Community Experience Distilled

Maven Build Customization

Discover the real power of Maven 3 to manage your Java projects more effectively than ever

Lorenzo Anardu
Umberto Antonio Cicero
Giacomo Veneri
Roberto Baldi
Riccardo Giomi

[PACKT] open source*
community experience distilled
PUBLISHING

Maven Build Customization

Discover the real power of Maven 3 to manage your Java projects more effectively than ever

Lorenzo Anardu

Roberto Baldi

Umberto Antonio Cicero

Riccardo Giomi

Giacomo Veneri

[PACKT] PUBLISHING

open source
community experience distilled

BIRMINGHAM - MUMBAI

Maven Build Customization

First published: October 2014

Production reference: 1251014

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

ISBN 978-1-78398-722-1

www.packtpub.com

Credits

Authors

Lorenzo Anardu

Roberto Baldi

Umberto Antonio Cicero

Riccardo Giomi

Giacomo Veneri

Reviewers

Cedric Gatay

Sharafat Ibn Mollah Mosharraf

Rohit Mukherjee

Commissioning Editor

Akram Hussain

Acquisition Editor

Kevin Colaco

Content Development Editor

Susmita Sabat

Technical Editors

Tanvi Bhatt

Siddhi Rane

Copy Editors

Sayanee Mukherjee

Laxmi Subramanian

Project Coordinator

Neha Thakur

Proofreaders

Simran Bhogal

Maria Gould

Ameesha Green

Paul Hindle

Indexers

Monica Ajmera Mehta

Rekha Nair

Priya Sane

Graphics

Valentina D'silva

Abhinash Sahu

Production Coordinators

Adonia Jones

Nilesh R. Mohite

Komal Ramchandani

Cover Work

Kyle Albuquerque

Komal Ramchandani

About the Authors

Lorenzo Anardu graduated in Computer Science from the University of Pisa with a thesis in Parallel Computing. He was born in Sardinia and currently lives in Tuscany where, since 2011, he started working at Autostrade Tech SPA. His main fields of interest are J2EE technology and Android development. He is an expert in optimization and high-performance computing. He has been working with Maven for 5 years, applying it in small and big projects for building and integration purposes. He loves to run.

Roberto Baldi graduated in Information Technology from the University of Florence. He is a senior Java developer with 10 years of experience developing backend and frontend applications. He also has experience as a software analyst, Android developer, and Linux system administrator. He has been working at Softec SPA since 2011. He is interested in programming languages, operating systems, and developer tools. He lives in Pistoia (Italy) with his wife, Chiara, and his son, Alessandro.

Umberto Antonio Cicero is a computer engineer; he received his degree from the University of Calabria. He was born, raised, and educated in Calabria (Italy) and currently lives in Tuscany. He works as a software developer, specialist in web environments and mobile applications. He has been involved in a large number of projects based on Java and Maven. In 2012, he started collaborating with Engineering SPA and started developing apps for Android. He loves Arduino and rock music.

Riccardo Giomi is a senior analyst and developer at Autostrade Tech SPA. He received the Laurea degree in Electronic Engineering in 1996 at the University of Florence. He usually works on large Java EE projects, all managed with Maven. He loves Java programming, mathematics, and playing the piano. He lives in Florence with his girlfriend Barbara.

Giacomo Veneri graduated in Computer Science from the University of Siena. He holds a PhD in neuroscience context with various scientific publications. He is an IEEE member, and is OCA Java 7 certified. He has 15 years of experience as an IT architect and a team leader. He is an expert in computer-assisted diagnosis in a real-time context and is experienced in the automotive, defense, and public sectors. He is currently an active developer and sponsor of various open source projects. He lives in Tuscany where he loves cycling.

About the Reviewers

Cedric Gatay has an engineering degree in Computer Science. He likes well-crafted and unit-tested code. He has a very good understanding of Java languages (giving courses in engineering schools and talking at local Java User Groups).

He has been working with Apache Maven since 2006, and from day one he has been the technical leader of a successful software company, editing a Wicket-based SaaS: SRMvision: `http://www.srmvision.com`.

He is also the founder of a collaborative blog for developers, Bloggure: `http://www.bloggure.info`. He is now a freelance developer member of the Code-Troopers team: `http://www.code-troopers.com`.

Sharafat Ibn Mollah Mosharraf graduated from the University of Dhaka in Computer Science and Engineering. He is currently working as a senior software engineer at Therap Services, LLC. He has expertise and experience in architecting, designing, and developing enterprise applications in Java, PHP, Android, and Objective-C. He loves researching as well as training people on state-of-the-art technologies to design, develop, secure, and maintain web and mobile applications. He also provides coaching for various teams participating in national software development contests. His areas of interest include user experience, application security, application performance, and designing scalable applications. He loves spending his free time with his family and friends.

I'd like to thank the author for writing such a wonderful book on advanced Maven concepts. It had been difficult for me to train people to master this topic due to a lack of detailed and organized resources. I'd also like to thank Neha Thakur, the project coordinator of the book. It was a pleasure working with her. And last, but not least, I thank my wife, Sadaf Ishaq, for bearing with me while I put my busy time reviewing the book. It's always been great to have you by my side!

Rohit Mukherjee is a final year student of computer engineering at the National University of Singapore (NUS). He spent some time in Zurich, Switzerland, studying graduate courses in computer science at ETH Zurich.

He has experience working in financial and healthcare technology, and enjoys working his way through the stack.

I would like to thank my parents and Pratish Mondal for their support.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `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` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

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

`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`, 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

Preface **1**

Chapter 1: Maven and Its Philosophy **7**

Core concepts of Maven **8**

Introduction to the transportation project **9**

Creating the project 11

Structuring the project 13

Summary **17**

Chapter 2: Core Maven Concepts **19**

Build lifecycles **19**

The default lifecycle 20

The clean lifecycle 26

Maven goals **26**

Getting help on plugin goals and parameters 27

Packaging types **28**

JAR 29

WAR 29

POM 30

EJB 30

EAR 31

Built-in lifecycles and default bindings 31

Adding and configuring Maven plugins **33**

The plugin-level configuration 33

The execution-level configuration 34

Managing dependencies **37**

Dependency scopes 38

Dependency version ranges 42

Transitive dependencies and the dependency tree 43

Dependency inheritance 46
The super and the effective POMs 46
Maven settings 50
Properties and resource filtering 51
Maven properties 51
Resource filtering 52
Building EE applications 54
Building WEB applications 54
Building enterprise applications 55
Configuring repositories 60
Enabling releases and snapshots 61
Best practices 62
Aggregate POMs 62
Dependency management 63
Plugin management 65
Summary 66
Chapter 3: Writing Plugins 67
A problem to solve 68
Developing a new plugin 68
Implementing Mojo 76
Testing Mojo 80
Best practices for testing 82
Integration testing 84
maven-plugin-plugin 87
Custom plugin – mantis-maven-plugin 93
Custom plugin implementations 93
Summary 98
Chapter 4: Managing the Code 99
Maven build profiles 99
What is a profile? 100
The structure of a profile 101
Profile activation 102
Sample build profiles 103
Maven Assembly Plugin 105
Fitting to environment 106
Building your own archive through the Assembly plugin 107
The descriptor file 108
The project configuration 109
Maven Site Plugin 114
Creating a simple site 114

Creating your own project site manually 114
Configuring the site for a submodule 120
Reporting the Javadoc 121
Skinning Maven sites 124
Maven site content 126
Summary 128
Chapter 5: Continuous Integration and Delivery with Maven 129
Key concepts of continuous integration and delivery 130
The repository management server 132
Installing Nexus 133
Installing Nexus on a Unix-based OS 133
Installing Nexus on Windows 133
Customizing Nexus 134
Testing the Nexus installation 134
Configuring the Nexus server 134
Testing the Nexus server 134
Managing repositories 135
Configuring official repositories 135
The User Managed Repository 136
Nexus access-level security 138
Integrating Ant 138
Installing Ant 138
Understanding Ant 138
Ant custom tasks 139
Maven-Ant integration 139
Ant-Maven integration 140
SCM integration 141
Maven SCM Plugin 142
Maven Release Plugin 143
Deploying on the remote repository 145
Continuous Integration and Delivery with Hudson or Jenkins 147
Installing Hudson 147
Configuring Hudson 148
Working with Hudson 149
Working with Hudson interactively 151
Maven-Hudson integration to deliver a new artifact 153
Testing software automation 154
Scheduling a test reporting 156
Integration tests 159
Static code analysis tools (FindBugs) 160
Bug fixing 162
A case study with MantisBT 163

A more realistic case – the transportation project **165**
Choosing the component to build 166
Preparing the version of a multimodule component 169
Configuring Hudson 171
Preparing the version of a multimodule with a flat structure (an alternative way) 172
Finalizing the release 174
Summary **175**
Chapter 6: Maven Android **177**
Prerequisites **177**
Creating your own Android application with an archetype **178**
Creating your own Android application **178**
Creating or modifying the AndroidManifest file 179
Defining a simple Maven POM file 181
Description tags 183
Building with Maven plugin goals 184
Declaring dependencies **185**
A compatibility library for API v4 186
The final POM file with dependencies 187
Useful instrumentations to test, sign, and zipalign **189**
The test profile 189
Signing and zipaligning the package 191
The bug detector (Lint) 194
Eclipse integration **196**
Installing the Android connector 196
Mavenized Android Project 196
Summary **197**
Appendix A: Integrating Maven – Gradle **199**
What is Gradle? **199**
How Gradle works **201**
Creating a simple project with Gradle **201**
Gradle's project configuration **202**
Deploying on the Maven repository **204**
Creating the project's POM **206**
Appendix B: Maven Integration for Eclipse **209**
Importing existing Maven projects **210**
Checking out Maven projects from SCM repositories **211**
Building Maven projects **213**
m2e plugin settings **215**
Managing the POM **216**

Managing repository indexes **219**
Managing dependencies and plugins **221**
m2e connectors and lifecycle mapping **222**
Managing Java EE projects **228**
Appendix C: Maven Global Settings **231**
The settings.xml file **231**
Servers 232
Proxies 233
Profiles 233
Appendix D: Maven Short References – Common Commands and Archetypes **235**
Commands **235**
Build 235
Deploy and release 236
Android 237
Miscellaneous 238
Archetypes **239**
Maven variables **240**
The default and clean Maven lifecycle **241**
Index **243**

Preface

As, someone we don't actually remember, said, a preface is something you write after, you put before, and you don't read neither after or before. So we're here trying to explain why someone who is not going to read this preface should read a book about Maven.

If you are looking for a book about Maven, you've probably faced some issues related to the management of a Java project. As a matter of fact, managing medium and big projects often results in problems related to the build, distribution, and documentation, leading to team cooperation and communication issues. In such environments, Maven emerged as one of the *state-of-the-art* tools to manage software projects.

This book will drive you to become a Maven expert. This book will provide you not only with the basic information to manage dependencies, but also with the knowledge to improve project management in your organization, resulting in resources savings and positively impacting the software quality.

The book is intended to provide an advanced treatise about Maven to a range of readers from software analysts to project managers. You will learn both Maven basic usage and how to exploit its advanced functionalities through an example project that will follow you throughout the book.

Finally, the four appendices attached to the book will treat specific topics in a very pragmatic way. You will read about the usage of Maven in Android projects, how to integrate Maven in your IDE or with other tools such as Gradle, and how to set up Maven.

What this book covers

Chapter 1, *Maven and Its Philosophy*, explores Maven's core concepts and describes the structure of the sample project that we will follow through this book.

Chapter 2, *Core Maven Concepts*, teaches you how Maven plugins are tied to the build lifecycles and how to use and configure plugins for various purposes. We will dive into the Maven dependency management system and speak about how to build multimodule projects containing Java WEB and EE applications.

Chapter 3, *Writing Plugins*, tells you how to develop a Maven plugin, since most of the work in Maven is done by plugins. You will also learn how to customize your builds with tasks that are not available in public plugins.

Chapter 4, *Managing the Code*, covers the most common procedures to create code, artifacts, additional documentation, and packages in order to create maintainable code in the production environment, improve software quality, and simplify the build phase.

Chapter 5, *Continuous Integration and Delivery with Maven*, covers how to implement a real Continuous Integration process with Maven and some popular tools such as Hudson, Nexus, and ANT, focusing on the Maven release and deploy process to perform a real releasing pipeline.

Chapter 6, *Maven Android*, shows you how to create an Android project with Maven. You will learn how Maven plugins and profiles make the applicants' planning stages easy, such as creating a simple structure, implementing and running the test, and running and deploying it on your devices.

Appendix A, *Integrating Maven - Gradle*, discusses the integration of Maven with an emerging tool named Gradle. Gradle is a new build automation tool, sponsored by some important IT companies. You will learn how to use a Maven repository from Gradle, and how to create a Project Object Model (POM) using Gradle's functionalities.

Appendix B, *Maven Integration for Eclipse*, teaches you some important tips to exploit Maven's features from Eclipse. As you are aware, working with Maven from Eclipse is one of the most common needs.

Appendix C, *Maven Global Settings*, discusses how to customize the global build environment of Maven: proxies, repositories, and security.

Appendix D, *Maven Short References – Common Commands and Archetypes*, gives you information about the common commands and archetypes. This small chapter resumes the most common Maven commands discovered in this book. You will find a practical quick reference of Maven here.

What you need for this book

The following are the requirements for this book:

- Software:
    - Maven version 3.2.x or later
    - Eclipse Luna or Kepler
    - JDK 7
    - Nexus OSS Sonatype
    - Hudson CI 3.2.0
    - Gradle 2.0
    - ANT
    - Android SDK
- Operating Systems:
    - Any operating system that supports JDK 6+

Please note that the code reported in this book has been tested on CentOS Linux and/or Windows 7.

Who this book is for

This book is intended for developers, project managers, and delivery managers who know a little bit about Maven and Java and want to extend their knowledge on building process automation in order to reduce human error.

It would be helpful to have familiarity with building and releasing best practices.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can add plugins to our project by right-clicking on our `pom.xml` file."

A block of code is set as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <!-- Project coordinates -->
    <groupId>com.mycompany.projects</groupId>
    <artifactId>my-first-maven-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>My First Maven Project!</name>
</project>
```

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

```
<reportSet>
  <reports>
    <report>javadoc</report>
    <report>test-javadoc</report>
  </reports>
</reportSet>
```

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

```
$  mvn site:deploy
```

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: "Open the new project window by navigating to **File** | **New** | **Project…**."

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`, 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`.

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` 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` if you are having a problem with any aspect of the book, and we will dos our best to address it.

1
Maven and Its Philosophy

If we ask software developers what Maven is, the majority will probably respond that Maven is a build tool. We can't say they are wrong, but this definition is not complete. If we want to be more precise, we should say that Maven is a project management tool that provides build and distribution functionalities, code generation, and communication features. Above all, Maven provides an advanced dependency management system that is able to retrieve transitive dependencies and download them from both local and remote repositories.

Maven is built with a plugin-based architecture; the core architecture provides a set of features that can be extended through a set of official or custom plugins downloadable from repositories.

Maven comes with the *convention over configuration* philosophy. The origin of this philosophy resides in the idea that accepting conventions resulting from a set of past experiences leads to advantages such as saving time, reuse, and maintenance simplification. Maven pursues this philosophy through the use of defaults, which means that unnecessary configurations should be avoided; a project should just work.

While the use of defaults is a powerful concept, users might want to customize some behavior. Maven meets the users' needs by allowing the customization of almost all defaults.

In this chapter, we will:

- Introduce Maven and explain its basic concepts
- Present the example project used to show the concepts that we will treat
- Start structuring the project, deepening some of the concepts introduced

Core concepts of Maven

In this paragraph, we will explore some of Maven's core concepts through a simple example. Maven defines project configurations through a **Project Object Model** (**POM**), which is stored in a file named `pom.xml`.

The following example of `pom.xml` defines a simple project. Such a simple POM file is capable of compiling and building the project without the need to specify any additional information:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <!-- Project coordinates -->
    <groupId>com.mycompany.projects</groupId>
    <artifactId>my-first-maven-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>My First Maven Project!</name>
</project>
```

Reading the example POM file, the following concepts emerge:

- **Inheritance**: This concept simply means that everything that is not specified in a POM file is inherited from the upper-level POM. At the top level, Maven provides a parent POM defining all default values, which were mentioned earlier. According to this principle, multimodular projects are often structured with a root POM file defining common settings and a `pom.xml` file related to each submodule to manage each module's peculiarities.
- **Overriding**: This concept derives from the preceding one. All the values defined in the lower levels of the POM hierarchy override the definitions in the upper layers. As we can see in the preceding example POM file, only the project's coordinates have been defined, any other value is inherited from the parent POM. A project is uniquely identified into Maven repositories by its coordinates, which is composed of `groupId`, `artifactId`, and `version`. Project coordinates are fundamental since they allow Maven to correctly manage modules and plugins.

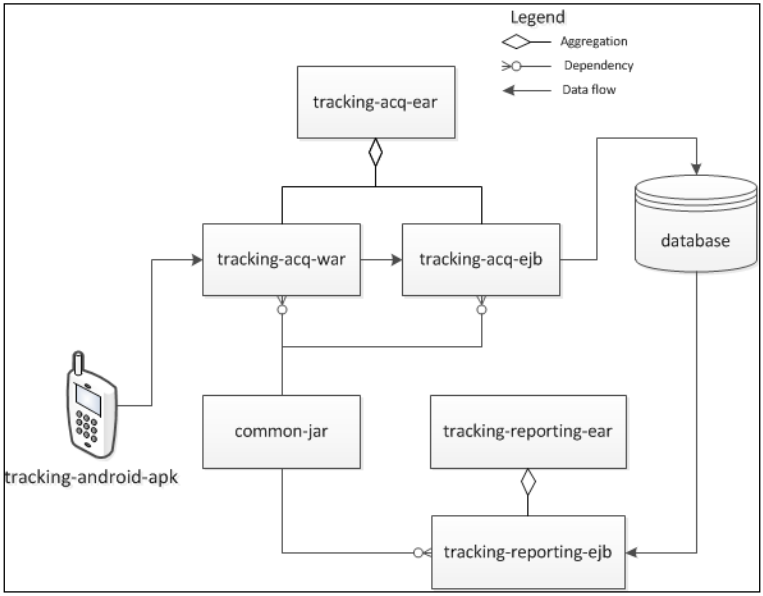
- **Modularity**: When we install Maven, we formally install only its core functionalities. Whenever we need some extra features, we can find it in some plugin. Plugins as well as software dependencies are downloaded from a set of configurable repositories.
- **Repository**: Maven downloads a project's dependencies and plugins through repositories. Maven only distinguishes between two types of repositories: local and remote. The local repository is a folder inside the machine in which a project is being developed, acting as a cache with respect to the remote repositories. For what concerns remote repositories, Apache provides a central repository containing thousands of common dependencies, which is the default one. Maven does not rely on this specific repository, thus allowing users to define their own custom repositories.

Introduction to the transportation project

We briefly explored some of Maven's core concepts. Before we start diving into details, we will introduce the project that will guide us across this book: *transportation project*.

This project aims to develop an application to track vehicles moving around the world and provides an integrated GUI for visualization and statistic calculations.

The example project that we will describe in this book is a complex multimodule J2EE application. Its functional architecture is shown in the following figure:



As we can see in the preceding figure, the project is composed of several modules that interact with each other and store their data in a shared database. All the modules composing the project follow a common naming convention. The first part of the name indicates the project. The second part is a descriptive name indicating the main functionality of the module. The final part indicates the packaging of each module. All the parts composing the name are separated by a dash.

The following list describes what these modules are in charge of:

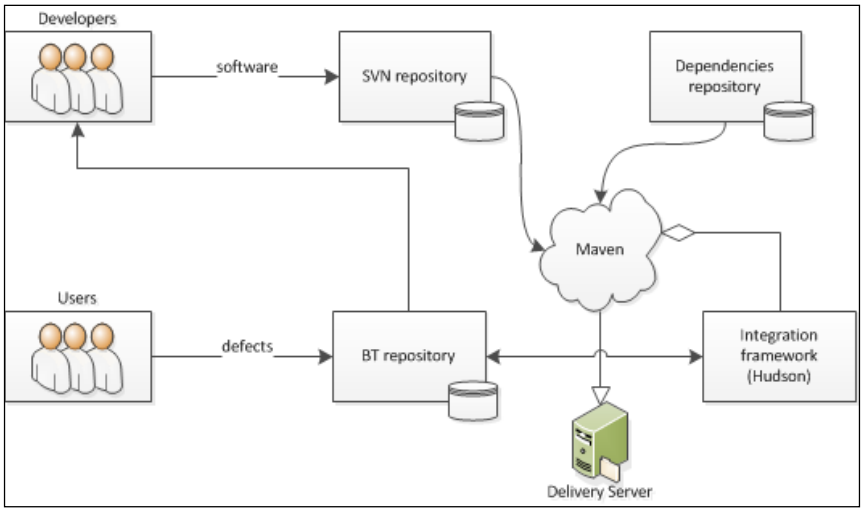
- `transportation-android-apk`: This is an Android application in charge of collecting GNSS coordinates and periodically sending them to the backend.
- `transportation-acq-ear`: This is an archive module containing all the functionalities of the backend interface.
- `transportation-acq-war`: This is a web application module exposing the backend functionalities to an app across the world using the REST technology. This module receives the application requests, validates them, and invokes the `transportation-acq-ejb` functionalities in order to perform its tasks.
- `transportation-acq-ejb`: This is an Enterprise Java Bean containing all the data acquisition APIs. This module is in charge of persisting the collected coordinates into the database.
- `transportation-reporting-ear`: This is an archive module containing all the reporting functionalities.
- `transportation-reporting-war`: This is a web application containing the reporting GUI.
- `transportation-reporting-ejb`: This is an enterprise Java Bean containing all of the business logic related to the statistics visualization.
- `transportation-common-jar`: This is a JAR file containing common utility classes.
- `transportation-statistics-batch-jar`: This is a scheduled standalone application in charge of statistical computations on the collected coordinates.

In order to explain some advanced Maven features, we assume that the project is developed in an integrated environment. This environment consists of several entities, managing different phases of the software's lifecycle:

- Source code repository: We assume that the code is available in an SVN repository, even though the kind of repository is not binding.
- Bug-tracking tool: In order to avoid dependencies from specific products such as MantisBT or Jira, we assume to have a custom database to track bugs.

- Custom-dependencies repository: This is proprietary software stored in the repository.
- Integrated-build and versioning environment: This environment relies on Maven features to perform most of its work. Since, in this case, we must target a specific tool, we will assume to work with Hudson.

In the following figure, we can see the overall picture of the development and build environment:



Life cycle of the software

In spite of the fact that Maven is agnostic with respect to operating systems and IDE, in the course of this book, we assume to develop the software using Eclipse IDE with the M2E-Maven Integration for Eclipse plugin, m2e.

You can download Eclipse from `https://www.eclipse.org/downloads/`.

You can find all details about m2e at `https://www.eclipse.org/m2e/`.

Creating the project

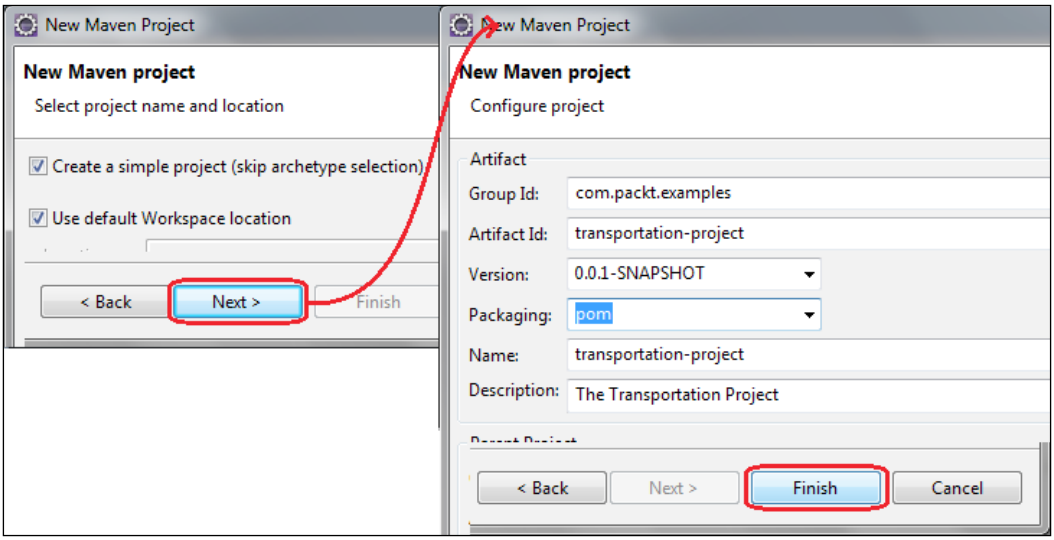
Now that we have a clear view of the project's structure and context, we can start getting our hands dirty.

In Maven, a multimodule project simply consists of a folder containing all submodule projects and a central POM file referencing these modules. This file is usually referred to as parent POM or aggregate POM. In this book, we will comply with this naming.

To start working, we simply need to open our IDE and create the project. Through the m2e plugin, Eclipse provides a wizard specific for Maven project creation.

The project creation starts as any other project. Open the new project window by navigating to **File** | **New** | **Project...**.

When the window opens, select the **Maven Project** option from the **Maven** folder. As we can see in the following screenshot, the last two steps consist of creating a simple project and filling in the form with the project's coordinates and packaging:



We finally have our Maven project. Our `pom.xml` file will look a bit desolate, but it will soon grow up. In the following sections, we will start structuring this POM file:

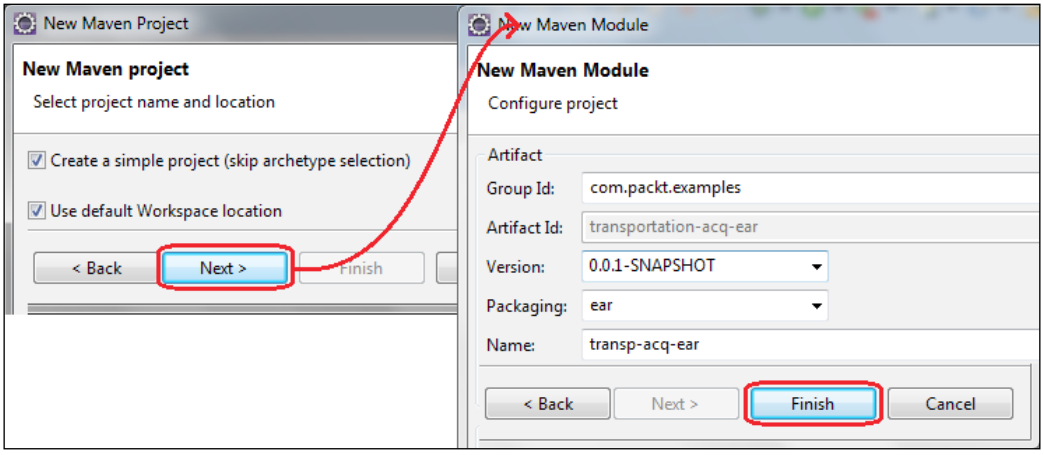
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.examples</groupId>
  <artifactId>transportation-project</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>transportation-project</name>
  <description>The Transportation Project</description>
</project>
```

A project packaging `pom` does not produce any software package; it simply defines a POM file referencing a set of specified modules, and provides the common settings such as repositories, dependencies, and plugins.

This is the case of an aggregate POM, but it is possible to create POM projects with only specific settings; such projects might be used as a dependencies of other projects or modules that inherit the specified settings.

Structuring the project

Actually, we created an empty project. In order to start structuring our project, we will start adding the project modules that we saw in the earlier sections. The easiest way to add a module is to exploit the m2e functionalities. Just right-click on the `pom.xml` file and navigate to **Maven** | **New Maven Module Project** from the context menu. After this, we can fill in the modules, as shown in the following screenshot:

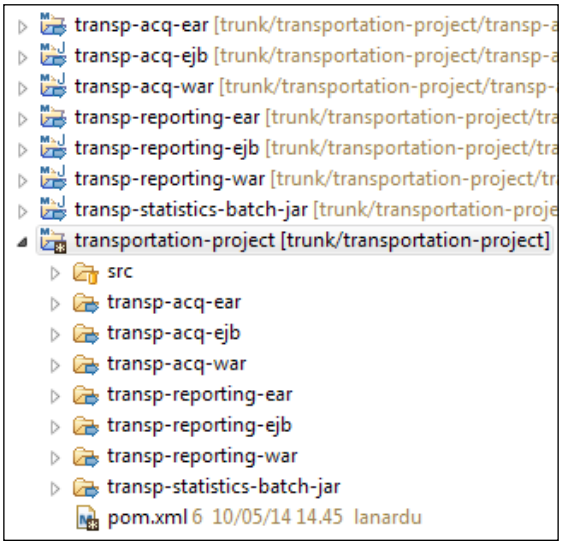


New Maven module creation

Now, our project will have the first module. We can iterate the same operation for the remaining modules, taking care to choose the right packaging for each module.

Remember that EJBs don't have proper packaging. They are often included in JAR packages.

Once we stop adding modules, we will see that m2e created all the submodules. Each module that is stored as a folder into the project has its own `pom.xml` file, which will specify its specific coordinates and settings. In order to distinguish between the POM file of the project and the POM files of its modules, we will call the **aggregate** POM that we created earlier. The following screenshot shows the added modules and submodules:

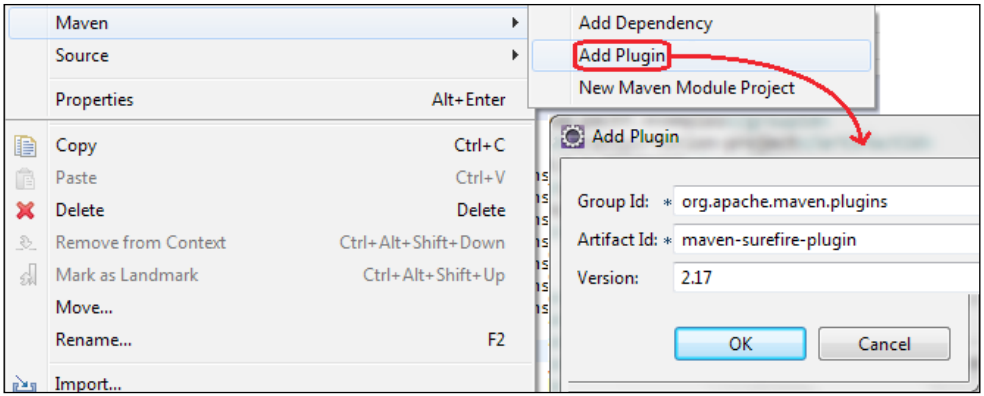


As we might notice, this multimodular project follows the standard that we discussed in the *Core concepts* section.

While terminating the structuration of our project, we take the chance to deepen two more concepts, plugins and dependencies.

As we mentioned earlier, Maven's core functionalities do not cover all needs; most of them are implemented in external plugins. We can add plugins to our project by right-clicking on our `pom.xml` file and navigating to **Maven** | **Add Plugin** from the context menu, as shown in the following screenshot. We can see that the **Version** field is not mandatory; if we don't specify its value, Maven will download the latest version.

In the following example, we add the **maven-surefire-plugin** to transportation project. This plugin is used during the test phase to execute the unit tests of the applications. It supports different unit-test frameworks such as JUnit and TestNG.



Since we want to use JUnit, we can simply add JUnit as a dependency of our project. We can add a new dependency the same way as we added a plugin before.

Of course, it is possible to add more plugins and dependencies by manually editing the POM file of each project. In order to add the Maven compiler plugin in our parent POM, we simply add the following tags as a child of the `<plugins>` tag:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration >
</plugin>
```

After we add all the modules and plugins described earlier, our `pom.xml` file will look like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.examples</groupId>
  <artifactId>transportation-project</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>transportation-project</name>
  <description>The Transportation Project</description>

  <modules>
```

```
    <module>transportation-acq-ear</module>
    <module>transportation-acq-war</module>
    <module>transportation-acq-ejb</module>
    <module>transportation-reporting-ear</module>
    <module>transportation-reporting-war</module>
    <module>transportation-reporting-ejb</module>
    <module>transportation-common-jar</module>
    <module>transportation-statistics-batch-jar</module>
  </modules>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.17</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration >
      </plugin>
    </plugins>
  </build>
</project>
```

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.

We have finally structured our project. Actually, it contains all of its modules, a single plugin, and one dependency.

Before exploring the concepts we introduced in this chapter in more detail, it is important to focus on the best practice described in the following snippet. This practice concerns dependency management and will allow us to avoid some common problems related to this topic.

The aggregate POM should be the only one defining the dependency version. The modules composing the project should use the dependencies in an anonymous way (that is, without specifying their versions).

Archive modules such as EARs should be the only way to physically contain the libraries; all the submodules should consider their dependencies as provided.

Summary

In this chapter, we explored Maven's core concepts and described the structure of the sample project that we will follow across this book.

We also discussed the concepts of dependency and plugins in detail, and explained how to practically manage them.

In the following chapter, we will dive into our project and discuss the advanced use and customization of Maven plugins in detail.

2
Core Maven Concepts

As we saw in the previous chapter, each Maven project is described by an XML configuration file called Project Object Model. What we have yet to see is how Maven will use the information contained in the POM, how we can clean and build our projects, which tasks we can decide to run, and finally, how Maven plugins take part in the build process. In order to answer all these questions, we'll dive into the core concepts of Maven, which are as follows:

- Build lifecycles
- Lifecycle phases and plugin goals
- Packaging types (JAR, WAR, EAR)
- Dependencies and repositories
- Resource filtering

Using all these features, you will learn how to set up and build a complex multimodule Java EE application. All the examples of this chapter refer to a direct usage of the Maven tool from the command line; in *Appendix B*, *Maven Integration for Eclipse* we will show you how to manage a Maven project from Eclipse IDE.

Build lifecycles

A **lifecycle** is a sequence of phases. In each phase, depending on the POM configuration, one or more tasks are executed. These tasks are called **goals**. Despite the enormous variety of work that can be accomplished by Maven, there are only three built-in Maven lifecycles: **default**, **clean**, and **site**.

The default lifecycle

The default lifecycle is responsible for the build process, so it's the most interesting. Among its phases, the most important phases are described in the following table:

Phase	Actions
process-resources	Filter the resource files and copy them in the output directory
compile	Compile the source code
process-test-resources	Filter the test resource files and copy them in the test output directory
test-compile	Compile the test source code
test	Run the unit tests
package	Produce the packaged artifact (JAR, WAR, EAR)
install	Install the package in the local repository so that other projects can use it as a dependency
deploy	Install the package in a remote repository

We'll speak later about local and remote Maven repositories.

When we invoke one phase from the command line, Maven executes all the phases of the lifecycle from the beginning up to the specified phase (included). In fact, one of the most common ways to run Maven is just to use the following syntax:

```
$ mvn <phase>
```

It will run all the portions of the respective lifecycle, ending with this phase.

Let's consider an example. Suppose that the POM file of our `transportation-acq-ejb` module is the following, and it is located in the `/transportation-project/transportation-acq-ejb` directory:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.packt.examples</groupId>
        <artifactId>transportation-project</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <artifactId>transportation-acq-ejb</artifactId>
    <packaging>jar</packaging>
```

```
    <name>transportation-acq-ejb</name>
    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

As we can see in the preceding code, the `transportation-acq-ejb` module's parent is the `transportation-project` parent project. We can add some sample Java classes and interfaces in the `transportation-acq-ejb` project. First, we add an EJB local interface, `MyEjb.java`:

```
package com.packt.samples;

import javax.ejb.Local;

@Local
public interface MyEjb
{
    public int myMethod();
}
```

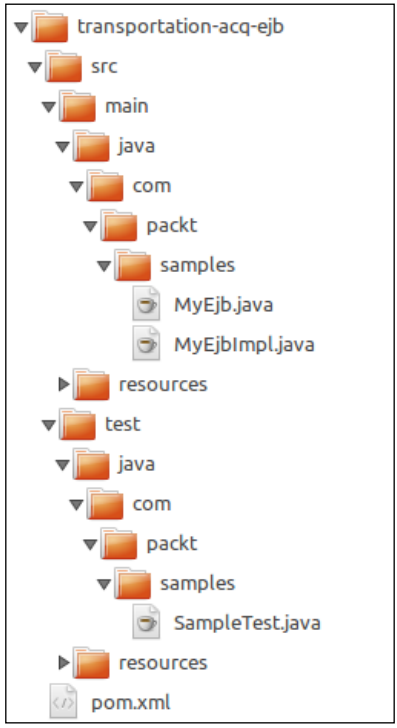
Then, we add a dummy implementation, `MyEjbImpl.java`:

```
package com.packt.samples;

import javax.ejb.Stateless;

@Stateless
public class MyEjbImpl implements MyEjb
{

    @Override
    public int myMethod()
    {
        return 0;
    }
}
```

Finally, we add a unit test class, `SampleTest.java`:

```
package com.packt.samples;

import static org.junit.Assert.*;

import org.junit.Test;

public class SampleTest
{
    @Test
    public void test()
    {
        assertTrue(true);
    }
}
```

The directory structure of the `transportation-acq-ejb` module is as shown in the following screenshot:



In a Maven project, we have to put the project sources under `/src/main/java` and the test sources under `/src/main/test`. These default conventional values can be overridden, as we'll see later in this chapter, but this is not recommended; remember the convention over configuration paradigm!

Execute the following command:

```
$ mvn install
```

We'll see the following output after executing the preceding command:

```
[INFO] Scanning for projects...
[...]
[INFO] ------------------------------------------------------------
[INFO] Building transportation-acq-ejb 0.0.1-SNAPSHOT
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @
transportation-acq-ejb ---
[…]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @
transportation-acq-ejb ---
[INFO] Compiling 2 source files to ~/transportation-project/
transportation-acq-ejb/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-
testResources) @ transportation-acq-ejb ---
[…]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile)
@ transportation-acq-ejb ---
[INFO] Compiling 1 source file to ~/transportation-project/
transportation-acq-ejb/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @
transportation-acq-ejb ---
[INFO] Surefire report directory: ~/transportation-
project/transportation-acq-ejb/target/surefire-repor
ts-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.packt.samples.SampleTest
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 sec
- in com.packt.samples.SampleTest


Results :


Tests run: 1, Failures: 0, Errors: 0, Skipped: 0


[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ transportation-acq-
ejb ---
[INFO] Building jar: ~/transportation-project/transportation-acq-ejb/
target/transportation-acq-ejb-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @
transportation-acq-ejb ---
[INFO] Installing ~/transportation-project/transportation-acq-ejb-0.0.1-
SNAPSHOT.jar to ~/.m2/repository/com/packt/examples/transportation-acq-
ejb/0.0.1-SNAPSHOT/transportation-acq-ejb-0.0.1-SNAPSHOT.jar
[INFO] Installing ~/transportation-project/transportation-acq-ejb/pom.
xml to ~/.m2/repository/com/packt/examples/transportation-acq-ejb/0.0.1-
SNAPSHOT/transportation-acq-ejb-0.0.1-SNAPSHOT.pom
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
```

If we run Maven for the first time, in addition to the preceding output shown, we'll see a lot of other output lines saying that project plugins and dependencies are being downloaded from the Maven central repository.

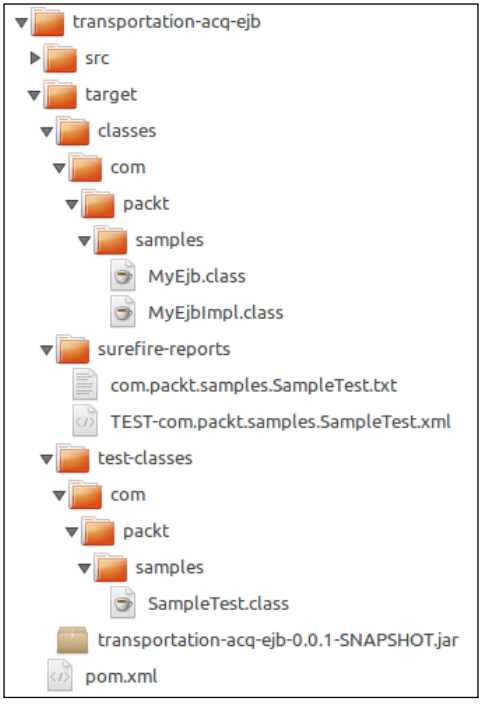
So, as we can see, the sequence of operations performed by Maven follows the steps specified in the default lifecycle. You will also notice that each action performed by Maven is delegated to a certain plugin. In order to compile the project, Maven will download and use the specified dependencies (in this case, Java EE API is needed to compile the EJB classes). Plugins and dependencies are downloaded on-demand, and they are saved in the **local repository**, which is located under the local user home in the `/.m2/repository` subdirectory by default.

For Linux users, the local repository is located under `~/.m2/repository`, where ~ means the user home directory that usually has the `/home/<username>` path.

For Windows users, the local repository is (usually) located under `C:\Users\<username>\.m2\repository`.

Once Maven downloads an artifact or a plugin, it will reuse its stored copy and never search the same version of this artifact or plugin in the Maven central repository or in other remote repositories that can be specified in our POM file again. The only exception to this rule regarding the snapshot versions is that if the version of a dependency or plugin is marked with the `-SNAPSHOT` suffix, this version is currently on development. For this reason, Maven will periodically attempt to download this artifact from all the remote repositories that have snapshots enabled in their configurations (refer the *Configuring repositories* section in this chapter).

If we look in the `/target` directory, we'll see all the work done by Maven; in this case, the compiled classes, unit test reports, and packaged artifact `transportation-acq-ejb-0.0.1-SNAPSHOT.jar`:



Build output of the transportation-acq-ejb module

Note that if instead of running the previous command, we run the `mvn package` command, the lifecycle execution will stop with the `package` phase and the artifact will not be installed in the local repository. This can be a problem if the artifact is needed by other projects as a dependency.

The clean lifecycle

The clean lifecycle is responsible for cleaning the build output. Its phases are as follows:

- Preclean
- Clean
- Postclean

If we run the `mvn clean` command, the target directory will be deleted, but not the artifact installed in the local repository:

```
$ mvn clean
```

Now, if we build the project again, we'll see that no downloads will take place.

Maven can also be used to generate project documentations in various formats and reports about the project. This is achieved through the **site lifecycle**. We'll discuss these features in *Chapter 4*, *Managing the Code*.

Maven goals

Now that we explored the concepts of lifecycle and phase, we have to answer questions such as what is executed in each phase and how we can customize the build process in order to accomplish the desired results. To answer these questions, we need to speak of goals.

A **goal** is a task contained in a Maven plugin. It can be invoked by directly running the following command:

```
$ mvn <plugin-prefix>:<goal-name>
```

A plugin prefix is a shortcut that allows us to refer to the plugin without having to specify its Maven coordinates `groupId`, `artifactId`, and `version`. We'll speak about this in the next chapter.

For example, from the `/transportation-project/transportation-acq-ejb` directory, we can run the following command:

```
$ mvn compiler:compile
[INFO] Scanning for projects...
[...]
[INFO] ------------------------------------------------------------------
[INFO] Building transportation-acq-ejb 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-cli) @
transportation-acq-ejb ---
[INFO] Compiling 2 source files to ~/transportation-project/
transportation-acq-ejb/target/classes
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

We can see that, in this case, Maven just compiles the Java sources. So, we can invoke the Maven executable by specifying a phase, a goal, or both. In fact, if we ask for help on the command line, we obtain the following output:

```
$ mvn -h
usage: mvn [options] [<goal(s)>] [<phase(s)>]
```

Getting help on plugin goals and parameters

We can list the available goals of a certain plugin through the Maven Help Plugin. For example, we can type on the command line:

```
$ mvn help:describe -Dplugin=compiler
[...]
Name: Maven Compiler Plugin
```

```
Description: The Compiler Plugin is used to compile the sources of your
project.
Group Id: org.apache.maven.plugins
Artifact Id: maven-compiler-plugin
Version: 3.1
Goal Prefix: compiler

This plugin has 3 goals:

compiler:compile
  Description: Compiles application sources

compiler:help
  Description: Display help information on maven-compiler-plugin.
  Call mvn compiler:help -Ddetail=true -Dgoal=<goal-name> to display
  parameter details.

compiler:testCompile
  Description: Compiles application test sources.

For more information, run 'mvn help:describe [...] -Ddetail'
```

With the `-Ddetail` parameter, we'll get information about the available parameters that can be specified through the `-D<parameter name>` syntax in case of direct invocation of the plugin goals. We can also try the following command:

```
$ mvn help:describe -Dplugin=help
```

This way, we'll obtain help on the Maven Help Plugin!

