product itself can be broken into modules developed by different teams dependent on each other. Dependencies in software development refer to the libraries or components required at various stages (compile, test, and runtime) of an application's development life cycle. The process of handling these dependencies, external or internal, for your application is called **dependency management**.

On the surface, it looks simple. All you have to do is take the JAR file and add it to your project. But when you actually have to handle it, problems arise. Some of the challenges are as follows:

- **Version management:** This will track the version of various dependencies you are using. Download the latest ones when they become available and replace the old ones. Ensure someone else in the team doesn't simply change the JAR file to a newer/older version.
- **Transitive dependencies:** This handles the chain of dependencies of the libraries you are dependent on, and also the dependencies of these dependencies.
- **Releasing your library:** If your library is part of a larger project, making your library available to others to use in an easy way is a challenge, especially when it is updated frequently (think about nightly snapshots).

In the early days of Java, when projects were small and didn't have many external dependencies, developers tended to manage dependencies manually by copying the required JAR files in the `lib` folder and checking it in their SCM/VCS with their code. This is still followed by a lot of developers, even today. But due to the aforementioned issues, this is not an option for larger projects.

In many enterprises, there are central servers, FTP, shared drives, and so on, which store the approved libraries for use and also internally released libraries. But managing and tracking them manually is never easy. They end up relying on scripts and build files.

Maven came and standardized this process. Maven defines standards for the project format to define its dependencies, formats for repositories to store libraries, the automated process to fetch transitive dependencies, and much more.

Most of the systems today either back onto Maven's dependency management system or on Ivy's, which can function in the same way, and also provides its own standards, which is heavily inspired by Maven. SBT uses Ivy in the backend for dependency management, but uses a custom DSL to specify the dependency.

## Quick introduction to Maven or Ivy dependency management

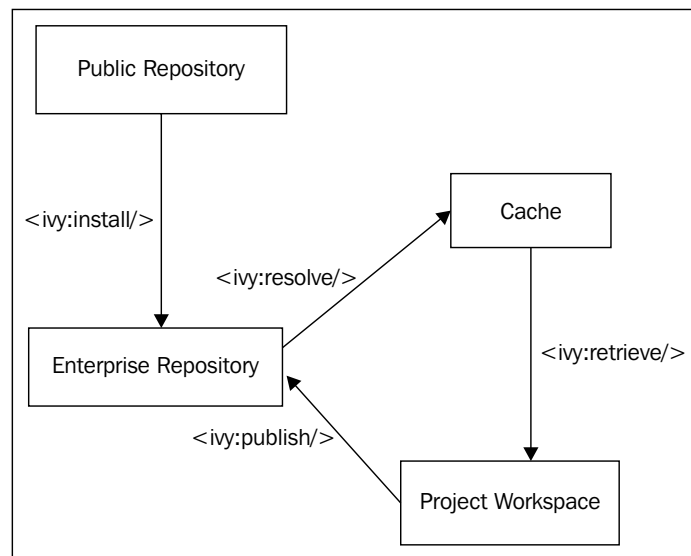
Apache Maven is not a dependency management tool. It is a project management and a comprehension tool. Maven is configured using a **Project Object Model (POM)**, which is represented in an XML file. A POM has all the details related to the project right from the basic ones, such as `groupId`, `artifactId`, `version`, and so on, to environment settings such as prerequisites, and repositories.

Apache Ivy is a dependency management tool and a subproject of Apache Ant. Ivy integrates publicly available artifact repositories automatically. The project dependencies are declared using XML in a file called `ivy.xml`. This is commonly known as the **Ivy file**.

Ivy is configured using a settings file. The settings file (`ivysettings.xml`) defines a set of dependency resolvers. Each resolver points to an Ivy file and/or artifacts. So, the configuration essentially indicates which resource should be used to resolve a module.

## How Ivy works

The following diagram depicts the usual cycle of Ivy modules between different locations:



The tags along the arrows are the Ivy commands that need to be run for that task, which are explained in detail in the following sections.

## Resolve

Resolve is the phase where Ivy resolves the dependencies of a module by accessing the Ivy file defined for that module.

For each dependency in the Ivy file, Ivy finds the module using the configuration. A module could be an Ivy file or artifact. Once a module is found, its Ivy file is downloaded to the Ivy cache. Then, Ivy checks for the dependencies of that module. If the module has dependencies on other modules, Ivy recursively traverses the graph of dependencies, handling conflicts simultaneously.

After traversing the whole graph, Ivy downloads all the dependencies that are not already in the cache and have not been evicted by conflict management. Ivy uses a filesystem-based cache to avoid loading dependencies already available in the cache.

In the end, an XML report of the dependencies of the module is generated in the cache.

## Retrieve

Retrieve is the act of copying artifacts from the cache to another directory structure. The destination for the files to be copied is specified using a pattern. Before copying, Ivy checks if the files are not already copied to maximize performance. After dependencies have been copied, the build becomes independent of Ivy.

## Publish

Ivy can then be used to publish the module to a repository. This can be done by manually running a task or from a continuous integration server.

## Dependency management in SBT

In SBT, library dependencies can be managed in the following two ways:

- By specifying the libraries in the build definition
- By manually adding the JAR files of the library

Manual addition of JAR files may seem simple in the beginning of a project. But as the project grows, it may depend on a lot of other projects, or the projects it depends on may have newer versions. These situations make handling dependencies manually a cumbersome task. Hence, most developers prefer to automate dependency management.

## Automatic dependency management

SBT uses Apache Ivy to handle automatic dependency management. When dependencies are configured in this manner, SBT handles the retrieval and update of the dependencies. An update does not happen every time there is a change, since that slows down all the processes. To update the dependencies, you need to execute the update task. Other tasks depend on the output generated through the update. Whenever dependencies are modified, an update should be run for these changes to get reflected. There are three ways in which project dependencies can be specified. They are as follows:

- Declarations within the build definition
- Maven dependency files, that is, POM files
- Configuration and settings files used for Ivy
- Adding JAR files manually

## Declaring dependencies in the build definition

The Setting key `libraryDependencies` is used to configure the dependencies of a project. The following are some of the possible syntaxes for `libraryDependencies`:

- `libraryDependencies += groupId % artifactID % revision`
- `libraryDependencies += groupId %% artifactID % revision`
- `libraryDependencies += groupId % artifactID % revision % configuration`
- `libraryDependencies ++= Seq(  
    groupId %% artifactID % revision,  
    groupId %% otherID % otherRevision  
)`

Let's explain some of these examples in more detail:

- `groupId`: This is the organization/group's ID by whom it was published
- `artifactID`: This is the project's name on which there is a dependency
- `revision`: This is the Ivy revision of the project on which there is a dependency
- `configuration`: This is the Ivy configuration for which we want to specify the dependency



Notice that the first and second syntax are not the same. The second one has a `%%` symbol after `groupId`. This tells SBT to append the project's Scala version to `artifactID`.

So, in a project with Scala Version 2.9.1, `libraryDependencies ++= Seq("mysql" %% "mysql-connector-java" % "5.1.18")` is equivalent to `libraryDependencies ++= Seq("mysql" % "mysql-connector-java_2.9.1" % "5.1.18")`.

The `%%` symbol is very helpful for cross-building a project. Cross-building is the process of building a project for multiple Scala versions. SBT uses the `crossScalaVersion` key's value to configure dependencies for multiple versions of Scala. Cross-building is possible only for Scala Version 2.8.0 or higher.

The `%%` symbol simply appends the current Scala version, so it should not be used when you know that there is no dependency for a given Scala version, although it is compatible with an older version. In such cases, you have to hardcode the version using the first syntax.



Using the third syntax, we could add a dependency only for a specific configuration. This is very useful as some dependencies are not required by all configurations. For example, the dependency on a testing library is only for the test configuration. We could declare this as follows:

```
libraryDependencies += Seq("org.specs2" % "specs2_2.9.1" % "1.12.3" %
  "test")
```

We could also specify dependency for the provided scope (where the JDK or container provides the dependency at runtime). This scope is only available on compilation and test classpath, and is not transitive. Generally, `javax.servlet-api` dependencies are declared in this scope:

```
libraryDependencies += "javax.servlet" % "javax.servlet-api" % "3.0.1"
  % "provided"
```

The revision does not have to be a single-fixed version, that is, it can be set with some constraints, and Ivy will select the one that matches best. For example, it could be latest integration or 12.0 or higher, or even a range of versions.