Packaging types

It's time to introduce one of the most important Maven concepts: how plugin goals are tied to lifecycle phases. This happens through **packaging types**. The packaging type is specified in the `pom.xml` descriptor through the `<packaging>` element, usually after its Maven coordinates. The default packaging type is `jar`. The plugin goals that are executed by default in each phase of the lifecycle depend on the packaging type of the project that we will build. This is because we need to execute different tasks for different packaging types. Let's see some details about the most common packaging types and their default bindings.

JAR

This is the default packaging type. It produces an archive in the JAR format. Its default bindings in the default lifecycle are shown in the following table. The plugin goal is expressed in the `<plugin-prefix>:<goal-name>` form.

Lifecycle phase	Plugin goal
process-resources	`resources:resources`
compile	`compiler:compile`
process-test-resources	`resources:testResources`
test-compile	`compiler:testCompile`
test	`surefire:test`
package	`jar:jar`
install	`install:install`
deploy	`deploy:deploy`

We can see that the plugin goals are the same as those we encountered when we built our sample module, `transportation-acq-ejb`.

WAR

The WAR packaging type binds the `war` goal of `maven-war-plugin` to the package phase. This goal creates a web application archive using the JSP pages and XML descriptors under `/src/main/webapp`, the compiled classes of the project, and all the JAR dependencies that have the `compile` or `runtime` scope. We'll speak later about dependencies and dependency scopes. The default bindings of the WAR packaging type are shown in the following table:

Lifecycle phase	Plugin goal
process-resources	`resources:resources`
compile	`compiler:compile`
process-test-resources	`resources:testResources`
test-compile	`compiler:testCompile`
test	`surefire:test`
package	`war:war`
install	`install:install`
deploy	`deploy:deploy`

POM

This is the packaging type of project parent POM files (such as the `transportation-project` parent POM of our sample) or aggregate POM files (that we'll discuss later in this chapter). Here, there aren't source files to process or compile; we only need to install and deploy the POM, along with its site descriptor, if present. Here are the default bindings of the POM packaging type:

Lifecycle phase	Plugin goal
package	`site:attach-descriptor`
install	`install:install`
deploy	`deploy:deploy`

EJB

This packaging type differs from the JAR packaging type in the `package` phase, in which the `ejb:ejb` goal of `maven-ejb-plugin` is used in place of the `jar:jar` goal of `maven-jar-plugin`. The `ejb:ejb` goal behaves like the `jar:jar` goal, but in addition, it checks for the presence of the `ejb-jar.xml` descriptor when the `ejbVersion` configuration parameter of the plugin is set to `2.x` (the default). If this descriptor is missing, an error is thrown. As the `ejb-jar.xml` descriptor is not needed when we use EJB version 3.x, we should explicitly configure `maven-ejb-plugin`, specifying EJB version 3.x. This way, we will obtain the exact same result as when using the `jar:jar` plugin goal, and ultimately, the JAR packaging type. For this reason, we used the JAR instead of the EJB packaging type in the EJB modules of our sample project. The following table shows the default bindings of the EJB packaging type:

Lifecycle phase	Plugin goal
process-resources	`resources:resources`
compile	`compiler:compile`
process-test-resources	`resources:testResources`
test-compile	`compiler:testCompile`
test	`surefire:test`
package	`ejb:ejb`
install	`install:install`
deploy	`deploy:deploy`

For more information on `maven-ejb-plugin`, we can ask for help from Maven using the Maven Help Plugin:

```
$ mvn help:describe -Dplugin=ejb -Ddetail
```

We can also get help on the official Maven site:

`http://maven.apache.org/plugins/maven-ejb-plugin/`

EAR

The EAR packaging type binds `maven-ear-plugin` to the `generate-resources` and `package` phases of the default lifecycle. It generates the `application.xml` descriptor and Java EE Enterprise Archive. Its default bindings are shown in the following table:

Lifecycle phase	Plugin goal
generate-resources	`ear:generate-application-xml`
process-resources	`resources:resources`
package	`ear:ear`
install	`install:install`
deploy	`deploy:deploy`

The `generate-resources` phase is not bound to any plugin goal for the packaging types that we saw before. In the next section, we'll show the complete list of the default lifecycle phases.

Built-in lifecycles and default bindings

We can find a complete reference for the built-in lifecycles on the Maven site, `http://maven.apache.org`, navigating to **Documentation** | **Introduction** | **The Build Lifecycle**. We can wonder where these lifecycles and default bindings are actually defined. All these definitions are in the Maven core library, `maven-core-<Maven version>.jar`, under the `/lib` subdirectory of the Maven installation. For example, in `maven-core-3.2.1.jar`, under the `META-INF/plexus` folder, we can find the `components.xml` and `default-bindings.xml` descriptors. These two descriptors contain the lifecycle definitions and their default bindings, respectively. Looking at the `components.xml` descriptor, we can see the following elements:

```
<component>
    <role>org.apache.maven.lifecycle.Lifecycle</role>
```

```
    <implementation>org.apache.maven.lifecycle.Lifecycle</
implementation>
    <role-hint>default</role-hint>
   <configuration>
        <id>default</id>
        <phases>
            <phase>validate</phase>
            <phase>initialize</phase>
            <phase>generate-sources</phase>
            <phase>process-sources</phase>
            <phase>generate-resources</phase>
            <phase>process-resources</phase>
            <phase>compile</phase>
            <phase>process-classes</phase>
            <phase>generate-test-sources</phase>
            <phase>process-test-sources</phase>
            <phase>generate-test-resources</phase>
            <phase>process-test-resources</phase>
            <phase>test-compile</phase>
            <phase>process-test-classes</phase>
            <phase>test</phase>
            <phase>prepare-package</phase>
            <phase>package</phase>
            <phase>pre-integration-test</phase>
            <phase>integration-test</phase>
            <phase>post-integration-test</phase>
            <phase>verify</phase>
            <phase>install</phase>
            <phase>deploy</phase>
        </phases>
    </configuration>
</component>
```

These are all the phases of the default lifecycle.

We will never need to inspect the Maven core jars! These details are documented in the Maven site, and there are other ways to discover the defaults of our projects. We'll see them later in this book.

Adding and configuring Maven plugins

Now that we explored the core concepts of Maven, we know that all the work is done by Maven plugins. We can say that there are no exceptions to this rule. Till now, we saw some core and packaging plugins such as the Maven Compiler Plugin, the Maven Install Plugin, the and the Maven JAR Plugin. We also learned how to explore their goals and properties using the Maven Help Plugin. What we have to know is how to customize the behavior of the plugins that are already bound by default to the build lifecycle and how to fill the lifecycle with the other required plugin goals.

The plugin-level configuration

If we need to configure a plugin, we can specify some common configuration parameters that will be used for all the invocations of the plugin within our project. This means that such parameters will be used both when we invoke a plugin goal directly from the command line (in the project directory) and when the plugin is invoked during a phase of the build lifecycle. We can achieve this putting a `<configuration>` element into the `<plugin>` element related to our plugin in the project POM. An abstract example is the following:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>...</groupId>
        <artifactId>...</artifactId>
        <version>...</version>
        <configuration>
          <param1>value1</param1>
          <param2>value2</param2>
          [...]
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

We have to remark on this:

- Even if the plugin is bound (by default) to the lifecycle, we need to declare it explicitly in the POM file if we want to set its configuration parameters with values that differ from their defaults.
- The configuration elements that we can specify are the same that we can see when invoking the Maven Help Plugin with the `-Ddetail` option. Using the Maven Help Plugin, we can also see the default values for all the plugin parameters.

If we look at the parent POM of our sample project introduced in the previous chapter, we will see that we set the source and target parameters to 1.7 in order to compile with JDK 1.7:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.5.1</version>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
</plugin>
```

If we use the version 2.5.1 of the Maven Compiler Plugin the default values for both these parameters are 1.5, so we need to change them.

The execution-level configuration

A plugin can be bound to more than one phase of the lifecycle. We can execute multiple goals or the same goal more than once in the same phase or in different phases. To obtain this, we can use multiple `<execution>` elements, each containing the `<configuration>` element to be considered for the execution. In the same manner, when a plugin is not bound by default to the build lifecycle, we have to specify an `<execution>` element with its configuration. The plugin-level and execution-level configuration can coexist, in which case, the execution-level configuration settings will override the plugin-level settings.

Let's consider an example. Suppose we have to generate the JAXB beans from a given XSD schema and we want to put them into the `transportation-common-jar` module of our sample project. We can use `jaxb2-maven-plugin` and bind it to the `generate-sources` phase of the lifecycle, as follows:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
```

```
      <artifactId>jaxb2-maven-plugin</artifactId>
      <version>1.6</version>
      <executions>
        <execution>
          <id>myExecution</id>
          <goals>
            <goal>xjc</goal>
          </goals>
          <configuration>
            <schemaDirectory>
                  src/main/resources/schema/
                  </schemaDirectory>
              <bindingDirectory>src/main/resources/xjb</bindingDirectory>
        <arguments>-extension</arguments>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

If we put one or more XSD files in the schema directory and we build the project, we'll see the following output:

```
$ mvn install
[...]
[INFO] --- jaxb2-maven-plugin:1.6:xjc (myExecution) @ transportation-
common-jar ---
[INFO] Generating source...
[...]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @
transportation-common-jar ---
[...]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @
transportation-common-jar ---
[...]
```

We'll notice that the JAXB beans have been generated in the `/target/generated-sources/jaxb` directory, which is the default value for the `outputDirectory` configuration parameter of the JAXB-2 Maven Plugin. Also, `schemaDirectory` and `bindingDirectory` have default values, but in this case, they have been overridden in the execution-level configuration.

For a complete description of this plugin, we can run the following command:

```
$ mvn help:describe -Dplugin=jaxb2 -Ddetail
```

We can also read the plugin documentation at the following URL:

`http://mojo.codehaus.org/jaxb2-maven-plugin/`

The compiler plugin is able to compile the generated sources in addition to those in `/src/main/java`.

We have to remark on this:

- The `<execution>` element and goal specification are needed. If they are missing, no goals will be executed. This is because no goals of `jaxb2-maven-plugin` are bound by default to the default lifecycle.
- We can specify a `<phase>generate-sources</phase>` child element of the `<execution>` element, but in this case, it is not needed because the binding of the `xjc` goal to the `generate-sources` phase is a default setting for `jaxb2-maven-plugin` and is defined within the plugin itself. We can discover the default phase for a plugin goal using the Maven Help Plugin, as suggested earlier.

The execution ID, which is the `<id>` child element of the `<execution>` element, is not mandatory. If it misses a value, `default` will be used. When we need to configure plugins that are already bound to the Maven lifecycle (for example, `compiler-maven-plugin` or `ear-maven-plugin`), we should know that each plugin goal invoked by the build process will have the `default-<goalName>` execution ID assigned to it. For example, `maven-compiler-plugin` is executed twice during the default lifecycle: during the `compile` phase, the `compile` goal is executed with the `default-compile` execution ID; during the `test-compile` phase, the `testCompile` goal is executed with the `default-testCompile` execution ID. This way, we'll be able to configure the two executions independently. We can verify this behavior looking at the Maven output of the previous examples. In the case of direct invocation of a plugin goal, the execution ID will always be `default-cli`. Let's see an example about the configuration of a plugin that is invoked directly: suppose we don't want to bind the JAXB-2 Maven Plugin to the `generate-sources` phase, and we want to invoke this plugin directly (and only once) to generate the JAXB beans under the `/src/main/java` source folder. All we have to do is modify the plugin configuration as follows:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
```

```
      <artifactId>jaxb2-maven-plugin</artifactId>
      <version>1.6</version>
      <executions>
        <execution>
          <id>default-cli</id>
          <configuration>
            <schemaDirectory>src/main/resources/schema/</schemaDirectory>
            <bindingDirectory>src/main/resources/xjb</bindingDirectory>
        <outputDirectory>src/main/java</outputDirectory>
            <arguments>-extension</arguments>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

This way, the `xjc` plugin goal will not be bound to the `generate-sources` phase because no goals are specified. If we want to generate the JAXB beans, we have to use the following command, and the configuration of the default-cli execution ID will be used:

```
$ mvn jaxb2:xjc
```

We need to notice that we cannot have multiple executions with different configurations for direct invocation because only the `default-cli` execution ID is available.

We can find more examples about this on the Maven site at the following URL:

`http://maven.apache.org/guides/mini/guide-default-execution-ids.html`

Managing dependencies

When we build a project, we usually need external libraries and archives of third parties, or those developed by us in other projects. These are called project dependencies. One Maven project will have other Maven projects as dependencies, and it will refer to them through their `groupId`, `artifactId`, and `version` Maven coordinates. When we declare a dependency in a project, this is first searched in the local repository, then in the Maven central repository and other remote repositories, if specified in the POM. When the dependency is found, it is downloaded and stored in the local repository for future reuse. As we are about to see, project dependencies can be available to the build process in different ways, depending on various attributes that we can specify when we declare them.

Dependency scopes

When we declare a dependency, we can specify a dependency **scope**. The scope indicates the classpaths in which the dependency will be included. There are five dependency scopes, and they are summarized in the following table:

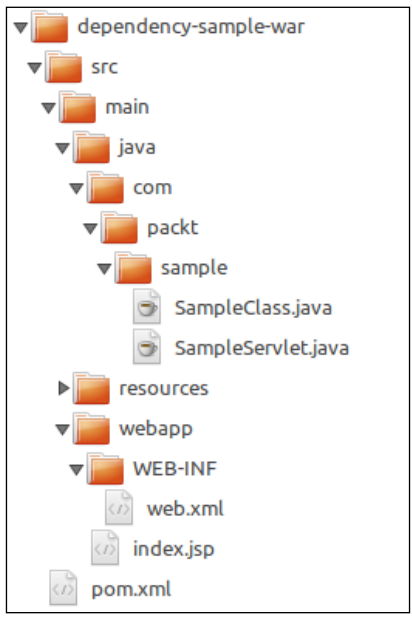
Scope	Description
compile	This is the default scope. Dependencies at compile scope will be available in all the classpaths with which Maven deals; they are used to compile and test our project, and they are packaged in WAR and EAR archives.
provided	Dependencies at this scope are available only during the `compile`, `test-compile`, and `test` phases. They are *not* packaged in WAR and EAR archives.
runtime	Runtime dependencies are used during the `test` phase and packaged in WAR and EAR archives. They are included in the runtime classpath of WEB and EE applications, but are *not* used to compile our project and its unit tests. We should use this scope if we need these dependencies just to run our project and its unit tests.
test	These dependencies are available only in the `test-compile` and `test` phases to compile and run the unit tests.
system	This scope is not recommended. It is similar to the `provided` scope, but we have to specify the full path of the artifact using the `<systemPath>` child element of the `<dependency>` element. Dependencies at this scope will not be searched in Maven repositories.

Let's see an example of a simple web application. Its POM file is as follows:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt.samples</groupId>
    <artifactId>dependency-sample-war</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
```

```
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
    <!— Default scope (compile) -->
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.16</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

The directory structure of the project is shown in the following screenshot:



The `SampleClass.java` code is as follows:

```
package com.packt.sample;

import org.apache.log4j.Logger;

public class SampleClass
{
    private static Logger log = Logger.getLogger(SampleClass.class);

    public void logMessage(String msg)
    {
        log.info(msg);
    }
}
```

If we try to build the project, we'll get the following error:

```
[...]
[INFO] Compiling 2 source files to ~\dependency-sample-war\target\classes
[INFO] -------------------------------------------------------------
[ERROR] COMPILATION ERROR :
[INFO] -------------------------------------------------------------
[ERROR] ~\dependency-sample-war\src\main\java\com\packt\sample\
SampleClass.java:[3,23] package org.apache.log4j does not exist
[...]
```

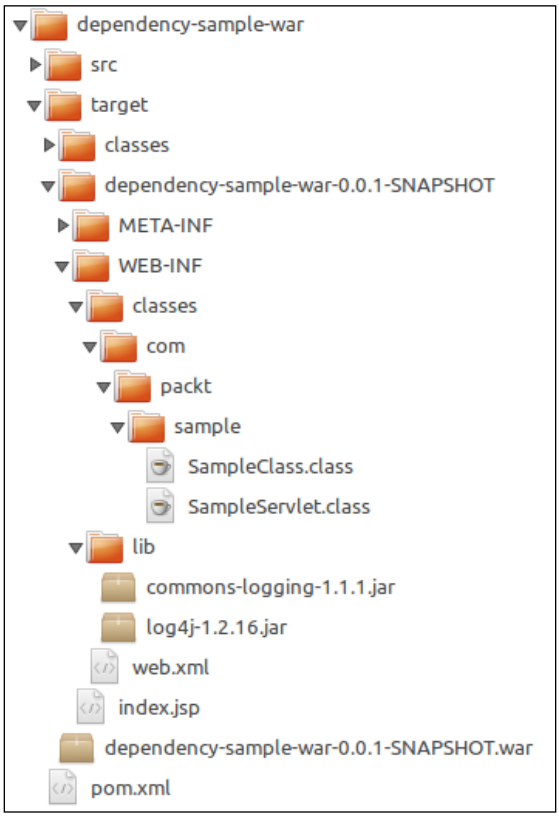
This is because the `log4j` dependency is not available at compile time. We will get a similar error if we use the JUnit API in a source under `src/main/java` rather than under `src/test/java` because the scope of the `junit` dependency is `test`.

If we change our `SampleClass.java` file as follows, then the build is successful:

```
package com.packt.sample;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class SampleClass
{
    private static Log log = LogFactory.getLog(SampleClass.class);

    public void logMessage(String msg)
```

```
        {
            log.info(msg);
        }
    }
```

Now, the build process will succeed; both the `SampleClass.java` and `SampleServlet.java` classes (the latter needs the servlet API contained in the `javaee-web-api` dependency) are compiled, and we'll see the output in the `target` folder, as shown in the following screenshot:



We can see the compiled classes under `target/classes`, the exploded WEB application, and the WAR archive. The `WEB-INF/lib` directory contains both the `compile` and `runtime` dependencies; it does not contain the `provided` and `test` dependencies.

Dependency version ranges

Instead of specifying a certain version number for a dependency, we can also specify a range of versions. The syntax to be used is the following:

- The `(<from version>,<to version>)` syntax specifies an excluding range
- The `[<from version>,<to version>]` syntax specifies an including range
- We can use the mixed forms `(,]` and `[,)`
- The version numbers before and after the comma are optional
- We can specify multiple ranges, which are separated by commas

Some examples are summarized in the following table:

Range	Meaning
(1.0, 1.7)	Any version between 1.0 and 1.7, both excluded
[1.0, 1.7]	Any version between 1.0 and 1.7, both included
[1.0, 2.0)	Any version; 1.0 included and 2.0 excluded
(1.0, 1.9]	Any version; 1.0 excluded and 1.9 included
[1.0]	Strictly 1.0, no other version will be accepted
(, 2.0)	Versions up to 2.0 excluded
[, 2.0)	Versions up to 2.0 excluded
(1.0,)	Versions greater than 1.0 (excluded)
(1.0,]	Versions greater than 1.0 (excluded)
(1.0, 1.9], [2.1, 3.0)	Any version in the specified ranges

We might wonder which version will be chosen by Maven when a range of versions is specified. We have to keep in mind that when we declare a dependency version (and not a range of versions), we simply give a suggestion about what version Maven should prefer. On the other hand, when we declare a version range, we tell Maven that we can't accept version numbers that are out of the specified range. Maven will use this kind of information to resolve conflicts with other declarations of the same dependency within the same build process. This can happen because of the transitive dependency mechanism or the dependency inheritance, which we'll see in the following sections. When two or more conflicting ranges are specified for the same dependency, the build process exits with an error.

Transitive dependencies and the dependency tree

When we have a project A that declares project B among its dependencies, and project B in turn depends on project C, then project A will also depend on project C. This is assured by the Maven dependency mechanism. In other words, we don't need to declare the dependency on project C in project A because project C is a **transitive dependency** of project A. This leads to great advantages in project dependency management because it permits you to use a certain dependency out of the box without caring whether it requires other artifacts, which are included automatically among the overall project dependencies.

Transitive dependency management depends on the scopes of the direct dependency (the project B of our sample) and the transitive dependency (the project C), as follows:

- If the scope of the transitive dependency (project C) is `compile`, then its scope in our project A will be the same as of the direct dependency (project B).
- If the scope of the transitive dependency is `test`, then it will not be a dependency of our project.
- If the scope of the transitive dependency is `provided`, then it will be a `provided` dependency of our project only if the scope of the direct dependency is also `provided`. In all other cases, it will not affect our project.
- Finally, if the scope of the transitive dependency is `runtime`, it will be a runtime dependency of our project if the direct dependency is `compile`; otherwise, its scope will be the same as that of the direct dependency.

This behavior is summarized in the following table. The intersection of the direct and transitive scopes will give the scope that will be assigned to the transitive dependency in our project.

	TRANSITIVE SCOPE (C)			
DIRECT SCOPE (B)	**compile**	**provided**	**runtime**	**test**
compile	compile	-	runtime	-
provided	provided	provided	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

This default behavior can be overridden in two different ways:

- We can specify the exclusion of a transitive dependency in the direct dependency declaration.
- We can declare a dependency with the `<optional>` attribute set to `true`, and it will not be considered as a transitive dependency of projects that depend on our project.

To take control of the dependencies of our project, know what the effective dependencies are, and from which other dependencies they come from, we can invoke the `dependency:tree` goal of the Maven Dependency Plugin. Let's take the sample `dependency-sample-war` and add the JAXB dependencies to the project, as follows:

```
[...]
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.1</version>
</dependency>

<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.1</version>
</dependency>
[...]
```

Now, if we invoke the Maven Dependency Plugin, we'll obtain the following result:

```
$ mvn dependency:tree
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ dependency-
sample-war ---
[INFO] com.packt.samples:dependency-sample-war:war:0.0.1-SNAPSHOT
[INFO] +- javax:javaee-web-api:jar:6.0:provided
[INFO] +- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- log4j:log4j:jar:1.2.16:runtime
[INFO] +- junit:junit:jar:4.8.1:test
[INFO] +- javax.xml.bind:jaxb-api:jar:2.1:compile
```

```
[INFO] |  +- javax.xml.stream:stax-api:jar:1.0-2:compile
[INFO] |  \- javax.activation:activation:jar:1.1:compile
[INFO] \- com.sun.xml.bind:jaxb-impl:jar:2.1:compile
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
```

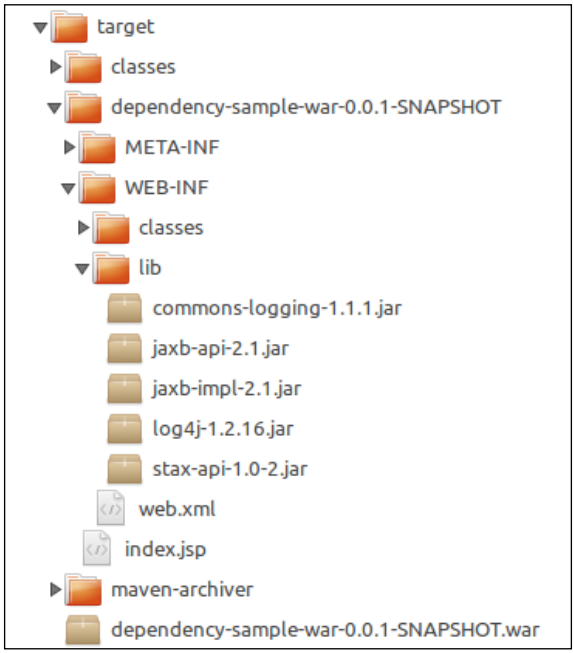
We can see that our project acquired two other dependencies, which are the `stax-api` version `1.0-2` and `activation` version `1.1`. Both these artifacts come from the `jaxb-api` dependency. Just to give an example, if we don't need the activation library in our project, we can exclude it as follows:

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <exclusions>
        <exclusion>
            <groupId>javax.activation</groupId>
            <artifactId>activation</artifactId>
        </exclusion>
    </exclusions>
    <version>2.1</version>
</dependency>
```

As we can see, in the exclusion element, only `groupId` and `artifactId` (and not `version`) of the transitive dependency have to be specified. This way, the dependency tree becomes the same as is shown:

```
$ mvn dependency:tree
[...]
[INFO] com.packt.samples:dependency-sample-war:war:0.0.1-SNAPSHOT
[INFO] +- javax:javaee-web-api:jar:6.0:provided
[INFO] +- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- log4j:log4j:jar:1.2.16:runtime
[INFO] +- junit:junit:jar:4.8.1:test
[INFO] +- javax.xml.bind:jaxb-api:jar:2.1:compile
[INFO] |  \- javax.xml.stream:stax-api:jar:1.0-2:compile
[INFO] \- com.sun.xml.bind:jaxb-impl:jar:2.1:compile
```

Our exploded WAR archive will have the structure shown in the following screenshot:



Dependency inheritance

We have to remember that all the Maven projects inherit everything from their parent POMs. Dependencies are not exceptions to this rule; if the parent of our POM declares some dependencies, our project will inherit these dependencies at the same scope they have in the parent project. For example, in our `transportation-project` POM, we declare the `junit` dependency with the `test` scope, so we don't need to declare it again in all the modules of our projects because they inherit this dependency by their parent. Of course, the `dependency:tree` plugin goal will display both the inherited as well as the transitive dependencies.

The super and the effective POMs

Even when a Maven POM does not refer to a parent project, it inherits implicitly from a parent POM that is embedded in the Maven core libraries. This parent POM is called the **super POM**. In Version 3.2.1 of Maven, the super POM is located in the `maven-model-builder-3.2.1.jar` archive under the `/lib` folder of the Maven installation directory. This JAR and the other core JARs in the same directory are not downloaded from remote repositories.

Browsing the `model-builder-3.2.1.jar` archive, we can find a `pom-4.0.0.xml` file under the `org.apache.maven.model` package, which is the super POM. This POM basically contains the definitions of the sources, resources, test sources, test resources, and output directories, and the declaration of the Maven central repository (but no project-default dependencies). Thanks to the super POM, Maven expects to find Java sources under `/src/main/java`, builds the project output in the `/target` directory, and searches for dependencies in the Maven central repository at `http://repo.maven.apache.org/maven2`. Remember the concept of convention over configuration!

We can be interested in the result of merging our project POM with its ancestors up to the super POM. This is provided by the `help:effective-pom` plugin goal. If we invoke this goal for our sample project, `dependency-sample-war`, we'll obtain the following result:

```
$ mvn help:effective-pom
[...]
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.samples</groupId>
  <artifactId>dependency-sample-war</artifactId>
  [...]
  <dependencies>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-web-api</artifactId>
      <version>6.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
      <scope>compile</scope>
    </dependency>
    [...]
  </dependencies>
  <repositories>
```

```
    <repository>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Central Repository</name>
      <url>http://repo.maven.apache.org/maven2</url>
    </repository>
  </repositories>
  [...]
  <build>
    <sourceDirectory>~\dependency-sample-war\src\main\java</
sourceDirectory>
    <scriptSourceDirectory>~\dependency-sample-war\src\main\scripts</
scriptSourceDirectory>
    <testSourceDirectory>~\dependency-sample-war\src\test\java</
testSourceDirectory>
    <outputDirectory>~\dependency-sample-war\target\classes</
outputDirectory>
    [...]
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>2.5</version>
        <executions>
          <execution>
            <id>default-clean</id>
            <phase>clean</phase>
            <goals>
              <goal>clean</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

```
      [...]
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <executions>
          <execution>
            <id>default-testCompile</id>
            <phase>test-compile</phase>
            <goals>
              <goal>testCompile</goal>
            </goals>
          </execution>
          <execution>
            <id>default-compile</id>
            <phase>compile</phase>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  <reporting>
    <outputDirectory>~\dependency-sample-war\target\site</
outputDirectory>
  </reporting>
</project>
```

As we can see, our project POM is merged with the super POM and with the built-in lifecycle default bindings. For example, we can see the bindings of the compiler plugin with the `compile` and `test-compile` phases, even if these bindings aren't declared in any of the module's POMs or the super POM. Notice that transitive dependencies are not merged—to see them, we have to invoke the `dependency:tree` goal.

Maven settings

The `/conf` folder of the Maven installation directory contains a `settings.xml` file that can be edited to customize some configuration properties used during our builds. This file is also referred to as the **Maven Global Settings** file. We can override these settings in a `settings.xml` file that we can create in the `~/.m2/` folder. While the Global Settings file is used by all the users of the same machine, the file under `~/.m2/` is used only by the local user, and it is called the **Maven Local Settings** file.

Remember that by ~ we mean the user home, which is usually located under `/home/<username>` for Linux users and under `C:\Users\<username>` for Windows users.

In the Maven settings, we can specify some properties and flags, as follows:

- The path of the local repository. This is `~/.m2/repository` by default.
- The `offline` flag prevents Maven from connecting to remote repositories (useful in case of network problems).
- The `<proxies>` element allows us to configure proxies used to connect to the network.
- The `<servers>` element allows us to specify the credentials of the Maven repositories to which we want to deploy our artifacts. We'll speak about deploying our projects to remote repositories in *Chapter 5*, *Continuous Integration and Delivery with Maven*.
- The `<profiles>` element is similar to the one that we can specify in our POMs. We'll speak of Maven profiles in *Chapter 4*, *Managing the Code*. This element should be used very carefully because a project should not depend too much on settings specified outside of its POM.

For example, it can be convenient to share a local repository between all the users of the same machine. This can be done specifying a path for the local repository in the Maven Global Settings file, which is accessible by all the users.

In order to see the result of the merging between the local and Global Settings, we can use the Maven Help Plugin, as follows:

```
$ mvn help:effective-settings
```

This goal is analogous to the `effective-pom` goal of the same plugin. We can find a complete reference of the Maven settings on the Maven site at `http://maven.apache.org/settings.html`.

Properties and resource filtering

In this section, we'll see how to use references to various types of properties in our POMs and how to use them to perform replacements in our project resources. This feature is called resource filtering.

Maven properties

Maven properties are referenced using the `${property-name}` syntax. They can be used as follows:

- Anywhere in the POM
- In all the project resources under `/src/main/resources` (and/or under any other resource directories defined in our POM)

We have to distinguish between implicit and user-defined properties. The implicit properties are as follows:

- **Project properties**: We can use the `${project.*}` syntax to reference the value of all elements of our effective POM. For example, `${project.groupId}` and `${project.build.directory}` refer to the `<project><groupId>` and `<project> <build><directory>` elements of our (effective) POM, respectively. Of course, we can only specify properties that are uniquely determined by their path. In other words, we cannot reference a `<dependency>` or `<plugin>` element.
- **Settings properties**: These are analogous to the project properties, but they refer to the Maven settings files through the `${settings.*}` syntax.
- **Environment properties**: We can refer to the environment variables through the `${env.<variable-name>}` syntax. For example, we can reference the `JAVA_HOME` or `PATH` variable using placeholders such as `${env.JAVA_HOME}` and `${env.PATH}`.
- **System properties**: We can reference all the properties accessible via `System.getProperties()` by the Maven Java process. Some examples are `${os.name}` and `${line.separator}`.

In addition to the implicit properties, we can define our arbitrary user-defined properties in the `<properties>` element of our POM, as follows:

```
<project>
  [...]
  <properties>
    <my.property>myValue</my.property>
    <other.property>Other value</other.property>
```

```
      <logback.version>1.0.7</logback.version>
    </properties>
    [...]
    <dependencies>
          <dependency>
              <groupId>ch.qos.logback</groupId>
              <artifactId>logback-classic</artifactId>
              <version>${logback.version}</version>
          </dependency>
          <dependency>
              <groupId>ch.qos.logback</groupId>
              <artifactId>logback-core</artifactId>
              <version>${logback.version}</version>
          </dependency>
    </dependencies>
</project>
```

This way, we can put in evidence and factorize some particular values that are used in multiple places in our POMs, for example, the dependency versions of platforms and frameworks that consist of more than one artifact.

Resource filtering

Resource filtering is disabled by default and can be activated in the `<resources>` child element of the `<build>` element of our POM, as shown in the following example. We have to set the `<filtering>` flag of the desired `<resource>` element to `true`:

```
<project>
  [...]
  <properties>
  [...]
  </properties>
  [...]
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
  [...]
</project>
```

This way, all the properties referenced in our resources will be replaced with their real values by the Maven Resource Plugin.

In addition to the Maven properties, resource filtering can also use properties defined in further property files, which are called **filters**. The properties contained in these files will be used only for resource filtering, and they cannot be referred in our POM. In the next example, we specify an additional property file, `app.properties`, to be used for resource filtering:

```
<build>
  <filters>
    <filter>src/main/filters/app.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

We can specify multiple resource directories with different settings for the `<filtering>` flag, as follows:

```
  <resources>
    <resource>
      <directory>src/main/resources-alt</directory>
      <filtering>true</filtering>
    </resource>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
```

In this case, only the resources in the `src/main/resources-alt` folder will be filtered.

Notice that we have to also specify the default `src/main/resources` directory when we add further resource directories because the `<resources>` element definition replaces the defaults completely.

Building EE applications

Now that we explored all the core features of Maven, we are ready to use them together to manage the build process of Java EE applications. Usually, EE applications consist of several WAR and JAR archives, and so the Maven way to manage them is to create a multimodule Maven project.

Building WEB applications

As we have already seen in the preceding examples, and also when we spoke of WAR packaging, we have to put the web application resources (JSP files, deployment descriptors, static images, and so on) under `/src/main/webapp`. This is the default value for the `warSourceDirectory` configuration property of the Maven WAR Plugin.

In addition, we can define other web resource directories and activate resource filtering for the additional resources. We can also enable the filtering of the deployment descriptors using the `filteringDeploymentDescriptors` configuration property, but other resources under the `/src/main/webapp` default directory cannot be filtered.

It seems that only the `web.xml` descriptor can be filtered setting the `filteringDeploymentDescriptors` property to `true`. Other proprietary descriptors such as `weblogic-web.xml` or `jboss-web.xml` are left unaltered. The recommended way to filter web application resources is to put them in additional web resource directories.

Here is an example of how to configure the Maven WAR Plugin to enable web resource filtering:

```
<filters>
  <filter>src/main/filters/webapp.properties</filter>
</filters>
<plugins>
  <plugin>
    <artifactId>maven-war-plugin</artifactId>
    <configuration>
      <filteringDeploymentDescriptors>
true
</filteringDeploymentDescriptors>
      <nonFilteredFileExtensions>
        <!-- default value contains jpg, jpeg, gif, bmp, png -->
        <nonFilteredFileExtension>pdf</nonFilteredFileExtension>
      </nonFilteredFileExtensions>
```

```
        <webResources>
          <resource>
            <directory>src/main/webResources</directory>
            <filtering>true</filtering>
          </resource>
        </webResources>
      </configuration>
    </plugin>
  </plugins>
```

Further information about the Maven WAR Plugin can be obtained invoking the Maven Help Plugin, as follows:

```
$ mvn help:describe -Dplugin=war -Ddetail
```

When we declare dependencies for a web application, we have to pay attention to the dependency scopes. As we know, all the direct and transitive dependencies resulting at the `compile` and `runtime` scopes will be included in the packaged archive. Often, we don't need to include artifacts in our WAR; we can encounter classpath problems doing this, which happens when these libraries are provided by the web application container. We have to remember to use the `provided` scope for these dependencies. In case of transitive dependencies, their scope should be overridden in our POM, or they can be excluded if they are not needed for the compilation of our project.

Building enterprise applications

Enterprise applications are packaged in EAR archives and can contain multiple EJB modules, WAR archives, and JAR libraries. All these artifacts must be referred to through their Maven coordinates, and some of them are usually siblings within the same multimodule Maven project.

Let's consider our `transportation-project` example and suppose that we want to build an EAR corresponding to the `transportation-acq-ear` module. This archive should contain the following:

- The `transportation-acq-ejb` module
- The `transportation-acq-war` module
- All the compile and runtime dependencies needed by EJB and WAR modules
- The `application.xml` descriptor

The common dependencies of EJB and WAR modules should be put in the library directory of the EAR and should not be repeated in the `/WEB-INF/lib` folders of the WAR modules.

We can achieve this result by just configuring the POMs of all these modules; the Maven EAR Plugin, bound by default to the package phase of the build lifecycle, will do the job and also generate the `application.xml` descriptor.

Suppose that the POM of the EJB module is as follows:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.packt.examples</groupId>
        <artifactId>transportation-project</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <artifactId>transportation-acq-ejb</artifactId>
    <name>transportation-acq-ejb</name>
    <dependencies>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.1.1</version>
        </dependency>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.1</version>
        </dependency>
    </dependencies>
</project>
```

Suppose that the POM of the WEB module is the following. Notice that all its dependencies have the provided scope:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.packt.examples</groupId>
        <artifactId>transportation-project</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
```

```
    <artifactId>transportation-acq-war</artifactId>
    <packaging>war</packaging>
    <name>transportation-acq-war</name>
    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-impl</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Then, we can edit the POM of the EAR module and customize the Maven EAR Plugin this way:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.packt.examples</groupId>
    <artifactId>transportation-project</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>transportation-acq-ear</artifactId>
  <packaging>ear</packaging>
  <name>transportation-acq-ear</name>
```

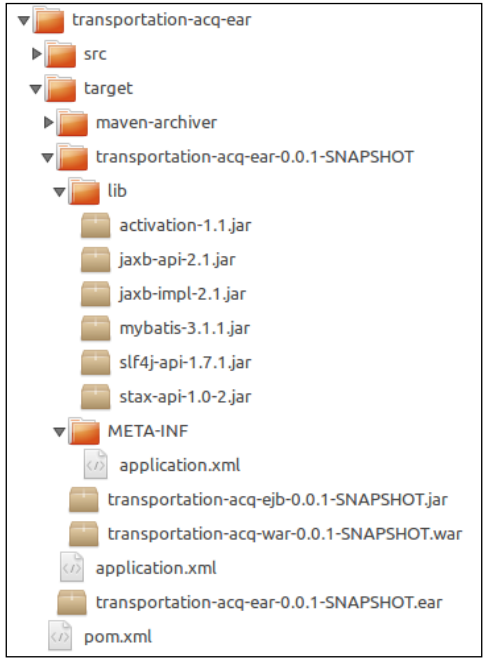
```
<description>Transportation Project Acquisition EAR</description>
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>transportation-acq-ejb</artifactId>
    <version>${project.version}</version>
    <type>ejb</type>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>transportation-acq-war</artifactId>
    <version>${project.version}</version>
    <type>war</type>
  </dependency>
  <dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>2.1</version>
  </dependency>
  <dependency>
      <groupId>com.sun.xml.bind</groupId>
      <artifactId>jaxb-impl</artifactId>
      <version>2.1</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-ear-plugin</artifactId>
      <configuration>
        <displayName>Java EE Application</displayName>
        <version>6</version>
        <generateApplicationXml>true</generateApplicationXml>
        <defaultLibBundleDir>lib</defaultLibBundleDir>
        <modules>
          <ejbModule>
            <groupId>${project.groupId}</groupId>
            <artifactId>transportation-acq-ejb</artifactId>
          </ejbModule>
          <webModule>
            <groupId>${project.groupId}</groupId>
```

```
                <artifactId>transportation-acq-war</artifactId>
              </webModule>
            </modules>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

We have to note the following points:

- The EJB and WEB modules have to be declared in the Maven EAR Plugin configuration
- The EJB and WEB modules also have to appear among the dependencies of the EAR module, and for these dependencies, we have to specify the attribute `type` (with values `ejb` and `war`, respectively)
- The dependencies of the WAR module have the `provided` scope, and they are reintroduced at the `compile` scope in the EAR module

The resulting archive will have the following structure:



The WAR module will not contain any JAR archive, and the content of the generated `application.xml` descriptor will be as follows:

```
<application>
  <description> Transportation Project Acquisition EAR</description>
  <display-name>Java EE Application</display-name>
  <module>
    <ejb>transportation-acq-ejb-0.0.1-SNAPSHOT.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>transportation-acq-war-0.0.1-SNAPSHOT.war</web-uri>
      <context-root>/transportation-acq-war</context-root>
    </web>
  </module>
  <library-directory>lib</library-directory>
</application>
```

If we want to customize the context root of the WEB module, this defaults to its `artifactId`. We should put the `<contextRoot>` child element in the `<webModule>` element of the Maven EAR Plugin configuration, as follows:

```
<webModule>
  <groupId>${project.groupId}</groupId>
  <artifactId>transportation-acq-war</artifactId>
  <contextRoot>/custom-context-root</contextRoot>
</webModule>
```

Finally, in the case of WEB modules, the dependencies at the `compile` scope, as we have seen before, will be packaged in the WAR archives, but they will not be transitive dependencies of the EAR module, so they will not be duplicated in the library directory of the EAR archive.

Configuring repositories

In addition to the Maven central repository, we can also configure other repositories to be used for plugin and dependency downloads. We have to remark that there are separate configuration elements for dependencies and plugin repositories, the `<repositories>` and `<pluginRepositories>` elements. For example, if we want to download both dependencies and plugins from the Java.net repository, we should declare the following:

```
<project>
[...]
<repositories>
```

```
  <repository>
    <id>java.net-Public</id>
    <name>Maven Java Net Snapshots and Releases</name>
    <url>https://maven.java.net/content/groups/public</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>java.net-Public</id>
    <name>Maven Java Net Snapshots and Releases</name>
    <url>https://maven.java.net/content/groups/public</url>
  </pluginRepository>
</pluginRepositories>
[...]
```

In *Chapter 5*, *Continuous Integration and Delivery with Maven*, we'll speak about the Nexus repository service that simplifies the repository management in Enterprise environments.

Enabling releases and snapshots

By default, Maven will attempt to download both releases and snapshots from the additional repositories. If we don't want these releases or snapshots to be searched on a remote repository, we have to disable them explicitly, as follows:

```
<repository>
  <id>sample-release-id</id>
  <name>A release repository</name>
  <url>...</url>
  <releases>
    <enabled>true</enabled>
  </releases>
  <snapshots>
    <enabled>false</enabled>
  </snapshots
</repository>
<repository>
  <id>sample-snapshot-id</id>
  <name>A snapshot repository</name>
  <url>...</url>
  <releases>
    <enabled>false</enabled>
  </releases>
  <snapshots>
```

```
      <enabled>true</enabled>
    </snapshots
  </repository>
```

If we look at the effective POM of any Maven project, we'll see that snapshots are disabled in the Maven central repository configuration.

In a multimodule project, the best choice is to declare an additional repository in the parent POM so that they will be available to all the modules of the project.

Best practices

In this last section, we'll speak about how to refactor POMs of multimodule projects in order to avoid errors and dependency conflicts.

Aggregate POMs

When we have a project consisting of several modules, we will sometimes want to build only a subset of them. If we build the parent project, all the modules will be compiled. On the other hand, building each module separately can be tedious, and we should remember to build the modules in the right order if they depend on each other. To accomplish all these needs, we can use an additional aggregate POM. Let's consider our transportation project example again and suppose that we want to clean and build not only the `transportation-acq-ear` module but also all the other modules on which it depends. We can create the following `transportation-acq-pom.xml` file in the project root directory (at the same level of the parent POM):

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.examples</groupId>
  <artifactId>transportation-acq</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>transportation-acq-ear</module>
    <module>transportation-acq-war</module>
    <module>transportation-acq-ejb</module>
  </modules>
</project>
```

As Maven uses the `pom.xml` file present in the project directory by default, we can build the aggregate POM instead of the parent POM using the `-f` parameter, as follows:

```
$ mvn -f transportation-acq-pom.xml clean install
[...]
[INFO] Reactor Build Order:
[INFO]
[INFO] transportation-acq-ejb
[INFO] transportation-acq-war
[INFO] transportation-acq-ear
[INFO] transportation-acq
[...]
```

As we can see, the Maven Build Reactor builds the modules taking account of the dependencies among them, and so it first builds the EJB module, followed by the WAR module, and finally the EAR module.

We can obtain the same result directly from the command line of the project directory, as follows:

```
$ mvn -pl transportation-acq-ear -am install
```

The `-pl` parameter allows us to specify a list of modules to build, and the `-am` parameter tells Maven to build the projects required by the list. Without the `-am` (or `--also-make`) parameter, only the modules of the list (in this case only the EAR module) will be built.

Dependency management

The Maven dependency mechanism can prove to be a double-edged weapon, especially in multimodule projects, or in case of conflicts between dependencies. Of course, we are speaking of conflicts regarding different versions or scopes of dependencies having the same `groupId` and `artifactId`. In these cases, the default Maven behavior is as follows:

- The version/scope declared in a project overrides the version/scope of the same dependency declared in a parent (or ancestor) POM.
- The version/scope declared in (or inherited by) our project prevails on the version/scope of a transitive dependency.

- If two or more conflicting transitive dependencies have different versions/scopes, the version/scope with the shortest path in the dependency tree will prevail. In the case of paths of the same length, the version/scope of the dependency assigned first in the POM will prevail.
- If a range of version is declared for a direct or transitive dependency, Maven will choose a version within the specified interval, but in the case of a conflict with other ranges of versions for the same dependency, the build process will exit with an error.

If we want to assure that all the modules of a project use the same dependency versions of certain artifacts even when they are transitive dependencies, we have to use a `<dependencyManagement>` element in our POM. In the case of multimodule projects, the dependency management configuration is usually specified in the parent POM.

For example, in the parent POM of our transportation project, we can insert an element such as this:

```
<project>
[...]
<dependencyManagement>
  <dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>6.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>6.0</version>
  <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.1</version>
  </dependency>

[…]
  </dependencies>
</dependencyManagement>
[...]
```

This means that the specified dependencies, when declared directly or assigned as transitive dependencies, will have the default versions and scopes defined in the *Dependency management* section. So, we don't need to explicitly declare the versions and scopes of these dependencies, for example, we can (or better, we should) declare the SLF4J and Java EE web API dependencies simply, as follows:

```
<dependencies>
  [...]
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
  </dependency>

  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
  </dependency>
  [...]
```

Note that the *Dependency management* section does not declare the dependencies that are specified in it, it only fixes their default version numbers and scopes. We have to declare these dependencies (in any case) explicitly in the `<dependencies>` element if they are needed.

If we declare versions and scopes in the `<dependencies>` element for dependencies that are declared in the dependency management section of the same POM (or of its parent POM), the values specified in the `<dependencies>` element will override those specified in the `<dependencyManagement>` element. On the other hand, versions and scopes of transitive dependencies will be always overridden by the values specified in the dependency management section.

Plugin management

Analogous to the dependency management configuration is the plugin management configuration. We can use a `<pluginManagement>` element under the `<build>` element to fix the plugin versions. For example, we can introduce the following element in a parent POM:

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
```

```
        <artifactId>jaxb2-maven-plugin</artifactId>
        <version>1.6</version>
      </plugin>
    </plugins>
    </pluginManagement>
```

This will help us fix the version of the JAXB-2 Maven Plugin across all the modules that need its declaration.

Other than plugin versions, we can also specify default plugin configurations and executions in the `<pluginManagement>` section, but we discourage this practice because it might make it difficult to customize configurations and executions, which might be different for different modules of the project.

Summary

In this chapter, we explored several Maven core concepts and the basic usage of the Maven tool. We saw that every Maven build process relies on a skeleton called build lifecycle. We were introduced to Maven plugins, and learned how they are tied to the phases of the build lifecycle. We dived into the Maven dependency management system and saw how features such as transitive dependencies and dependency inheritance help us to maintain consistency in our projects. Other interesting features such as Maven properties, Maven settings, and resource filtering were also explained. Finally, we learned how to build WEB applications and multimodule EE applications constituted by JAR archives, WAR modules, and EJB modules packaged in EAR archives.

Some other core concepts such as build profiles and site generation will be explained in *Chapter 4*, *Managing the Code*. As all of the Maven work is done by plugins, we'll speak about how to develop a Maven plugin in the next chapter in case we need to customize our builds with tasks that are not available in public plugins.

3
Writing Plugins

As we saw in the previous chapters, Maven isn't a monolithic self-standing product. Instead, it is a pluggable and evolving tool. The sake of extensibility is achieved through the Maven plugin system.

The need for extensibility comes from the nature of the environment Maven operates in. The need for flexibility comes from a vast community of users with different exigencies for their products producing different applications.

Plugins extend Maven's core functionality, allowing it to accomplish many custom tasks. In the previous chapter, we used plugins to build modules with different packaging used as common product distributions (WAR, EAR, EJB). Other plugins make possible custom packaging for autoexecutable products (executable JAR), generating web service implementations based on XSD or WSDL definition, and many other functionalities.

Most of the plugins are developed by Apache. Despite that, everyone can develop their own plugin to fit their needs. A plugin could cover specific project requirements or simply extend the execution of an operation to different lifecycle phases.

In this chapter, we will learn about the following topics:

- Writing a simple Maven plugin
- How to test our plugin
- How to publish your plugin in a local repository

Thanks to `maven-plugin-plugin`, we can use many different programming languages in order to build an executable Maven plugin. It supports different programming languages such as Java, C#, Ruby/JRuby, Scala, Groovy, and Ant. On the other hand, `maven-plugin-testing-harness` provides unit test and integration functionalities.

A problem to solve

A common issue that many developers face is release policies. Often, companies impose to trace every step of every project release.

Projects are composed of many modules, and for each of these modules we have to trace its history. In order to declare a module as **released**, we have to trace which issues were resolved in that release, and how many issues remain open.

For issue tracking and management, we hosted an instance of **Mantis bug tracker** in a private server. All issue information is stored in a MySQL database instance.

You can find more information on Mantis here: `http://www.mantisbt.org/`.

The version of the project model differs from the Mantis version of a module. In order to fill this gap, we created an XML file named `release_structure.xml` and stored it in the `${basedir}` folder of each versioned module.

In this file, we store information about the actual production version of the module. The release structure is updated every time a new module version is released. If the project's version is not specified, the actual `project.version` value is used.

In order to resolve this automation problem, we implemented a plugin named `mantis-maven-plugin`. Whenever a build is performed, our plugin queries for the release version of the project ID passed through configurations. Once it resolves the project, it gets all the resolved issues related to the project, and marks them as `released` in that build version.

Our plugin performs all these operations by means of the `release_structure.xml` file.

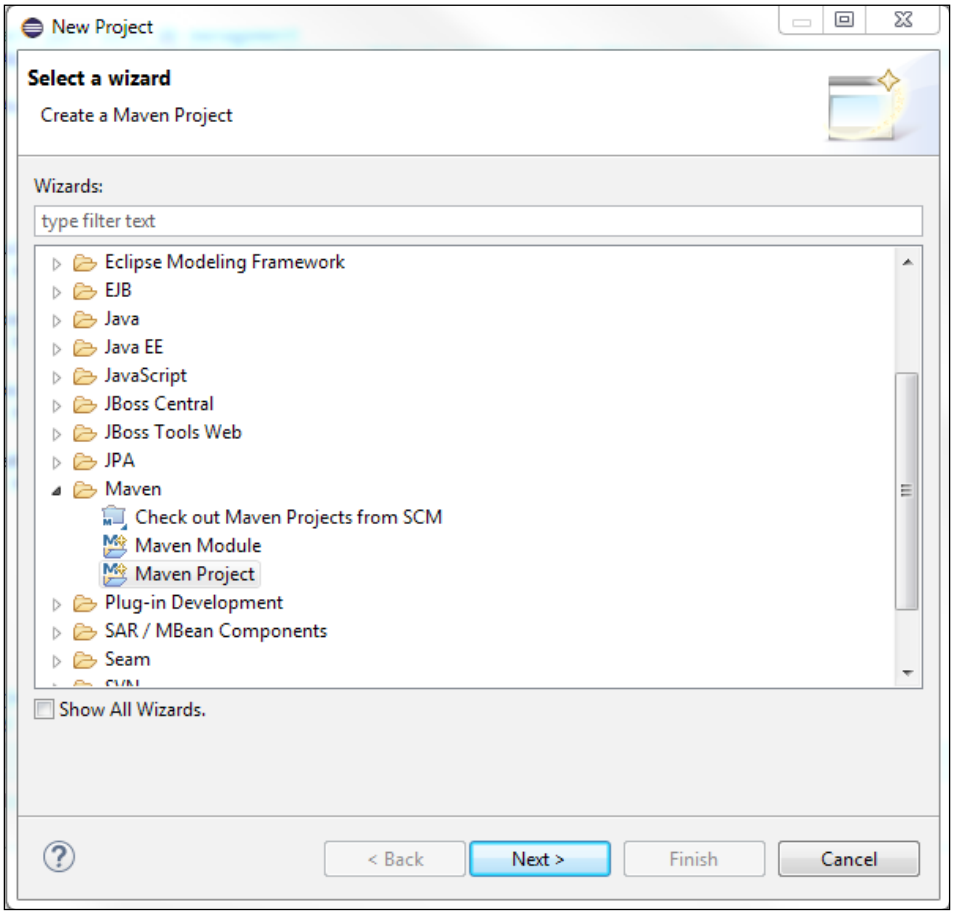
Starting from this problem, we can explain how to build a custom plugin.

Developing a new plugin

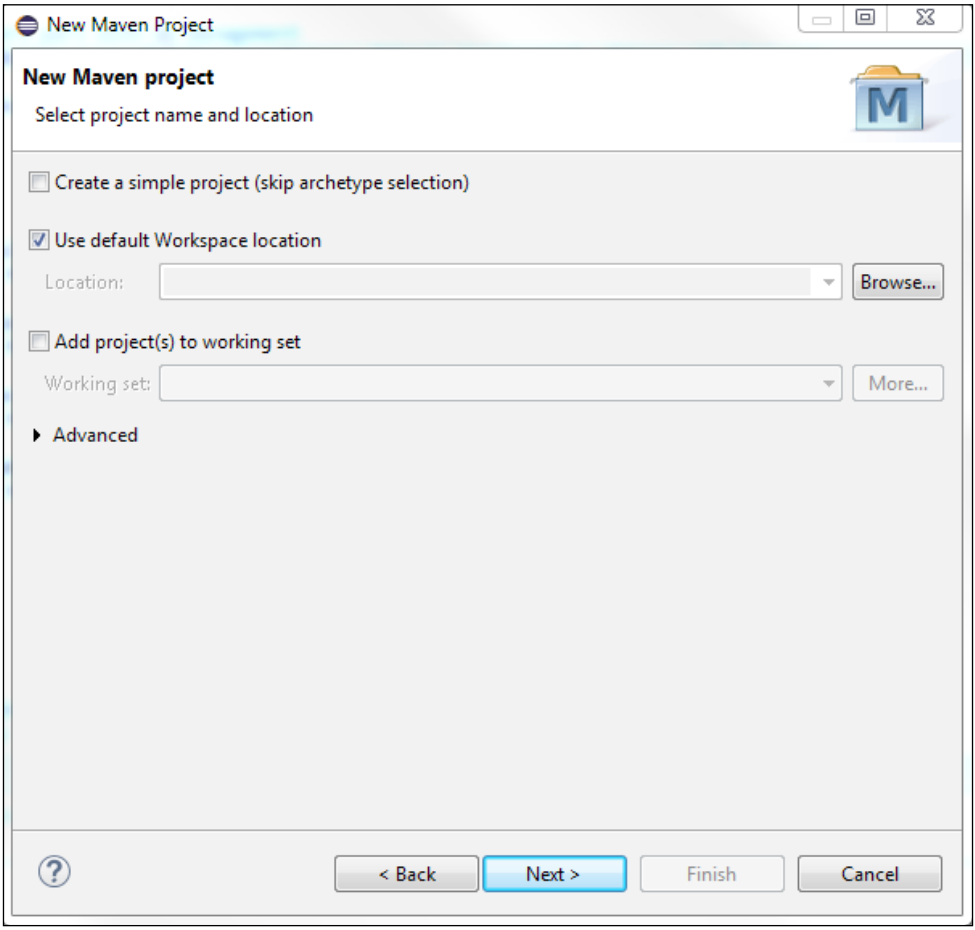
For the development of a Java Maven plugin, we will use the same tool chain we presented in *Chapter 1*, *Maven and Its Philosophy*.

Using Eclipse's utility, we can create a new project with **Maven archetype**. So, we obtain a skeleton for the new Maven plugin project. This is a good starting point for our project. Perform the following steps:

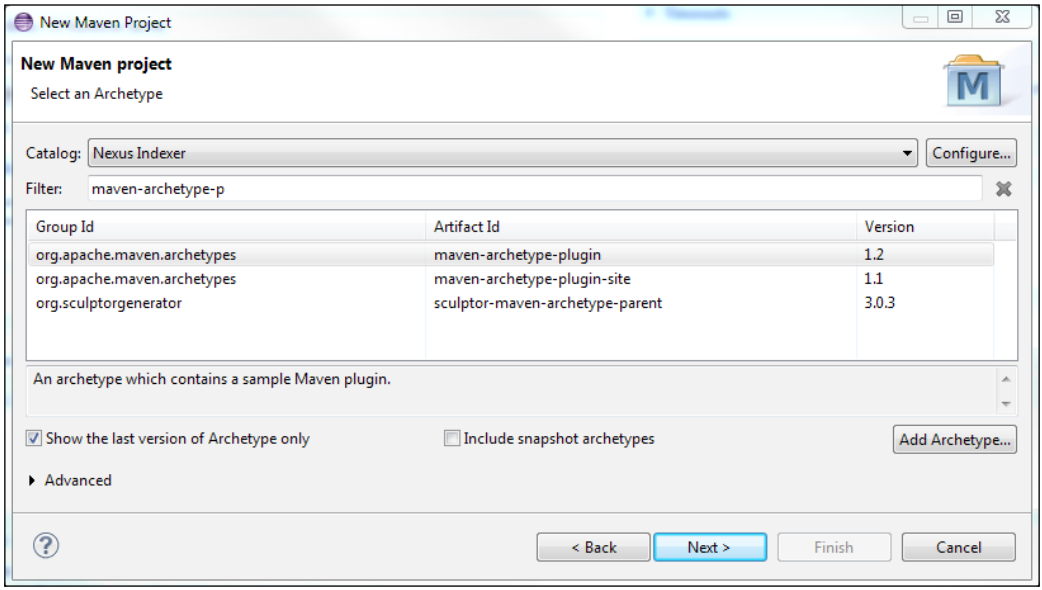
1. First of all, we will create a new Eclipse project as a new Maven project from the menu, as shown in the following screenshot:



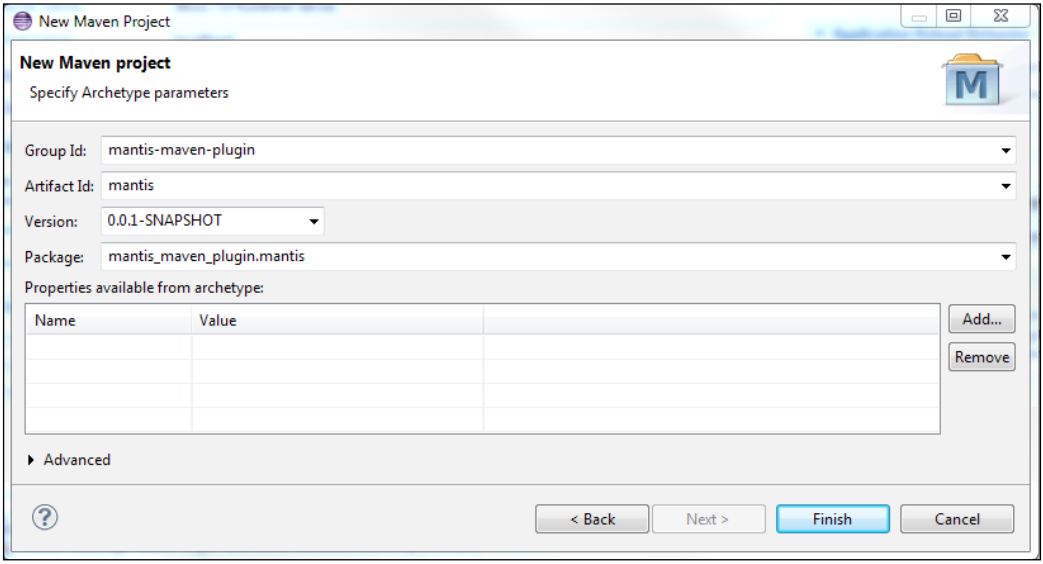
2. Then, we can choose the location for our project and click on **Next**:



3. Now, we have to select the artifact type for our project. First, we choose **Nexus Indexer**, then we select `maven-archetype-plugin`, as shown in the following screenshot, and then click on **Next**:



4. The last step in the creation procedure is the selection of the project's coordinates, as shown in the following screenshot:



The encouraged naming convention for `artifactId` is `<ourplugin>-maven-plugin`.

The name pattern, `maven-<pluginname>-plugin`, is reserved for official Apache Maven plugins. Such plugins are maintained by the official Apache Maven team and have `groupId` as `org.apache.maven.plugins`.

Using this name pattern is an infringement of the Apache Maven Trademark.

The archetype we chose during the creation phase created a `pom.xml` file with some dependencies imported by default:

- `maven-plugin-api`: The articfact generation is set as default version 2.0. This is the basic library containing plugin utility classes.
- `maven-plugin-annotation`: This contains the annotation system. More details are available at `http://maven.apache.org/plugin-tools/maven-plugin-tools-annotations/`.
- `maven-testing-plugin-harness`: This is a library developed by Apache, for developing unit tests based on JUnit.
- `junit`: This is imported with the scope test for testing purposes.

The POM file generated by our IDE will also contain a build profile that is created for integration tests, named `maven-invoker-plugin`.

Before you start writing the plugin source code, we have to slightly edit the POM file that has been generated. First, we need to change the `plexus-utils` default dependency with `maven-core`. The second and last modification consists of aligning the version of `maven-plugin-api` with the version we chose for `maven-core`.

The result of our work is the following POM file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.mojo</groupId>
  <artifactId>mantis-maven-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>mantis-plugin Maven Mojo</name>
  <url>http://maven.apache.org</url>
```

```
  <properties>
    <mavenVersion>3.2.1</mavenVersion>
  </properties>

  <dependencies>
    <!-- Maven dependencies -->
    <dependency>
      <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
    <version>${mavenVersion}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-core</artifactId>
      <version>${mavenVersion}</version>
     </dependency>

     <dependency>
       <groupId>org.apache.maven.plugin-tools</groupId>
       <artifactId>maven-plugin-annotations</artifactId>
       <version>3.2</version>
       <scope>provided</scope>
     </dependency>

     <!-- MySql driver -->
     <dependency>
       <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
       <version>5.1.30</version>
     </dependency>

   <!-- Test dependencies -->
  <!-- Mandatory in order to works with maven-plugin-testing-
harness v. 3.1.0 -->
   <dependency>
   <groupId>org.apache.maven</groupId>
   <artifactId>maven-compat</artifactId>
   <version>3.2.1</version>
   <scope>test</scope>
  </dependency>

 <dependency>
   <groupId>org.apache.maven.plugin-testing</groupId>
  <artifactId>maven-plugin-testing-harness</artifactId>
  <version>3.1.0</version>
```

```
    <scope>test</scope>
   </dependency>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  </dependencies>

  <build>
  <pluginManagement>
    <plugins>
     <plugin>
     <groupId>
       org.apache.maven.plugins
     </groupId>
       <artifactId>maven-plugin-plugin</artifactId>
       <version>3.2</version>
       <executions>
      <execution>
        <id>mojo-descriptor</id>
          <goals>
            <goal>descriptor</goal>
          </goals>
      </execution>
      </executions>
      <configuration>
        <skipErrorNoDescriptorsFound>
        true
        <skipErrorNoDescriptorsFound>
      </configuration>

    </plugin>
  </plugins>
  </pluginManagement>
  </build>
   <profiles>
      <profile>
        <id>integration-tests</id>
        <build>
          <plugins>
            <plugin>
              <groupId>org.apache.maven.plugins</groupId>
```

```
            <artifactId>maven-invoker-plugin</artifactId>
            <version>1.7</version>
            <configuration>
              <debug>true</debug>
              <cloneProjectsTo>
                ${project.build.directory}/it
              </cloneProjectsTo>
              <pomIncludes>
                <pomInclude>*/pom.xml</pomInclude>
              </pomIncludes>
              <postBuildHookScript>verify</postBuildHookScript>
              <localRepositoryPath>
                ${project.build.directory}/local-repo
              </localRepositoryPath>
              <settingsFile>src/it/settings.xml</settingsFile>
              <goals>
                <goal>clean</goal>
                <goal>test-compile</goal>
              </goals>
            </configuration>
            <executions>
              <execution>
                <id>integration-test</id>
                <goals>
                  <goal>install</goal>
                  <goal>integration-test</goal>
                  <goal>verify</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
```

The `maven-plugin-annotations` dependency has the `provided` scope. Since annotations are not needed during plugin execution, this dependency can be excluded from the built package.

The `maven-compat` Version 3.2.1 is *mandatory* in order to make it possible to us the `maven-plugin-testing-harness` Version 3.1.0.

Once we perform a Maven update from our IDE, we can start to code the first Mojo.

Implementing Mojo

What is a Mojo? A simple definition for Mojo is that Mojo is a Maven goal. This is not far from reality.

In order to be more accurate, we can say that a Mojo is a **Maven plain Old Java Object**. Each Mojo is an executable goal in Maven, and a plugin is a distribution of one or more related Mojos.

In practice, to create a Mojo, we must create a class extending the abstract class, as follows:

```
org.apache.maven.plugin.AbstractMojo
```

Such a class provides a utility method for common operations. Furthermore, it provides the abstract method, `public void execute() throws MojoExecutionException`, to perform all the *dirty work* when the plugin is executed.

In order to associate our Mojo to a goal for our plugin, we have to use an annotation in Java style on our class definition:

```
@Mojo(name = "mark-resolved",
      defaultPhase = LifecyclePhase.PACKAGE,
      requiresOnline = true,
      requiresProject = true,
      threadSafe = true)
public class MarkResolved extends AbstractMojo
```

With the `@Mojo` annotation, we indicate that the `MarkResolved` class is a Mojo, and its goal is `mark-resolved`. With other parameters of annotation, we specified the default phase when the plugin is executed. In the same annotation, we can indicate other desirable characteristics as follows:

- `requiresOnline`: This indicates that the operation requires online mode to be executed
- `requiresProject`: This indicates that Mojo requires a project in order to be executed
- `threadSafe`: This marks Mojo to be thread safe; with this flag it can support parallel execution during parallel build

This kind of notation is quite different from the old-fashioned way. The old, and still supported, annotation system provides the use of comments. With such a system, our Mojo definition would look like the following code:

```
/**
 * @goal mark-resolved
 * @phase package
 * @requiresOnline true
 * @requiresProject true
 * @threadSafe true
 */
public class MarkResolved extends AbstractMojo
```

As we can see, the functionality and parameters are the same. The only change is represented by the declaration syntax.

All the plugins have a configuration section in which the users can set parameters used for accomplishing the plugin goal. Parameters can also be passed through the command line:

```
$ mvn plugin-name:goal -Dmy.custom.parameter=somevalue
```

In order to match the configuration properties with Mojo's fields, we have to annotate our Mojo fields as follows:

```
@Parameter( property="basedir", required=true)
protected File baseDir;
```

The `@Parameter` annotation's function is to pass the base directory of the project. Within the annotation, we specified the name of the configuration parameter with `property="basedir"`.

We also specified whether the parameter is mandatory or not with the `required=true` notation.

Our `basedir` property is automatically injected into our object without the need for other coding. We can thank the Plexus Inversion of Control framework for this.

With the old annotation, we could obtain the same result with the following code:

```
/**
 * @parameter expression="${basedir}"
 * @required
 */
protected File baseDir;
```

In this way, all the annotated Mojo fields will be associated with a relative parameter without the need for getter and setter methods.

The direct injection is only possible if the name of the field matches the name of the property specified within the annotation, `property="propertyNameToInject"`.

If we want to inject a property with a different name with respect to the field, we have to specify a setter method for such a property.

In order to use a property name different from the field name, we have to indicate the configuration property name used in the plugin configuration. We can do this through the `alias="dbUserName"` notation.

This indication is not enough; we also have to create a setter for the new aliased property name:

```
public void setDbUserName(String dbUserName) {
    this.dbUser = dbUserName;
}
```

As we can see, we linked the `dbUserName` property to the `dbUser` private field.

Using this method, we can remap the configuration properties on different Mojo field names. With this feature, we can remap private fields that are incomprehensible to users, to friendly name parameters.

Another powerful feature is the capability of assigning a default value if there is no value for nonmandatory parameters:

```
@Parameter(property="project.desctiption", defaultValue="${project.
description}")
protected String projectDesctiption;
```

The `projectDescription` field is a nonmandatory field so, if the configuration does not provide a value for this parameter, we can assign the value of the internal property inherited from the Maven's central POM, `{project.description}`.

Note that using Maven 3.0, all the `pom.*` properties are deprecated. All the properties must be named `project.*`.

We have some specific properties, defined in the Maven's central POM, as follows:

- `${basedir}`: This is a built-in property representing the directory in which the `pom.xml` is stored.
- `${version}`: This is a built-in property, equivalent to `${project.version}`, containing the version of the project.
- `${project.build.directory}`: This is a property defined in the Maven's central POM, containing the path of the build directory. Its default value is `target`.
- `${project.build.outputDirectory}`: This is a property defined in the Maven's central POM, containing the directory in which the class files are stored during the build process. Its default value is `target/classes`.
- `${project.name}`: This contains the name of the project.
- `${project.build.finalName}`: This contains the final name of the file created when the built project is packaged.
- `${settings.localRepository}`: This refers to the path of the user's local repository.
- `${env.M2_HOME}`: This is an environment variable containing the `Maven2` installation folder.
- `${java.home}`: This is an environment variable specifying the path to the current `JRE_HOME` folder.

Notice how some of these properties have a common name prefix. Such a prefix is used to specify the scope of the property. So, properties defined at the project level will be prefixed with `project`, while properties defined in the user's `settings.xml` file will be prefixed with `settings`.

The convention for referring to the parent project's variables provides using the `${project.parent}` syntax.

Maven also gives users the possibility of defining custom properties. We can simply add a property into our `pom.xml` file with the following syntax:

```
<project>
...
 <properties>
   <my.property>hello</my.property>
 </properties>
...
</project>
```

If we add this kind of snippet into our project, the `${my.property}` property would result in `hello`.

As we saw before, Maven links each property to a getter/setter method. Thus, properties such as `${project.build.directory}` will be matched with the `getProject().getBuild().getDirectory()` method chain.

In order to implement a Mojo, we need to extend the `AbstractMojo` abstract class. This class provides some common utility methods.

The first inherited method that we will see is `getLog()`. This method returns a logger of type `org.apache.maven.plugin.logging.Log`. Such a logger allows you to write on output of the plugin's execution. The following line of code will produce the `[INFO] Jdbc Driver :: com.mysql.jdbc.Driver` output:

```
log.info("Jdbc Driver :: " + jdbcDriver);
```

The plugin logger provides three different levels of output: info, error, and debug.

Another inherited method is `getPluginContext()`. This method returns `java.util.Map` containing all the property mappings defined in Mojo's context. These mappings contain all the properties explained before, that are visible from Mojo's POM. We can access the `basedir` property using the following notation:

```
String baseDir = (String) getPluginContext("basedir");
```

Using the setter method, we set a custom plugin context to our Mojo, as follows:

```
public void setPluginContext(Map pluginContext)
```

We can get the plugin context map using `getPluginContext`, add some parameters, and set the new map with `setPluginContext`.

Best practices discourage this kind of manipulation of the context map.

Testing Mojo

An essential need during development is to test the code. In order to satisfy this need, we used a specific library named `maven-plugin-testing-harness`. Such a library has been designed specifically to test Mojo functionalities.

To start using `testing-harness` for our tests, we need to create a test class in the test package. As in the Mojo development, our class must extend this abstract class in order to inherit testing and utility methods:

```
org.apache.maven.plugin.testing.AbstractMojoTestCase
```

As a first step, we must override the following two basic methods for initialization and ending:

```
@Override
protectedvoid setUp() throws Exception {
  super.setUp();
}

@Override
protectedvoid tearDown() throws Exception {
  super.tearDown();
}
```

The `setUp` method is in charge of creating a default Maven project configuration, and adding it to `PlexusContainer`. For IoC's `tearDown` method simply dismisses `PlexusContainer`.

`PlexusContainer` was chosen by the Maven team because at that time, it was the only implementation of the **Inversion of Control** (**IoC**) pattern.

Later on, Spring came out. Spring is a general framework encapsulating Inversion of Control using XML/Annotations and other patterns build scalable web applications.