## A URL for the dependency JAR

If the dependency is not published to a repository, you can also specify a direct URL to the JAR file:

```
libraryDependencies += groupId %% artifactID % revision from directURL
```

`directURL` is used only if the dependency cannot be found in the specified repositories and is not included in published metadata. For example:

```
libraryDependencies += "slinky" % "slinky" % "2.1" from "http://
  slinky2.googlecode.com/svn/artifacts/2.1/slinky.jar"
```

## Extra attributes

SBT also supports Ivy's extra attributes. To specify extra attributes, one could use the `extra` method. Consider that the project has a dependency on the following Ivy module:

```
<ivy-module version="2.0" xmlns:e="http://ant.apache.org/ivy/extra">
  <info organization="packt"
    module = "introduction"
    e:media = "screen"
    status = "integration"
    e:codeWord = "PP1872"
  </ivy-module>
```

A dependency on this can be declared by using the following:

```
libraryDependencies += "packt" % "introduction" % "latest.integration"
extra(
  "media"->"screen", "codeWord"-> "PP1872")
```

The extra method can also be used to specify extra attributes for the current project, so that when it is published to the repository its Ivy file will also have extra attributes. An example for this is as follows:

```
projectID << projectID {id =>
  id extra( "codeWord"-> "PP1952")
}
```

## Classifiers

Classifiers ensure that the dependency being loaded is compatible with the platform for which the project is written. For example, to fetch the dependency relevant to JDK 1.5, use the following:

```
libraryDependencies += "org.testng" % "testng" % "5.7" classifier
"jdk15"
```

We could also have multiple classifiers, as follows:

```
libraryDependencies +=
  "org.lwjgl.lwjgl" % "lwjgl-platform" % lwjglVersion classifier
  "natives-windows"
  classifier "natives-linux" classifier "natives-osx"
```

## Transitivity

In logic and mathematics, a relationship between three elements is said to be transitive. If the relationship holds between the first and second elements and between the second and third elements, it implies that it also holds a relationship between the first and third elements.

Relating this to the dependencies of a project, imagine that you have a project that depends on the project `Foo` for some of its functionality. Now, `Foo` depends on another project, `Bar`, for some of its functionality. If a change in the project `Bar` affects your project's functionality, then this implies that your project indirectly depends on project `Bar`. This means that your project has a transitive dependency on the project `Bar`.

But if in this case a change in the project `Bar` does not affect your project's functionality, then your project does not depend on the project `Bar`. This means that your project does not have a dependency on the project `Bar`.

SBT cannot know whether your project has a transitive dependency or not, so to avoid dependency issues, it loads the library dependencies transitively by default. In situations where this is not required for your project, you can disable it using `intransitive()` or `notTransitive()`. A common case where artifact dependencies are not required is in projects using the Felix OSGI framework (only its main JAR is required). The dependency can be declared as follows:

```
libraryDependencies += "org.apache.felix" % "org.apache.felix.  
framework" % "1.8.0" intransitive()
```

Or, it can be declared as follows:

```
libraryDependencies += "org.apache.felix" % "org.apache.felix.  
framework" % "1.8.0" notTransitive()
```

If we need to exclude certain transitive dependencies of a dependency, we could use the `excludeAll` or `exclude` method.

```
libraryDependencies += "log4j" % "log4j" % "1.2.15" exclude("javax.  
jms", "jms")  
  
libraryDependencies +=  
  "log4j" % "log4j" % "1.2.15" excludeAll(  
    ExclusionRule(organization = "com.sun.jdmk"),  
    ExclusionRule(organization = "com.sun.jmx"),  
    ExclusionRule(organization = "javax.jms")  
  )
```

Although `excludeAll` provides more flexibility, it should not be used in projects that will be published in the Maven style as it cannot be represented in a `pom.xml` file. The `exclude` method is more useful in projects that require `pom.xml` when being published, since it requires both `organizationID` and `name` to exclude a module.

## Download documentation

Generally, an IDE plugin is used to download the source and API documentation JAR files. However, one can configure SBT to download the documentation without using an IDE plugin.

To download the dependency's sources, add `withSources()` to the dependency definition. For example:

```
libraryDependencies +=  
  "org.apache.felix" % "org.apache.felix.framework" % "1.8.0"  
  withSources()
```

To download API JAR files, add `withJavaDoc()` to the dependency definition. For example:

```
libraryDependencies +=
  "org.apache.felix" % "org.apache.felix.framework" % "1.8.0"
  withSources() withJavadoc()
```



The documentation downloaded like this is not transitive. You must use the `update-classifiers` task to do so.

## Dependencies using Maven files

SBT can be configured to use a Maven POM file to handle the project dependencies by using the `externalPom` method. The following statements can be used in the build definition:

- `externalPom()`: This will set `pom.xml` in the project's base directory as the source for project dependencies
- `externalPom(baseDirectory{base=>base/"myProjectPom"})`: This will set the custom-named POM file `myProjectPom.xml` in the project's base directory as the source for project dependencies

There are a few restrictions with using a POM file, as follows:

- It can be used only for configuring dependencies.
- The repositories mentioned in POM will not be considered. They need to be specified explicitly in the build definition or in an Ivy settings file.
- There is no support for `relativePath` in the parent element of POM and its existence will result in an error.

## Dependencies using Ivy files or Ivy XML

Both Ivy settings and dependencies can be used to configure project dependencies in SBT through the build definition. They can either be loaded from a file or can be given inline in the build definition.

The Ivy XML can be declared as follows:

```
ivyXML := <dependencies>
  <dependency org="org.specs2" name="specs2" rev="1.12.3">
  </dependency>
</dependencies>
```

The commands to load from a file are as follows:

- `externalIvySettings()`: This will set `ivysettings.xml` in the project's base directory as the source for dependency settings.
- `externalIvySettings(baseDirectory{base=>base/"myIvySettings"})`: This will set the custom-named settings file `myIvySettings.xml` in the project's base directory as the source for dependency settings.
- `externalIvySettingsURL(url("settingsURL"))`: This will set the settings file at `settingsURL` as the source for dependency settings.
- `externalIvyFile()`: This will set `ivy.xml` in the project's base directory as the source for dependency.
- `externalIvyFile(baseDirectory(_/"myIvy"))`: This will set the custom-named settings file `myIvy.xml` in the project's base directory as the source for project dependencies.

When using Ivy settings and configuration files, the configurations need to be mapped, because Ivy files specify their own configurations. So, `classpathConfiguration` must be set for the three main configurations. For example:

- `classpathConfiguration in Compile := Compile`
- `classpathConfiguration in Test := Test`
- `classpathConfiguration in Runtime := Runtime`

## Adding JAR files manually

To handle dependencies manually in SBT, you need to create a `lib` folder in the project and add the JAR files to it. That is the default location where SBT looks for unmanaged dependencies. If you have the JAR files located in some other folder, you could specify that in the build definition. The key used to specify the source for manually added JAR files is `unmanagedBase`. For example, if the JAR files your project depends on are in `project/extras/dependencies` instead of `project/lib`, modify the value of `unmanagedBase` as follows:

```
unmanagedBase <=<= baseDirectory {base => base/"extras/dependencies"}
```

Here, `baseDirectory` is the project's root directory.

`unmanagedJars` is a task which lists the JAR files from the `unmanagedBase` directory. To see the list of JAR files in the interactive shell type, type the following:

```
> show unmanaged-jars
[info] ArrayBuffer()
```

Or in the project folder, type:

```
$ sbt show unmanaged-jars
```

If you add a Spring JAR (`org.springframework.aop-3.0.1.jar`) to the dependencies folder, then the result of the previous command would be:

```
> show unmanaged-jars
[info] ArrayBuffer(Attributed(/home/introduction/extras/dependencies/org.
springframework.aop-3.0.1.jar))
```

It is also possible to specify the path(s) of JAR files for different configurations using `unmanagedJars`. In the build definition, the `unmanagedJars` task may need to be replaced when the jars are in multiple directories and other complex cases.

```
unmanagedJars in Compile += file("/home/downloads/ org.
springframework.aop-3.0.1.jar")
```

## Resolvers

Resolvers are alternate resources provided for the projects on which there is a dependency. If the specified project's JAR is not found in the default repository, these are tried. The default repository used by SBT is Maven2 and the local Ivy repository.