All public methods with the prefix `test` will be executed during the test execution phase. To launch the test case, `test`, we can use both Eclipse or the Maven command line. Execute this command from the project's root directory:

```
$ mvn test
```

The inheritance hierarchy of the `AbstractMojotestCase` class refers to the `junit.framework.TestCase` class. This allows us to use all the Junit methods for assertions.

In order to test a plugin, we need a project context for fetching project properties. Since we can't load the test POM inside the test case, we must use the following method:

```
public static File getTestFile(final String path)
```

This method gets the POM file from the location passed as the input parameter.

In order to start our Mojo, we need to load it from the POM file loaded through the `getTestFile` method. We can easily perform this operation using the following inherited method:

```
protected Mojo lookupMojo(String goal, File pom) throws Exception
```

Otherwise, we can load a Mojo with a particular configuration using the following code:

```
protected Mojo lookupConfiguredMojo(MavenProject project, String goal)
throws Exception
```

It is also possible to perform multiple launches to test different goals, as follows:

```
protected MojoExecution newMojoExecution(String goal)
```

We can also extract the plugin configuration using the following code:

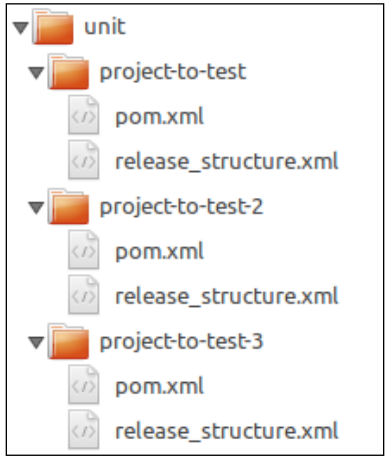
```
protected PlexusConfiguration extractPluginConfiguration(String
artifactId, File pom)throws Exception
```

The `PlexusConfiguration` object contains all the configuration within a map, so we can test whether some configuration is present or not and elaborate those values.

Best practices for testing

In order to test a plugin, we must create a condition for the plugin execution. In other words, we need one or more test POM files. By using different POM files, we can cover a wide range of test cases and plugin settings.

We can organize the test structure inside the `test/resources` directory as follows:



A testing POM must contain a simple project where we can use our plugin. An example POM file for testing is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.mojo</groupId>
  <artifactId>project-to-test</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Test MyMojo</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
       <artifactId>mantis-maven-plugin</artifactId>
       <version>0.0.1-SNAPSHOT</version>
    <configuration>
       <basedir>
        ${basedir}/src/test/resources/unit/project-to-test
       </basedir>
    <rsName>release_structure.xml</rsName>
    <databaseUrl>http://localhost:8090</databaseUrl>
    <jdbcDriver>com.mysql.jdbc.Driver</jdbcDriver>
    <dbUserName>testName</dbUserName>
    <dbPassword>testPassword</dbPassword>
    <databaseName>mantisIssue</databaseName>
    <projectId>test</projectId>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

The following code uses the example POM file to access our Mojo as a Java object, and performs some functional tests:

```
public void testPomAndGoalForNoProperty() throws Exception {

File pom = getTestFile( "src/test/resources/unit/project-to-test/pom.
xml" );

    assertNotNull( pom );
    assertTrue( pom.exists() );

    MarkResolved myMojo = (MarkResolved) lookupMojo("mark-resolved",
pom );

    assertNotNull( myMojo );

    myMojo.execute();
    }
```

First, we get the test POM file, using the `getTestFile` method. We pass the relative path to the test POM as a parameter.

Subsequently, we exploit the Junit assertion mechanism in order to check whether the test POM exists. We perform the same checks on the Mojo object. We can extract the Mojo from the POM file using the `lookupMojo` method described earlier.

Finally, we execute the Mojo, invoking its `execute` method directly. Using different POM files, we can test different plugin goals as different Mojos. In general, it is better to define a test POM for each goal to test.

Integration testing

In the classic software lifecycle, unit tests are naturally followed by the integration phase. Earlier, we studied the unit testing of a Maven plugin, and now we will deepen the integration phase.

The integration phase is covered by `maven-invoker-plugin` because this plugin can run a set of Maven projects (features that we didn't use at all for our plugin project) and can verify the output generated from the project launched. The ability to verify the output generated from the project that is executed is accomplished by a script that could be a bash script or a groovy script. This plugin is included in a specific build profile for the integration phase.

The integration build profile, by default, is named `run-its`. It is possible to rename it.

Whenever we want to perform integration tests, we have to run the following command:

```
$ mvn integration-test -Prun-its
```

When we run such a command, Maven executes the plugin related to the profile, and creates a local repository structure in the `${basedir}/target` directory. Once the local repository has been created, Maven tries to install the plugin under testing and verifies the correctness of the installation process. In order to perform the check for the correct installation of the plugin, we used the configuration option of `maven-invoker-plugin`: `postbuildhookscript` called `verify.bsh`.

The configuration part of the plugin is as follows:

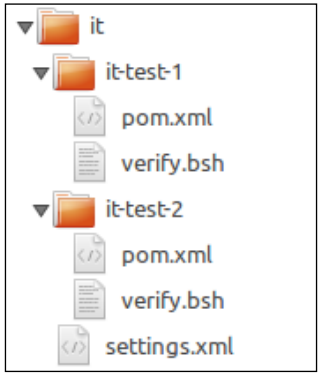
```
<postBuildHookScript>verify</postBuildHookScript>
```

It is important to adopt a custom local repository for integration tests, such as the `${basedir}/target` folder. Thus, we avoid polluting the local Maven repository.

As we can see from the configuration of `maven-invoker-plugin`, we configured the plugin in order to perform the installation, the integration test, and verify the goals:

```
<executions>
  <execution>
     <id>integration-test</id>
     <goals>
        <goal>install</goal>
        <goal>integration-test</goal>
        <goal>verify</goal>
     </goals>
  </execution>
</executions>
```

The structure created to perform tests is as follows:



The `settings.xml` file contains settings for `maven-invoker-plugin`. These settings allow Maven to locate the test repository.

The `pom.xml` files within the folders simulate the project environment in which we want to test our plugin.

We had two test projects to test two different conditions: `it-test-1` verifies the correct plugin installation and `it-test-2` verifies whether the database was updated correctly. All tests were performed by `verify.bsh`.

If scripts found no expected behavior, they throw an exception.

When we execute the `integration-test` command on the `pom.xml` file, we obtain the sequence of operations declared in `maven-invoker-plugin`:

```
[INFO] ------------------------------------------------------------------
[INFO] Building mantis-plugin Maven Mojo 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------
 [INFO] --- maven-invoker-plugin:1.7:install (integration-test) @ mantis-
maven-plugin ---
[INFO] Installing projects/mantis-plugin/pom.xml to target/local-repo/
com/example/mojo/mantis-maven-plugin/0.0.1-SNAPSHOT/mantis-maven-plugin-
0.0.1-SNAPSHOT.pom
[INFO] Installing /Users/robertobaldiprojects/mantis-plugin/target/
mantis-maven-plugin-0.0.1-SNAPSHOT.jar to /target/local-repo/com/example/
mojo/mantis-maven-plugin/0.0.1-SNAPSHOT/mantis-maven-plugin-0.0.1-
SNAPSHOT.jar
[INFO]
[INFO] --- maven-invoker-plugin:1.7:integration-test (integration-test) @
mantis-maven-plugin ---
[WARNING] File encoding has not been set, using platform encoding
MacRoman, i.e. build is platform dependent!
[INFO] Building: it-test-1/pom.xml
[INFO] ..SUCCESS (5.3 s)
[INFO] Building: it-test-2/pom.xml
[INFO] run script verify.bsh
[INFO] ..SUCCESS (3.4 s)
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
```

As we can see from the preceding logs (the only interesting lines were reported for shortness), first the plugin is installed. Then, Maven finds the two test projects and executes our plugin on the projects. After that, it executes `postbuildhookscript` in order to check the correct installation of the plugin (test-1) and the correct execution of the plugin on the database (`test-2`).

If something goes wrong during one of the checks, then the following error is returned:

```
[INFO] Building: it-test-1/pom.xml
[INFO] ..SUCCESS (4.3 s)
[INFO] Building: it-test-2/pom.xml
[INFO] run script verify.bsh
[INFO] ..FAILED (3.5 s)
[INFO]   The post-build script did not succeed. Database is not updated
correctly!!
```

maven-plugin-plugin

After the Mojo has been compiled and packaged into a JAR, Maven can invoke it as a plugin. The main difference between a common JAR and a plugin JAR lies in a file named `plugin.xml`, stored inside the JAR's directory, `META-INF/maven/plugin.xml`.

In a plugin's JAR artifact, this file is the plugin descriptor. It contains all the information that Maven needs to recognize a JAR as a plugin's JAR artifact:

- The list of Mojo classes
- The list of plugin's configurations
- The list of dependencies and requirements needed

Manually writing all the information can be a long and error-prone task. Fortunately, Maven provides a plugin for doing this dirty work. The `maven-plugin-plugin` provides the `descriptor` goal to generate the plugin descriptor file. The following sample shows how to configure `maven-plugin-plugin` to generate a descriptor:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-plugin-plugin</artifactId>
  <version>3.2</version>

  <executions>
    <execution>
```

```
        <id>mojo-descriptor</id>
        <goals>
          <goal>descriptor</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <skipErrorNoDescriptorsFound>
        true
      </skipErrorNoDescriptorsFound>
    </configuration>
  </plugin>
```

The next sample shows the head section of the generated plugin descriptor:

```
<plugin>
  <name>mantis-plugin Maven Mojo</name>
  <description></description>
  <groupId>com.example.mojo</groupId>
  <artifactId>mantis-maven-plugin</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <goalPrefix>mantis</goalPrefix>
  <isolatedRealm>false</isolatedRealm>
  <inheritedByDefault>true</inheritedByDefault>
```

The plugin descriptor also contains one section for each goal covered by the plugin. Each of these sections contain the goal execution's requirements and configurations. In the following sample, we can see the configurations related to the `mark-resolved` goal:

```
<mojos>
    <mojo>
      <goal>mark-resolved</goal>
      <description>
        Goal touching a timestamp file.
      </description>
      <requiresDirectInvocation>
        false
      </requiresDirectInvocation>
      <requiresProject>true</requiresProject>
      <requiresReports>false</requiresReports>
      <aggregator>false</aggregator>
      <requiresOnline>true</requiresOnline>
```

```
<inheritedByDefault>true</inheritedByDefault>
<phase>package</phase>
<implementation>
  com.example.mojo.plugin.MarkResolved
</implementation>
<language>java</language>
<instantiationStrategy>
  per-lookup
</instantiationStrategy>
<executionStrategy>
  once-per-session
</executionStrategy>
<threadSafe>true</threadSafe>
<parameters>
  <parameter>
    <name>basedir</name>
    <type>java.io.File</type>
    <required>true</required>
    <editable>true</editable>
    <description>The base directory.</description>
  </parameter>
```

The plugin's configurations represent the Mojo fields. In the following sample, we can see how, for each parameter, we declare its type and, optionally, its default value:

```
<configuration>
  <basedir implementation="java.io.File">
    ${basedir}
  </basedir>
  <databaseUrl implementation="java.lang.String">
    ${databaseUrl}
  </databaseUrl>
  <projectDescription>
      implementation="java.lang.String"
      default-value="${project.description}">
    ${project.description}
  </projectDescription>
  <projectId implementation="java.lang.String"
      default-value="${project.artifactId}">
    ${projectId}
  </projectId>
  <projectVersion
```

```
            implementation="java.lang.String"
            default-value="${project.version}">
          ${projectVersion}
        </projectVersion>
        <rsName implementation="java.lang.String">
          ${rsName}
        </rsName>
      </configuration>
```

The preceding sample shows the dependencies of our plugin. As we can see, the dependencies declaration is the same as the one used for normal projects:

```
<dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <type>jar</type>
      <version>3.2.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-model</artifactId>
      <type>jar</type>
      <version>3.2.1</version>
    </dependency>
```

All plugins have a useful help goal explaining how to use that plugin, and which configurations it needs. Since every plugin goal is a Mojo, it would be expensive work to implement a custom Mojo only to satisfy the help goal. In order to realize this task, we can take advantage of a `maven-plugin-plugin` goal named `helpmojo`, as follows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-plugin-plugin</artifactId>
  <version>3.2</version>
  <execution>
    <id>generated-helpmojo</id>
    <goals>
      <goal>helpmojo</goal>
    </goals>
  </execution>
 </executions>
</plugin>
```

After executing a compile command, we will find a new Mojo class generated by the `helpmojo` goal at the following location:

```
target
  |
  |-generated-source
     |
     |-our packaging directory structure
      |
      |-HelpMojo.java
```

The generated class, `HelpMojo`, is a standard Mojo class, with all its necessary fields and a preimplemented `execute` method, where the goal task logic is implemented:

```
@Mojo( name = "help", requiresProject = false, threadSafe = true )
publicclass HelpMojo extends AbstractMojo
/**
     * If <code>true</code>, display all settable properties for each
goal.
     *
     */
    @Parameter( property = "detail", defaultValue = "false" )
    private boolean detail;

    /**
     * The name of the goal for which to show help. If unspecified,
all goals will be displayed.
     *
     */
    @Parameter( property = "goal" )
    private java.lang.String goal;

    /**
     * The maximum length of a display line, should be positive.
     *
     */
    @Parameter( property = "lineLength", defaultValue = "80" )
    private int lineLength;

    /**
     * The number of spaces per indentation level, should be positive.
     *
     */
    @Parameter( property = "indentSize", defaultValue = "2" )
    private int indentSize;
```

The only task in charge to the help goal, is to print out all the information related to the implemented plugin. All the information is extracted from an XML file located at the following location:

```
META-INF
  |
  |-maven
  |
  |-com.example.mojo
    |
    |-mantis-maven-plugin
      |
      |-plugin-help.xml
```

To check whether the help Mojo generated by `maven-plugin-plugin` has all the correct information, we need to perform `mvn install` on our project (a previous build is needed in order to make a successful install possible). After a successful return of the `install` command, we can execute `mvn help ourPluginGroupId:ourpluginArtifactId:help`.

If all the previous operations were successful, we can read the help information as follows:

```
[INFO] mantis-plugin Maven Mojo 0.0.1-SNAPSHOT
This plugin has 3 goals:
mantis:help
  Display help information on mantis-maven-plugin.
  Call mvn mantis:help -Ddetail=true -Dgoal=<goal-name> to display
parameter details.


mantis:issue-info
  Passing module articaftId return all open mantis issue, if exist any,
for that project.


mantis:mark-resolved
  Perform an update on all issue attached to projectId passed as
configuration parameter.
```

All the information related to a single goal is written in the class comments of the matching Mojo definition, as shown in the following example:

```
/**
 * Passing module artifactId return all open
 * mantis issue, if there exist any, for that project.
```

```
 *
 *
 */
@Mojo(name = "issue-info", defaultPhase = LifecyclePhase.NONE,
requiresOnline = true, requiresProject = true, threadSafe = true)
public class IssueInfo extends MarkResolved
```

To launch our plugin's help goal (`mvn mantis:help`) with user friendly notation, we have to edit the `settings.xml` file and add our plugin group to the `pluginGroups` tag:

```
<pluginGroups>
    <pluginGroup>com.example.mojo</pluginGroup>
</pluginGroups>
```

In this way, we can execute all goals for our plugin with user-friendly notation.

Custom plugin – mantis-maven-plugin

In order to deepen all the concepts described earlier, we will expose a real-world experience. We will build up a plugin for automating the publication process of our transportation project.

Referring to the problem introduced in the beginning of this chapter, we move forward to its implementation.

Custom plugin implementations

In order to resolve this automation problem, we implemented a plugin named `mantis-maven-plugin`. Whenever a build is performed, our plugin queries for the release version of project ID that is passed through configurations. Once it resolves the project, it gets all the resolved issues related to the project and marks them as `released` in that build version.

Our plugin performs all these operations by means of the `release_structure.xml` file.

If the project ID was not present, then get `projectId` as the default value.

The `mantis-maven-plugin` architecture is quite simple. The main plugin goal is implemented by `MarkResolved.java`. This class simply retrieves all the parameters needed for a correct execution.

The `IssueInfo.java` class extends the `MarkResolved` functionalities to implement the second goal.

The last goal for our plugin is implemented in `HelpMojo.java`. As you probably imagine, this class has been generated through `maven-plugin-plugin` using the `helpmojo` goal.

In the following code snippet, we can see how all the concepts that were exposed earlier were implemented.

In order to accomplish the database update, we implemented a class named `MySqlAccess.java`. Such a class deals with database connections and performs update operations. Since these functionalities have been implemented through standard libraries, we won't complicate this class.

Instead, we find more interesting `DataReader.class`. This class, used by `getProjectVersion` to get XML data, uses utility classes from the `org.codehaus.plexus.util` package. As you remember, this is the same package used by `AbstractMojo`:

```
@Mojo(name = "mark-resolved", defaultPhase = LifecyclePhase.PACKAGE,
requiresOnline = true, requiresProject = true, threadSafe = true)
public class MarkResolved extends AbstractMojo {

  @Parameter( property = "basedir", required = true )
  protected File basedir;

  @Parameter(property="rsName", required = true)
  protected String rsName;

  @Parameter(required = true)
  protected String jdbcDriver;

  @Parameter(property="projectId", required = true,
defaultValue="${project.artifactId}")
  protected String projectId;

  @Parameter(property="projectVersion", required=true,
defaultValue="${project.version}")
  protected String projectVersion;

  @Parameter(property="projectDescription", defaultValue="${project.
description}")
  protected String projectDescription;

  @Parameter(property="databaseUrl", required = true)
```

```
protected String databaseUrl;

@Parameter(required=true, alias="dbUserName")
protected String dbUser;

@Parameter(required=true, alias="dbPassword")
protected String dbPswd;

@Parameter(required=true, alias="databaseName")
protected String dbName;

@Component(role=MavenProject.class)
protected MavenProject projectArtifact;

protected MySqlAccess mySqlAccess;

protected DataReader dataReader;

protected Log log;

public void setDbUserName(String dbUserName) {
  this.dbUser = dbUserName;
}

public void setDbPassword(String dbPassword) {
  this.dbPswd = dbPassword;
}


public void setDatabaseName(String databaseName) {
  this.dbName = databaseName;
}

public void execute() throws MojoExecutionException {
  String projectV = null;
  log = getLog();

  log.info("base dir :: " + basedir);

  log.info("Jdbc Driver :: " + jdbcDriver);

  log.info("urldb passed :: " + databaseUrl);
```

```
    log.info("projectId :: " + projectId);

    try {

      projectV = getProjectVersion();

      if (projectV != null && !projectV.isEmpty()) {
        log.info("Current module version to update is :: " +
projectV);

        // Start part with db management
        mySqlAccess = new MySqlAccess(dbUser, dbPswd, databaseUrl,
dbName, jdbcDriver);

        mySqlAccess.updateStatus(projectId, projectV);


      }


    } catch (XmlPullParserException e) {
      throw new MojoExecutionException("Error on parsing xml file for
version to update");

    } catch (IOException e) {
      throw new MojoExecutionException("Error on accessing xml file
for version to update ");

    } catch (SQLException e) {
      throw new MojoExecutionException("Error on mantis database
execution");

    } catch (ClassNotFoundException e) {
      throw new MojoExecutionException("Error on instantiation for
driver class");
    }

  }


  protected String getProjectVersion() throws XmlPullParserException,
IOException {
    String prjVersion = null;
```

```
    File xmlReleaseStructure = null;

    if (this.projectVersion == null || this.projectVersion.isEmpty())
{
      if (basedir != null && (projectVersion == null ||
projectVersion.isEmpty())) {
        xmlReleaseStructure = new File(basedir.getAbsolutePath() +
File.separator + rsName);

        if (xmlReleaseStructure.exists()) {
          dataReader = new DataReader();
          prjVersion = dataReader.getVersionFromXml(xmlReleaseStructu
re, projectId);

        }
      }
    } else {
      prjVersion = this.projectVersion;

    }

    return prjVersion;
  }

}
```

As mentioned earlier, we use the `release_structure.xml` file to get the information about the project name and version. The file has the following structure:

```
<releaseStructure>
  <module>
    <name>test</name>
    <version>2.4</version>
  </module>
</releaseStructure>
```

Since a software module can be composed of several submodules, the XML structure allows more than one occurrence for the `module` tag. This structure allows us to get data in a simple way through the `DataReader` class functionalities:

```
import java.io.File;
import java.io.IOException;

import org.codehaus.plexus.util.ReaderFactory;
import org.codehaus.plexus.util.xml.Xpp3Dom;
```

```
import org.codehaus.plexus.util.xml.Xpp3DomBuilder;
import org.codehaus.plexus.util.xml.pull.XmlPullParserException;

public class DataReader {

  public String getVersionFromXml(File xmlReleaseStructure, String
projectId) throws XmlPullParserException, IOException {
    String version = null;
    Xpp3Dom xmlDom = Xpp3DomBuilder.build( ReaderFactory.newXmlReader(
xmlReleaseStructure ) );
    Xpp3Dom[] modules = xmlDom.getChildren("module");

    for (Xpp3Dom child : modules) {
      String nameModule = child.getChild("name").getValue();

// If name module isn't null and not empty then check for
// searching name
      if (nameModule != null && !nameModule.isEmpty()) {
        if (projectId.equalsIgnoreCase(nameModule)) {
          // If we found correct module, then get version
          version = child.getChild("version").getValue();

          break;
        }
      }
    }
    return version;
  }
    }
```

Summary

In this chapter, we saw how to create a custom plugin starting from a real-world problem. We learned how to create a plugin project, we saw which libraries to import in order to make the project work, and which class we need for developing basic functions for our purpose. We also saw how to structure tests for plugin development and how to perform simple integration tests.

The next chapter will show how Maven can be integrated with new and various instruments in order to automate operations and different tasks.

4
Managing the Code

In the previous chapters, we studied how Maven core works and how to write a Maven plugin by implementing custom functionalities. In this chapter, we will talk about some extended Maven concepts and functionalities.

This chapter will show us the following concepts in detail:

- How **build profiles** are structured
- How to customize your build phase to face different environments
- How to use **Maven Assembly Plugin** to build customized archives out of your project
- How to use **Maven Site Plugin** to create a wiki-style website containing all the information related to the project

Maven build profiles

In the previous chapters, we saw how to configure the build of our project. However, we did not address the problem of build portability in the previous chapters. With *build portability*, we measure how easily we can port a build configuration across different environments. A nonportable build will need more configurations and hacks compared to a portable build.

Of course, portability is sometimes not entirely possible. Some plugins and some applications' configurations might depend on resources that are related to a specific environment.

In order to address such circumstances and facilitate build portability across environments, Maven introduces the concept of a build profile. A build profile is a set of POM elements that you can optionally activate by overriding the corresponding tags in a POM file. This is the only point in which you need to define environment-specific settings.

Profiles modify the POM file at build time by overriding the POM settings according to the configurations set in the profile. In some ways, the profiles are similar to the `mvn -f` command, providing maintainability to the POM file.

In the following sections, we will see how profiles are structured, and we will instantiate them in our transportation project.

What is a profile?

Maven allows the definition of profiles in different levels.

A build profile might contain project-specific settings. In this case, it must be defined in the `pom.xml` file of a single project. Based upon whether we want to centralize all the profiles' information related to a single project, we can also define them in the profile descriptor file contained in the `${basedir}/profiles.xml` folder. Profiles must be defined in a specific POM file element, which is named `profiles`.

As shown in the following snippet, all profiles are identified by a unique ID:

```
<profiles>
    <profile>
        <id>profile1-id</id>
        <!-- configurations and other stuff -->
    </profile>
    <profile>
        <id>profile2-id</id>
        <!-- configurations and other stuff -->
    </profile>
</profiles>
```

Profiles can also be defined at the user level. In this case, the profiles section, which we saw earlier, will be defined in the `${USER_HOME}/.m2/settings.xml` file and will be visible for all the projects in the machine.

Profiles defined at the global level are contained in the `${M2_HOME}/conf/settings.xml` file.

Since profiles can be defined across many layers, Maven provides a specific goal to track the active build profiles of a project. Running `mvn active-profiles` will show all the active profiles of our project.

The structure of a profile

As we said before, build profiles are designed to port a build configuration across different environments. We also said that this result is achieved by overriding some of the POM settings; but which settings can be overridden? The obvious answer to this question is that we can override almost all the properties that we defined in the POM file.

The following snippet from the book titled *Maven: The Complete Reference* shows the full structure of the plugins:

```
<profiles>
    <profile>
        <id>...</id>
        <activation>...</activation>
        <build>
            <defaultGoal>...</defaultGoal>
            <finalName>...</finalName>
            <resources>...</resources>
            <testResources>...</testResources>
            <plugins>...</plugins>
        </build>
        <reporting>...</reporting>
        <modules>...</modules>
        <dependencies>...</dependencies>
        <dependencyManagement>...</dependencyManagement>
        <distributionManagement>...</distributionManagement>
        <repositories>...</repositories>
        <pluginRepositories>...</pluginRepositories>
        <properties>...</properties>
    </profile>
</profiles>
```

As we can see, the profile can override almost all the POM sections. It is possible to customize the build package with different modules and dependencies, and it is also possible to define custom resources and properties to fit in the environment settings. We can customize the database connection, the namespace of a WSDL, or the `web.xml` configuration of a WAR module.

Smart readers will probably notice that two tags of the preceding code sample do not appear in a normal POM file. The `id` and `activation` tags are specific to the `profile` element. As we said earlier, only the `id` element can uniquely identify the profile within the project.

The `activations` element explains when the profile has to be used. The concept of profile activation will be explained in more detail in the following section.

Profile activation

As we explained earlier, profiles are needed to activate environment-specific settings during the build phase. The main implication of this fact is that more than one profile will exist within a single project. The concept of activation of a profile is strictly related to its nature; we might want to enable a specific profile when we build the project for a specific operating system or JDK version, or even by default.

Maven provides several different ways to activate a build profile.

A profile might be set to activate by default. If the activation element contains the `activeByDefault` tag that is set to `true`, the profile will be active, unless some other profile in the same POM file is activated:

```
<activation>
  <activeByDefault>true</activeByDefault>
</activation>
```

Usage of the `activeByDefault` flag is discouraged. This flag activates the profile if no other profile is active. This implies that the default activation is inhibited by another profile's activation. When you run a multimodule build, the default activation will miss if a build profile has been activated, even if it has been defined in other modules.

The best practice to obtain default activation consists in relying on the absence of a property:

```
<activation>
  <property>
    <name>dummy.property.name</name>
  </property>
</activation>
```

You can just define the property in another profile if you want to inhibit the profile's activation:

```
<properties>
  <dummy.property.name>dummyValue</dummy.property.name>
</properties>
```

As hinted before, a build profile can also be activated for a specific operating system or JDK version. In these cases, the specific child elements are `os` and `jdk`. As we can see in the following code snippet, the `os` element allows the user to specify the name, family, version, and architecture of the operating system to activate the profiles. All of these elements are optional.

The following snippet shows the structure of this element for a specific operating system and architecture:

```
<activation>
  <os>
    <name>Windows 7</name>
    <version>6.1</version>
    <family>Windows</family>
    <arch>amd64</arch>
  </os>
</activation>
```

The `jdk` element allows you to specify a range of JDK versions to be used. The version's range has to be defined through the version range system. It is possible to exclude a specific version using the exclamation mark, as shown in the following example:

```
<activation>
  <jdk>!1.5</jdk>
</activation>
```

Other activation methods rely on the presence or lack of a property or file. With the `property` element, we can specify to activate the build profile if a property exists, if it does not, or if it has a specific value. As in the preceding example, the lack of the property is specified by the use of an exclamation mark. The `file` element allows us to specify whether a file exists with the specific `exists` child element of the `missing` element.

Finally, Maven provides the possibility of manually activating one or more build profiles through the `-P` option. It is possible to indicate more than one profile with a comma-separated notation, as shown here:

```
mvn clean package -P profile_id1,profile_id2,!profile_id3
```

Sample build profiles

Throughout this book, we'll show the use of the build profiles mechanism through our example project. We will specifically write two different profiles to build the `transportation-acq-war` module into a development environment and on the production environment.

Our build profiles will be identified by the `dev` and `prod` IDs. The first profile will be active when a fake property doesn't exist, and the production profile will only activate with a specific set of configurations.

Each of these profiles will define a set of properties containing environmental settings. The classic example of such settings, highlighted in the next code snippet, is the database connection string; this string sets different values for different profiles and allows developers to work avoiding to care about environmental details. Once the build has been set up, Maven will set the correct configuration according to the environment that we are building on.

In the following code snippet, we can see the resulting profiles section of our POM file. In addition to everything we've already covered, we also add some different plugin configurations.

During the development phase, we will run unit tests with a custom configuration to ignore test failures, and in the production phase, we will build a WAR file containing a custom `web.xml` file:

```
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <property>
        <name>!dev.profile.trigger</name>
      </property>
    </activation>

    <properties>
      <db.driverClass>
        oracle.jdbc.driver.OracleDriver
      </db.driverClass>
      <db.url>jdbc:oracle:thin:@127:0:0:1:1521/DBService</db.url>
      <db.user>dbuser</db.user>
      <db.password>dbuser123</db.password>
    </properties>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.17</version>

          <configuration>
            <testFailureIgnore>true</testFailureIgnore>
            <includes>
              <include>**/test/java/*.java</include>
            </includes>
```

```
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>

    <profile>
      <id>production</id>

      <activation>
        <jdk>[1.6,)</jdk>
        <os>
          <family>Unix</family>
          <arch>amd64</arch>
        </os>
      </activation>

      <properties>
        <db.jndi>jdbc/productionOracle</db.jndi>
        <!-- dev profile inhibition -->
        <dev.profile.trigger>I'm a dummy value</dev.profile.trigger>
      </properties>

      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>

            <configuration>
              <webXml>src/main/webapp/WEB-INF/prod/web.xml</webXml>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
```

Maven Assembly Plugin

The Maven plugin generates a structure that fits our needs, but in some cases, we have to produce an output different from the common plugin structure; in such cases, the appropriate choice is **Maven Assembly Plugin**.

Fitting to environment

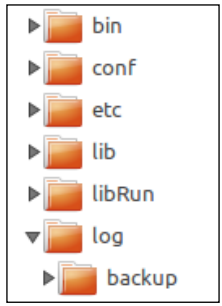
Our particular case of study has to face a deployment of batch application. In the specific batch application called **BatchHandler**, we have to perform read operations for data acquisition, produce output on files, and send files via FTP to be stored in other locations. In order to be deployed, our batch has to satisfy some specifications:

- It must be a ZIP file
- Once the ZIP file is unpacked, we must have a precise directory structure

To accomplish this specification without making strange operations in order to create the package, we use `maven-assembly-plugin`.

As described before, our batch application is a simple JAR with some dependency, so we manage to add the assembly plugin to our project's POM.

The structure within the ZIP file should be like the one shown in the following screenshot:



All the directories contain some specific files as described here:

- The `bin` directory contains the script to launch the application in an environment-agnostic fashion (Windows, Linux, or Mac OSX)
- The `conf` directory contains all the configurations for correct application execution
- The `etc` directory contains all the configurations related to the environment variables and database connections
- The `lib` directory contains all the libraries that are needed for batch execution
- The `libRun` directory contains the batch application's JAR
- The `log` directory contains some configurations to log
- The `backup` directory contains the log configuration backup

In order to generate the whole structure in a single step, we use a custom generation option for `maven-assembly-plugin` using the assembly descriptor, as described previously.

Building your own archive through the Assembly plugin

The common meaning of assembly is to merge a group of files, directories, or dependencies into an archive format and distribute it to someone or into some environment.

The Assembly plugin can be used to aggregate a project with its dependencies, source code, documentation, and other files as configuration files into a single archive.

If we simply need to aggregate our project with the most common files present in a project structure, we can use a predefined model. When the predefined descriptor can't accomplish a specific user's target, a more powerful tool to manage the assembly architecture and files comes to the rescue: the descriptor file.

In all the examples, we used Version 2.4 of `maven-assembly-plugin`. This version provides a single goal.

All other goals are deprecated and will be removed in the future versions of the plugin.

Goals such as `assembly:assembly`, `assembly:attached`, `assembly:directory`, and `assembly:directory-inline` are deprecated because they break normal build processes and promote nonstandard build practices.

Since the `assembly:single-directory` goal is redundant, it has been deprecated in favor of the `dir` goal. Moreover, the `assembly:format` and `assembly:unpack` goals have been deprecated in favor of a far more comprehensive Maven Dependency Plugin.

Maven Assembly Plugin's most important function is represented by `descriptorRef`. This element represents the key for all packaging operations because it describes the structure to be created in the output.

As we said before, with `descriptorRef`, you can specify a predefined descriptor because Maven provides a set of descriptors covering all the common usages:

- With the `jar-with-dependencies` descriptor, the plugin can generate an executable JAR such as Maven Shade Plugin.

- The `bin` descriptor allows the creation of a redistributable archive, starting from your project. Such archives can be in any of the three formats: `ZIP`, `tar.gz`, or `tar.bz2`. The resulting project JAR is included, and it is possible to specify other files such as *readme*, *license*, or *notice*.
- The `src` descriptor produces an output packaging similar to the `bin` descriptor output. This descriptor adds content from the `src` project directory, which enables you to redistribute source code in conjunction with the executable. Output formats are the same as that of the `bin` descriptor.
- The `project` descriptor consists of the sum of the `bin` and `src` descriptors. This descriptor creates an archive containing all the elements from our project structure. Only the target directory will be excluded from packaging. Also, in this case, the output formats are `ZIP`, `tar.gz`, and `tar.bz2`.

The predefined descriptors cover almost all the common needs; if someone needs specific behavior, the assembly plugin accepts a custom XML descriptor as input. This descriptor allows you to specify how to create the output archive and the contents to be included. This kind of operation can be accomplished using the **descriptor file**.

The descriptor file

The descriptor file has different sections to describe various interactions with a project's files. We will describe the principal sections in order to understand the descriptor structure and functionality:

- As we saw before, the assembly section usually describes a compressed archive (`tar`, `tar.gz`, or `ZIP`) starting from the original project. It's possible to create a compressed file within the project's JAR artifact, a directory within dependencies, usually called `lib`, and another directory called `bin` within scripts in order to execute the application in a standalone mode.
- The `containerDescriptorHandler` is used to filter the files to aggregate them into the assembly archive. It's possible to aggregate different types of descriptor fragments, such as XML files for project configuration.
- With `moduleSet`, it is possible to include sources or binaries from different modules that are declared in a project's `pom.xml` file.
- The `sources` element allows us to define configuration options to add a project's source code into our assembly file.
- The `fileSet` element allows us to include files or a group of files into the assembly.

- The `binaries` element is useful for including a project's module binary files in the resultant package.
- We can exploit a set of options provided by `dependencySet` to manage project dependencies by the inclusion and exclusion of the output assembly package.
- The `unpackOptions` element provides us with the possibility to manage item extraction from the archive in order to filter, exclude, or include resources.
- The `file` element allows us to specify the inclusion of individual files. It also permits us to change the destination filename.
- The `groupVersionAlignement` element gives us the possibility to align a group of artifacts to a specific version, passed as a configuration parameter.
- The `repository` element is particularly useful whenever we need to deploy archives to internal repositories. It allows us to reorganize the project's dependencies into a small Maven repository and include it in the output archive.

Currently, only the artifacts from the central repository are allowed values for the element repository.

Thanks to the information provided within the descriptor file, we can create different structures for the plugin output.

The project configuration

In our project, the Assembly plugin configuration has the following form:

```
<plugin>
 <artifactId>maven-assembly-plugin</artifactId>
 <version>2.4</version>
 <configuration>
   <descriptorRefs>
     <descriptorRef>
       ${descriptorDir}/assembly-descriptor.xml
     </descriptorRef>
   </descriptorRefs>
 </configuration>
 <executions>
   <execution>
     <id>make-assembly</id>
     <phase>package</phase>
```

```
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

We bind execution to the package phase and pass a custom descriptor as `descriptorRef`.

The `${descriptorDir}` variable stores the path to the descriptor file used by the assembly plugin.

In the preceding section, we described all the components of a descriptor file. Now, we will understand how it works. The following is the first element that we have:

```
<id>run</id>
```

It just sets the ID for assembly and represents a symbolic name for files from the project. The ID value will also be attached to generate a final filename. Another function of the `id` element is to be an artifact's classifier at deploying time:

```
<includeBaseDirectory>false</includeBaseDirectory>
```

As the tag name suggests, this option tells the plugin whether to include the project's base directory in the output archive. In our example, we set it to false, since we don't want to include the base directory.

With the following tag, we specify to use the ZIP format as the output:

```
<formats>
        <format>zip</format>
</formats>
```

As we said before, this option accepts many formats, such as `ZIP`, `tar`, `tar.gz`, `tar.bz2`, `jar`, `dir`, and `war`. All formats but dir are well known: that is not a file format but a directive to create an exploded directory and not a compressed file or a Java archive.

In the `fileSets` tag, we specify which file we want to put in the output directories. In the case of the `bin` directory, all the source directory contents will be copied into the destination directory:

```
<fileSet>
  <directory>src/main/resources/scripts</directory>
  <outputDirectory>bin</outputDirectory>
</fileSet>
```

The same operation was performed for the `log` and `etc` directories. In the `conf` directory, we included a subset of the files present in the source directory. Through the `include` directive, we configure this behavior:

```
<fileSet>
  <directory>src/main/resources</directory>
  <outputDirectory>conf</outputDirectory>
  <includes>
    <include>app.properties</include>
    <include>log4j.xml</include>
    <include>extract_ldap.param</include>
    <include>extract_ldap.param.sample.INT</include>
    <include>extract_ldap.param.sample.INT1</include>
    <include>extract_ldap.param.sample.PREPROD</include>
    <include>extract_ldap.param.sample.PROD</include>
  </includes>
</fileSet>
```

All the JAR files related to the libraries and application batches are included using the `dependencySets` tag. As in the previous section, we can use the `include` and `exclude` directives to manage the set of JAR files that we want to copy:

```
<dependencySets>
  <dependencySet>
    <outputDirectory>libRun</outputDirectory>
    <unpack>false</unpack>
    <scope>runtime</scope>
    <outputFileNameMapping>
                  ${artifactId}.jar
            </outputFileNameMapping>
    <includes>
      <include>${artifact}</include>
    </includes>
  </dependencySet>
  <dependencySet>
    <outputDirectory>lib</outputDirectory>
    <unpack>false</unpack>
    <scope>runtime</scope>
    <excludes>
      <exclude>${artifact}</exclude>
    </excludes>
  </dependencySet>
</dependencySets>
```

The `includes` directive defines a set of files and directories to be included in the output archive. If no pattern is specified, all the files are included. Similarly, the `excludes` tag represents a set of files to exclude. On the other hand, if the `excludes` tags are empty, no files will be excluded.

Since the `excludes` tag takes priority over the `includes` tag, if we leave both the elements empty, all the files will be included in the output archive.

In the preceding snippet, we can see two different dependency sets. The first set specifies through the use of appropriate tags:

- Output directory
- To copy without unpacking the archive
- The set of libraries to copy, based on the scope and name pattern of the library to copy

Since the `include` statement specified the filename pattern described in `outputFileNameMapping`, only the libraries fitting the pattern will be included inside the `libRun` directory. In our examples, the pattern includes only the generated JAR file.

The second `dependencySet` looks very similar to the first one. The main difference relies in the usage of the `excludes` statement in order to exclude a set of dependencies. Using the `${artifact}` pattern, we exclude only the batch application JAR.

Put together all the configurations that we saw previously into the original descriptor file and complete it with its header results, as shown in the following code:

```
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">

  <id>run</id>
  <includeBaseDirectory>false</includeBaseDirectory>
  <formats>
    <format>zip</format>
  </formats>

  <dependencySets>
    <dependencySet>
      <outputDirectory>libRun</outputDirectory>
      <unpack>false</unpack>
```

```
      <scope>runtime</scope>
      <outputFileNameMapping>
        ${artifactId}.jar
      </outputFileNameMapping>
      <includes>
        <include>${artifact}</include>
      </includes>
    </dependencySet>
    <dependencySet>
      <outputDirectory>lib</outputDirectory>
      <unpack>false</unpack>
      <scope>runtime</scope>
      <excludes>
        <exclude>${artifact}</exclude>
      </excludes>
    </dependencySet>
  </dependencySets>

  <fileSets>
    <fileSet>
      <directory>src/main/resources/config</directory>
      <outputDirectory>etc</outputDirectory>
    </fileSet>
    <fileSet>
      <directory>src/main/resources</directory>
      <outputDirectory>conf</outputDirectory>
      <includes>
        <include>app.properties</include>
        <include>log4j.xml</include>
        <include>extract_ldap.param</include>
        <include>extract_ldap.param.sample.INT</include>
        <include>extract_ldap.param.sample.INT1</include>
        <include>extract_ldap.param.sample.PREPROD</include>
        <include>extract_ldap.param.sample.PROD</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>src/main/resources/scripts</directory>
      <outputDirectory>bin</outputDirectory>
    </fileSet>
    <fileSet>
      <directory>src/main/tmp</directory>
      <outputDirectory>tmp</outputDirectory>
    </fileSet>
    <fileSet>
      <directory>src/main/resources/log</directory>
      <outputDirectory>log/backup</outputDirectory>
```

```
      </fileSet>
    </fileSets>
  </assembly>
```

As a result, we manage to generate a single file named `BatchHandler-run.zip` containing all the directories and libraries that we need to put our batch into action.

Maven Site Plugin

Maven Site Plugin is a very useful mechanism used to generate some basic information, such as the Javadoc, project and module descriptions, dependencies, and management tools, for the project.

Maven Site Plugin provides a fine-grained way to customize the final outcome using the APT language. In this section, we will learn how to produce a basic site, grabbing information from the `pom.xml` file and the source code of our project and avoiding extra documentation. In the last paragraph, we will customize a module to provide more information.

Creating a simple site

To create a site easily, we first choose a directory for a sample project using the following archetype:

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook
-DartifactId=sample-project
```

This archetype creates a simple project named `sample-project`:

Then, we run the following command:

```
$   mvn site
```

The preceding command will create a `target/site` folder.

To see the result, open the `index.html` file or connect to the `http://localhost:8080/` URL after running the following command:

```
$   mvn site:run
```

Maven will use the default configuration to create the site.

Creating your own project site manually

Now, we are ready to customize the site for our multimodule project structure.

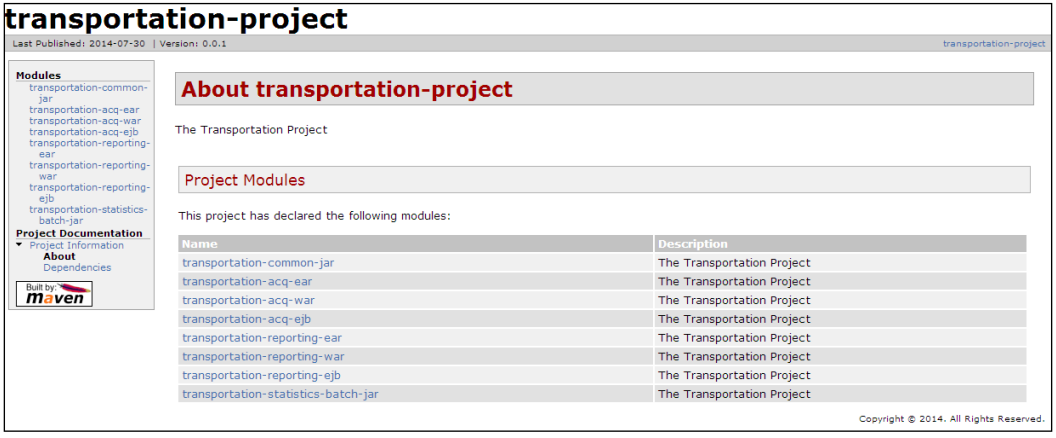
Open the `pom.xml` file of `transportation-project` and add the following plugin in the `reporting` element:

```
[…]
<reporting>
  <plugins>
   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.7</version>
      <reportSets>
        <reportSet>
          <reports>
             <report>index</report>
            <report>dependencies</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
</reporting>
[…]
```

Use the following command line to run the goal:

```
$  mvn clean install site
```

The `install` command is required if it is not already installed. Maven will create the `target/site` folder, including our HTML pages, one site for each submodule. Opening the `index.html` file, you should get a result as shown in the following screenshot:



The transportation project's index page

Unfortunately, we cannot navigate the site correctly. To explore each submodule, we must deploy the site into the final deploy directory.

We can simply define the deploy directory by creating a new property named `siteDirectory`:

```
[…]
<properties>
    <siteDirectory>C://siteDirectory</siteDirectory>
</properties>
[…]
```

After we define the property, we have to set the `distributionManagement` element, as shown in the following snippet:

```
[…]
<distributionManagement>
      <site>
      <id>transportation-site</id>
      <url>file://${siteDirectory}</url>
    </site>
</distributionManagement>
[…]
```

The `url` tag describes the deploy location. It's possible to save it into both a local and remote directory (at the moment, only SSH is supported).

Now, we can deploy the site running the following command:

```
$  mvn site:deploy
```

Maven should produce the following result:

```
[INFO] ------------------------------------------------------------------------
[INFO] Building transportation-statistics-batch-jar 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-site-plugin:3.3:deploy (default-cli) @ transportation-
statistics-batch-jar ---
file://C://siteDirectory/ - Session: Opened
[INFO] Pushing C:\\WSSites\transportation-project\transportation-
statistics-batch-jar\target\site
```

```
[INFO]    >>> to file://C://siteDirectory/transportation-statistics-
batch-jar
file://C://siteDirectory/ - Session: Disconnecting
file://C://siteDirectory/ - Session: Disconnected
[INFO] ------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] transportation-project ..................... SUCCESS [  1.268 s]
[INFO] transportation-common-jar ................... SUCCESS [  0.357 s]
[INFO] transportation-acq-ejb ..................... SUCCESS [  0.264 s]
[INFO] transportation-acq-war ..................... SUCCESS [  0.217 s]
[INFO] transportation-acq-ear ..................... SUCCESS [  0.340 s]
[…]
```

It's also possible to define other interesting reports by editing `maven-project-info-reports-plugin` and adding the highlighted reports:

```
[…]
<reporting>
  <plugins>
   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.7</version>
      <reportSets>
        <reportSet>
          <reports>
             <report>index</report>
            <report>dependencies</report>
            <report>project-team</report>
            <report>cim</report>
            <report>issue-tracking</report>
            <report>scm</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
</reporting>
[…]
```

Let's further explore the reports we added in the POM snippet previously.

First, we can define the project team by adding the following code at the project root of our `pom.xml` file:

```
<project …>
  […]
  <developers>
    <developer>
      <id>ucicero</id>
      <name>Umberto Cicero</name>
      <email>mymail@example.com</email>
      <organization>MyOrganization</organization>
      <roles>
        <role>book-author</role>
        <role>developer</role>
      </roles>
    </developer>
  </developers>
  […]
```

The `cim` report is the **Continuous Integration Management** (**CIM**) system based on triggers or timings such as Hudson CI (see *Chapter 5, Continuous Integration and Delivery with Maven*).

Other possible reports are License and Mailing Lists.

We can configure the `cim` report using the `ciManagement` element given here:

```
[…]
<ciManagement>
  <system>Hudson CI</system>
  <url>http:// localhost:8080/</url>
  <notifiers>
    <notifier>
      <type>mail</type>
      <address>mymail@example.com</address>
    </notifier>
  </notifiers>
</ciManagement>
[…]
```

To carry out information on issue management, it's possible to set our issue tracker, such as **JIRA** or **MantisBT**. Set the `issueManagement` element as follows:

```
[…]
<issueManagement>
  <system>MantisBT</system>
```

```
    <url>http://localhost/mantisbt</url>
  </issueManagement>
  [...]
```

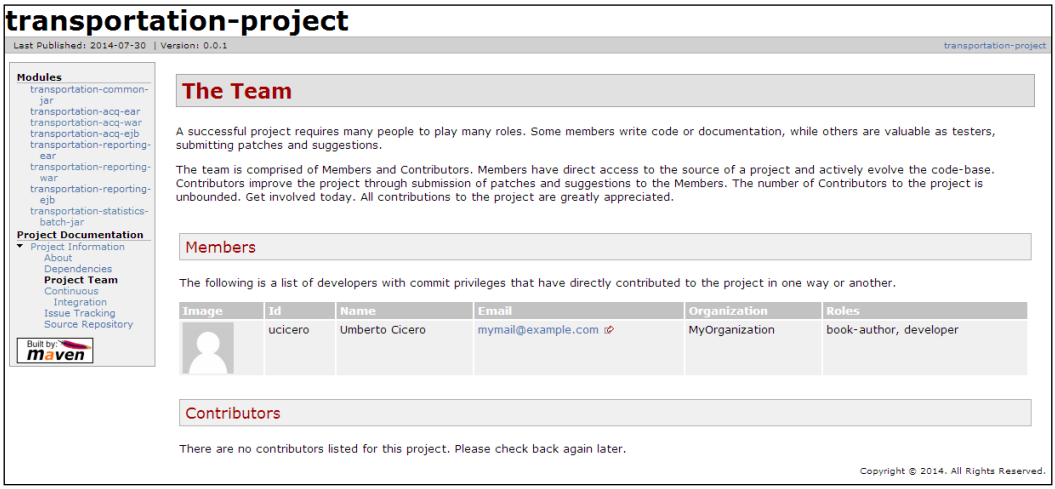
Finally, we can define `scm` or **Source Code Management** (**SCM**). It is used to provide information related to the code management system in order to maintain the project in the code repository (check *SVN*, *GIT*, *CVS*, *Mercurial*, and so on, in *Chapter 5*, *Continuous Integration and Delivery with Maven*). To configure SCM, include the following code in the `pom.xml` file:

```
[...]
<scm>
  <connection>
    scm:svn:https://my-scm-host/trunk/project/
  </connection>
  <developerConnection>
    scm:svn:https://my-scm-host/trunk/project/
  </developerConnection>
  <url>https://my-scm-host/trunk/project </url>
</scm>
[...]
```

We are finally ready to rebuild our project's site and deploy it. Just run the following compact command:

```
$  mvn clean site:site site:deploy
```

The following screenshot shows the team's `project.html` page:



The project's team page

This configuration is the same for each module of the project since it is inherited from the parent POM.

Configuring the site for a submodule

In this section, we'll see how to configure a site for a single submodule. We chose the `transportation-acq-ear` module as an example.

First, we open the `pom.xml` file of `transportation-acq-ear`, and then add a new person at `project-team`:

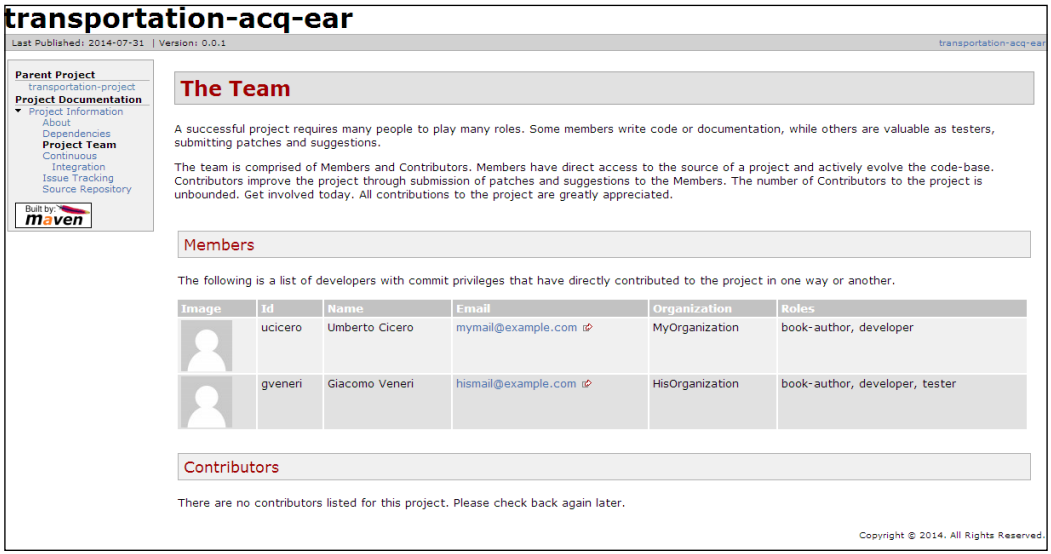
```
[…]
  <developers>
    <developer>
      <id>ucicero</id>
      <name>Umberto Cicero</name>
      <email>mymail@example.com</email>
      <organization>MyOrganization</organization>
      <roles>
        <role>book-author</role>
        <role>developer</role>
      </roles>
    </developer>
    <developer>
      <id>gveneri</id>
      <name>Giacomo Veneri</name>
      <email>hismail@example.com</email>
      <organization>HisOrganization</organization>
      <roles>
        <role>book-author</role>
        <role>developer</role>
        <role>tester</role>
      </roles>
    </developer>
  </developers>
[…]
```

You need not include the site plugin because it is inherited from the parent POM.

Run the following command again:

```
$  mvn clean site:site site:deploy
```

The following screenshot shows a different team for the `transportation-acq-ear` module as a result of our operation:



The transportation-acq-ear's project team

Reporting the Javadoc

It is also possible to publish the Javadoc on the generated site through **Javadoc Plugin**. Add the following snippet in the `reporting` element of the `pom.xml` file of our multimodule project:

```
<project>
  [...]
  <reporting>
   <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.9</version>
        <reportSets>
          <reportSet>
            <reports>
              <report>javadoc</report>
              <report>test-javadoc</report>
            </reports>
          </reportSet>
          <reportSet>
            <id>aggregate</id>
```

```
            <inherited>false</inherited>
            <reports>
              <report>aggregate</report>
            </reports>
          </reportSet>
        </reportSets>
      </plugin>
      […]
  </reporting>
  […]
</project>
```

The `aggregate` directive tells Maven to aggregate the Javadoc of each module on a global directory. The highlighted code, `<report>test-javadoc</report>`, requires the processing of the Javadoc for the test code.

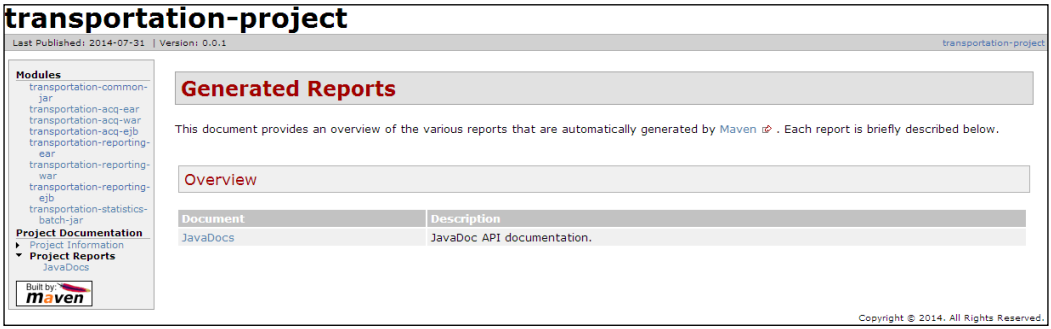
Execute the following command:

```
$ mvn clean site:site site:deploy
```

We should see a result like this:

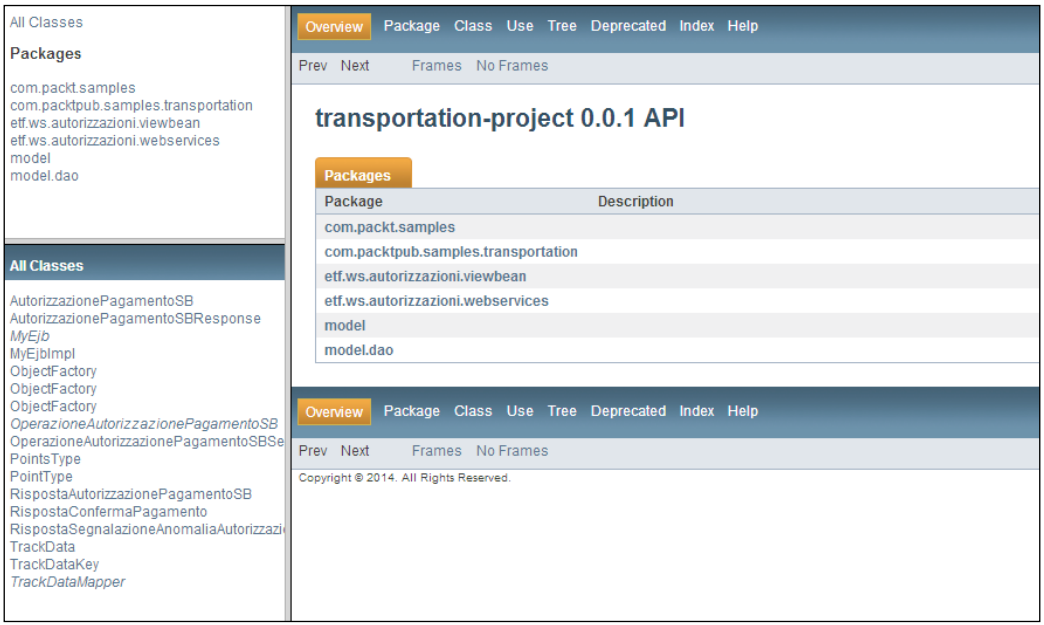
```
[…]
[INFO] >>> maven-javadoc-plugin:2.9:javadoc (report:javadoc) >
generate-sources
@ transportation-reporting-ear >>>
[INFO]
[INFO] <<< maven-javadoc-plugin:2.9:javadoc (report:javadoc) <
generate-sources
@ transportation-reporting-ear <<<
[INFO]
[INFO] >>> maven-javadoc-plugin:2.9:test-javadoc (report:test-javadoc)
> generat
e-test-sources @ transportation-reporting-ear >>>
[INFO]
[INFO] --- maven-ear-plugin:2.8:generate-application-xml (default-
generate-appli
cation-xml) @ transportation-reporting-ear ---
[INFO] Generating application.xml
[INFO]
```

Maven will create a new item on the site's menu, linking it to the Javadoc:



The Javadoc link

Clicking on the link, you will see a result like this:



The Javadoc result

Skinning Maven sites

If you don't like the classic Maven site skin, or if you want a more charming template, Maven provides you with the possibility of customizing it. All we have to do is create the following directory in our project:

Create an XML file named `site.xml` inside the directory. To customize our site, we will use **Skin** (**Maven Fluido Skin**), which is included by default in the site plugin. Other available Skins are:

- Maven Application Skin
- Maven Classic Skin
- Maven Default Skin
- Maven Stylus Skin

Define the `site.xml` file as follows:

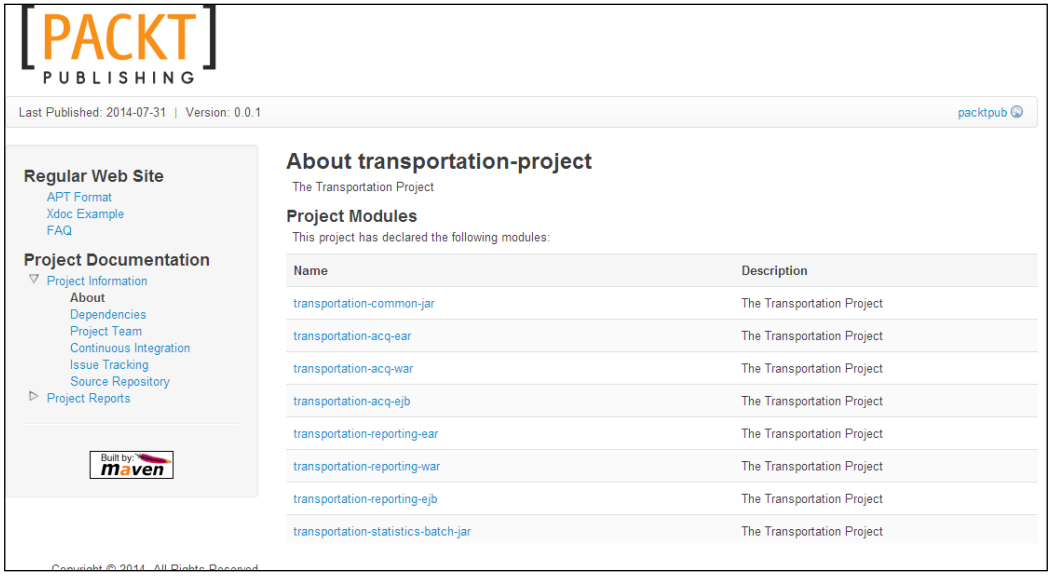
```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="Maven"
  xmlns="http://maven.apache.org/DECORATION/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/DECORATION/1.0.0
    http://maven.apache.org/xsd/decoration-1.0.0.xsd">

  <bannerLeft>
    <name>A Maven Site</name>
    <src>
      http://ww1.prweb.com/prfiles/2011/10/26/9257737/gI_61351_
Packt%20Publishing%20logo.PNG
    </src>
    <href>http://www.packtpub.com/</href>
  </bannerLeft>
  <skin>
    <groupId>org.apache.maven.skins</groupId>
    <artifactId>maven-fluido-skin</artifactId>
    <version>1.3.1</version>
  </skin>
  <custom>
    <fluidoSkin>
      <sideBarEnabled>true</sideBarEnabled>
      <googlePlusOne />
    </fluidoSkin>
```

```
    </custom>

    <body>
      <links>
        <item name="packtpub" href="http://www.packtpub.com/"/>
      </links>

      <menu name="Regular Web Site">
        <item name="APT Format"
  href="http://maven.apache.org/doxia/references/apt-format.html"/>
        <item name="Xdoc Example" href="xdoc.html"/>
        <item name="FAQ" href="faq.html"/>
      </menu>

      <menu ref="reports"/>

      <footer>Packt. All rights reserved.</footer>
    </body>
  </project>
```

If we run the following command, we will get a site that has a new (and very cool) look:

```
$ mvn clean site:site site:deploy
```

The following is an example of a site with a new look:

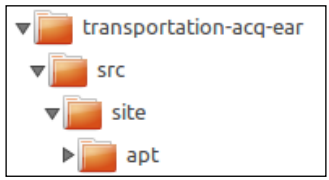


The transportation project site with maven-fluido-skin

Maven site content

Maven Site Plugin supports an easy way to provide additional details about the project. Customizing the main page of each submodule is very easy; we only need to write a few text lines.

Maven Site Plugin allows us to write content in different languages. In our example, we will use **Almost Plain Text** (**APT**). Navigate to the directory shown in the following screenshot:



Create a simple text file named `index.apt` that contains the following data; take care to preserve the same indentation:

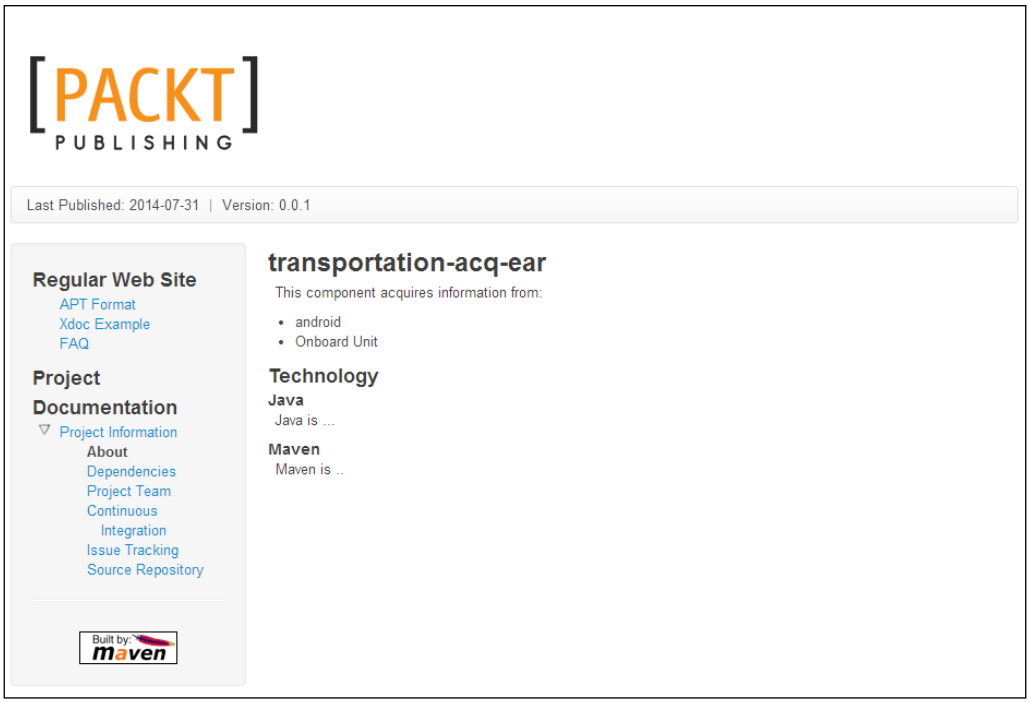
```
            ------
            Title: transportation-acq-ear
            ------

    transportation-acq-ear

      This component acquires information from:

      * android

      * Onboard Unit Device

    * Technology

    ** Java

      Java is ...

    ** Maven

      Maven is ..
```

Now, copy the `site.xml` file that was previously defined in the folder shown in the following screenshot:

Finally, run the following command:

```
$ mvn clean site:site site:deploy
```

The resulting index page will look as follows:



Custom index transportation-project-acq-ear

Following the same procedure, it is possible to build different APT pages.

Summary

This chapter covered the common procedures to create code, artifacts, additional documentation, and packages in order to create maintainable code in the production environment.

We discussed the following topics in detail:

- How to build different artifacts according to a given profile. Profiles are great tools that can change the final package according to the environment settings. On big projects, it is common to change the security policy or database JNDI name in order to fit the destination environment (for example, development, preproduction, test, and production).
- How to change the `web.xml` file to accomplish operations.
- How to produce a proof package, including additional resources (for example, for a batch) required for correct component execution.
- How to produce a simple site hosting the Javadoc and project's information. Documentation is a boring activity as it's normally detached from the code.
- How to produce a minimalistic site reporting Javadoc, dependencies, and generic information about SCM or CI.

In the following chapter, we will see how to exploit Maven's features to exercise the **Continuous Integration and Delivery** pattern through integration with some *state-of-the-art* open source tools.

5

Continuous Integration and Delivery with Maven

Continuous Integration is a common practice proposed by the **extreme programming** (**XP**) methodology to integrate various components developed over a long period of time by different groups. **Continuous Delivery** is a series of processes designed to ensure software quality and to deploy them safely as the final version to a production-like environment. Continuous Integration is *de facto* complimentary to Continuous Delivery and anticipates this stage.

The Continuous Integration workflow can be summarized as follows:

- A developer submits changed code to the Software Configuration Management system
- The code build is automated (periodically or triggered by code changes)
- Unit tests are executed
- The component is deployed on the integration-test environment
- The integration tests are (automatically) executed

The Continuous Delivery workflow can be summarized as follows:

- The code is officially tagged, versioned, and released
- The components are deployed on the official environment test (preproduction)
- The functional tests are executed by the test team
- The components are approved and released for the production environment

Other authors refer to Continuous Integration as part of Continuous Delivery. In this book, we will avoid a complete discussion of these concepts; we will refer to the complete process as **Continuous Integration and Delivery** (**CID**). CID consists of continuous software releasing during the day, software versioning, a tool to align the development environment (development) with the shared environment (mainline), a platform to release and manage the versioned components, continuous unit test management, integration test management, and functional test execution.

CID needs a complete build-and-integration tool, allowing the development team to build and manage the versions, integrate a bug-fixing tool, perform some basic tests, and release the software to the test team. In this chapter, we will build a complete Maven CID environment using some common open source tools such as Nexus, Jenkins/Hudson, MantisBT, and Ant. There are other powerful commercial products, but generally speaking, the main concepts of CID can be discovered and applied through these tools.

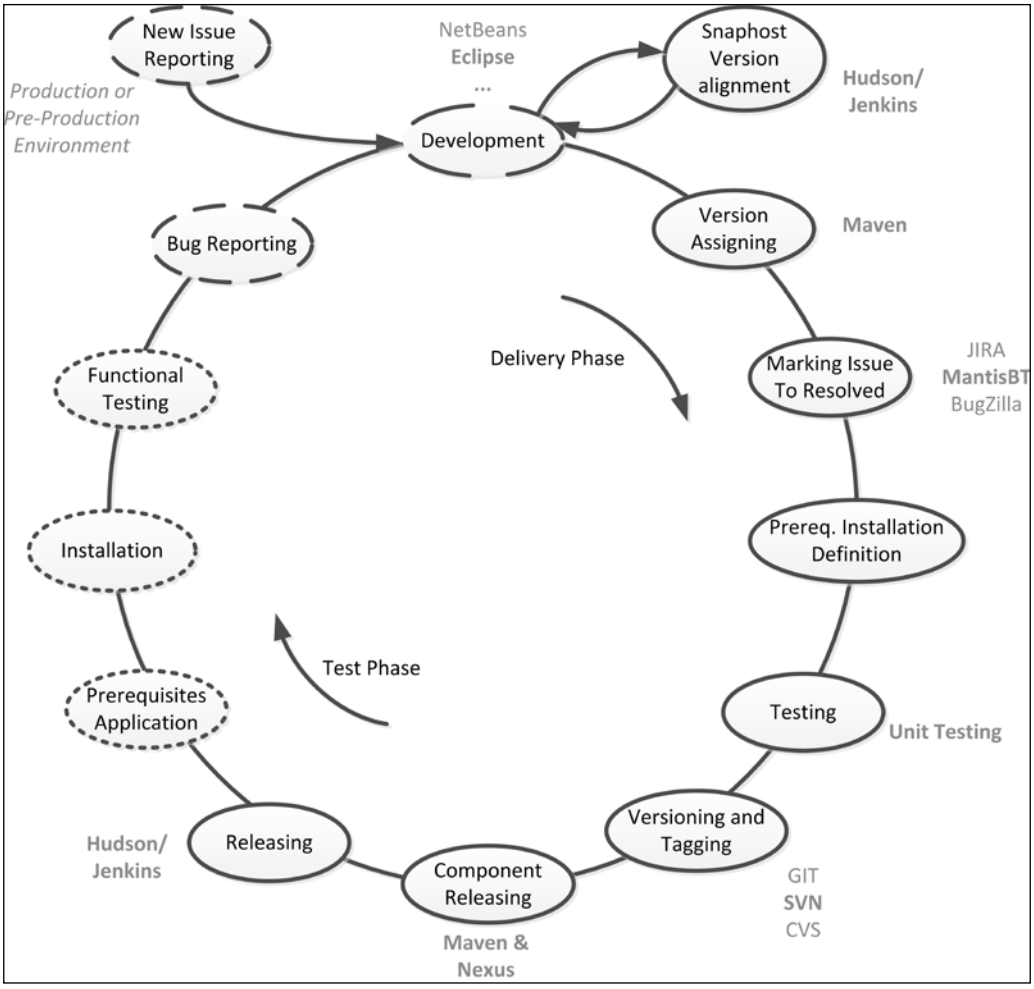
In particular, we will cover the following topics:

- Setting up the Maven repository
- Setting up the Software Configuration Management system
- Setting up the Software Version Management system
- Setting up the Build Integration tool
- Setting up the Test Integration tool
- Customizing CID

Key concepts of continuous integration and delivery

CID's flowchart is a set of repeatable steps implementing the software-releasing phase for a production-like environment. This phase aims to package the application and related components, assign the correct version number, archive the code, and provide the correct information to test and install the new patch.

The following diagram shows CID's flowchart. Following change requests due to bugs or new features, developers make the new changes to the source code and test the components on the development environment. Developers mark the issue as resolved and provide the needed information (prerequisites), such as configuration, the database changes, and preinstallation actions (Development Phase), to install the components.



Release lifecycle

Throughout CID's workflow, the Build Phase starts the following:

1. A version that is assigned to each component (artifact)
2. Source code that is tagged
3. Source code that is built and tested through a test platform (*Test Phase*)
4. The issues that are marked as released on the component version
5. Each component that is packaged
6. A patch with one or more new components that are notified

Finally, the test team installs and validates the patch using the following steps:

1. The prerequisites (the database changes or configuration) are applied
2. The components are installed (*Deploy Phase*)
3. According to the test plan, functional, integration, and regression tests are executed by the delivery team

These last steps are a part of the delivery (and validation) stage. In this book, we will not discuss about the final delivery stage as we assume that all released versions will be validated and promoted (for production) by a delivery team, rather than by an automation system. Therefore, we will refer to the keyword *release version* in the sense of *release candidate version for production*.

The previous diagram reports some technologies used in this chapter to implement CI's pipeline. In particular, **Nexus** is used to administer a shared and remote Maven repository, **Subversion** (**SVN**) to manage source code, **MantisBT** as a bug-tracking tool, **Hudson/Jenkins** to manage the build pipeline, and, obviously, **Maven** to manage the software's version and to compile and package the components. Different technologies, such as **GIT** or **JIRA** will shortly be discussed to complete the overall view. Finally, **Maven Ant** integration will be discussed to customize some basic steps.

The repository management server

On medium/large projects, it is common practice to control the external Maven repositories through an internal repository manager. Maven's central repository is very convenient for the users of Maven, but it is recommended to maintain your own repositories to ensure stability within your organization. Just as **Software Configuration Management** (**SCM**) tools are designed to manage source artifacts, repository managers have been designed to manage external dependencies and artifacts generated by your own build.

Consider an organization that has 100 developers split into different groups, each group working on a different part of the system without an easy way to share internal dependencies, and every group creating an ad hoc filesystem-based repository or building the system in its entirety so that dependencies are installed in every developer's local repository. Indeed, if your application is being continuously built and deployed using a tool such as Hudson (which we will discuss later), a developer can get a specific module from a large project build and not have to constantly compile the entire source at any given time.

Internal repository managers offer some advantages, such as:

- Sharing released artifacts with other developers or end users
- Caching software artifacts from remote repositories
- Applying fine-grained security and access policies
- Blocking or stabilizing some obsolete or not fully compliant or specific artifacts

Nexus Open Source or Professional is one of the most common repository managers (an alternative is **Artifactory**); Nexus has a very flexible infrastructure and allows us to configure multiple environments for different teams.

Installing Nexus

There are two distributions of Nexus: Nexus Open Source and Nexus Professional. For our purpose, we will use **Nexus Open Source** (referred to as Nexus for short), which is distributed under the GNU Affero General Public License Version 3. Nexus is a Java web application and can be downloaded from `http://www.sonatype.org/downloads/nexus-latest-bundle.tar.gz`.

Nexus can be run with a Jetty instance that runs on port 8081 by default, but should be installed on a different servlet container.

Installing Nexus on a Unix-based OS

We can install Nexus on a Unix-based OS by launching the following script:

```
$ cp nexus-oss-webapp-<version>-bundle.tgz /usr/local
$ cd /usr/local
$ sudo tar xvzf nexus-oss-webapp-<version>-bundle.tgz
$ ln -s nexus-oss-webapp-<version> nexus
$ /usr/local/bin/nexus start
```

Installing Nexus on Windows

Unzip the file content in `<NEXUS HOME>` and, with administrative privileges, run the installation program located in `<NEXUS HOME>/bin/jws`.

Customizing Nexus

Finally, the location of the work directory can be customized by altering the `nexus-work` property in `/usr/local/conf/nexus.properties` or `<NEXUS HOME>/conf/nexus.properties`.

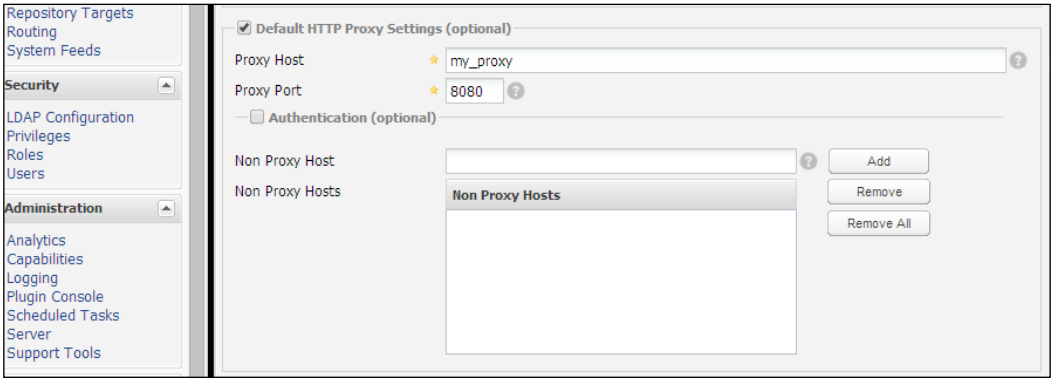
Testing the Nexus installation

To test the correct installation, we can open the browser to `http://<nexus_host>:8081/`.

The default username and password are `admin` and `admin123`, respectively.

Configuring the Nexus server

The Nexus server can be easily configured through the administrative console. On the left-hand side menu, navigate to **Administration** | **Server** to view the administrative settings console. Since Nexus has to access the remote repository, it is strongly encouraged to configure the proxy. On the administrative settings console (called nexus), enable the **Default HTTP Proxy** settings and configure them using your organization proxy (see the following screenshot):

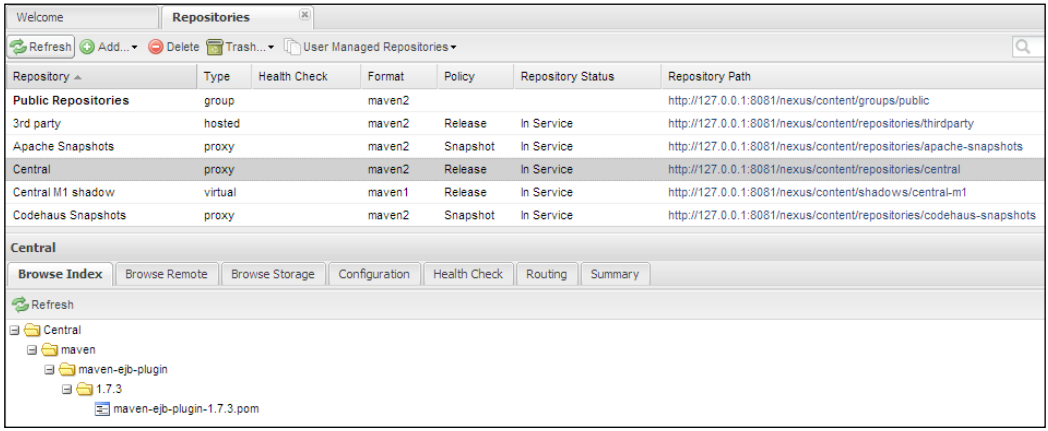


Nexus proxy settings

Testing the Nexus server

Since Nexus is configured to proxy the most common public repositories such as Central Repository or Apache Repository, to test the Internet connection of Nexus, we can download `maven-ejb-plugin` directly from our local Nexus instance using this link: `http://<nexus_host>:8081/nexus/content/repositories/central/maven/maven-ejb-plugin/1.7.3/maven-ejb-plugin-1.7.3.pom`.