The simplest ways of adding a repository are as follows:

- `resolvers += name at location`. For example:  

```
resolvers += "releases" at "http://oss.sonatype.org/content/
repositories/releases"
```
- `resolvers += Seq (name1 at location1, name2 at location2)`.  
For example:  

```
resolvers += Seq("snapshots" at "http://oss.sonatype.org/content/
repositories/snapshots",
  "releases" at "http://oss.sonatype.org/content/repositories/
releases")
```
- `resolvers := Seq (name1 at location1, name2 at location2)`.  
For example:  

```
resolvers := Seq("sgodbillon" at "https://bitbucket.org/
sgodbillon/repository/raw/master/snapshots/",
  "Typesafe backup repo" at "http://repo.typesafe.com/
typesafe/repo/",
  "Maven repo1" at "http://repo1.maven.org/")
)
```

You can also add their own local Maven repository as a resource using the following syntax:

```
resolvers += "Local Maven Repository" at "file://" + Path.userHome.  
absolutePath + "/.m2/repository"
```

An Ivy repository of the file types URL, SSH, or SFTP can also be added as resources using `sbt.Resolver`. Note that `sbt.Resolver` is a class with factories for interfaces to Ivy repositories that require a hostname, port, and patterns. Let's see how to use the `Resolver` class.

For filesystem repositories, the following line defines an atomic filesystem repository in the test directory of the current working directory:

```
resolvers += Resolver.file("my-test-repo", file("test"))  
transactional()
```

For URL repositories, the following line defines a URL repository at `http://example.org/repo-releases/`:

```
resolvers += Resolver.url("my-test-repo", url("http://example.org/  
repo-releases/"))
```

The following line defines an Ivy repository at `http://joscha.github.com/play-easymail/repo/releases/`:

```
resolvers += Resolver.url("my-test-repo", url("http://joscha.github.  
com/play-easymail/repo/releases/")) (Resolver.ivyStylePatterns)
```

For SFTP repositories, the following line defines a repository that is served by SFTP from the host `example.org`:

```
resolvers += Resolver.sftp("my-sftp-repo", "example.org")
```

The following line defines a repository that is served by SFTP from the host `example.org` at port 22:

```
resolvers += Resolver.sftp("my-sftp-repo", "example.org", 22)
```

The following line defines a repository that is served by SFTP from the host `example.org` with `maven2/repo-releases/` as the base path:

```
resolvers += Resolver.sftp("my-sftp-repo", "example.org", "maven2/  
repo-releases/")
```

For SSH repositories, the following line defines an SSH repository with user-password authentication:

```
resolvers += Resolver.ssh("my-ssh-repo", "example.org") as("user",  
"password")
```

The following line defines an SSH repository with an access request for the given user. The user will be prompted to enter the password to complete the download.

```
resolvers += Resolver.ssh("my-ssh-repo", "example.org") as("user")
```

The following line defines an SSH repository using key authentication:

```
resolvers += {
  val keyFile: File = ...
  Resolver.ssh("my-ssh-repo", "example.org") as("user", keyFile,
    "keyFilePassword")
}
```

The next line defines an SSH repository using key authentication where no keyFile password is required to be prompted for before download:

```
resolvers += Resolver.ssh("my-ssh-repo", "example.org") as("user",
  keyFile)
```

The following line defines an SSH repository with the permissions. It is a mode specification such as chmod:

```
resolvers += Resolver.ssh("my-ssh-repo", "example.org")
  withPermissions("0644")
```

SFTP authentication can be handled in the same way as shown for SSH in the previous examples.

Ivy patterns can also be given to the factory methods. Each factory method uses a `Patterns` instance which defines the patterns to be used. The default pattern passed to the factory methods gives the Maven-style layout. To use a different layout, provide a `Patterns` object describing it. The following are some examples that specify custom repository layouts using patterns:

```
resolvers += Resolver.url("my-test-repo", url)(
  Patterns("[organisation]/[module]/[revision]/[artifact].[ext]") )
```

You can specify multiple patterns or patterns for the metadata and artifacts separately.

For filesystem and URL repositories, you can specify absolute patterns by omitting the base URL, passing an empty patterns instance, and using Ivy instances and artifacts.

```
resolvers += Resolver.url("my-test-repo") artifacts
  "http://example.org/[organisation]/[module]/[revision]/
  [artifact].[ext]"
```



When you do not need the default repositories, you must override `externalResolvers`. It is the combination of resolvers and default repositories. To use the local Ivy repository without the Maven repository, define `externalResolvers` as follows:

```
externalResolvers <=<= resolvers map { rs =>
  Resolver.withDefaultResolvers(rs, mavenCentral = false)
}
```

## Summary

In this chapter, we have seen how dependency management tools such as Maven and Ivy work and how SBT handles project dependencies. This chapter also talked about the different options that SBT provides to handle your project dependencies and configuring resolvers for the module on which your project has a dependency.

# 4

## Full Build Definitions

Until now, we have configured projects in SBT using a Scala-based DSL. But configuring in such a manner may not be sufficient in all kinds of projects. That's where writing a full configuration comes to the rescue. Full configuration in SBT is done using a Scala object. Since it's a Scala object, we can write Scala code within our build definition. Projects and their settings are configured in the object we create.

Let's look at a basic `.scala` build definition:

```
import sbt._
import sbt.Keys._

object DemoBuild extends Build {

  lazy val demo = Project(
    id = "demo",
    base = file("."),
    settings = Project.defaultSettings ++ Seq(
      name := "Demo",
      organization := "packt",
      version := "0.1-SNAPSHOT",
      scalaVersion := "2.9.2"
      // other settings here
    )
  )
}
```

A `.scala` build file defines an object that extends the `Build` trait defined in SBT. The object can have one or more SBT projects defined. In the preceding code, `DemoBuild` is our object that defines a project with `id demo`. The settings for the project `demo` are set by appending to the project's default settings in a manner similar to the declaration in the `.sbt` file.

The `.scala` build file gives us the flexibility to use Scala code for the settings. This is very useful in multiproject builds as things common to multiple projects can be declared once and used multiple times.

The build definition can also be a combination of `.sbt` and `.scala` files. It is strongly recommended to use `.sbt` files for build definition. The `.scala` files should be used only when we need to share code among expressions or for a multiproject build.

## Build definition project

The build definitions are based on Scala code. How will SBT understand the given configuration? Well, the answer to that is that SBT treats the build files as a project. Let's look at a simple project that uses a Scala build file to understand this better.

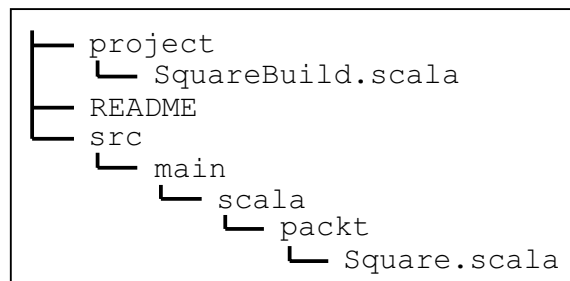
Consider that `Square.scala` is a program to print the square of a given number using a function, `calculate()`, and its `.scala` build file, `SquareBuild.scala`, is as follows:

```
import sbt._
import sbt.Keys._

object SquareBuild extends Build {

  lazy val square = Project(
    id = "square",
    base = file("."),
    settings = Project.defaultSettings ++ Seq(
      name := "square",
      organization := "packt",
      version := "1.0",
      scalaVersion := "2.9.2"
    )
  )
}
```

The following schema shows the project's structure before running any SBT command:



Run the code using SBT and then look at the project's structure. There will be a `target` folder within the base directory and another one in the `project` folder of the base. The `project` folder in the base directory is used by SBT to build the build definition specified by the `.sbt` and `.scala` files.

Another interesting feature of SBT is that we can configure the project that SBT creates based on the build definition. This can be done by writing a build definition within the `project` folder of the base directory. And this process can be done until any number of levels, since SBT sets the build configuration only after going through all the lower levels.

## **.sbt and .scala**

In a Scala build file, `sbt.Keys` needs to be imported explicitly, whereas all this is implied in a `.sbt` build file. The code to configure default settings using an `.sbt` build file is concise and smaller compared to a `.Scala` build file. But, `.Scala` build files provide greater flexibility in manipulating and/or customizing the settings.

The build definition can be a combination of `.sbt` build files and `.Scala` build files. In order to implement this, we need to understand how these files relate to each other. To see how a combination of `.sbt` and `.Scala` build files works, let's add a `build.sbt` file in the base directory of the `project square` with the following snippet of code:

```
scalaVersion = "2.10.0"
```

Then, run the following commands in the SBT interactive shell:

```
> clean
[success]
> reload
[info] Loading project definition from /square/project
[info] Set current project to square (in build file:/square/)
> update
[info] Updating {file:/square/}square...
[info] Resolving org.scala-lang#scala-library;2.10.0 ...
[info] Done updating.
```

Now, check the version of Scala used in the project using the following command:

```
> scala-version
[info] 2.10.0
```

This means that SBT loaded the configuration from the `.sbt` build file. Check the name of the project. Is it default-something? No? Then what is it? Square? How? SBT loads all the `.scala` build files and only then does it load the `.sbt` files. So, if any key has been set in both a `.scala` build file and a `.sbt` build file, its value will be overridden with the one given in the `.sbt` file. Therefore, a build object defined in the `.scala` build file is available in the `.sbt` build files. Since the settings in a `.sbt` file are project-scoped by default, to apply them on a multiproject build, `ThisBuild` should be specified as the configuration. For example:

```
scalaVersion in ThisBuild = "2.10.0"
```

So, the order of precedence for settings from low to high is as follows:

`.scala` build files < `.sbt` build files

settings scoped at project level < settings scoped at build level

## Working with full build definitions

We have already seen a working example of a full build definition in its simplest form. But, `SquareBuild.scala` can easily be replaced by a `.sbt` build file, and it is not a very good use case for a full build definition.

Let's look at parts of the build definition used in **Scalaz**. Scalaz is a Scala library for functional programming. It provides purely functional data structures and defines a set of foundational type classes (for example, `Functor` and `Monad`) and corresponding instances for a large number of types. Refer to the `build.scala` file of Scalaz for more details.

The dependencies are defined in an object as follows:

```
object Dependency {
  val ServletApi = "javax.servlet" % "servlet-api" % "2.5"
  def ScalaCheck(scalaVersion: String) = {
    val version = scalaVersion match {
      case "2.8.1" => "1.8"
      case "2.9.2" | "2.9.3" | "2.10.1" => "1.10.0"
    }
    "org.scalacheck" %% "scalacheck" % version
  }
  ...
}
```

Here, the version of `ScalaCheck` and the specifications are set based on the `ScalaVersion` key, since the project is built over a range of Scala versions.

Common settings are defined in another object, `standardSettings`, which is referenced in the build definition of the project as follows:

```
lazy val scalaz = Project(  
  id      = "scalaz",  
  base    = file("."),  
  settings = standardSettings,  
  ...  
)
```

In the previously mentioned scenario, using just the `.sbt` build files can be extremely cumbersome. Here, a full build definition truly serves the purpose of simplifying the build definition.

It is also possible to separate out the objects that are required for the build into multiple files. All the files should be placed in the `project` folder. For example, the object dependency can be moved to its own file and the build would still work in the same manner. This prevents cluttering of the build definition with independent objects. Another well-known project that uses a full definition build is **spark**. Spark is a Scala framework for iterative and interactive cluster computing.

## Multiproject builds

A build definition that consists of multiple project configurations is termed as a multiproject build. These are extremely useful when your project is a combination of two or more modules. If they depend on one another, you could also specify the dependencies so that whenever a change is made to one project, it is reflected in the projects that depend on it.

A single Scala file which declares the projects and their configuration can be used as the build definition of the complete project. Or, each module can have its own `.sbt` build file and then a Scala build file that specifies the dependencies, and so on. This is recommended as the build definition in SBT files takes precedence over the definition in the Scala files. But, the second method can become very difficult to manage and maintain.

Create a project, `power`, that uses the `square` project written earlier. Listed next are the paths for the source code:

- `Square.scala(/power/square/src/main/scala/dep/Square.scala)`
- `Power.scala(/power/src/main/scala/packt/Power.scala)`

The build file, `PowerBuild.scala (/power/project/PowerBuild.scala)`, is as follows:

```
import sbt._
import sbt.Keys._
object PowerBuild extends Build {

  lazy val square = Project(
    id = "square",
    base = file("square"),
    settings = Project.defaultSettings ++ Seq(
      name := "square",
      organization := "dep",
      version := "1.0",
      scalaVersion := "2.9.2"
    )
  )

  lazy val power = Project(
    id = "power",
    base = file("."),
    settings = Project.defaultSettings ++ Seq(
      name := "Power",
      organization := "packt",
      version := "1.0",
      scalaVersion := "2.9.2"
    )
  ) dependsOn(square)
}
```

Notice the `dependsOn()` method right after the declaration of the project `power`. This tells SBT that the project `power` depends on the project `square`. Once this dependency is set, the project `square` and its functions are accessible within the project `power` just by giving an `import` statement. To import a project, it should be referred by its `organizationID` followed by `projectName`. For example:

```
import dep.Square._
```

All SBT operations on a multiproject can be performed in the same manner as on a single project.

So, to run the project `power` from the interactive SBT shell, type the following:

```
> run 8
[info] Running packt.Power 8
64
```

Any update within the project on which another project is dependent is reflected directly without reloading the project. So, if I change `Square.calculate()` to return two times the square of a given number, I can see the change in the output of the power function:

```
> run 8
[info] Running packt.Power 8
128
```

This depends on what the function assumes that the dependency is on the compile configuration by default. So, in the preceding example, it is assumed that the compile configuration of `power` depends on the compile configuration of `square`. To explicitly specify a different configuration, the syntax is:

```
dependsOn(projectName % "configuration1->configuration2")
```

Here, `configuration1` is the configuration of a project which is dependent on another project and `configuration2` is the configuration of the project whose name is specified before the `%` symbol:

```
dependsOn(square)
```

This is shorthand for:

```
dependsOn(square % "compile->compile")
```

If the `->` symbol and `configuration2` are not specified, `configuration2` defaults to compile:

```
dependsOn(square % "test")
```

This is shorthand for:

```
dependsOn(square % "test->compile")
```

In case of dependencies on multiple configurations, each configuration should be separated by a semicolon. For example:

```
dependsOn(square % "compile->compile;test->test")
```

Or, you can use this:

```
dependsOn(square % "compile;test->test")
```

What if your project depends on multiple modules? The `dependsOn()` method can have one or more arguments. So, if the project `power` depended on `square` and `cube` (as a task, create `cube` in `power`), we could write:

```
dependsOn(square, cube)
```



Or, with explicit configuration dependencies, we could write:

```
dependsOn(square % "compile;test->test", cube % "compile;test->test")
```

All that seems good, but how will you combine different modules that do not depend on one another? This can be achieved by using the `aggregate()` method. So, if `power` was a combination of `square` and `cube` rather than being dependent on them, its declaration would be:

```
lazy val power = Project(
  id = "power",
  base = file("."),
  settings = Project.defaultSettings ++ Seq(
    name := "Power",
    organization := "packt",
    version := "1.0",
    scalaVersion := "2.9.2"
  )
) aggregate(square, cube)
}
```

Simply aggregate group projects together. Running a task on the base project will cause the task to be run in parallel on all the projects grouped under it. On compiling `power`, `square` and `cube` are also compiled. But, executing `run` will work only on the base project.

Generally, in a multiproject, some modules depend on others while some don't. However, we would still want to group them together. This can be achieved in SBT using a combination of the `aggregate()` and `dependsOn()` methods. Suppose that `cube` depends on `square`. Then, our build definition would be:

```
import sbt._
import sbt.Keys._

object PowerBuild extends Build {

  lazy val square = Project(
    id = "square",
    ...
  )

}
```

```
lazy val cube = Project(  
  id = "cube",  
  ...  
)dependsOn(square)  
  
lazy val power = Project(  
  id = "power",  
  ...  
)  
)aggregate(square, cube)  
}
```

We can see the grouped projects and switch between them just like a project on GitHub with multiple branches. To see the list of projects which are part of the aggregation, run the task projects:

```
> projects  
[info] In file:/power/  
[info]      cube  
[info]    * power  
[info]      square
```

And to switch to a project, run `project ProjectName`:

```
> project cube  
[info] Set current project to Cube (in build file:/power/)  
  
> run 9  
[info] Running dep2.Cube 9  
Cube=729
```

## Summary

In this chapter, we have seen what Scala build files are and how they relate to `.sbt` build files and the build definition project. We have also seen how Scala build definitions are used in actual projects. We created a multiproject build definition for both independent and dependent projects.



# 5

## Compile, Test, and Run

There is much more to a Scala project built in SBT than just the build definition. This chapter talks about the various commands and options, such as logging, testing, and so on, that are provided by SBT.

### Commands

What is a command? Every statement executed from the SBT prompt is a command. A command can either be an SBT task or command.