The `maven-ejb-plugin` artifact is downloaded on the local Nexus working directory and is cached for further applications. We can browse the local cached repository (see the following screenshot) on the left-hand side menu; the **Repositories** item opens a list of repositories and we can use the **Browse Index** menu to explore the index of Nexus.



Nexus browse index

Managing repositories

When Nexus works properly, we can configure our `pom.xml` file to use it. We have to configure both the official and custom repositories to allow the download of artifacts.

Configuring official repositories

Maven is configured by default to use the official central repository, `http://repo1.maven.org`.

To change the declared default repository to point to the installed Nexus repository, we have to configure the `<repository>` and `<pluginRepository>` tags in the `pom.xml` file:

```
<repositories>
    <repository>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Proxied Maven Repository</name>
      <url> http://<nexus_host>:8081/nexus/content/
      repositories/central/
```

```
</url>
    </repository>
  </repositories>
<pluginRepositories>
    <pluginRepository>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Proxied Maven Plugin Repository</name>
<url>http://<nexus_host>:8081/nexus/content/repositories/central/</
url>
    </pluginRepository>
  </pluginRepositories>
```

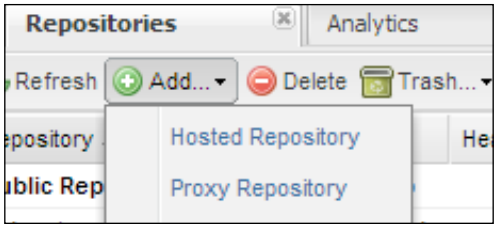
To add other proxy repositories, access the online Nexus console and, from the repositories view, navigate to **Add…** | **Hosted Repository** (see the following screenshot); you can then configure the repository. In this case, the repository has been declared as a *not snapshot* repository through the following tag:

```
[…]
<snapshots>
       <enabled>false</enabled>
</snapshots>
[…]
```

By default, the `snapshots` and `releases` tags are `true`, which means that the repository is enabled for both the snapshot and release artifacts (refer to *Chapter 2*, *Core Maven Concepts*).

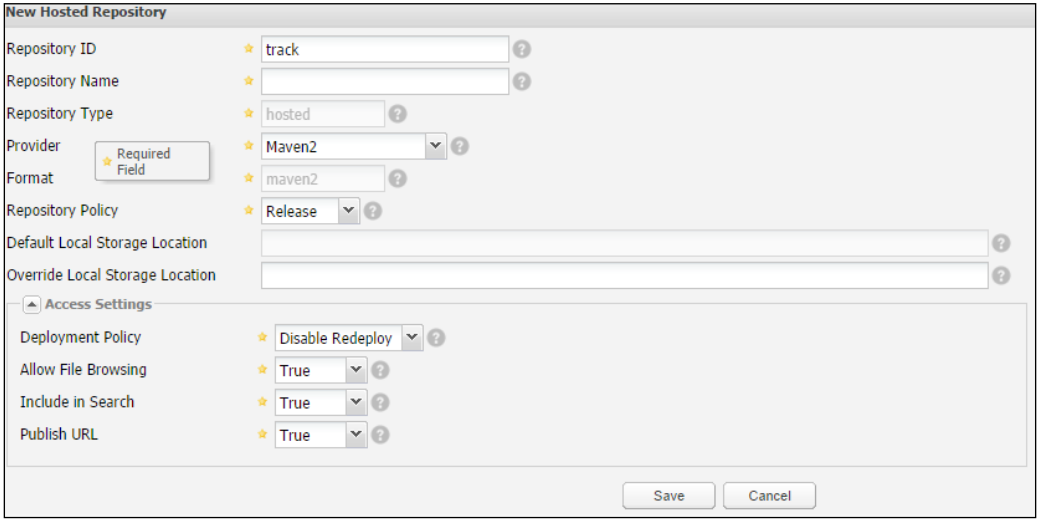
The User Managed Repository

The **User Managed Repository** is the most important functionality provided by Nexus. Configuring a User Managed Repository is quite simple, requiring only a couple of steps. From the online Nexus console, navigate to **Add…** | **Proxy Repository** (see the following screenshot):



Adding a new Nexus repository

Nexus requires the name and ID of the repository (see the following screenshot). The ID is the identifier of the repository; it will be part of the URL and cannot contain spaces.



The Nexus repository manager

By default, Nexus is configured with three user repositories:

- **Snapshots**: This is used to develop artifacts for your organization
- **Releases**: This is used for the released artifacts of your organization
- **3rd party**: This is used for the artifacts provided by other parties

For our purposes, these repositories are adequate.

Nexus access-level security

Nexus provides other important functionalities such as fine-grained access-level security. By default, Nexus allows an anonymous user to work with all the repositories. If your organization needs some specific policies for each group, the security view allows you to define an external LDAP server or add custom users to Nexus. Finally, through the privileges view, it is easy to grant or revoke some specific levels.

Integrating Ant

Apache Ant is one of the most important automation build tools. It is based on XML, it has been written in Java, and the concept is similar to *Make*.

Despite Maven and Ant being complementary, with the introduction of Maven, Ant has lost its importance and popularity; Ant was developed to compile, test, and package Java applications (or other applications), and Maven shares the same purpose. Ant provides a wide range of plugins such as Maven. To conclude, most Maven plugins are now outperforming Ant. Ant, however, is precious when we need to orchestrate some Maven executions or modulate Maven parameters.

Installing Ant

Ant is a command-line tool. To install Ant, you can download the binary from `http://ant.apache.org/bindownload.cgi`. Unzip the distribution file into a folder.

Set the `JAVA_HOME` environmental variable in your Java environment, set `Ant_HOME` to the directory you uncompressed Ant to, and add `${Ant_HOME}/bin` (Unix) or `%Ant_HOME%\bin` (Windows) to your PATH.

Run the following command for help to be displayed:

```
$ <Ant_HOME>/bin/ant --help
```

Understanding Ant

Ant uses a simple XML file called `build.xml` located in the running directory. The `build.xml` file must contain a set of tasks called `target`. The following code defines a global property called `dist` and a simple task creating the `dist` directory:

```
<project name="MyProject" default="dist-task" basedir=".">

  <!-- set global properties for this build -->
  <property name="dist-task"  location="dist"/>
```

```
    <target name="dist-task">
  <!-- Create the dist directory -->
      <mkdir dir="${dist}"/>
    </target>
  </project>
```

You can execute the `build.xml` file by running the following command:

```
$ <Ant_HOME>/bin/ant
```

You can also use the following command explicitly:

```
$ <Ant_HOME>/bin/ant –f build.xml dist-task
```

Ant custom tasks

Like Maven, Ant provides a set of APIs to create a custom plugin called `task`.

A list of the official Ant tasks can be found at `http://ant.apache.org/manual/tasksoverview.html`.

Through these tasks, we can package (WAR, JAR, and EAR), zip, compile, make a Javadoc, work with a filesystem, SSH and SFTP, work with CVS, read a properties file, and so on.

Other unofficial tasks have been developed by the community; the following points show the most popular Ant tasks contributed by the community:

- `if`, `then`, `else`: These are used to control Ant tasks
- `Svnant`: This is used to operate with Subversion
- `maven-ant`: This is used to integrate Maven with Ant

Maven-Ant integration

If you plan to integrate Maven and Ant, you probably need the **Maven-ant task**.

Download the plugin from `https://maven.apache.org/ant-tasks/download.html` and copy the JAR file into the `<Ant_HOME>/lib` directory.

The next step is to customize the `build.xml` file to use the plugin. The following code snippet is a minimal Ant project:

```
<project name="MyProject"
   default=" maven-task "
      basedir="."
```

```
        xmlns:artifact="antlib:org.apache.maven.artifact.ant">
    <target name="maven-task">
    <artifact:mvn pom="path/to/my-pom.xml"
         mavenHome="/path/to/maven-3.0.x">
          <arg value="install"/>
      </artifact:mvn>
    </target>
  </project>
```

The namespace definition of the `project` tag declares the plugin to Ant. The Maven task, `artifact:mvn`, executes the `install` goal of the `my-pom.xml` file. The `mavenHome` attribute is optional, but it is good practice to provide the Maven installation directory.

The `maven-ant` plugin supports other tasks. The following code snippet reads the `my-pom.xml` file and prints the POM version number.

```
<project name="MyProject"
   default=" maven-task "
      basedir="."
     xmlns:artifact="antlib:org.apache.maven.artifact.ant">

  <target name="maven-task">
     <artifact:pom id="mypom" file=" path/to/my-pom.xml" />
     <echo>The version is ${mypom.version}</echo>
  </target>
</project>.
```

Ant-Maven integration

If your organization has an old library of Ant procedures that are not easy to refactor, you can use the **Maven AntRun Plugin** to call Ant from Maven. The following code snippet calls the `my-task` task during the installation phase:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.7</version>
        <executions>
          <execution>
            <phase>install</phase>
            <configuration>
              <target>my-task</target>
            </configuration>
```

```
            <goals>
              <goal>run</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Despite this plugin appearing very useful, it is strongly encouraged to use the Maven plugin instead of a custom Ant task. It is good practice to prepare the Maven environment through Ant, without interfering with Maven.

SCM integration

SCM (**Software Configuration Management**), which is also referred to as **version control**, is the core of the project. If your organization uses an SCM system, there is an easy way to place information into the POM file. The following code snippet declares a local SVN repository:

```
[…]
   <scm>
    <connection>
        scm:svn:http://127.0.0.1/svn/my-project
     </connection>
    <developerConnection>
        scm:svn:http://127.0.0.1/svn/myproject
    </developerConnection>
    <tag>HEAD</tag>
    <url>http://127.0.0.1/websvn/my-project</url>
  </scm>
[…]
```

The Maven SCM plugin or site generation uses these SCM configurations in the POM file to perform some tasks and goals. Eclipse, through the M2Eclipse plugin, will materialize your Maven project from SCM using these configurations.

Maven SCM supports the repositories SVN, Git, CVS, Jazz, Bazaar, Mercurial, Perforce, StarTeam, and CM Synergy.

Maven SCM Plugin

Maven SCM Plugin is a vendor-independent plugin to gain access to SCM supported by Maven. It allows you to execute the basic versioning functionalities of check-in, check-out, update, tag, branch, add, and remove.

The following code snippet declares the plugin in the `pom.xml` file:

```
[…]
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-scm-plugin</artifactId>
      <version>1.9</version>
      <configuration>
        <connectionType>connection</connectionType>
      </configuration>
    </plugin>
  </plugins>
</build>
[…]
```

The `connectionType` tag declares the SCM connection to use.

You can check your configuration is correct by executing the **validation goal**:

```
$ mvn scm:validate
```

The result is as follows:

```
…
[INFO] connectionUrl scm connection string is valid.
[INFO] project.scm.connection scm connection string is valid.
[INFO] project.scm.developerConnection scm connection string is valid.
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
…
```

You can perform an SVN checkout by executing the **checkout goal**:

```
$ mvn scm:checkout
```

The result is as follows:

```
…
[INFO] Executing: svn checkout http://localhost/svn/example/ /ws
[INFO] Working directory: C:\wsxample\target
[DEBUG] Checked out revision 0.
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
…
```

If the checkout does not work, check that you have correctly installed the SCM client on the client machine, whether it's SVN, Git, and so on. The SCM Maven plugin works with the native client. The client must be accessible and the `bin` directory must be declared on the `PATH` settings.

Other useful available goals are as follows:

- `scm:branch -Dbranch=<name>`: This is used to branch the project
- `scm:checkin`: This is used to commit changes
- `scm:tag -Dbranch=<name>`: This is used to tag a certain revision
- `scm:list`: This is used to get the list of project files
- `scm:update`: This is used to update the working copy with the latest changes

Maven Release Plugin

Maven Release Plugin is the most important plugin in a CI project; the plugin allows us to tag the source code using the SCM information and change the POM release and snapshot version. It is based on two phases: the prepare phase and the perform/rollback phase. As the first step, the plugin asks the user for a specific version, prepares the release version, and makes the tag of the component. As the second step, Maven deploys (or rolls back) the changes.

The plugin was introduced after Maven 2, but it is strongly recommended to use Maven Version 3.0.4 or later. The plugin makes some assumptions:

- The start version must be a snapshot version (`SNAPSHOT`)
- The `allowTimestampedSnapshots` and `ignoreSnapshots` dependencies must be released or set to `true`
- No changes have to be committed

The plugin performs some actions:

- It changes the current version to a release version by removing the `SNAPSHOT` suffix
- It tags all the changes
- It packages and installs the current version on the local repository
- It sets the current version to a new snapshot version and commits it (by default, *a +1* will be added to the final version number)

To add the plugin, you have to configure the `pom.xml` file, as follows:

```
[…]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.5</version>
         <configuration>
   <tagBase>
http://127.0.0.1/websvn/my-project/tags
</tagBase>
   </configuration>
</plugin>
    </plugins>
   </build>
[…]
```

The next step is to launch Maven, as follows:

```
$ mvn release:clean release:prepare
```

Maven will ask for the new released version ID, tag name, and new development release (`SNAPSHOT`) ID:

```
What is SCM release tag or label for "Chp5Release Test"? (com.mycompany.
project
s:my-first-maven-project) my-first-maven-project-1.0: :
…
```

```
What is SCM release tag or label for "Chp5 Release Test"? (com.mycompany.
project
s:my-first-maven-project) my-first-maven-project-1.0: :
…
What is the new development version for "Chp5 Release Test"? (com.
mycompany.proj
ects:my-first-maven-project) 1.1-SNAPSHOT: :
…
[INFO] Release preparation complete.
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
```

Then, you can deploy the changes on a remote repository:

```
$ mvn release:perform
```

Maven will check out from SCM and release the project (`deploy` and `site-deploy`) on the remote repository. This step requires the definition of the repository location.

Deploying on the remote repository

The released software must be deployed on the remote (Nexus) repository in order to be accessible by every developer. Generally speaking, it is a good idea to create different repositories for each team of your organization on the remote server in order to control access to the artifacts; in our case, we use the default Nexus repositories. If your organization doesn't require strict access to some artifacts, the default Nexus installation is what you are looking for.

To publish the released plugin on Nexus, we have to define the release or snapshot repository, and we can call the `maven deploy` command:

```
$ mvn deploy
```

You can also use the following command if you use Maven Release Plugin:

```
$ mvn release:perform
```

In the `pom.xml` file, we have to define the repository manager (Nexus) URL:

```
[…]
<properties>
    <repo-id>nexus_releases</repo-id>
    <repo-name>Nexus Release</repo-name>
    <repo-url>
      http://localhost/nexus/content/repositories/releases/
    </repo-url>
</properties>
[…]
    <distributionManagement>
      <repository>
        <id>${repo-id}</id>
        <name>${repo-name}</name>
        <url>${repo-url}</url>
     </repository>
     </distributionManagement>
[…]
```

If the repository requires authentication, we have to define the credentials in the `settings.xml` file:

```
    <server>
      <id>nexus_releases</id>
      <username>admin</username>
      <password>admin123</password>
    </server>
```

Formally speaking, the example proposed here is not correct, and Maven offers two types of repository declarations: `repository` and `snapshotRepository`. A `snapshotRepository` declaration is used only for the snapshot artifact. A `repository` declaration is used for the release and snapshot artifacts if the snapshot repository has not been declared. To conclude, a more correct definition of the repositories is as follows:

```
[…]
  <distributionManagement>
<repository>
   <id>${repo-id}</id>
   <name>${repo-name}</name>
   <url>${repo-url}</url>
</repository>
```

```
<snapshotRepository>
   <id>${repo-snapshot-id}</id>
   <name>${repo-snapshot-name}</name>
   <url>${repo-snapshot-url}</url>
</snapshotRepository>
</distributionManagement>
[…]
```

In this case, Maven will deploy the release artifact in the release repository and snapshot artifact (version `x.x.x-SNAPSHOT`) in the snapshot repository.

Continuous Integration and Delivery with Hudson or Jenkins

Hudson CI, or the new fork Jenkins CI, is a Java web server application to automate and control the building of projects.

Hudson is very similar to Jenkins, and they differ only in the sense of licensing policies, hosting, and some minimal functionalities; some authors work with both systems, but in this book, we will refer only to Hudson.

Installing Hudson

You can download Hudson (formally known as Hudson Open Source Continuous Integration Server from the Eclipse Foundation) from `http://eclipse.org/hudson/download.php`.

The next step is to launch the embedded Jetty servlet container:

```
$ java -jar Hudson.<version>.war
```

Jenkins can be downloaded from `http://mirrors.jenkins-ci.org/war/latest/jenkins.war`.

To launch Jenkins or deploy a WAR file on a servlet container, execute the following command:

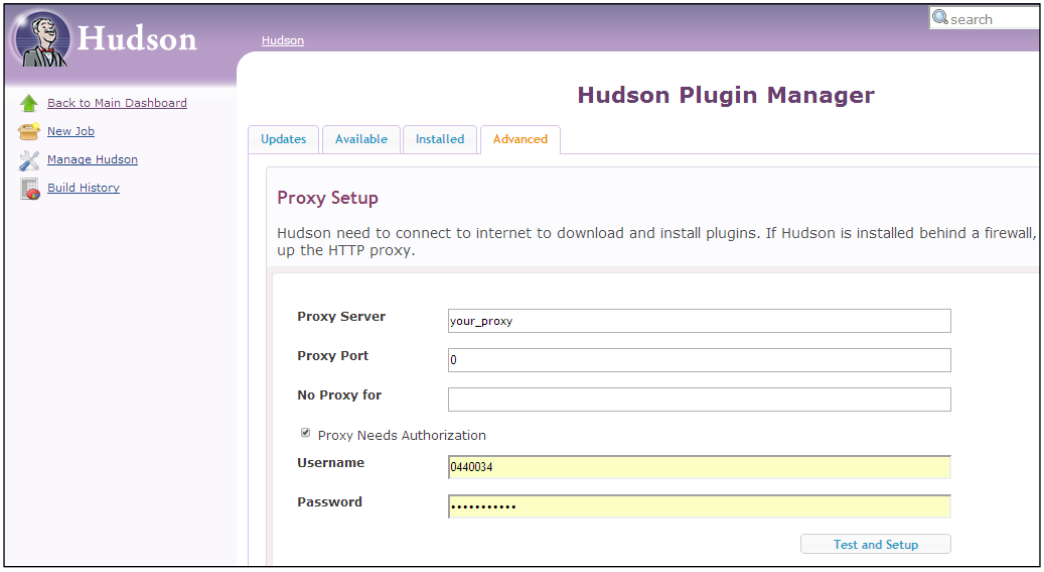
```
$ java -jar jenkins.war
```

Then, we can connect to `http://localhost:8080` and click on **Finish**.

Configuring Hudson

To configure Hudson (and Jenkins) CI, we need to accomplish the following steps:

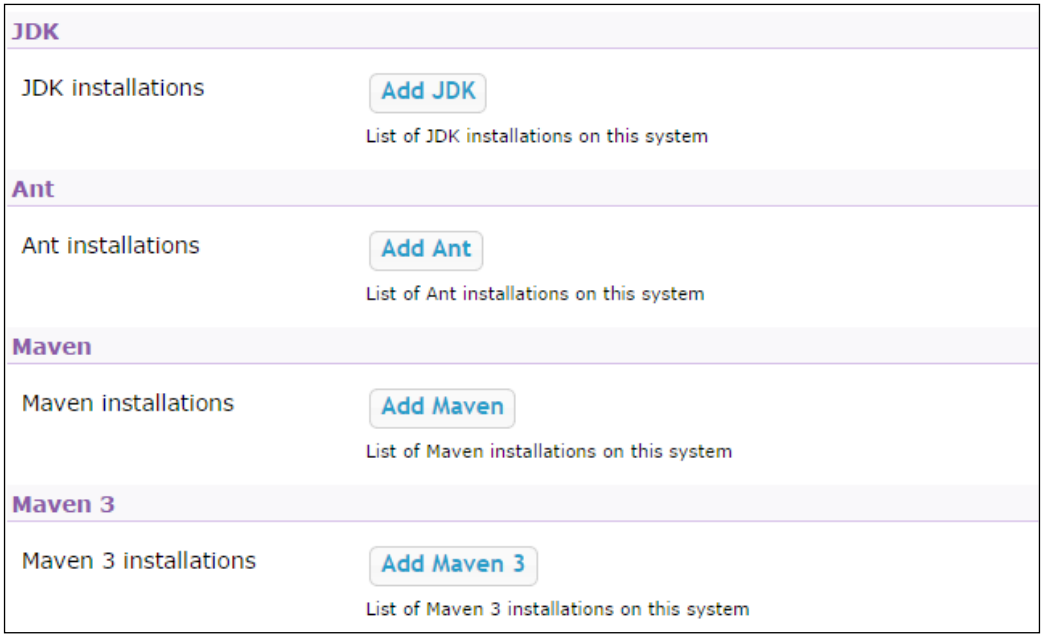
1. **Configuring the proxy**: Navigate to **Manage Hudson | Manage Plugins | Advanced**, and then configure the proxy server (see the following screenshot):



Hudson proxy configuration settings

2. **Installing plugins**: Navigate to **Manage Hudson | Manage Plugins**, then check the **Hudson Maven 3**, **Hudson Subversion Plugin**, **Maven2 Legacy Integration plugin**, and **Email-ext plugin** options if they are not yet installed, and click on **Install**.
3. **Configuring JDK, Maven, and Ant** (see the following screenshot): Navigate to **Manage Hudson | Configure System**, and then perform the following steps:
    - Click on **Add Jdk**, uncheck **Install automatically**, and provide the name and Java home
    - Click on **Add Maven3**, uncheck **Install automatically**, and provide the name and Maven home (refer to *Chapter 1*, *Maven and Its Philosophy*)

- Click on **Add Ant**, uncheck **Install automatically**, and provide the name and Ant home



Hudson installation configuration settings

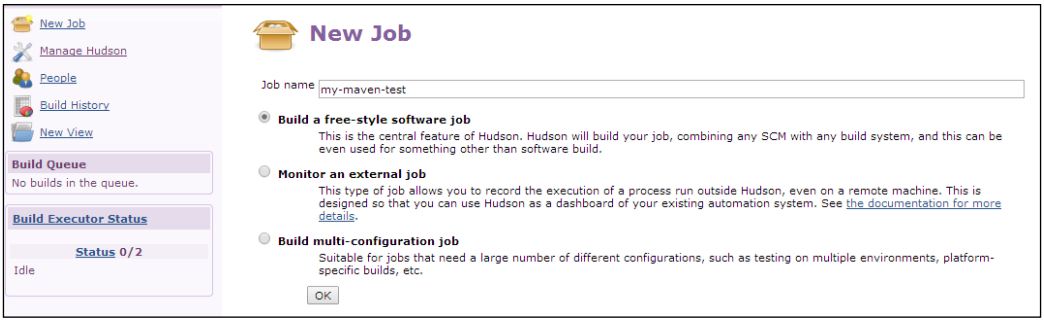
4. **Configuring security (optional)**: The security configuration is not strictly required for the purpose of this book, but it is strongly encouraged to enable security. Navigate to **Manage Hudson | Configure Security**, click on **Enable security**, and enable **Hudson's own user database** or configure **LDAP**.
5. Finally, restart Hudson.

Working with Hudson

Hudson (and Jenkins) is based on jobs; jobs can be scheduled or launched by users. Jobs can execute a shell script, a Maven goal, an Ant target, or a specific action defined by a plugin. In our examples, we will work with Maven and Maven-Ant jobs.

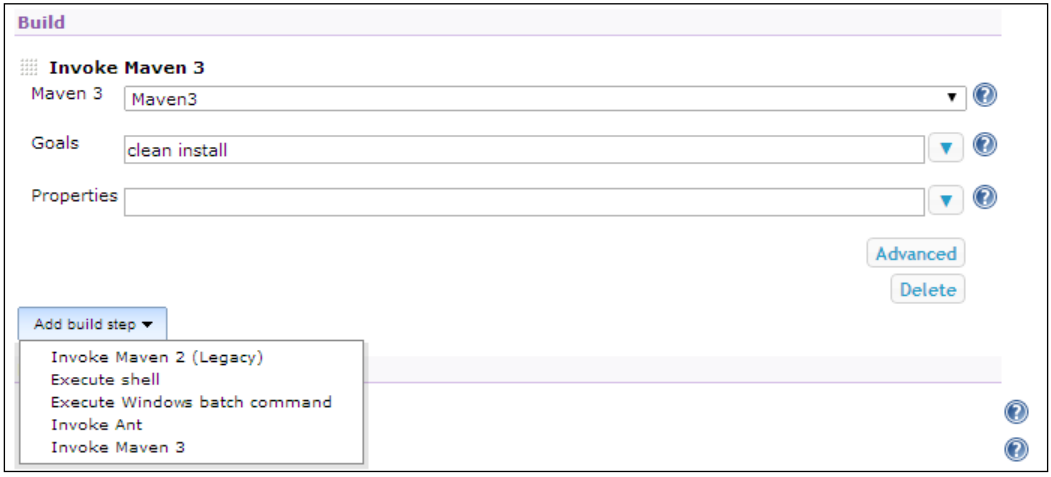
To create a new job, perform the following steps:

1. Click on the **New Job** option from the left-hand side menu, ensure that the free-style job is checked, and provide the job name (see the following screenshot):



Hudson's new job

2. On the **Configuration Job** panel, do the following:
    - Navigate to **Advanced Options** | **Custom Workspace** and provide the path of the workspace
    - From the **Build** section (see the following screenshot), navigate to **Add build step** | **Invoke Maven 3**



Hudson's invoke new step option

3. Click on **Save** at the end of the page.

In the workspace directory, enter the following `pom.xml` code:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>My Hudson Test</name>
  <groupId>com.mycompany.projects</groupId>
  <artifactId>my-first-maven-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
</project>
```

Finally, execute the job. Click on **build job** from the Hudson home screen or from the menu on the left-hand side. See the output console to monitor the build. Hudson launches Maven on the given workspace directory.

The following output will be shown:

```
Started by user anonymous
[INFO] Using Maven 3 installation: Maven3
…
[INFO] ------------------------------------------------------
[INFO] Building My Hudson Test 1.0-SNAPSHOT
[INFO] ------------------------------------------------------
…
[DEBUG] Waiting for process to finish
[DEBUG] Result: 0
Finished: SUCCESS
```

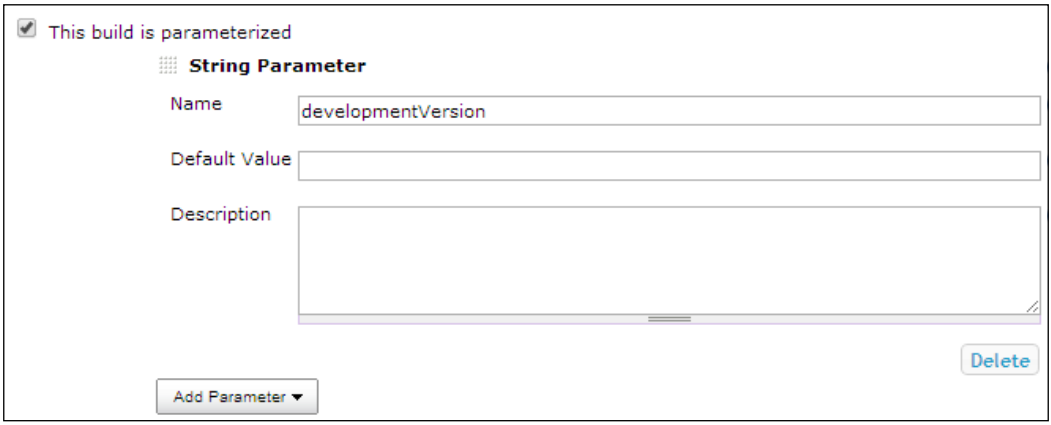
Working with Hudson interactively

The previous example is completely didactic since it does not introduce anything interesting. Hudson provides an easy way to customize a job by asking the user for some parameters. In this example, we will change the `pom.xml` version through Hudson and Maven Release Plugin.

From the Hudson home screen, select the job configured in the previous example (or duplicate it). From the left-hand side menu, perform the following steps:

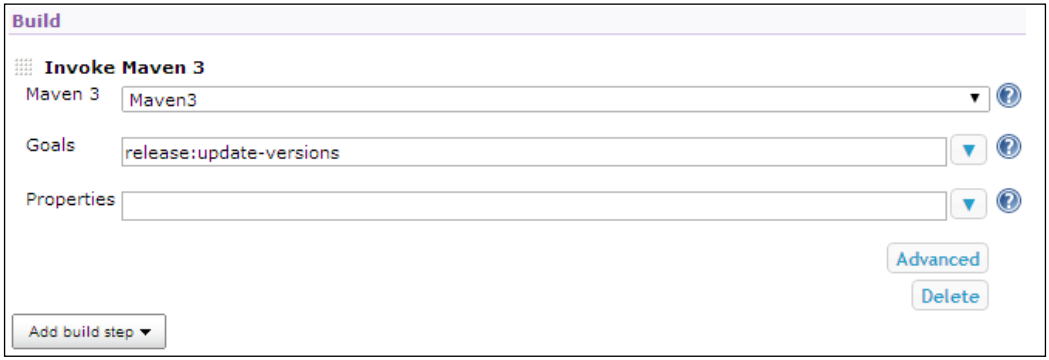
1. Click on **Configure Job**.
2. Enable **This build is parameterized**.
3. Navigate to **Add Parameter** | **String parameter**.

4. In the **Name** field, write `developmentVersion` (see the following screenshot):



Configuring the development version in Hudson

The next step is to change the Maven goal to `release:update-versions`. In the build section, change `clean install` to `release:update-versions` (see the following screenshot):



Configuring the Maven release step in Hudson

Finally, we need to add the `maven-release-plugin` to the `pom.xml` file. Open the `pom.xml` file located in your workspace and add the following lines:

```
[…]
<build>
<plugins>
  <plugin>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.5</version>
  <configuration>
```

```
      <releaseProfiles>release</releaseProfiles>
        <goals>deploy assembly:single</goals>
    </configuration>
    </plugin>
  </plugins>
  </build>
  […]
```

Now, we are ready to launch the job. Select the configured job from the Maven home screen. Click on the **Build Now** button.

Hudson will require the version number; provide the version number in the `x.x.x-SNAPSHOT` format.

The result appears like this:

```
[INFO] Transforming 'My Hudson Test 2'...
[INFO] Cleaning up after release...
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
```

Finally, reopen the `pom.xml` file located in your workspace; it will appear changed to the given version number.

All the Maven actions are performed with the batch mode option, `-B`. You do not need to specify it.

Maven-Hudson integration to deliver a new artifact

The following list shows the basic steps to release a new artifact (see the following screenshot):

1. Test the software (unit and integration test) and quality assurance.
2. Fix bugs on the issue-tracking system.
3. Assign a new version to the `pom.xml` file.
4. Make a new `svn` tag and create the package.
5. Assign a new `SNAPSHOT` version to the `pom.xml` file and commit changes to the development stage.

This work can be easily done by **Maven Release Plugin**.

This plugin can be easily integrated with Hudson/M2 Release Plugin. To install Hudson M2 Release Plugin, navigate to **Manage Hudson | Plugin Manager** from the available plugin list, look for the plugin and enable it, click on **Install**, and restart Hudson. We do not use this plugin.

Testing software automation

The building step is formally complete when all the unit tests are passed. **Unit tests** are typically written and run by developers to ensure that the code meets its design and functionalities, but it is good practice to rerun all the tests during the software release. To enable Maven to run the unit tests, we need to define Surefire Plugin (see *Chapter 3*, *Writing Plugins*) in `pom.xml`:

```
[…]
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.17</version>
 </plugin>
[…]
```

Surefire Plugin will launch all the unit tests (Junit or TestNG) defined by the developer.

From Version 2.7, Surefire Plugin will inspect the unit library (Junit version or TestNG), and it will activate the right provider automatically.

Sometimes, you might need to skip the execution of unit tests. There are two ways to skip tests:

- Launch Maven with the `skipTest` parameter. Use one of the following three commands:

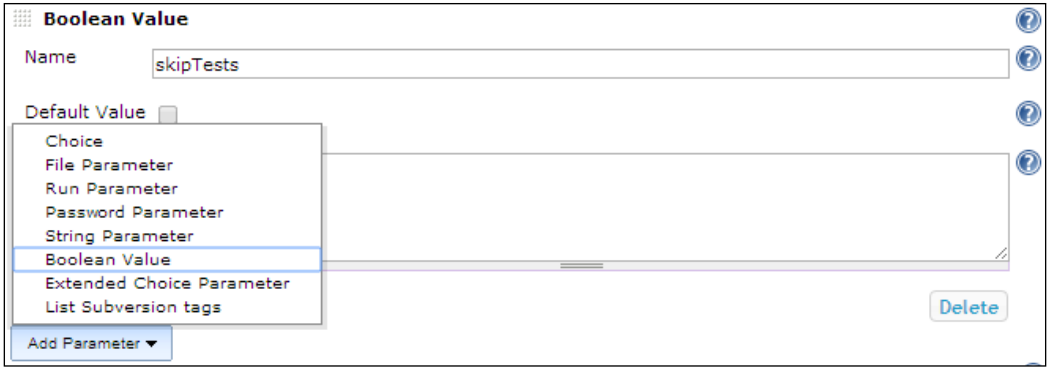
  ```
  $ mvn clean install -DskipTests
  $ mvn clean install -Dmaven.skip.test=true
  $ mvn test -Dmaven.skip.test=true
  ```

- Configure Surefire Plugin to skip all the tests:

```
[…]
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.17</version>
    <configuration>
  <skipTests>true</skipTests>
    </configuration>
 </plugin>
[…]
```

Finally, we can configure Hudson to ask the user about the test policy to apply. Perform the following steps from the left-hand side menu:

1. Click on **Configure Job**.
2. Enable **This build is parameterized**.
3. Click on **Add Parameter** | **Boolean parameter**.
4. In the **Name** field, write `skipTests` (see the following screenshot):



Hudson's configure skip test parameter

5. The next step is to use the following property:

```
[…]
    <configuration>
  <skipTests>${skipTests}</skipTests>
    </configuration>
[…]
```

Through these few steps, the user will control the execution of the unit test in order to enable or disable it. From a theoretical point of view, automatic unit test execution should be enabled and executed every time before any release; also, it is bad practice to allow the final user to control this step. Unfortunately, the devil is in the details, and this practice is the common practice in case of emergency.

A good compromise is to exclude some specific tests providing (through Hudson's parameter) a regular expression. A regular expression must be defined with the `%regex[expression]` syntax and defined into the `configuration` section:

```
[…]
    <configuration>
      <excludes>
         <exclude>%regex[.*[My|You].*Test.*]</exclude>
      </excludes>
    </configuration>
[…]
```

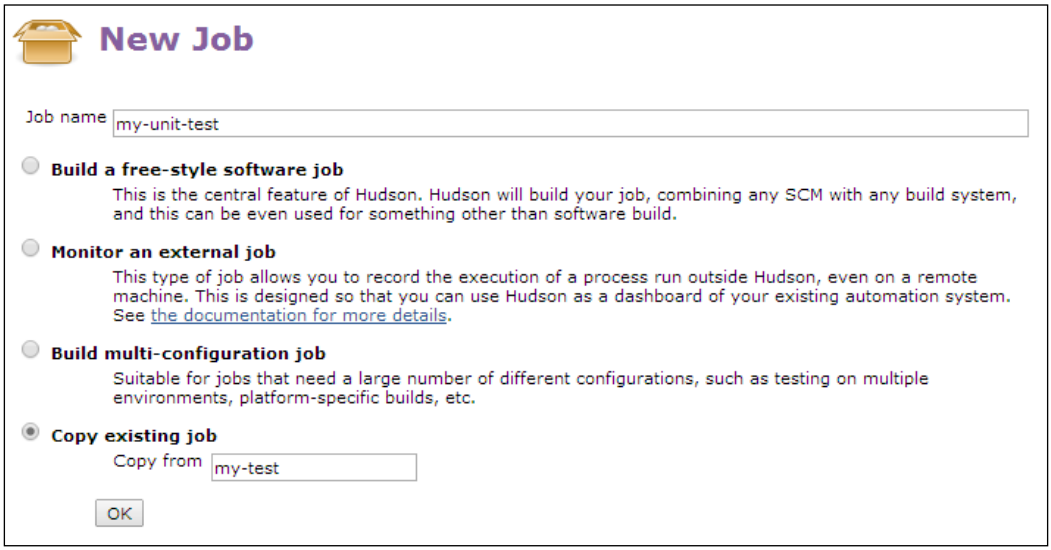
From our point of view, this solution is equivalent to the usage of the proposed `skipTests` parameter since the user can specify all the tests. A second common approach is to use a scheduled Hudson job reporting the status of the report periodically.

Scheduling a test reporting

One common way to grant a good level of software quality is to execute some common tasks periodically and publish these results on a site area. Hudson provides a large number of plugins to execute the unit (NUnit, JSUnit, JUnit, and TestNG) performance and functional tests and publish results on external (or internal) sites. Luckily, Surefire Plugin for JUnit tests is natively integrated, and it can be easily configured without any further installation.

A common practice is to define a new Hudson job scheduled every day (or trigger SCM changes).

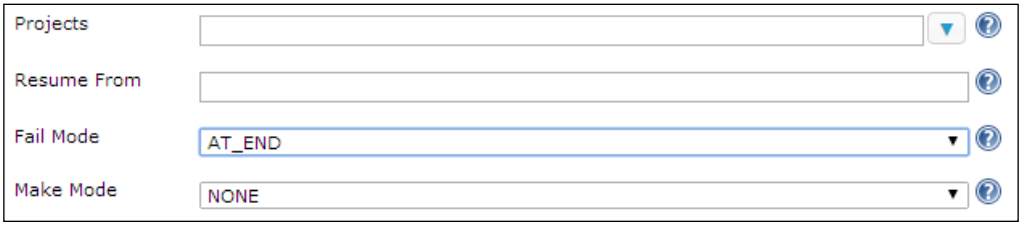
From the Hudson home screen, click on the **New Job** option from the left-hand side menu and copy settings from an existing job or create a new one (see the following screenshot):



Configuring a new unit test job in Hudson

Configure the workspace and Maven build process like the previous job, but set **Fail Mode** to `AT_END`:

1. From the **Build** section, click on **Advanced**.
2. Select `AT_END` for the **Fail Mode** parameter (see the following screenshot):

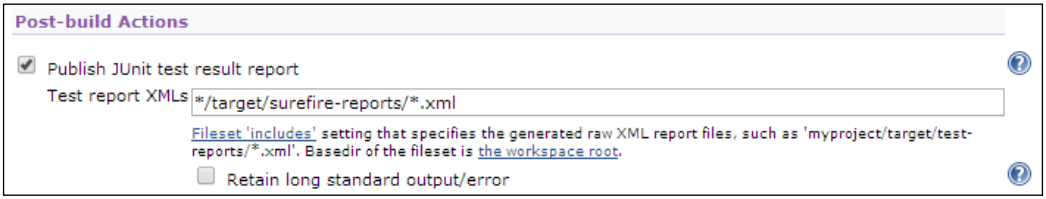


Hudson's configure build step

This parameter allows Maven to build all the modules, postponing any intermediate error up to the end of the entire build process.

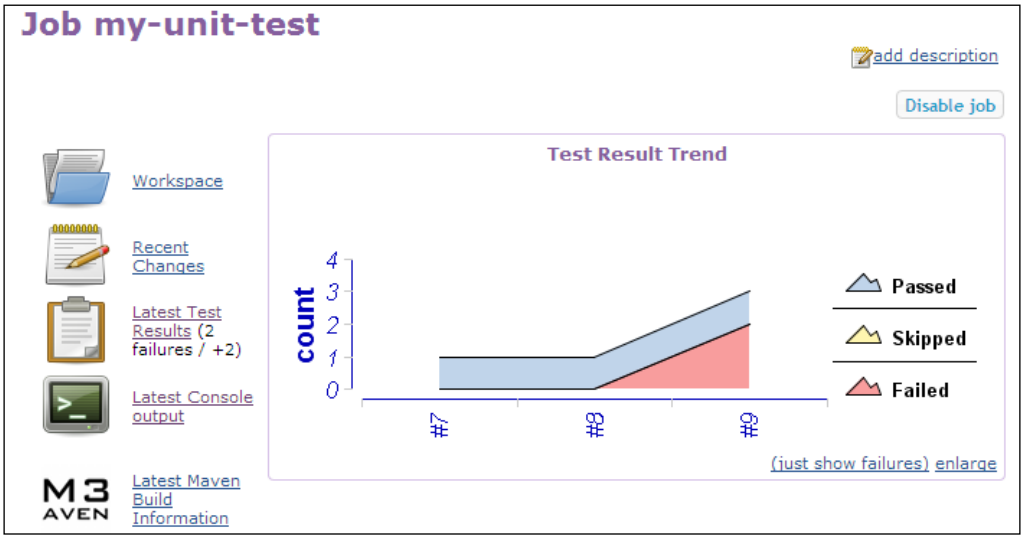
The next step is to configure Hudson to publish the Surefire Plugin result:

1. From the **Post-Build** section, enable **Publish JUnit test result report**.
2. In the `Test report XMLs` field, write `**/target/surefire-reports/*.xml` (see the following screenshot):



Hudson's configure Surefire report

These settings allow Hudson to parse the Surefire Plugin test result and publish it in the test result section (or in a Hudson dashboard). To see the test result trend, you can execute the job we just configured and see the latest test result (see the following screenshot):



Hudson's test report

Finally, we can schedule the job to run every day at midnight:

1. From the **Build-Triggers** section, enable **Build periodically**.
2. In the **Schedule** field, write `@midnight`.

Hudson will execute the job every day at midnight and publish the test results. Alternatively, from the **From Build-Triggers** section, you can poll SCM for changes.

Aligning the shared development environment

If the test task just described does not fail, it is a good idea to deploy the nightly build software on a shared environment. Unfortunately, there is a wide range of available plugins for all the platforms, and it depends on the strategy adopted (Weblogic, JBOSS, GlassFish, Tomcat, Job scheduling, Linux, and so on); therefore, it is not possible to suggest a unique strategy. In *Chapter 2*, *Core Maven Concepts*, we introduced the deployment plugin for JBOSS. The following table provides some basic Maven plugins that can be adopted to perform this step:

Plugin	Description
`weblogic-maven-plugin`	This is the Weblogic deploy plugin. It is not available in the central repository.
`jboss-as-maven-plugin`	This is the JBoss deploy plugin.
`tomcat7-maven-plugin`	This is the Tomcat 7 deploy WAR plugin.
`wagon-ssh-external`	This plugin is used to deploy on an SSH external host.
`maven-glassfish-plugin`	This plugin is used to deploy on a GlassFish server.

When the plugin has been configured, we can add a Maven 3 build step after the `clean install` task from the previous job:

1. Click on **Configure Job**.
2. From the **Build step** section, add a new Maven 3 step.
3. In the **goal** field, write `jboss-as:deploy`.

Alternatively, you can use the dedicated Hudson plugins.

Integration tests

Now, the environment is ready to perform the integration tests. Integration tests can be configured like the unit tests of a dedicated project (which is already deployed) using **Maven Failsafe Plugin**.

To install Failsafe Plugin, configure `pom.xml` as follows:

```
<project>
  […]

  <plugins>
      <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.17</version>
    </plugin>
        </plugins>

      [...]
    </project>
```

When the plugin has been configured, we can add a Maven 3 build step after the `jboss:deploy` task from the previous job:

1. Click on **Configure Job**.
2. From the **Build step** section, add a new Maven 3 step.
3. In the `goal` field, write `failsafe:integration-test failsafe:verify`.

Since we are using Surefire Plugin and Failsafe Plugin together, we have to make sure to use the correct naming convention to make it easier to identify which tests are being executed by which plugin. Failsafe Plugin will look for tests with the `**/IT*.java`, `**/*IT.java`, and `**/*ITCase.java` patterns by default, but, since it is a branch of Surefire Plugin, it supports the `include` and `exclude` syntaxes.

Alternatively, you can develop some dedicated unit tests on each artifact and activate them through Maven profiles.

Static code analysis tools (FindBugs)

The previous job can be refined and improved by integrating a static code analysis tool. Static code analysis tools inspect the code for potential bugs, missing code styles, high complexities, missing tests, or wrong patterns. Static code analysis tools require the definition of a specific Maven plugin integrated with a Hudson Plugin. The following table reports a short list of the most common static analysis tools:

Maven Plugin	Hudson Plugin	Description
`findbugs-maven-plugin`	`FindBugs-Plug-in`	FindBugs analyzes Java code from 1.0 to 1.7.
`maven-checkstyle-plugin`	`Checkstyle Plug-in`	Checkstyle checks Java code to adhere to a coding standard.

Maven Plugin	Hudson Plugin	Description
`maven-pmd-plugin`	`PMD Plug-in`	PMD finds common programming flaws such as unused variables, empty catch blocks, and unnecessary object creation. It supports Java, JavaScript, XML, and XSL.
`cobertura-maven-plugin`	`Cobertura Plug-In`	Cobertura checks the unit test coverage.

Other commercial-related products integrated with Hudson are JIRA, Clover, and Sonar.

FindBugs is one of the most interesting and complete tools. Since a complete discussion on all the plugins is outside the scope of this book, we can discover the power of the code analysis tools by only experiencing FindBugs.

Firstly, we need to install a Hudson Plugin by performing the following steps:

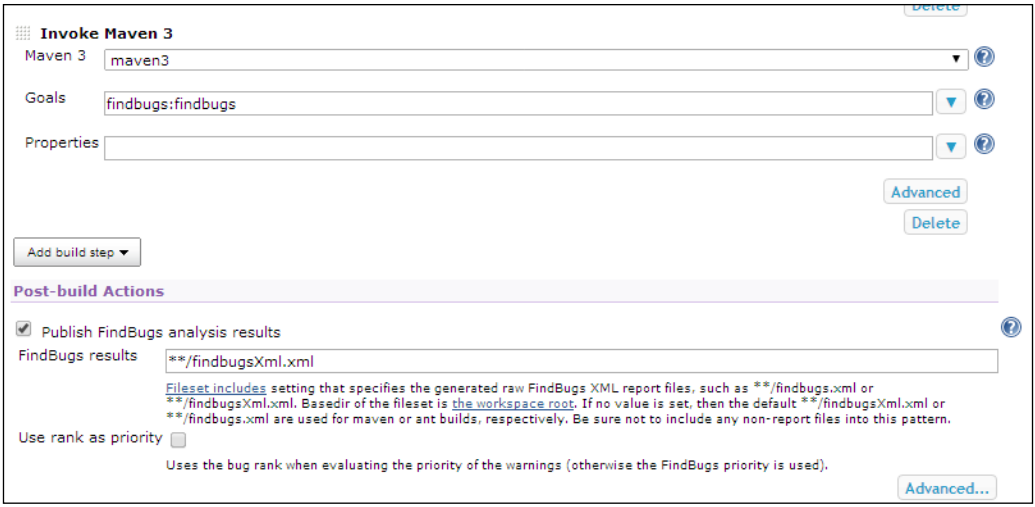
1. Navigate to **Manage Hudson | Manage Plugins**, check the **FindBugs Plug-in** option if not yet installed, and click on **Install**.
2. In the `pom.xml` file, define the FindBugs Plugin as follows:

```
<project>
  […]
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>findbugs-maven-plugin</artifactId>
        <version>2.5.4</version>
<configuration>
   <xmlOutput>true</xmlOutput>
   <threshold>Default</threshold>
</configuration>
      </plugin>
    </plugins>
  </reporting>
  […]
</project>
```

The next step is to configure the previous job:

1. Click on **Configure Job**.
2. From the **Build step** section, add a new Maven 3 step.

3. In the **Goals** field, write `site` (see the following screenshot).
4. From the **Post-build Actions** section, enable **Publish findbugs analysis tools**.
5. In the **FindBugs results** field, write `**/findbugsXml.xml` (see the following screenshot):



Hudson's configure FindBugs report

The following lines should be displayed:

```
[FINDBUGS] Collecting findbugs analysis files...
[FINDBUGS] Collecting findbugs analysis files...
[FINDBUGS] Finding all files that match the pattern **/findbugsXml.xml
[FINDBUGS] Parsing 1 files in …
[FINDBUGS] Successfully parsed file …\findbugsXml.xml of module  with … warnings.
```

To see the FindBugs report, you can execute the job just configured. To see the latest result, click on **FindBugs Warnings** from the left-hand side menu.

Bug fixing

In the previous sections, we exposed the basic concepts and steps to accomplish CI's flowchart in the form of versioning, tagging, unit testing, and releasing. To complete the release phase, however, we need to integrate the bug-reporting system in order to mark the issues fixed by the current release as released/resolved.

Hudson (or Jenkins) integrates several plugins for popular bug-tracking systems such as JIRA, MantisBT, and Bugzilla.

Unfortunately, these plugins are not under continuous development and support only old versions.

In *Chapter 3*, *Writing Plugins*, we learned how to develop a plugin to update MySQL DB. We will now extend this plugin to integrate MantisBT.

A case study with MantisBT

MantisBT is a popular bug-tracking system for multiteam environments. Generally speaking, it is based on an internal relational DB (typically, MySQL) and some PHP pages.

When a new component is released, it is a good idea to mark all resolved bugs as released and assign the version numbers; this step concludes the final stage of the release process before proceeding to the functional tests.

Firstly, we need to configure the plugin developed in *Chapter 3*, *Writing Plugins*. In the `pom.xml` file of the main project, configure `mantis-maven-plugin` as follows:

```
<project>
   […]
   <build>
    <plugins>
     <plugin>
  <artifactId>mantis-maven-plugin</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <configuration>        <databaseUrl>${dbUrl}</databaseUrl>
<jdbcDriver>${dbDriver}</jdbcDriver>
  <projectId>${project.artifactId}</projectId>  <dbUserName>${usr}</
dbUserName>
  <dbPassword>${pswd}</dbPassword>
  <databaseName>${dbName}</databaseName>
  </configuration>
     </plugin>
   </plugins>
   </build>
   […]
</project>
```

Parameters of the DB should be defined in the `properties` section. The plugin developed in *Chapter 3, Writing Plugins*, is a generic plugin that does not need database access implementation, but for our purpose, we need a utility class to work with the DB.

The following code provides a short implementation of the proposed utility:

```
public class MySqlAccess {
  /* Parameters are : bug_id, tag_id */
  private final String INSERT_INTO_MANTIS_BUG_TAG_TABLE = "INSERT INTO
mantis_bug_tag_table(?, ?)";

  /* Parameters are : projectId   */
  private final String SELECT_FROM_MANTIS_BUG_TABLE = "SELECT id,
tagId FROM mantis_bug_table where status=80 and project_id = ?";

  /* Parameters are : versionName, projectId */
  private final String UPDATE_MANTIS_BUG_TABLE = "UPDATE mantis_bug_
table SET status=85, fixed_in_version=? where status=80 and project_id
= ?";

  /* Parameters are : versionName, projectId */
  private final String UPDATE_MANTIS_BUG_TABLE_2 = "UPDATE mantis_bug_
table SET status=90, fixed_in_version= ? where status!=90 and project_
id = ? and (category_id=34 OR category_id=35)";

  private Connection connect = null;
  private Statement statement = null;
  private PreparedStatement preparedStatement = null;
  private ResultSet resultSet = null;
  private String userName;
  private String password;
  private String dbUrl;
  private String dbName;
  private String jdbcDriver;

  public MySqlAccess(String userName,
String password, String dbUrl, String dbName, String jdbcDriver)
throws SQLException, ClassNotFoundException {
    this.userName = userName;
    this.password = password;
    this.dbUrl = dbUrl;
    this.dbName = dbName;
```

```
    this.jdbcDriver = jdbcDriver;


          Class.forName(jdbcDriver);
    this.connect = DriverManager.getConnection("jdbc:mysql://" + dbUrl
          + "/" + dbName + "?" + "user=" + userName + " &password="+
password);
  }

  public void updateStatus(String versionName, String projectId)
throws SQLException {

          preparedStatement = connect.prepareStatement(UPDATE_MANTIS_
BUG_TABLE);

    preparedStatement.setString(1, "versionName");
    preparedStatement.setString(2, "projectId");
    preparedStatement.executeUpdate();

          preparedStatement = connect.prepareStatement(UPDATE_MANTIS_
BUG_TABLE_2);

    preparedStatement.setString(1, "versionName");
    preparedStatement.setString(2, "projectId");
    preparedStatement.executeUpdate();
  }


  }
```

With these settings, we just implemented a simple way to finalize the release process. Keep in mind that a CI process should be easy, fast, and automatic; the basic idea is to shorten the distance between the development and test with continuous and automatic build and release cycles.

A more realistic case – the transportation project

It's time to bring together all the bricks just discussed. In *Chapter 1*, *Maven and Its Philosophy*, we introduced a real project called the transportation project. Here, we apply a CI process to the proposed project.

The following list shows the organization of the proposed project:

- `transportation-acq-ear`
    - `transportation-acq-ejb`
    - `transportation-acq-war`
    - `transportation-common-jar`
- `transportation-reporting-ear`
    - `transportation-reporting-war`
    - `transportation-reporting-ejb`
    - `transportation-common-jar`
- `transportation-statistics-batch-jar`

The project is based on two `ear` components sharing a common `jar` library and a single `batch` component. All the artifacts have the same `pom.xml` parent file. In this case, we will ignore the dependencies from third-party libraries (JEE, Spring, and so on), since these libraries are not active components during the release process; in other words, we do not need to release or assign a version to third-party libraries.

The proposed project elicits some issues:

- The developer requires to release each component (`transportation-acq-ear`, `transportation-reporting-ear`, or `transportation-statistics-batch-jar`) with no other components: how to choose the project to build using the CI tool?
- Two components share a common component (`transportation-common-jar`): how to assign the version number and release it?
- The single `batch` component, `transportation-statistics-batch-jar`, can be released without any dependencies: how to assign the version number and how to release it?
- Two components (`transportation-acq-ear` and `transportation-reporting-ear`) have three subcomponents: how to assign the version number?

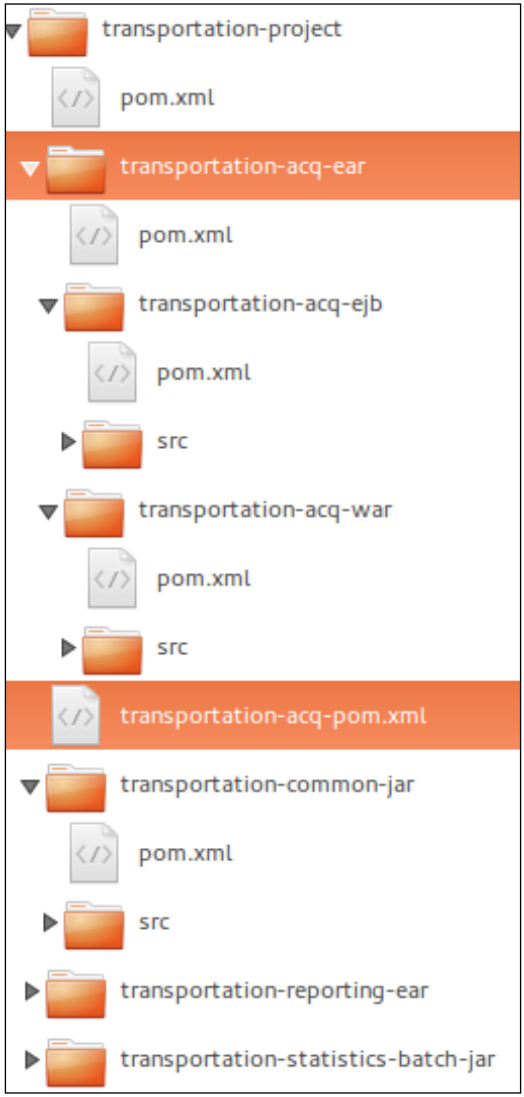
Choosing the component to build

When a large project consists of many components/modules such as a web API, a client, and some libraries, not everything has to be re-released every time a component changes. Maven allows the user to specify an alternate `pom.xml` file through the `-f` option:

```
$ mvn -f my-pom.xml
```

Therefore, on a multimodule project, it is common practice to define an aggregator `pom.xml` file that lists the modules to be executed as a group.

The following `pom.xml` file defines the POM aggregator (called `transportation-acq.xml`) for the multimodule project, `transportation-acq-ear`. We assume that the `transportation-acq-ear`, `transportation-acq-ejb`, and `transportation-acq-war` projects have been included on a subdirectory called `transportation-acq` (the name must be coherent with the name of the aggregator's artifact ID). Therefore, the final directory structure is as follows:



Configure the `pom.xml` file of the project as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<parent>
  <groupId>com.packt.examples</groupId>
  <artifactId>transportation-project</artifactId>
  <version>0.0.1 </version>
</parent>
<artifactId>transportation-acq</artifactId>
<version>0.0.1-SNAPHOST</version>
<packaging>pom</packaging>

<modules>
<module>transportation-acq/transportation-acq-ejb</module>
<module>transportation-acq/transportation-acq-ear</module>
<module>transportation-acq/transportation-acq-war</module>

</modules>
</project>
```

We can run the following command to build only the `transportation-acq-ear` and its related modules:

```
$ mvn -f transportation-acq.xml clean install deploy
```

Order is not important; Maven will sort the modules such that dependencies will always be built before the dependent modules.

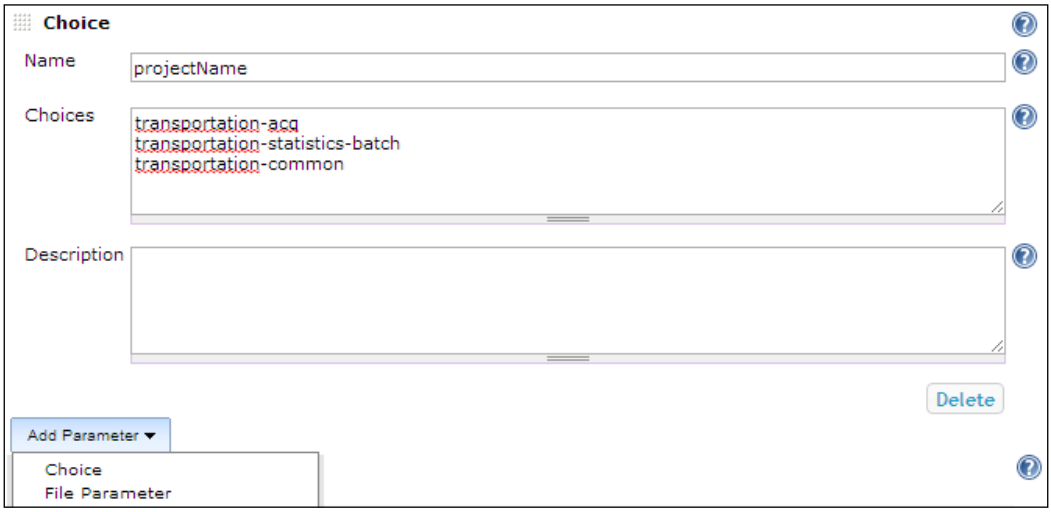
You can build specific reactors through the `-pl -am` options.

The next step is to configure Hudson to ask the user which project aggregator POM should be built on.

From the Hudson home screen, click on the **New Job** option from the left-hand side menu and copy settings from an existing job or create a new one. Perform the following steps:

1. Click on **Configure Job**.
2. Navigate to **Add Parameter** | **Choice parameter**.

3. In the **Name** field, write `projectName`.
4. In the **Choices** field, write the list of projects (see the following screenshot):



Configuring artifact parameters in Hudson

Finally, we have to configure the build section. In the build section, perform the following steps:

1. Click on **Advanced**.
2. In the **POM File** field, write `${projectName}/pom.xml`.

Hudson is ready to ask the user which module to build.

The `${projectName}/pom.xml` parameter is a simple convention adopted by authors, but you can adopt your own convention. The aggregator filename and the name configured on Hudson, however, must be coherent.

Preparing the version of a multimodule component

The release process of a single component has been extensively described in the previous sections. In a multimodule component, however, dependencies to common artifacts and module versioning require a different configuration. Indeed, even if the submodules are versioned with the same version number of the EAR project, common projects should follow a different lifecycle.

In the transportation project, the proposed strategy will require the team of developers to release common projects (`transportation-common-jar`) such as a unitary component, and assign the same version of the EAR project to each submodule (`transportation-acq-ejb`, `transportation-acq-war`, `transportation-reporting-ejb`, or `transportation-reporting-war`).

This mixed strategy is a good compromise between the release automation and fine-grained controls of a component's versions. Alternatively, developers can set the version to each component manually and use Maven SCM Plugin only to tag the code. This way, however, is not exactly the idea of CI, where all processes (not developing) must be as automatic and easy as possible.

We configured Maven Release Plugin as explained in the previous sections, and we can execute the following command:

```
$ mvn --batch-mode \
-DallowTimestampedSnapshots=true -DignoreSnapshots=true \
-f transportation-acq.xml \
clean install release:clean release:prepare
```

The outcome is as follows:

```
[…]
[INFO] Transforming 'transportation-acq-ejb'...
[INFO] Transforming 'transportation-acq-war'...
[INFO] Transforming 'transportation-acq-ear'...
[…]
[INFO] Tagging release with the label transportation-acq-pom-0.0...
[…]
```

The `-batch-mode` option will force Maven to assign the version number automatically, without any further user interaction. All the submodules will be released with the same version, upgrading the final development (`SNAPSHOT`) version with *a +1* on the latest number.

We can force the version number, as follows:

```
$ mvn --batch-mode -f transportation-acq.xml \
-DallowTimestampedSnapshots=true -DignoreSnapshots=true \
-DreleaseVersion=0.0.2 \
clean install release:clean release:prepare
```

The project will be delivered with the provided versions.

Obviously, the plugin allows us to provide the number for each component using the following syntax:

```
-DdevelopmentVersion=1.3-SNAPSHOT
```

We can also use the following syntax:

```
-Dproject.dev.groupId:projectName=1.3-SNAPSHOT
```

Alternatively, use the following syntax:

```
-Dproject.rel.groupId:projectName=1.3
```

However, we prefer a more easy strategy.

Configuring Hudson

Hudson can be easily configured by asking the user for the `releaseVersion` parameter.

Perform the following steps from the left-hand side menu:

1. Click on **Configure Job**.
2. Navigate to **Add Parameter** | **String parameter**.
3. In the **Name** field, write `releaseVersion`.

Finally, configure a new Maven build step, as follows:

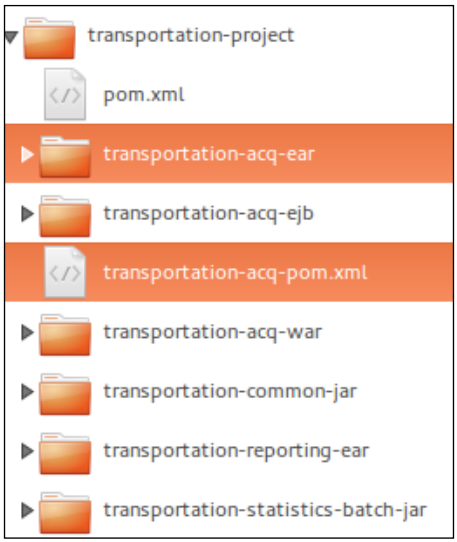
1. From the **Build** section, navigate to **Add build Step** | **Invoke Maven 3**.
2. Replace `clean install` with `release:clean release:prepare`.
3. From the **Build** section, navigate to **Add build Step** | **Invoke Maven 3**.
4. Replace `clean install` with `release:perform`.

Hudson will show the following output:

```
[…]
[artifact:mvn] [INFO] Building transportation-acq 0.0.1-SNAPSHOT
[…]
[artifact:mvn] [INFO] Building transportation-acq 0.0.1
[…]
[artifact:mvn] [INFO] Tagging release with the label transportation-acq
-0.0.1
[…]
[artifact:mvn] [INFO] Transforming 'transportation-acq'...
[artifact:mvn] [INFO] Not removing release POMs
[artifact:mvn] [INFO] Checking in modified POMs...
[…]
```

Preparing the version of a multimodule with a flat structure (an alternative way)

In some cases, it is not acceptable to adopt a hierarchical structure of folders, and we might therefore prefer a flat environment, as is shown in the following screenshot:



In this case, the current version of Maven Release Plugin will not tag the submodules.

In the previous sections, we introduced the Ant Maven task. We can use this task to loop over modules in order to release artifact by artifact. The Ant script works with the For task, and the Ant Maven task will cycle between modules and release each artifact.

The script reads the aggregator POM, `transportation-acq.xml`, through the `Xmlproperty` task.

In the root directory of your project, create a `build.xml` file with the following content:

```
<project name="maven-release" default="release" basedir="."
xmlns:artifact="antlib:org.apache.maven.artifact.ant">

<taskdef name="for" classname="net.sf.antcontrib.logic.ForTask"
classpath="${basedir}/lib/ant-contrib-1.0b3.jar" />

<typedef resource="org/apache/maven/artifact/ant/antlib.xml"
uri="antlib:org.apache.maven.artifact.ant"
path="${basedir}/lib/maven-ant-tasks-2.1.3.jar" />
```

```
    <target name="release">

      <xmlproperty file="${basedir}/${projectName}.xml"
         collapseAttributes="true"/>

     <for param="line"
        list="${project.modules.module}"
        delimiter=",">
        <sequential>
        <echo>@{line}</echo>
       <artifact:mvn pom="${basedir}/@{line}/pom.xml">
        <arg value="-DallowTimestampedSnapshots=true" />
        <arg value="-DignoreSnapshots=true" />
        <arg value="release:clean"/>
        <arg value="release:prepare"/>

       </artifact:mvn>
    </sequential>
    </for>
    </target>
  </project>
```

In Hudson, we need to replace the Maven release step with an Ant step and configure a new Ant build step:

1. From the **Build** section, navigate to **Add build step** | **Invoke Ant**.
2. In the **Targets** field, write `release`.
3. Replace `Invoke Maven 3` with `release:perform`.

Hudson will show the following output:

```
[…]
[artifact:mvn] [INFO] Building transportation-acq-ejb 0.0.1-SNAPSHOT
[…]
[artifact:mvn] [INFO] Building transportation-acq-ejb 0.0.1
[…]
[artifact:mvn] [INFO] Tagging release with the label transportation-acq-
ejb-0.0.1
[…]
[artifact:mvn] [INFO] Transforming 'transportation-acq-ejb'...
[artifact:mvn] [INFO] Not removing release POMs
[artifact:mvn] [INFO] Checking in modified POMs...
[…]
```

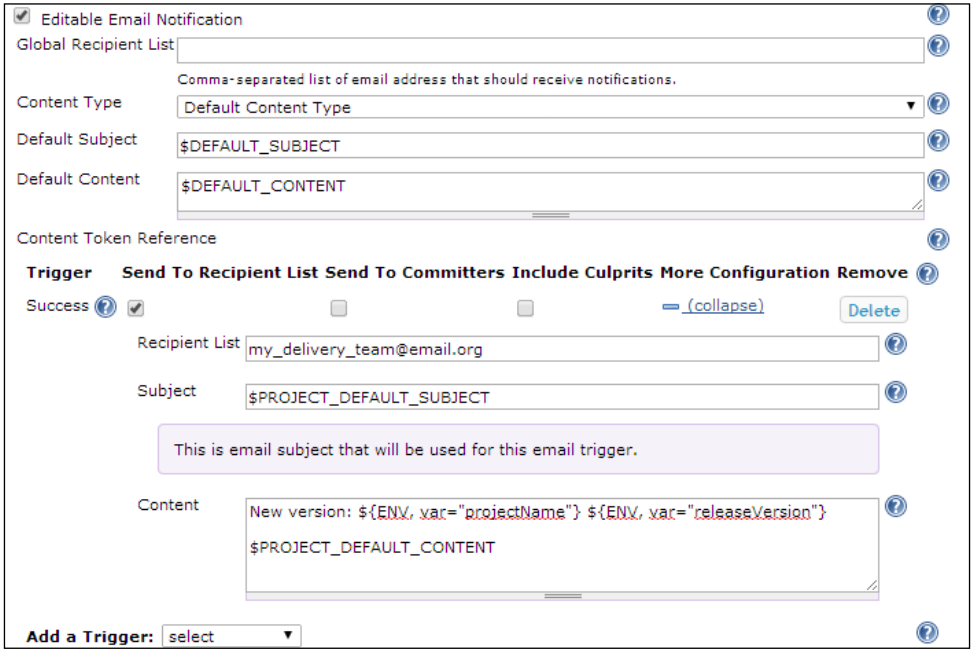
This strategy is strongly discouraged by authors, but should be a good work-around for old projects released in a production environment.

Finalizing the release

In the previous sections, we tested, versioned, and tagged the software. Then, we packaged and released the component on the repository server. Now, it's time to alert the delivery team about the new available version.

Hudson (or Jenkins) allows us to send an e-mail after the build step. Perform the following steps from the left-hand side menu:

1. Click on **Configure Job**.
2. In the **Post-build Actions** section, click on **Editable Email Notification**.
3. Click on **Advanced** and add **Success Trigger**.
4. Click on **Expand**.
5. In the **Recipient List** field, write the list of the e-mails of users to be notified.
6. In the **Content** field, write `New version: ${ENV, var="projectName"} ${ENV, var="releaseVersion"}` (see the following screenshot):



Hudson's configure notification phase

In the case of build success, the delivery team will be notified.

Sometimes, we also need to communicate some related information such as the prerequisites for installation (new features or configurations of DB); this information can be easily sent in the body of an e-mail or (better) attached as a child issue of the bug issue just solved. Indeed, in the previous sections, we showed how to integrate and automate the issue-tracking system (MantisBT). The most popular bug-tracking systems (MantisBT and JIRA) support this function, and it is common to use these features.

Our environment is now ready to be built; we can now release and deploy our amazing ideas.

Summary

In this chapter, we covered how to implement a real Continuous Integration process with Maven and some popular tools such as Hudson, Nexus, and Ant. We covered the Maven release and its deploy process to perform a real releasing pipeline.

We proposed SVN as our Software Control Management system, but the same principles can be applied to SVN, Git, CVS, Jazz, Bazaar, Mercurial, Perforce, StarTeam, and CM Synergy by only configuring the SCM section.

We discussed how to prepare a build environment to work with enterprise repositories in a multiteam company and integrate with popular issue-tracking systems; several companies use JIRA to track issues and manage the release process. JIRA is a powerful tool, but a complete discussion on JIRA-Maven integration is out of the scope of this book.

Finally, we looked at how to apply the Continuous Integration process in our custom project.

6
Maven Android

In this chapter, we will talk about **Android Maven Plugin** and its usefulness to a team to build, deploy, release, and test Android applications with Apache Maven. In a nutshell, we will see:

- How to configure the Maven Android environment
- How to build and package **Android Application Package** (**APK**)
- How to test an application

Prerequisites

We assume that the following packages are installed:

- JDK 1.6+ as required for Android development
- Android SDK (r21.1 or later; the latest version is best supported), preferably installed with all platforms
- Maven 3.0.5 (advised) or higher

We set the environment variable, `ANDROID_HOME`, to the path of our installed Android SDK. For example, if the SDK is installed at `/opt/adt-bundle/sdk`, this can be achieved with the given commands:

- On a Unix/bash-based system, use the following command:

```
export ANDROID_HOME=/opt/adt-bundle/sdk
```

- On a Windows-based system, use the following command:

```
set ANDROID_HOME=C:\opt\adt-bundle\sdk
```

Then, add `$ANDROID_HOME/tools` as well as `$ANDROID_HOME/platform-tools` to your `$PATH` (or add `%ANDROID_HOME%\tools` and `%ANDROID_HOME%\platform-tools` on Windows).

Creating your own Android application with an archetype

The simplest way to create the skeleton of our application is through the usage of an **archetype**.

The archetype that we will use is `ANDROID-QUICKSTART-ARCHETYPE`.

The only step to perform is to run the following command:

```
mvn archetype:generate \
-DarchetypeArtifactId=android-quickstart \
-DarchetypeGroupId=de.akquinet.android.archetypes \
-DarchetypeVersion=0.1.0 -DgroupId=com.androidmavenproject \
-DartifactId=android-maven-project
```

After the execution, our project is ready and can be customized.

Creating your own Android application

Alternatively, we can create the Android application manually. We will show two ways to create a project manually:

1. Create an empty project called `AndroidMavenProject` using Android tools.

Check out `http://developer.android.com/training/basics/firstapp/creating-project.html` to find the official guide to create a project using Android tools.

2. In the root project directory, create the subdirectory structure shown in the following screenshot:

Creating or modifying the AndroidManifest file

The `AndroidManifest` file is a powerful file in the Android platform that allows us to describe the functionality and requirements of our Android applications.

Create the `AndroidManifest.xml` file in the root of the project. The `AndroidManifest.xml` file has the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.androidmavenproject"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="19" />

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
      android:name="com.androidmavenproject.MainActivity"
      android:label="@string/app_name" >
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.
        LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Create a new layout into resources in the `activity_main.xml` file located at `res/layout`, and define the visual structure of your app. The `AndroidManifest.xml` file has the following code:

```
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingBottom="@dimen/activity_vertical_margin"
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight="@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
  tools:context="com.androidmavenproject.
  MainActivity$PlaceholderFragment" >

  <TextView
    android:id="@+id/text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</RelativeLayout>
```

Finally, write the `MainActivity.java` class at `src/main/com/androidmavenproject`. This class contains the following code:

```
package com.androidmavenproject;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }

  @Override
  public void onStart() {
    super.onStart();
    TextView textView = (TextView) findViewById(R.id.text_view);
    textView.setText("Hello world!");
  }
}
```

Defining a simple Maven POM file

In this section, we will learn how to define Maven projects with an XML file named `pom.xml`. This file provides the project's name, version, dependencies, and in particular, the Maven Android plugins and its configurations (see also *Chapter 1*, *Maven and Its Philosophy*).