In SBT, each task represents a specific operation. Most of the tasks are run from the SBT prompt as commands while some are used internally by SBT. Based on the impact, tasks can be split into:

- Project-level tasks
- Configuration-level tasks

Tasks such as `clean`, `update`, `publish`, and so on, are categorized as project-level tasks, while tasks such as `compile`, `console`, `run`, `test`, `run-main`, and so on, fall into the category of configuration-level tasks, as they can be executed for different configurations.

In SBT, a command is just another type like keys. A command has access to the current state of the build and can modify that state. So, commands are very much like task keys with additional privileges. Some of the commonly used commands are as follows:

- `exit` or `quit`: These are used to end an interactive session or build. You could also use `Ctrl + D` on Unix-based systems and `Ctrl + Z` on Windows systems for the same.

- `eval <expression>`: This can be used to evaluate any Scala expression. It could be as simple as this:

```
> eval "Hello"+"world"
[info] ans: java.lang.String = Helloworld
```

Or, it can be a little intense like the following:

```
> eval Array(1,2,3).map(n=>println("Number "+ n))
Number 1
Number 2
Number 3
[info] ans: Array[Unit] = [Lscala.runtime.BoxedUnit;@25e157bf
```

Or, it can be complex like this:

```
> eval {val n= Array(1,2,3);val number =
Array("one","two","three"); val result=n.map(i=>i+"-"+number(i-
1));result.foreach(r=>println(r));}
1-one
2-two
3-three
[info] ans: Unit = null
```

- `help <commandName>`: This displays the help related to the given `commandName` that can either be a key or a command. If `commandName` is not given, it displays some of the main commands.
- `~ <commandName>`: This is used for a triggered execution of the given command. As and when changes in the project code are saved, this command is executed.
- `< fileName>`: This can be used to run the commands in the given file. For example, create a file called `commands` with the following content:

`test`

```
> < commands
[info] IntroSpec
.....
```

```
[info] Passed: : Total 4, Failed 0, Errors 0, Passed 4, Skipped 0
[success]
```

This file can have multiple commands, but each command should start in a new line. Empty lines or lines starting with # are ignored. For example, update the `commands` file to the following:

```
#compile, test and run
compile
test
run

> < commands
[info] Updating default-e2e6e3...
...
[info] Done updating.
[info] Compiling 1 Scala source to
.....
[info] IntroSpec
.....
[info] Passed: : Total 4, Failed 0, Errors 0, Passed 4, Skipped 0
[success]
[info] Running ...
[success]
```

This command can also be used to run multiple files, and the different file names should be space separated.

- `+ <commandName>`: This causes the specified command to be executed for all the Scala versions assigned to the `crossScalaVersions` key. So, if the build definition has three versions, for all the three versions to compile, executing and compiling is sufficient.  
`crossScalaVersions := Seq( "2.8.1", "2.9.2", "2.10.1")`
- `++ <version> <commandName(optional)>`: This command updates the Scala version of the project to the given version. If `commandName` is specified, it runs that command after updating the version. The Scala version set for the project using this command is reset for that session only if another command modifies it or if the project is reloaded. For example:  
`> ++ 2.10.1`

Or, consider this:

```
> ++ 2.10.0 update
```

- `; commandA ; commandB`: This causes the execution of `commandB` if and only if `commandA` was executed successfully:

```
>;compile;test
[info] Compiling 1 Scala source to
[error] .... ';' expected but string literal found.
[error] one error found
[error] (compile:compile) Compilation failed
```

This could go on for multiple commands. For example:

```
; compile ; test ; run
```

- `inspect <settingKey>`: This command displays detailed information about the given `settingKey`.
- `set <setting-expression>`: This command is used to set the value of `settingKey` from the SBT prompt. The values set using this command are limited to the session and are lost on reloading the project.
- `session <command>`: The `session` command is used to manipulate settings modified using the `set` command for the current session, that is, temporary settings which are valid only for that session. The commands that can be used with the `session` command are as follows:
  - `clear`: This clears the current project's temporary settings for that session.
  - `list`: This prints a list of temporary settings with numbers which can be used later to remove the settings.
  - `remove <range-spec>`: This removes all the settings specified in `range-spec`. The parameter `range-spec` can be a list of comma-separated numbers or a number range or a combination of both, where each number represents a temporary setting as listed using the `list` command.
  - `save`: This persists the temporary settings into a `.sbt` file. If the project does not have a `.sbt` file, SBT creates a `.sbt` file within the project's base directory and appends the session settings to it.
  - `clear-all`, `list-all`, and `save-all`: You will see that `clear`, `list`, and `save` have equivalent commands at build level, respectively. When `save-all` is used, the temporary settings are appended to the first `.sbt` file in the project.

- `reload [plugins|return(optional)]`: Without the optional arguments, the build is reloaded. Therefore, this command should be run after updating the build definition. If `plugins` is given as the argument, the current project changes to the build definition project. This can be used to manipulate the build definition. If `return` is given as the argument, the main project is set as the current project. For example:

```
> reload plugins
[info] Loading project definition from /home/square/project
> reload
[info] Loading project definition from /home/square/project
> reload return
[info] Set current project to default-afcf4 (in build file:/home/square/)
> reload
[info] Set current project to default-afcf4 (in build file:/home/square/)
```

## Logging

**Logging** is the process of recording events as they occur in files that are known as **logfiles**. Logfiles can be used for analyzing the execution or figuring out errors in the code. SBT provides an internal logging mechanism that can be configured as per your requirements. After running a command, you just need to run the `last` command. For example:

```
> reload
[info] Loading project definition from /home/.../projectName/project
[info] Set current project to Introduction (in build file:/home/.../projectName)

> last
[info] Loading project definition from /home/.../projectName/project
[debug] Running task... Cancelable: false, check cycles: false
[debug]
[debug] Initial source changes:
[debug]   removed: Set()
[debug]   added: Set()
[debug]   modified: Set()
```



```
[debug] Removed products: Set()
[debug] Modified external sources: Set()
[debug] Modified binary dependencies: Set()
[debug] Initial directly invalidated sources: Set()
[debug]
[debug] Sources indirectly invalidated by:
[debug]   product: Set()
[debug]   binary dep: Set()
[debug]   external source: Set()
[debug] Initially invalidated: Set()
[debug] Copy resource mappings:
[debug]
[info] Set current project to Introduction (in build file:/home/.../
projectName)
```

The last command can also be used to get details of commands that are run by default. For example, when you execute `test` without updating and compiling the source code, `update` and `compile` are executed before the tests are actually done. So, to see the logs for update, use the following:

```
>last update
....
[debug] resolve done (77ms resolve - 6ms download)
[info] Done updating.
```

Or, to see compile logs, use the following:

```
> last compile
...
[debug] Scala compilation took 3.249011536 s
[debug]   Invalidated direct: Set()
[debug] Incrementally invalidated: Set()
[debug] Modified binary dependencies: Set()
[debug] Initial directly invalidated sources: Set()
```

When using Scala Version 2.10 or higher, the `print-warnings` command can be used. This displays all warnings from the previous command. The verbosity, stack trace, and buffering of logs can all be configured with ease.

The setting key `logLevel` can be used to set the extent of verbosity of the logs displayed on the screen. The type of value for it is an enumeration value. There are four levels of logging:

- `Debug`
- `Info`
- `Warn`
- `Error`

The highest level includes all the levels below it. For example, if I set log level to `Error`, only error logs will be displayed:

```
logLevel := Level.Error
```

This results in the following output:

```
> test
[success]
> compile
[success]
```

Similarly, `persistLogLevel` can be used to set the extent of verbosity of the logs persisted into a file. Its value type is the same as that of `logLevel`.

The stack trace of most exceptions is hidden by default in SBT. This can be configured using another setting key called `traceLevel`. The type of value for this is `Int`. The levels of a stack trace corresponding to different integer values are shown next:

- Negative number: No stack trace
- Zero: Stack trace up to the first SBT stack frame
- Positive number: Stack trace up to that many stack frames

Similarly, `persistTraceLevel` can be used to set the extent of a stack trace persisted into a file.

The log output of a test is buffered until the whole class finishes so that it doesn't get mixed up when executing in parallel. If this feature is not desired, you could disable it using the `logBuffered` setting key. Its value type is `Boolean` and the default value is `true`. To disable it, add the following statement to your build definition:

```
logBuffered := false
```

Besides these, you could also configure SBT to use a logging framework. All you need to do is add its dependency and configuration file. For example, to use logback, add the library dependencies as follows:

```
libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.0.1"
```

Then, include the `logback.xml` configuration file in `/src/main/resources`. If you wish to use the logger for tests as well, add a `logback-test.xml` file in `/src/test/resources`. After doing so, you can refer to the logger in your source code as you normally would.

## Forking the JVM

**Fork** is a Unix terminology. **Forking** a process essentially means cloning a process in such a way that the original and the clone execute independently of each other. Also, the two processes can execute concurrently.

Why do we need to fork the JVM?

When a user runs code using `run` or `console` commands, the code is run on the same virtual machine as SBT. In some cases, running of code may cause SBT to crash, such as a `System.exit` call or unterminated threads (for example, when running tests on code while simultaneously working on the code). If a test causes the JVM to shut down, you would need to restart SBT. In order to avoid such scenarios, forking the JVM is important.

You do not need to fork the JVM to run your code if the code follows the constraints listed as follows, else it must be run in a forked JVM:

- No threads are created or the program ends when user-created threads terminate on their own
- `System.exit` is used to end the program and user-created threads terminate when interrupted
- No deserialization is done or deserialization code ensures that the right class loader is used

To configure forking for your project, you can use the `fork` Setting key. The type of its value is `Boolean`. It can only be set for the `test` commands and the `run` commands. To enable forking for all possible commands (`run` and `test`), use the following:

```
fork:=true
```

This sets `fork` as `true` for the `test`, `test-only`, `test-quick`, `run`, `run-main`, `test:run`, and `test:run-main` tasks.

Enable `fork` for `run` tasks only as follows:

```
fork in run := true
```

Enable `fork` for `test:run` and `test:run-main` only as follows:

```
fork in (Test,run) := true
```

Enable `fork` for main `run` tasks only as follows:

```
fork in (Compile,run) := true
```

And enable it for all test tasks as follows:

```
fork in test := true
```

The options to be provided to the forked JVM and the Java installation to be used are also configurable. This can be done using the `javaOptions` and `javaHome` Setting keys. For example:

```
javaOptions in (Test,run) += "-Xmx8G"

javaHome in run := file("/path/to/jre/")
```

The forked process uses the same Java and Scala version being used for the build and the working directory while the JVM options are the same as that of the current process if not specified explicitly in the build definition.

The input of the SBT process is not forwarded to the forked process by default. To enable this, the `connectInput` setting key should be used. Its value should be of the type `Boolean`. For example:

```
connectInput in run := true
```

The forked output is sent to the logger by default. It can be configured using the `outputStrategy` Setting key. For example:

```
outputStrategy := Some(BufferedOutput(log: Logger))
```

This sends the output to the logger only after the process terminates.

If a new Java process is to be forked, you could simply use the SBT fork API.

## Parallel execution

In SBT, task execution occurs in parallel by default. This is done to utilize all the available processors. Even tests are run in parallel. To make this simpler, each test class is mapped to its own task.

From SBT 0.12.0 onwards, there is more control for the user over task concurrency. Earlier, users could only enable or disable parallel execution using the Setting key `parallelExecution`.

SBT now has a feature to tag tasks based on their purpose and utilization. These tags can also be used to restrict task concurrency. So, the system now depends on proper tagging and restrictions. Users can now selectively leverage task concurrency where required and also limit it if necessary.

Each tag represents a resource and has a weight associated with it. The weight represents the tasks relative utilization of the resource. A task can be assigned a tag using either the `tag()` or the `tagw()` method. For example:

```
compile <=> myCompileTask tag(Tags.CPU, Tags.Compile)

download <=> downloadImpl.tagw(Tags.Network -> 3)
```

The `tag()` method sets the weight as 1 by default, while the `tagw()` method expects tag-weight pairs.

But, where are we restricting the number of tasks executed in parallel using the tags? This restriction is configured using the `concurrentRestrictions` Setting key. For example:

```
concurrentRestrictions in Global := Seq(
  Tags.limitSum(2, Tags.CPU, Tags.Untagged)
  Tags.limit(Tags.Network, 10),
  Tags.limit(Tags.Test, 1),
  Tags.exclusive(Tags.Update)
  Tags.limitAll( 15 )
)
```

Short-lived tasks are generally not tagged by users. Such tasks are automatically tagged with the label `Untagged`.

The `limit()` method sets the maximum allowed number of tasks with a given tag and the `limitAll()` method sets the maximum number of tasks that can be run in parallel at any given time. The value provided as the limit should be greater than or equal to 1, else SBT will generate an error.

The `limitSum()` method sets the maximum allowed number of tasks for a combination of different tags.

The `exclusive()` method takes a tag as its argument. If a tag is specified as exclusive, it is always run in isolation, that is, no other tasks are run in parallel with it even if they also have a tag that is specified as exclusive. Such tasks are run only when no other task is running and another task cannot start unless it is completed.

Furthermore, you can specify a custom function of the type `Map[Tag, Int] => Boolean`. The `Map[Tag, Int]` input represents the weighted tags of a set of tasks. The result is `true` if and only if the set of tasks are allowed to run concurrently. This custom function should be written in such a way that if a single weighted tag is passed, it should return `true`, else SBT throws a warning about the restriction and executes the task. For example:

```
...
Tags.customLimit { (tags: Map[Tag, Int]) =>
  val exclusive = tags.getOrElse(Tags.Update, 0)
  // the total number of tasks in the group
  val all = tags.getOrElse(Tags.All, 0)
  // if there are no exclusive tasks in this group, this rule adds no
  restrictions
  exclusive == 0 ||
    // If there is only one task, allow it to execute.
    all == 1
}
...
```

The built-in tags can be categorized as follows:

- **Semantic tags:** These are task-based tags. The different semantic tags are as follows:
  - `Compile`
  - `Test`
  - `Publish`
  - `Update`
  - `Untagged`
  - `All` (this tag is added to all tasks automatically)

- **Resource tags:** These are resource utilization-based tags. The different resource tags are as follows:
  - Network
  - Disk (filesystem)
  - CPU

Some of the tasks in SBT are tagged by default. They are as follows:

- **compile:** Compile, CPU
- **test:** Test
- **update:** Update, Network
- **publish, publish-local:** Publish, Network

In addition to this, users can create their custom tags just by declaring them in the build definition as follows, using the value's name (`Custom` in this case) as the tag label:

```
val Custom = Tags.Tag("custom")
```

## SBT scripts and REPL

SBT has two alternative entry points, the Scala REPL and a Scala script, which are experimental.

SBT can be used to compile and execute a Scala script containing SBT dependency declarations and/or other settings. It can also be used to start the Scala REPL with the dependencies on the classpath.

This requires additional setup. In the folder where you have the SBT script, add the following two scripts:

- The `scalas` script is as follows:

```
#!/bin/bash
```

```
java -Dsbt.main.class=sbt.ScriptMain -Dsbt.boot.directory=/
home/${USER}/.sbt/boot -jar `dirname $0`/sbt-launch.jar "$@"
```

- The `screpl` script is as follows:

```
#!/bin/bash
```

```
java -Dsbt.main.class=sbt.ConsoleMain -Dsbt.boot.directory=/
home/${USER}/.sbt/boot -jar `dirname $0`/sbt-launch.jar "$@"
```

With the `scalas` script, we can run a standard Scala script that can configure SBT. SBT settings should be enclosed within a comment block. The comment block starts with `/**` and ends with `*/`.

For example, create a simple Scala script, `demo.scala`, as shown as follows:

```
#!/usr/bin/env scalas
!#

/**
scalaVersion := "2.9.1"

*/

println("Hello World")
```

To run this script on a Unix-based system, run the following statements:

```
$chmod u+x demo.scala
$./demo.scala
```

The `screpl` script can be used to start Scala REPL with the desired dependencies. To start the REPL, simply run the `screpl` script:

```
$ screpl
```

```
[info] Set current project to default-f5f373 (in build file:/home/ .sbt/
boot/ivy-console/)
```

```
Welcome to Scala version 2.9.2 (Java HotSpot(TM) 64-Bit Server VM, Java
1.7.0_17).
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala>
```

The resolvers and dependencies can be passed as arguments. For example:

```
screpl "sonatype-releases at https://oss.sonatype.org/content/
repositories/snapshots/" "org.scalaz%%scalaz-core%7.0-SNAPSHOT"
```



## Classpath, sources, and resources

SBT has a clear distinction between classpath and source. In SBT, classpath refers to the path of the JAR files on which the project/configuration has a dependency, whereas source refers to the path of the Scala/Java files. The resources are just like in any other build tool. These three are further divided into the following categories:

- **managed:** It is managed by SBT. For example, dependencies declared in the build definition.
- **unmanaged:** It's manually managed by the user. For example, dependencies added in the `lib` folder.