Create a file named `pom.xml` at the root of the project with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.androidmavenproject</groupId>
  <artifactId>android-maven-project</artifactId>
  <version>0.1.0</version>
  <packaging>apk</packaging>
  <name>Android Maven project</name>

  <properties>
    <!-- use UTF-8 for everything -->
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8
    </project.reporting.outputEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.google.android</groupId>
      <artifactId>android</artifactId>
      <version>4.1.1.4</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <sourceDirectory>src</sourceDirectory>

    <plugins>
      <plugin>
        <groupId>com.jayway.maven.plugins.android.generation2
        </groupId>
```

```
            <artifactId>android-maven-plugin</artifactId>
            <version>3.9.0-rc.2</version>
            <extensions>true</extensions>
            <configuration>
              <sdk>
                <path>${env.ANDROID_HOME}</path>
                <platform>19</platform>
              </sdk>
              <deleteConflictingFiles>true</deleteConflictingFiles>
              <undeployBeforeDeploy>true</undeployBeforeDeploy>
            </configuration>
          </plugin>
          <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
              <source>1.6</source>
              <target>1.6</target>
            </configuration>
          </plugin>
        </plugins>
      </build>

    </project>
```

If you use Eclipse, especially the `m2e` plugin, you might get the following error:

```
Plugin execution not covered by lifecycle configuration: com.jayway.
maven.plugins.android.generation2:android-maven-plugin:3.6.1:generate-
sources (execution: default-generate-sources, phase: generate-sources)

Plugin execution not covered by lifecycle configuration: com.jayway.
maven.plugins.android.generation2:android-maven-plugin:3.6.1:proguard

(execution: default-proguard, phase: process-classes)
```

Add the following contents to remove the life cycle configuration error caused by the `m2e` plugin:

```
    <build>
      <pluginManagement>
        <plugins>
          <plugin>
            <groupId>org.eclipse.m2e</groupId>
            <artifactId>lifecycle-mapping</artifactId>
            <version>1.0.0</version>
            <configuration>
```

```
            <lifecycleMappingMetadata>
              <pluginExecutions>
                <pluginExecution>
                  <pluginExecutionFilter>
                    <groupId>com.jayway.maven.plugins.android.
                    generation2</groupId>
                    <artifactId>android-maven-plugin</artifactId>
                    <versionRange>[3.9.0-rc.2,)</versionRange>
                    <goals>
                      <goal>generate-sources</goal>
                      <goal>proguard</goal>
                    </goals>
                  </pluginExecutionFilter>
                  <action>
                    <execute />
                  </action>
                </pluginExecution>
              </pluginExecutions>
            </lifecycleMappingMetadata>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  […]
```

Description tags

In this chapter, we will examine only specific tags for an Android application.

The first tag that we will examine is the `<packaging>` element that specifies an APK. The APK value is allowed only after including the `com.jayway.maven.plugins.android.generation2` plugin.

As you can see, we define the Android plugin, `com.jayway.maven.plugins.android.generation2`, in the `<build>` section, with its configuration enclosed in the appropriate configuration tags. Here (in the `platform` tag), we define the Android SDK platform to use during the build (API Level 4 is platform 1.6).

The following line tells Maven that the plugin contributes to a package and/or as a type handler:

```
[…]
<extensions>true</extensions>
[…]
```

Building with Maven plugin goals

We are ready to use the most common goals to build the project; in this section, we will see:

- How to create a JAR file
- How to install libraries
- How to deploy
- How to run the application on your device

Use the `compile` goal to build the compiled `.class` files in the `target/classes` directory:

```
$ mvn compile
```

If you want to work with the `.class` files directly, run the `package` goal.

To take the compiled code and package it in its distributable format, such as JAR, run any test and use the following goal:

```
$ mvn package
```

Performing the preceding command on our project will generate a JAR file named `android-maven-project -0.1.0.jar` on the `target` directory.

The construction of the packaged name is based on the artifact ID and version.

Since we set the value of `packaging` to `apk`, the result will be an APK file on the `target` directory, which is ready to be deployed and launched on a device or emulator.

If you want to install the application via Maven on your Android device, you can use the following command:

```
$ mvn android:deploy
```

If more than one device is available, you can specify the relevant device in your `pom.xml` file. Maven can also start and stop an Android virtual device automatically for you.

To list all attached devices and emulators found with the Android debug bridge, use the following command:

```
$ mvn android:devices
```

If the `android.devices` property is not set, it will use all attached devices. To specify a device, set the `android.device` property; it is possible to use the special values, `usb` and `emulator`, as shown in the following code:

```
<properties>
  <android.device>usb</android.device>
  […]
</properties>
```

Finally, you can also start the application using the following command:

```
$ mvn android:run
```

To see all available goals, use the `$ mvn android:help` command.

Declaring dependencies

Now, we will see how to insert and use third-party libraries easily. For our scope, we will use a very simple and functional library, which is called **RoboGuice 2**, and which allows us to inject our view, resource, system service, or any other object into our activity (we call this activity `RoboActivity`).

To do this, we have to modify the `MainActivity.java` class. This class contains the following code:

```
package com.androidmavenproject;

import roboguice.activity.RoboActivity;
import roboguice.inject.ContentView;
import roboguice.inject.InjectView;
import android.os.Bundle;
import android.widget.TextView;

@ContentView(R.layout.activity_main)
public class MainActivity extends RoboActivity {

  @InjectView(R.id.text_view)
  TextView name;

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    name.setText("Hello world!");
  }
}
```

We take advantage of the functionalities offered by this library to set the content view and inject the `TextView` instance.

The next step is to insert the dependent library into the `pom.xml` file:

```
[…]
<dependency>
  <groupId>org.roboguice</groupId>
  <artifactId>roboguice</artifactId>
  <version>2.0</version>
</dependency>
[…]
```

A compatibility library for API v4

Another common add-on is the compatibility library for API v4 and higher. Obviously, we need to include the library in the final package (`.apk`), and this does not have the provided scope.

If you generate a project with the Eclipse wizard, make sure that no support for library v4 is automatically included into the `libs` folder; if it is, delete it.

The next step is to include `support-v4` in the `pom.xml` file:

```
[…]
<dependency>
  <groupId>com.google.android</groupId>
  <artifactId>support-v4</artifactId>
  <version>r7</version>
</dependency>
[…]
```

Finally, modify the `MainActivity` class to use `android.support.v4.app.FragmentActivity` with the following code:

```
package com.androidmavenproject;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.widget.TextView;

public class MainActivity extends FragmentActivity {
  @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
    }

    @Override
    public void onStart() {
      super.onStart();
      TextView textView = (TextView) findViewById(R.id.text_view);
      textView.setText("Hello world!");
    }
  }
```

The final POM file with dependencies

We are ready to run our application with dependencies. First, we need to resolve the dependencies; for this, run the following command:

```
$ mvn package
```

The following is the code of the final `pom.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.androidmavenproject</groupId>
  <artifactId>android-maven-project</artifactId>
  <version>0.1.0</version>
  <packaging>apk</packaging>
  <name>Android Maven project</name>
  <properties>
    <!-- use UTF-8 for everything -->
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8
    </project.reporting.outputEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.google.android</groupId>
      <artifactId>android</artifactId>
      <version>4.1.1.4</version>
```

```
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>org.roboguice</groupId>
        <artifactId>roboguice</artifactId>
        <version>2.0</version>
      </dependency>
      <dependency>
        <groupId>com.google.android</groupId>
        <artifactId>support-v4</artifactId>
        <version>r7</version>
      </dependency>
    </dependencies>
    <build>
      <sourceDirectory>src</sourceDirectory>
      <plugins>
        <plugin>
          <groupId>com.jayway.maven.plugins.android.generation2
          </groupId>
          <artifactId>android-maven-plugin</artifactId>
          <version>3.9.0-rc.2</version>
          <extensions>true</extensions>
          <configuration>
            <sdk>
              <path>${env.ANDROID_HOME}</path>
              <platform>19</platform>
            </sdk>
            <deleteConflictingFiles>true</deleteConflictingFiles>
            <undeployBeforeDeploy>true</undeployBeforeDeploy>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.1</version>
          <configuration>
            <source>1.6</source>
            <target>1.6</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

Useful instrumentations to test, sign, and zipalign

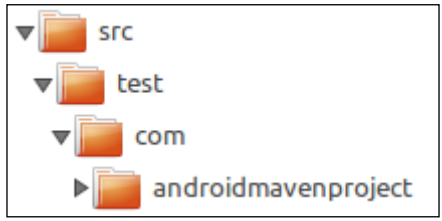
In this section, we will talk about some useful tools for our code.

The test profile

In this section, we will create a profile to install the application automatically and run instrumentation tests at every build.

For convenience, we will refer to our profile as `testProfile` and perform the following steps:

1. Create a new directory for the test classes; follow the folder structure shown as follows:



2. Then, specify the source test directory in the POM file:

```
[…]
<build>
  <sourceDirectory>src</sourceDirectory>
  <testSourceDirectory>test</testSourceDirectory>
</build>
[…]
```

3. Create a property to enable/disable the instrumentation tests:

```
[…]
<properties>
  <skipTests.value>true</skipTests.value>
</properties>
[…]
```

4. Add the JUnit test dependency:

```
[…]
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>
[…]
```

5. Create a test class named `Tests` in the previously created directory, `src/test/com/androidmavenproject`:

```
package com.androidmavenproject;

import junit.framework.Assert;
import org.junit.Test;

public class Tests {

  @Test
  public void testEquals() {
    Assert.assertEquals("Hello World", "Hello World");
  }
}
```

6. Now, add the `maven-surefire-plugin` artifact ID to the `build` tag and run the application. We can see the `maven-surefire-plugin` artifact ID's following code:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.12.4</version>
  <configuration>
    <skipTests>${skipTests.value}</skipTests>
  </configuration>
</plugin>
```

You should get a result like the following:

```
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @
android-maven-project ---
[INFO] Tests are skipped.
```

7. Finally, create the `testProfile` profile in the POM file:

```
[…]
<profiles>
  <profile>
    <id>testProfile</id>
    <properties>
      <skipTests.value>false</skipTests.value>
    </properties>
  </profile>
</profiles>
[…]
```

   Note that the property to skip the test is set to `false`.

8. Now, we are ready to test the app; run Maven activating the profile from the command line with the `-P` flag:

```
$ mvn clean install -PtestProfile
```

   We should get a result like the following:

```
T E S T S
-------------------------------------------------------
Running com.androidmavenproject.Tests
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.003 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Signing and zipaligning the package

To sign and zipalign the package, perform the following steps:

1. Create and add a new profile called `releaseProfile` that will serve to release the APK:

```
[…]
<profile>
  <id>releaseProfile</id>
  <activation>
    <property>
      <name>performRelease</name>
      <value>true</value>
```

```
    </property>
  </activation>
</profile>
[…]
```

2. Create the properties to sign the app, as follows:

```
[…]
<properties>
  <sign.keystore>pathtokeystorefile</sign.keystore>
  <sign.alias>aliasname</sign.alias>
  <sign.keypass>somepassword</sign.keypass>
  <sign.storepass>somotherpassword</sign.storepass>
</properties>
[…]
```

3. Replace all `sign.*` properties with your `keystore` values. Finally, add the JAR-signing process into the build profile:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jarsigner-plugin</artifactId>
      <executions>
        <execution>
          <id>signing</id>
          <goals>
            <goal>sign</goal>
            <goal>verify</goal>
          </goals>
          <phase>package</phase>
          <inherited>true</inherited>
          <configuration>
            <removeExistingSignatures>true
            </removeExistingSignatures>
            <archiveDirectory/>
            <includes>
              <include>${project.build.directory}/
              ${project.artifactId}.apk</include>
            </includes>
            <keystore>${sign.keystore}</keystore>
            <alias>${sign.alias}</alias>
            <storepass>${sign.storepass}</storepass>
            <keypass>${sign.keypass}</keypass>
```

```
            <verbose>true</verbose>
          </configuration>
        </execution>
      </executions>
    </plugin>
[…]
```

4. Finally, the signed APK has to be zipaligned; also, deactivate the debug sign. To zipalign, add the following procedure:

```
[…]
<plugin>
  <groupId>com.jayway.maven.plugins.android.generation2
  </groupId>
  <artifactId>maven-android-plugin</artifactId>
  <inherited>true</inherited>
  <configuration>
    <sign>
      <debug>false</debug>
    </sign>
    <zipalign>
      <verbose>true</verbose>
      <inputApk>
        ${project.build.directory}/${project.artifactId}
        .apk
      </inputApk>
      <outputApk>
        ${project.build.directory}/${project.artifactId}-
        signed-aligned.apk
      </outputApk>
    </zipalign>
  </configuration>
  <executions>
    <execution>
      <id>alignApk</id>
      <phase>package</phase>
      <goals>
        <goal>zipalign</goal>
      </goals>
    </execution>
  </executions>
</plugin>
[…]
```

5. Now, we can produce your signed and zipaligned APK with the simple command, ready to be published on the market; run the following command:

```
$ mvn clean install –PreleaseProfile
```

The output will be something like the following:

```
[INFO] --- maven-jarsigner-plugin:1.3.2:verify (signing) @
android-maven-project ---
[INFO] 2 archive(s) processed
[INFO]
[INFO] --- android-maven-plugin:3.0.0-alpha-11:zipalign (alignApk)
@ android-maven-project ---
[INFO] Running command: …\eclipse\adt-bundle-windows-x86_64\sdk\
tools\zipalign.exe
[INFO] with parameters: [-f, 4, …\workspace\AndroidMavenProject\
target\android-maven-project-0.1.0.apk, …\workspace\
AndroidMavenProject\target\android-maven-project-0.1.0-aligned.
apk]
[INFO] Attach …\workspace\AndroidMavenProject\target\android-
maven-project-0.1.0-aligned.apk to the project
```

As you can see, the zipalign procedure will take the APK file as input and produce a new output called `${project.artifactId}-signed-aligned.apk`.

The bug detector (Lint)

There are several tools to identify potential bugs in our code. In this section, we will see the dedicated plugin for Android, and this tool is called **lint**.

Android lint is a new tool introduced in ADT 16 (and Tools 16). Lint can check missing translations, unused resources, and other common mistakes in Android programming.

Now, let's see a simple example to generate an HTML report using the errors of our project. We need to change the Android plugin by adding a few simple lines:

```
[…]
<configuration>
  <sdk>
    <path>${env.ANDROID_HOME}</path>
    <platform>19</platform>
  </sdk>
  <deleteConflictingFiles>true</deleteConflictingFiles>
  <undeployBeforeDeploy>true</undeployBeforeDeploy>
```

```
    <lint>
      <skip>false</skip>
      <enableHtml>true</enableHtml>
      <enableXml>false</enableXml>
    </lint>
  </configuration>
  […]
```

We disable the XML output and enable HTML because it is more readable and we can get a better overview of HTML pages.

We just add `android:lint` to the Maven commands with the following command:

```
$ mvn clean install android:lint
```

The results will be written to `/target/lint-results/lint-results-html`.

You should get a result like the following:

```
[INFO] --- android-maven-plugin:3.9.0-rc.2:lint (default-cli) @ android-
maven-project ---
[INFO] Performing lint analysis.
[INFO] Writing Lint HTML report in C:\android\workspace \
AndroidMavenProject\target\lint-results\lint-results-html
[INFO] Running command: …\eclipse\adt-bundle-windows-x86_64\sdk\tools\
lint.bat
[INFO] with parameters: [--showall, --html, …\workspace \
AndroidMavenProject\target\lint-results\lint-results-html, --sources, …\
workspace \AndroidMavenProject\src, …\workspace \AndroidMavenProject,
--exitcode]
[INFO] Lint analysis completed successfully.
```

It is possible to control lint invocation by adding `failOnError` to the `lint` tag.

If `failOnError` is `true`, any lint error (not warning) will stop the build. The default is set to `false`. This flag is useful for continuous integration (see *Chapter 5, Continuous Integration and Delivery with Maven*) builds as it allows us to enforce lint's usage.

To increase lint's granularity, use the tag `ignoreWarnings=true|false`.

If true, we don't report lint warnings, only errors are reported. By default, it is set to `false`, as in `warningsAsErrors=true|false`.

If `true`, all lint warnings will be treated as errors. By default, it is set to `false`.

Eclipse integration

Android's official development effort provides solid support for Eclipse integration, and we want to make sure that the Android Maven plugin helps bridge Maven, Android, and Eclipse.

First, you will need to install Eclipse Indigo, Juno, or **Android Developer Tools** (**ADT**), which is a plugin for Eclipse that provides a professional-grade development environment to build Android apps.

Installing the Android connector

The m2e Android plugin is an M2E connector that adds Maven support to Eclipse or ADT.

Install it via Eclipse Marketplace, and perform the following steps:

1. Select **Help** | **Eclipse Marketplace...** and search for `android m2e`.
2. Click on the **Install** button next to the Android connector for Maven that appears, and follow the path through the wizard dialog to install the plugin and all its dependencies.
3. Accept the terms of the license and click on **Finish**.
4. If you cannot access Marketplace, select **Help** | **Install New Software** and paste the `http://download.eclipse.org/technology/m2e/releases/` repository. Then, check **Maven Integration for Eclipse**, click on the **Next** button, accept the terms of the license, and click on **Finish**.
5. Restart your Eclipse workspace.

Mavenized Android Project

If you have an Android project configured, right-click on it and chose **Configure** | **Convert to Maven Project**.

If you are starting with a new project, you can use the Maven Android archetypes to create Android projects completely within Eclipse using the following steps:

1. Create a new Maven Project by navigating to **File** | **New** | **Project**.
2. Select **Maven** | **Maven Project**.
3. When prompted to select an archetype, click on **Add Archetype**.
4. In the **Archetype Group Id** field, enter `de.akquinet.android.archetypes`.

5. In the **Archetype Artifact Id** field, enter `android-quickstart`.
6. In the **Archetype Version** field, enter `0.1.0`.
7. When prompted, enter your desired project group and the artifact ID, version, and platform property for the Android version (the default is `16`).
8. Click on **Finish**.

Summary

In this section, we covered how to create an Android project with Maven. We learned how Maven plugins and profiles made the applicants' planning stages, such as creating a simple structure, implementing and running a test, dependency management, signing and zipaligning an application, detecting potential bags, and running and deploying it on devices, easy.

In addition, we saw how to run our commands from the command line and integrate them with Eclipse and ADT.

A

Integrating Maven – Gradle

Gradle is gradually becoming an important and stable tool for project automation (at the time of writing this book, the latest version used is 2.0). We decided to mention this tool because its percentage of adoption is growing fast; moreover, it was adopted as an official building tool for Android apps by Google at Google I/O 2013. After an official announcement of the Gradle adoption by Google, this tool was completely integrated into the developer IDE Google Android Studio; the new Eclipse plugin will be introduced by the end of 2014, as announced by Gradleware (the Gradle developing team). All these facts, together with Gradle's ability to download dependencies from Maven repositories, have made Gradle eligible to be mentioned in this book.

What is Gradle?

Gradle is a project automation tool that was brought up on the concepts of Apache Ant and Apache Maven. Among Apache's tools, Gradle does not use an XML tag language in order to define the project structure and operation/task to execute; it introduces a Groovy-based **domain-specific language** (**DSL**).

Ant and Maven (more often Maven) define a lifecycle that invokes different tasks in a specific order, and every defined task is associated with a specific phase of the lifecycle. Despite Maven's and Ant's behavior, Gradle uses **directed acyclic graph** (**DAG**) to determine the order in which tasks can be run; using this structure, Gradle can determine which task has to be executed before or after, without a standard order of execution.

A match point between Gradle and Apache Maven is the capability to manage multiproject builds. Gradle can support incremental builds by determining which parts of a subproject are up to date, so a task that depends on these parts does not have to be re-executed.

The most interesting Gradle feature is represented by the ability to use Maven repositories for dependency management (Ivy repositories can be used too). It is possible to use remote and local repositories and declare nonstandard Maven repositories as custom repositories.

Like Maven, Gradle makes use of a plugin that provides additionally functionalities to accomplish common tasks to build and assemble projects in packages such as JAR, WAR, and EAR used by the Java programming language. The Android plugin compiles and assembles an app with all the tools to publish and sign the generated APK.

Actually, Gradle can build different programming languages:

- Java: This adds Java compilation, testing, and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins.
- Groovy: This adds compilation, testing, and generation of documentation.
- Scala: This adds compilation, testing, and generation of documentation
- ANTLR: This generates source files for production and testing.

There are many other incubating plugins for other languages, such as:

- `assembler`
- `c`
- `cpp`
- `objective-C`
- `objective-cpp`
- `windows-resources`: Adds support for Windows resources in native binaries

Other kinds of plugins are represented by **integration** plugins. They are:

- `application`: Adds tasks to run and build Java projects at the command line
- `jetty`: Deploys your web application to a Jetty web container embedded in the build
- `ear`
- `war`
- `osgi`
- `maven`: Adds support to deploy artifacts on Maven repositories

These plugins are only a few representative numbers to help explain what Gradle is. More documentation can be found online in order to get a deeper understanding of the Gradle mechanics.

How Gradle works

Gradle executes a series of commands called **task** declared inside the `build.gradle` file. The syntax to declare tasks is Groovy-based, as described before. A simple example of how to declare a task is:

```
task goGradle {
    doLast {
        println 'Gradle Task'
    }
}
```

The command to execute this simple task is:

```
$ gradle goGradle
```

The output for this command is:

```
Gradle Task
```

Another syntax to define the same task is:

```
task goGradle << {
    println ''Gradle Task'
}
```

In the first task definition, we use the `doLast` block to wrap actions to perform; we can use other instructions such as `doFirst` to decide task ordering. Thanks to `doFirst` and `doLast`, Gradle accomplishes its main characteristic to use a DAG for a task's order.

More Gradle functionalities are tasks, and they are executed with the command-line syntax explained.

If you want to know more about how Gradle's tasks work, you can consult the online manual at `http://www.gradle.org/`.

Creating a simple project with Gradle

Gradle can be used to create a Java project; in our case, we can create a common JAR project to explain a simple configuration to use Maven repositories within Gradle.

First, we must download the current version of Gradle; we can download the latest version 2.0. Once we get a ZIP file, we unzip it and put it into a folder as follows:

```
C:\gadle-2.0
```

Add `GRADLE_HOME/bin` to your `PATH` environmental variable to launch the Gradle command from every location. Obviously, you must have an installation of Java on your machine if you want to build a Java project.

All Gradle projects contain a file called `build.gradle` that contains the instructions to build and assemble projects through the command line.

Gradle's project configuration

First, we add plugins into `build.gradle` used for the project:

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

With these lines, we specify the use of three plugins: the Java plugin to compile and assemble our project, the Eclipse plugin to generate files that are used to import the project into the Eclipse IDE (if you want to use Eclipse), and the Maven plugin to deploy artifacts into the Maven repository.

In the next section, we will declare the data, Java version compatibility, group ID, and project version, which are relative to the project:

```
sourceCompatibility = 1.6
group = 'org.gradle.test'
version = '1.1'
```

Thanks to the Groovy syntax, we can declare what the `jar` manifest contains, in a simple and elegant way:

```
jar {
    manifest {
      attributes 'Implementation-Title': 'Gradle Test',
                 'Implementation-Version': version
    }
}
```

The version number refers to the global variable, `version`, just declared in the previous statement.

In the `repositories` section, we can see how Gradle makes use of Maven official repositories and our custom repository:

```
repositories {
  mavenCentral()
  mavenLocal()

  maven { url
  "http://ourserver:8080/nexus/content/repositories/ourrepo" }

}
```

Thanks to the `mavenCentral()` object, Gradle downloads dependencies from the Maven 2 repository (`http://repo1.maven.org/maven2`), and the `mavenLocal()` object indicates to get dependencies from the `PATH_TO/.m2` local repository.

As we can see, we add the custom repository while looking for dependencies.

The following code snippet shows us how to declare dependencies:

```
configurations {
  deployerJars
}
dependencies {
    compile 'commons-collections:commons-collections:3.2'
    testCompile 'junit:junit:4.+'
    deployerJars "org.apache.maven.wagon:wagon-http:2.2"
}
```

The first line of the code represents a local variable to import utility libraries used to perform a Maven deploy. In the second statement, we have the `dependencies` declaration since we can use different scopes for dependency import in Maven, as shown in the following table:

Scope	Function/objective	Default
`compile`	Required to compile the source	
`runtime`	Required at runtime	Includes the compile time dependencies
`testCompile`	Required in order to compile a test source	Includes compiled production and compile time
`testRuntime`	Required in order to run tests	Includes compile, runtime, and test dependencies

To import a version greater or equal to a certain version, Gradle uses the `+` notation for a JUnit import case.

We assigned the `wagon-http` dependency to the `deployerJars` variable used to deploy the JAR into a repository.

We are able to perform a build of the project with this minimal configuration. To perform a build positioning in the project's base directory, launch the following command:

```
$ gradle build
```

This input will generate a simple output:

```
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL
```

In this case, we can execute without having the declared task, `build`, which is possible because of the Java plugin.

The Java plugin adds a build file command to configuration, without declaration in the `build.gradle` file.

Deploying on the Maven repository

Until now, we explained how to build a JAR project with Gradle; let's add a task to deploy an artifact on the repository with the following lines:

```
uploadArchives {
  repositories {
    ext.configuration = configurations.deployerJars
    mavenDeployer {
      repository(url:
      "http://ourserver/nexus/content/repositories/releases") {
```

```
          authentication(userName: "user", password: "boh!")
        }
      }
    }
  }
```

This code snippet contains the task and object inherited from the Maven plugin.

In the preceding example, we used `uploadArchives` to perform the artifact's upload. The `uploadArchives` task requires parameters such as which repository to use for the deploy operation passed within the `repositories` object. To perform the artifact's upload, we used `mavenDeployer` within the object that contains the `url` repository and authentication credential; this object is `repository`, which contains `authentication` within the specification for `username` and `password`, valorized with our server authentication credentials. In order to enable `mavenDeployer` to create a connection to the server, the `configuration` variable contains `deployerJars` within the `wagon-http` library. Using this task, we can upload an artifact to our Maven server.

In the code snippet, we used the new **extra** properties' `ext` syntax to dynamically add content to objects:

```
ext.configuration = configurations.deployerJars
```

The old fashion way to declare configuration is called **dynamic** properties:

```
configuration = configurations.deployerJars
```

This example can upload an artifact with the Maven dependency notation:

```
<dependency>
  <groupId>org.gradle.test</groupId>
  <artifactId>gradle-project</artifactId>
  <version>1.1</version>
</dependency>
```

If we had to change a `pom` property at the moment of deployment, we can use the following syntax:

```
uploadArchives {
  repositories {
    ext.configuration = configurations.deployerJars
    mavenDeployer {
      repository(url:
      "http://ourserver/nexus/content/repositories/releases") {
        authentication(userName: "user", password: "boh!")
      }
```

```
        pom.version = '1.0' pom.artifactId = 'gradle-project-second'
      }
    }
}
```

Also, add the following two properties:

```
pom.version = '1.0'
pom.artifactId = 'gradle-project-second'
```

As a result, we will have this POM content on the published library:

```
<dependency>
  <groupId>org.gradle.test</groupId>
  <artifactId>gradle-project-second</artifactId>
  <version>1.0</version>
</dependency>
```

Creating the project's POM

Gradle's Maven plugin can create a complete POM file. To make this operation possible, we can create an appropriate task:

```
task writeNewPom << {
  pom {
    project {
      inceptionYear '2014'
      licenses {
        license {
          name 'The Apache Software License, Version 2.0'
          url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
          distribution 'repo'
        }
      }
    }
  }.writeTo("$buildDir/pom.xml")
}
```

The `pom` object gives us all elements for the POM file, and we can override the default element inherited from the Gradle project configuration with other elements in order to customize POM creation. As the final instruction, perform a write of the POM file to the building directory with the name `pom.xml` using the following method:

```
.writeTo("$buildDir/pom.xml")
```

The resulting POM file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.gradle.test</groupId>
  <artifactId>gradle-project</artifactId>
  <version>1.0</version>
  <inceptionYear>2014</inceptionYear>
  <licenses>
    <license>
      <name>The Apache Software License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.+</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>commons-collections</groupId>
      <artifactId>commons-collections</artifactId>
      <version>3.2</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

B

Maven Integration for Eclipse

The Eclipse IDE provides support for Maven through the m2eclipse plugin, which has been recently renamed to m2e. The newer versions of the Eclipse IDE (starting from Kepler) come with the m2e plugin available without needing to be installed as an additional component. The m2e plugin uses components called m2e connectors (or Maven plugin connectors) that work as a bridge between Maven and Eclipse and are able to trigger the execution of the Maven plugins declared in our POMs during the automatic build process of the IDE. These connectors are searched in the Eclipse repositories when they are needed, depending on the used Maven plugins.

This way, we can work on a Maven project within the Eclipse IDE as if it were a native Eclipse project. In the next paragraphs, we are going to summarize the most important use cases.

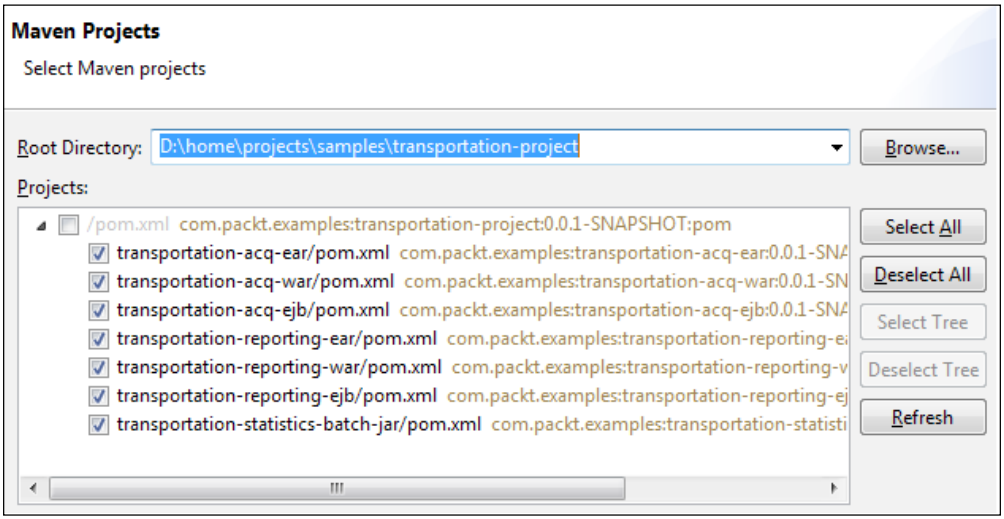
The m2e plugin is not a Maven plugin, it is a plugin for Eclipse! The m2e plugin must not be confused with the Maven Eclipse Plugin `org.apache.maven.plugins:maven-eclipse-plugin` (with the prefix `eclipse`). The latter is a Maven plugin that statically generates/regenerates the Eclipse project files every time we invoke the `eclipse:eclipse` goal.

In the first chapter, we created our sample project parent POM using the Eclipse IDE. We can very easily create new Maven projects and modules from the Eclipse menu by navigating to **New | Project… | Maven**. While creating the sample parent POM and its child modules, we could see that other than the `pom.xml` descriptor, the Eclipse project files had also been created.

In addition, we can import existing Maven projects into the Eclipse IDE, or we can check them out from an SCM repository like SVN or CVS.

Importing existing Maven projects

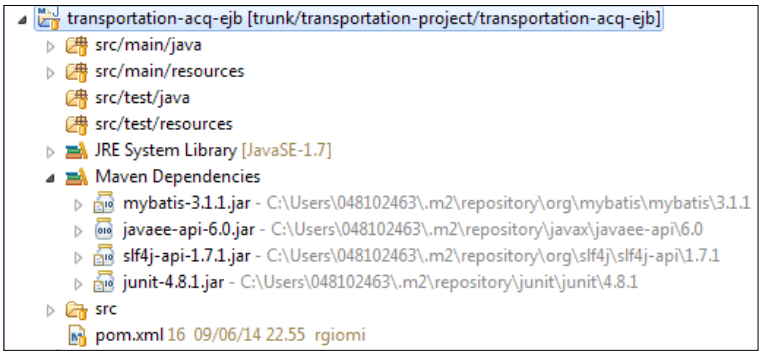
By navigating to **Import…** | **Existing Maven Projects…**, we can select the project to import in the Eclipse IDE as shown in the following screenshot:



Import existing Maven projects

In this sample, we suppose that we created the parent project POM from the Eclipse IDE, and all the other by hand outside Eclipse, so that we need to import them into the IDE.

If we look in the **Package Explorer** view, we can see that all the project structures are recognized: the directories `/src/main/java`, `/src/main/resources`, and so on are displayed as source folders and the dependencies are visible under the `Maven Dependencies` classpath folder. This is shown in the following screenshot:



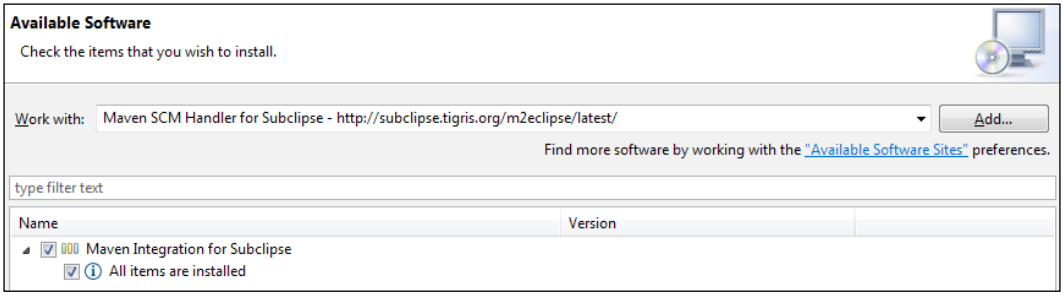
Structure of an imported Maven project

If we look in the **Navigator** view or in the filesystem, we can see the Eclipse project configuration that is formed by the `.project` and `.classpath` files and by the `.settings` directory. These files and directories have been created by Eclipse itself while importing the project, on the basis of the Maven POM.

Checking out Maven projects from SCM repositories

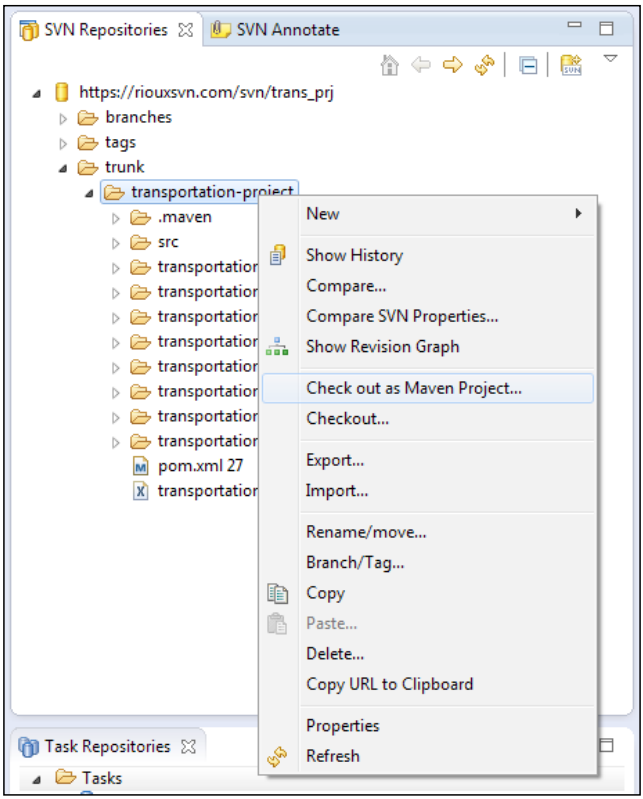
If we have to check out a Maven project from an SCM repository such as SVN or CVS, we can check it out directly as a Maven project. Usually, the Eclipse configuration resources are not committed on the SCM; even better, they are added to the `.svnignore` or `.cvsignore` files. This is because the m2e plugin is able to recreate all the necessary Eclipse configuration starting from the POM.

If our SCM repository is Subversion, we could install the Subclipse plugin from the Eclipse Marketplace by navigating to **Help | Eclipse Marketplace…**; just type `subclipse` in the search textbox and then select the appropriate plugin. In addition, we have to install the Maven SCM handler for Subclipse if we want to check out the Maven project directly, as we are about to show. We can install this component by navigating to **Help | Install New Software…**, adding the update site `http://subclipse.tigris.org/m2eclipse/latest/`, and finally selecting the Maven integration for the Subclipse checkbox, as shown in the following screenshot:



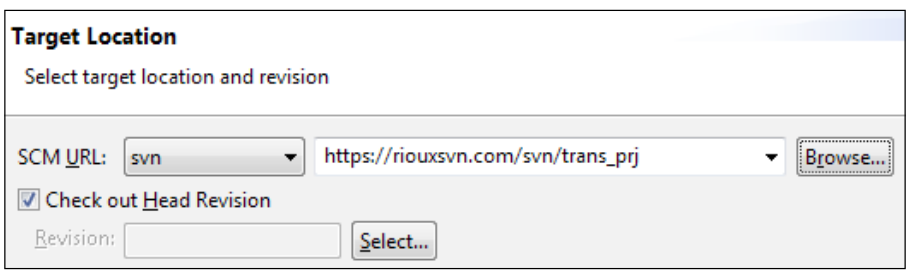
Installation of the Maven SCM Handler for Subclipse

At this point, we are able to check out the project by opening the **SVN Repository Exploring** perspective (by navigating to **Window** | **Open Perspective** | **Other**...), then right-clicking on the project folder and selecting the **Check out as Maven Project…** menu item, as shown in the following screenshot:



Check out from the repository exploring perspective

Alternatively, we can check out our projects directly by navigating to **File** | **Import** | **Maven** | **Check out Maven Projects from SCM** and then filling the **SCM URL** dropdown, as shown in the following screenshot:

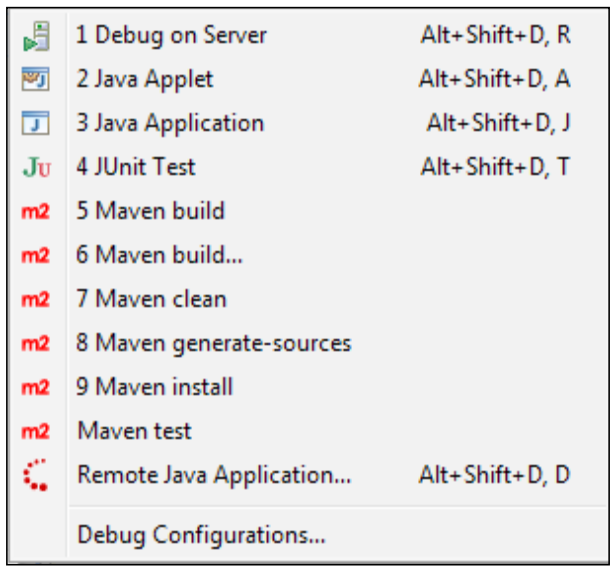


Check out from the main menu

If our SCM repository is CVS, we can install the Maven SCM Handler for CVS using the update URL `http://repository.tesla.io:8081/nexus/content/sites/m2e.extras/m2eclipse-cvs/0.13.0/N/0.13.0.201304101743/` and then proceed in a similar manner.

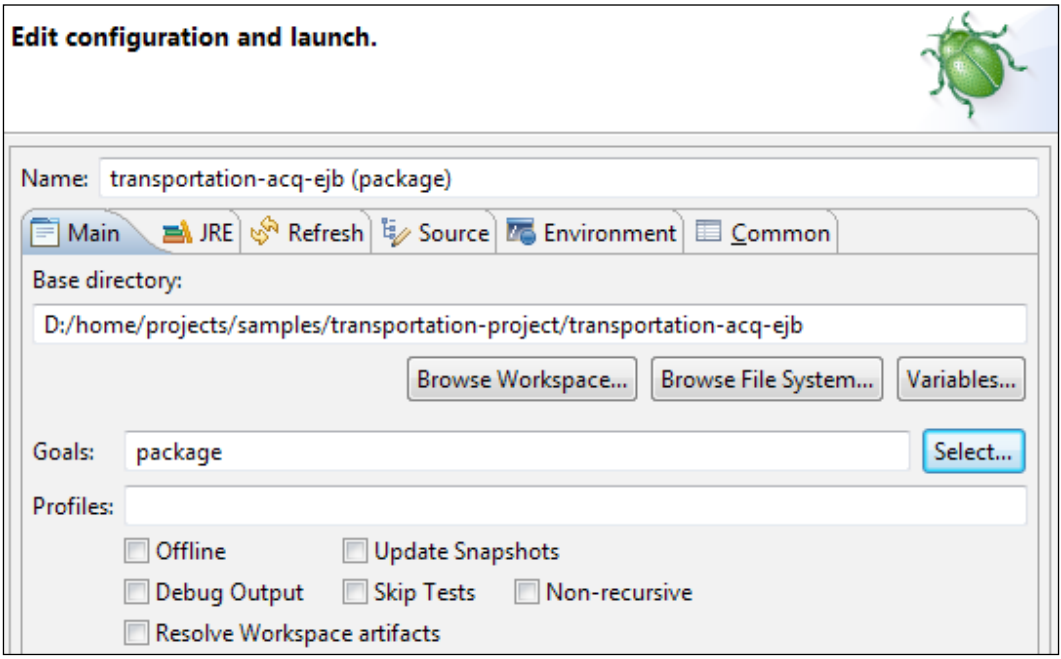
Building Maven projects

Once our Maven project has been integrated in the Eclipse IDE, all the phases of the build lifecycle, up to the compile phase, are executed automatically in the background by the IDE itself. So, we can see that under the `/target` folder, there are the compiled `.class` files corresponding to our project sources and test sources as well as the filtered project and test resources. If we want to build our project till the needed phase, we have to invoke Maven explicitly: this can be done following the **Run As** (or **Debug As**) menu and then selecting the desired phase, as shown in the following screenshot:



Maven build options in the Run As menu

If we want to launch Maven with a phase or a goal that is not listed in the pop-up menu, we can click on the **Maven Build...** menu item and fill the dialog window shown in the following screenshot:



Edit the Maven build configuration

Here, other parameters can also be specified, for example, the **Skip Tests** checkbox (corresponding to the `-Dmaven.test.skip=true` parameter on the command line) if we want to skip the compilation and execution of the unit tests, or the **Debug Output** checkbox (corresponding to the `-X` parameter) to enable debug output, or the **Non-recursive** checkbox (corresponding to the `-N` parameter) that avoids building child modules.