Classpaths are further classified as **internal** and **external** in addition to managed and unmanaged. Internal classpaths are a result of inter-project dependency; for example, if a project has a dependency on another project within the build. External classpaths are the union of managed and unmanaged classpaths.

Look at the following outputs for a clear understanding of this:

```
> show managed-classpath
[info] List(Attributed(/home/user/.sbt/boot/scala-2.9.1/lib/scala-library.jar))
[success]

> show unmanaged-classpath
[info] ArrayBuffer()
[success]

> show test:managed-classpath
[info] List(Attributed(/home/user/.sbt/boot/scala-2.9.1/lib/scala-library.jar), Attributed(/home/user/.ivy2/cache/org.specs2/specs2_2.9.1/jars/specs2_2.9.1-1.12.3.jar), Attributed(/home/user/.ivy2/cache/org.specs2/specs2-scalaz-core_2.9.1/jars/specs2-scalaz-core_2.9.1-6.0.1.jar))
[success]
```

```
> show external-dependency-classpath
[info] ArrayBuffer(Attributed(/home/user/.sbt/boot/scala-2.9.1/lib/scala-
library.jar))
[success]

> show internal-dependency-classpath
[info] ArrayBuffer()
[success]

> show sources
[info] ArrayBuffer(/home/user/demo/src/main/scala/Demo.scala)
[success]

> show unmanaged-sources
[info] ArrayBuffer(/home/user/demo/src/main/scala/Demo.scala)
[success]



> show managed-sources
[info] List()
[success]

> show resources
[info] List(/home/user/demo/src/main/resources)
[success]

> show unmanaged-resources
[info] ArrayBuffer(/home/user/demo/src/main/resources)
[success]
```

```
> show managed-resources  
[info] List()  
[success]
```

Attributed is a type that associates a heterogeneous map with each classpath entry.

 sources and resources is a combination of unmanaged sources, managed-sources, and resources respectively. 

How is the Scala file in `src/main/scala` in `unmanaged-sources` and the `src/main/resources` in `unmanaged-resources`? This is because the files in these folders are not generated or updated by SBT. The files generated/downloaded by SBT while building/updating a project only fall into the categories of `managed-sources` or `managed-resources`.

The related keys are as follows:

- **Classpath:** `unmanaged-classpath`, `managed-classpath`, `external-dependency-classpath`, and `internal-dependency-classpath`.
- **Sources:** `unmanaged-sources`, `managed-sources`, `sources`, and `source-generators`. Note that `source-generators` is a task that generates source files (`managed-sources`).
- **Resources:** `unmanaged-resources`, `managed-resources`, `resources`, and `resource-generators`.

It is also possible to exclude source files from a build. This can be done using `excludeFilter`. For example:

```
excludeFilter in unmanagedSources := "demo2.scala"
```

## Test

Testing gives you an insight into the workings of an application. It helps in understanding the extent to which the application does its job and the cases where it fails to do so. This information is highly valuable for a developer, and SBT understands that. It provides support for running tests in continuous mode so that you can see the impact the moment a change is made.

All the major Scala testing frameworks can be used within an SBT project just by adding them as a project dependency for the test configuration.

To use `specs2`, include the following:

```
libraryDependencies += "org.specs2" %% "specs2" % "1.14" % "test"
```

For `scalacheck`, add the following:

```
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.10.1" %  
"test"
```

And for `ScalaTest`, add the following:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "1.6.1" %  
"test"
```

There is also support for JUnit. Use `junit-interface` in the build definition:

```
libraryDependencies += "com.novocode" % "junit-interface" % "0.10-M4"  
% "test"
```

SBT has three tasks which can be executed to run tests. They are `test`, `test-quick`, and `test-only`. Let's see the difference between each of them by running them on a project with two test classes: `WordSpec.scala` and `NumberSpec.scala`. These classes have been derived from the `IntroSpec` class in *Chapter 1, Hello World with SBT*.

```
> test  
[info] Compiling 2 Scala sources to /home/.../introduction/target/  
scala-2.9.1/test-classes...  
[info] NumberSpec  
...  
[info] WordSpec  
...  
[info] Passed: : Total 4, Failed 0, Errors 0, Passed 4, Skipped 0  
[success]
```

```
> test-quick  
[info] No tests to run for test:test-quick  
[success]
```

```
>test-only N*  
[info] NumberSpec  
...  
[info] Passed: : Total 2, Failed 0, Errors 0, Passed 2, Skipped 0  
[success]
```

From the results, we can see that the `test` task runs all the tests while the `test-only` task runs a specified task. But what does `test-quick` do? The `test-quick` task runs only those tests that have been affected by changes in source code or those that failed earlier. Modify `NumberSpec` and then run `test-quick`. Only the tests defined in `NumberSpec` will be run.

There is another way of filtering the tests that should be run. This can be done using the `testOptions` setting key. For example, add the following code to your build definition:

```
testOptions in Test := Seq(Tests.Filter(s => s.startsWith("N")))
```

Then, reload your project. When you run the `test` command, only tests in `NumberSpec` will be run:

```
> test
[info] NumberSpec
...
[info] Passed: : Total 2, Failed 0, Errors 0, Passed 2, Skipped 0
[success]
```

This is because `testOptions` is applied before any of the SBT testing tasks are executed.

Another interesting feature of SBT is that tests can be run in a separate JVM. The `fork` Setting key comes in handy here. To enable execution of all tests in a single external JVM, add the following code to your build definition:

```
fork in Test := true
```

All of this was just about unit tests. Is it possible to run other kinds of tests using SBT? Yes, you could also run integration tests and, if required, add your customized test configuration.

Consider that we have an integration test, `HelloWorldSpec` (refer to this in the code). To run an integration test defined in this class, we would need to add this file in `src/it/scala` and add it in the build definition:

```
import sbt._
import Keys._

object DemoBuild extends Build
{
```

---

```

lazy val demo =
  Project("demo", file("."))
    .configs( IntegrationTest )
    .settings( Defaults.itSettings : _*)
    .settings(
      name := "Introduction",
      version := "1.0",
      scalaVersion := "2.9.1",
      libraryDependencies ++= Seq("org.specs2" %% "specs2" % "1.12.3" %
        "it,test")
    )
  }

```

Here, `configs(IntegrationTest)` adds the predefined integration test configuration. The `IntegrationTest` configuration is labeled as `it`. So, `settings(Defaults.itSettings: _*)` is used to add compilation, packaging, testing actions, and settings in the `IntegrationTest` configuration.

In the preceding example, a library dependency is specified for both `it` and `test` configurations, but different dependencies can be specified for each configuration.

Now, the integration test can be run using the `test` task as follows:

```
> it:test
```

All the testing tasks can be executed for integration tests just by scoping them into the `IntegratedTest` configuration. So, the commands for integration testing are `it:test`, `it:test-quick`, and `it:test-only`.

Adding a custom test configuration is also done in a similar manner. In the `.scala` build file, define a configuration object:

```
lazy val CustomTest = config("ct") extend(Test)
```

Then, define the configuration and settings as follows:

```

configs(CustomTest)
settings( inConfig(CustomTest) (Defaults.testSettings):_*)

```

Library dependencies if any should be specified for the `ct` configuration. All the Scala test code for the `CustomTest` configuration should be in `src/ct/scala` and the commands for running tests would be `ct:test`, `ct:test-quick`, and `ct:test-only`.

## Summary

In this chapter, we have seen the different commands that SBT provides. We have also seen how to check the default logs, how to configure SBT to use a logging framework, how to fork the JVM, and how to customize parallel execution. This chapter also gives an introduction to leveraging SBT and running a simple Scala file or starting the Scala REPL with required dependencies. Creating, configuring, and executing tests have also been covered.

# Index

## Symbols

`++=` 17, 18  
`+=` 17, 18  
`<<=` 17  
`<<=, <<` 18  
`~=` 17, 18  
`$=` 17, 18  
`~` character 13  
`; commandA ; commandB` 52  
`+ <commandName> command` 51  
`~ <commandName> command` 50  
`< fileName command` 50  
`.sbt`  
    about 15, 41, 42  
    syntax 16  
`.scala build definition` 39-42  
`++ <version> <commandName(optional)>`  
    command 51

## A

`aggregate()` method 46  
Another Neat Tool. *See* ANT  
ANT 6  
Apache Ivy. *See* Ivy  
Apache Maven 26  
API JAR files  
    downloading 33  
Attributed 64

## B

Bar 31  
build definition

    .scala build definition 39, 40  
    full build definition, working with 42, 43  
    Multiproject builds 43-47  
**build definitions** 40, 41  
**build.sbt** 16  
**built-in tags**  
    resource tags 60  
    semantic tags 59

## C

`calculate()` function 40  
**classifiers** 31  
**Classpath**  
    about 62-64  
    external 62  
    internal 62  
**clean** 21  
**clean command** 12  
**command**  
    ; commandA ; commandB 52  
    + <commandName> 51  
    ~ <commandName> 50  
    < fileName 50, 51  
    ++ <version> <commandName  
        (optional)> 51about 49  
    eval <expression> 50  
    exit command 49  
    help <commandName> 50  
    inspect <settingKey> command 52  
    quit command 49  
    reload [plugins|return(optional)]  
        command 53  
    session <command> command 52  
    set <setting-expression> command 52



**compile** 21  
**configuration-level tasks** 49  
**connectInput setting** 57  
**console** 21

## D

**dependencies**  
  Ivy files used 33  
  Ivy XML used 33  
  Maven files used 33  
**dependency JAR**  
  URL 30  
**dependency management**  
  about 25, 28  
  automatic dependency management 28  
  classifiers 31  
  dependencies, declaring in build  
    definition 29, 30  
  dependency JAR, URL 30  
  extra attributes 30, 31  
  transitivity 31, 32  
**dependsOn() method** 44, 45  
**directURL** 30  
**doc** 21

## E

**eval <expression> command** 50  
**excludeAll method** 32  
**exclude method** 32  
**exclusive() method** 59  
**exit command** 49  
**externalIvyFile()** 34  
**externalIvySettings()** 34  
**externalPom()** 33

## F

**Foo** 31  
**forking** 56

## H

**help <commandName> command** 50

## I

**input keys**  
  about 18  
  run 22  
  runMain 22  
  testOnly 22  
  testQuick 22  
**inspect <settingKey> command** 52  
**intransitive()** 32  
**isSnapshot** 20  
**Ivy**  
  about 26  
  modules 26, 27  
  publish phase 28  
  resolve phase 27  
  retrieve phase 28  
**Ivy file**  
  about 26  
  used, for dependencies 33, 34  
**Ivy XML**  
  used, for dependencies 33, 34

## J

**JAR files**  
  adding, manually 34, 35  
**javaSource** 21  
**junit-interface** 7  
**JVM**  
  forking 56, 57

## K

**keys**  
  about 18  
  example 19  
  input keys 18, 22  
  setting keys 18, 20  
  task keys 18, 21

## L

**last command** 54  
**library**  
  releasing 25

## **libraryDependencies**

about 20, 29

syntax 29

**limitAll()** method 58

**limit()** method 58

**limitSum()** method 59

**logBuffered** 55

**logfiles** 53

**logging**

about 53-55

levels 55

## **M**

**Mac**

SBT, installing on 8

**Make tool** 5

**Map[Tag,Int] input** 59

**Maven.** *See also* **Apache Maven**

**Maven**

dependencies used 33

## **N**

**notTransitive()** 32

## **O**

**offline** 20

## **P**

**package**

SBT, installing from 8

**packageBin** 21

**packageSrc** 21

**parallelExecution** 20

**Patterns object** 37

**persistLogLevel** 55

**PollInterval** 20

**POM**

about 26

file, restrictions 33

**project**

compiling 11

creating 9, 10

running 11

testing 11

**project level** 22

**project-level tasks** 49

**Project Object Model.** *See* **POM**

**publish** 21 28

**publishLocal** 21

**publishMavenStyle** 20

**publishTo** 20

## **Q**

**quit command** 49

## **R**

**reload [plugins | return(optional)] command**  
53

**resolve phase** 27

**Resolvers**

about 20, 35

repository, adding 35-37

**resources** 62, 64

**resource tags** 60

**retrieve phase** 28

**run** 22

**runMain** 22

## **S**

**SBT**

about 6, 7

dependency management 28

installing 7

installing, from package 8

installing, manually 8

installing, on Mac 8

parallel execution 58-60

Scala REPL 60

Scala script 60

scopes 22-24

shell 12

**sbt.bat file** 8

**SBT commands**

triggerring, on saves 13

**SBT configurations**

compile 23

runtime 23

test 23

- sbt.Keys** 41
- sbt.Resolver**, **sbt.Resolver class** 36
- SBT shell** 12
- Scala REPL** 60
- Scala SBT**
  - directory structure 9
- Scala script**
  - about 60
  - creating 61
- scalaSource** 21
- scalaVersion** 18, 20
- ScalaVersion key** 42
- Scalaz** 42
- scopes**
  - about 22, 23
  - configuration 23
  - project level 22
  - task level 23
- semantic tags** 59
- servlet-api dependencies** 30
- session <command>** 52
- session command** 52
- setting keys**
  - about 18, 20
  - isSnapshot 20
  - libraryDependencies 20
  - name 20
  - offline 20
  - organization 20
  - parallelExecution 20
  - path-related setting keys 21
  - PollInterval 20
  - publishMavenStyle 20
  - publishTo 20
  - resolvers 20
  - scalaVersion 20
  - version 20
- setting keys, path-related**
  - javaSource 21
  - scalaSource 21
  - sourceDirectories 21
- Simple Build Tool**. *See* **SBT**
- SMT**
  - tasks 60
- software development** 25

- source** 62, 64
- sourceDirectories** 21
- Spark** 43
- Square.scala** 40
- String** 15
- System.exit** 56

## T

- tag()** method 58
- tagw()** method 58
- task keys**
  - about 18, 21
  - clean 21
  - compile 21
  - console 21
  - doc 21
  - packageBin 21
  - packageSrc 21
  - publish 21
  - publishLocal 21
  - test 21
  - update 21
- task level**
  - scoping at 23
- tasks**
  - about 49
  - configuration-level tasks 49
  - project-level tasks 49
- test** 21
- testing** 64-67
- testOnly** 22
- testQuick** 22
- ThisBuild** 42
- traceLevel** 55
- transitive dependencies** 25

## U

- unmanagedJars** 34
- update** 21

## V

- version management** 25



## Thank you for buying **Getting Started with SBT for Scala**

### **About Packt Publishing**

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

### **About Packt Open Source**

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### **Writing for Packt**

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



## Matplotlib for Python Developers

ISBN: 978-1-847197-90-0

Paperback: 308 pages

Build remarkable publication quality plots the easy way

1. Create high quality 2D plots by using Matplotlib productively
2. Incremental introduction to Matplotlib, from the ground up to advanced levels
3. Embed Matplotlib in GTK+, Qt, and wxWidgets applications as well as web sites to utilize them in Python applications



## Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-849516-86-0

Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

Please check **[www.PacktPub.com](http://www.PacktPub.com)** for information on our titles



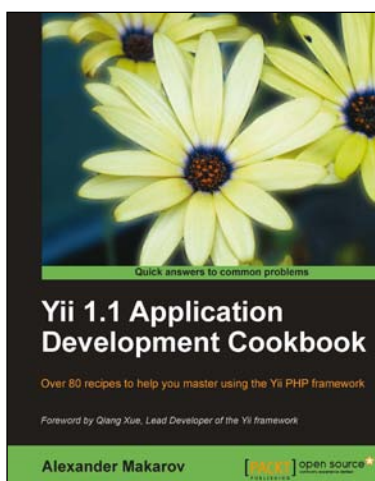
## Responsive Web Design with HTML5 and CSS3

ISBN: 978-1-849693-18-9

Paperback: 324 pages

Learn responsive design using HTML5 and CSS3 to adapt websites to any browser or screen size

1. Everything needed to code websites in HTML5 and CSS3 that are responsive to every device or screen size
2. Learn the main new features of HTML5 and use CSS3's stunning new capabilities including animations, transitions and transformations
3. Real world examples show how to progressively enhance a responsive design while providing fall backs for older browsers



## Yii 1.1 Application Development Cookbook

ISBN: 978-1-849515-48-1

Paperback: 392 pages

Over 80 recipes to help you master using the Yii PHP framework

1. Learn to use Yii more efficiently through plentiful Yii recipes on diverse topics
2. Make the most efficient use of your controller and views and reuse them
3. Automate error tracking and understand the Yii log and stack trace

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles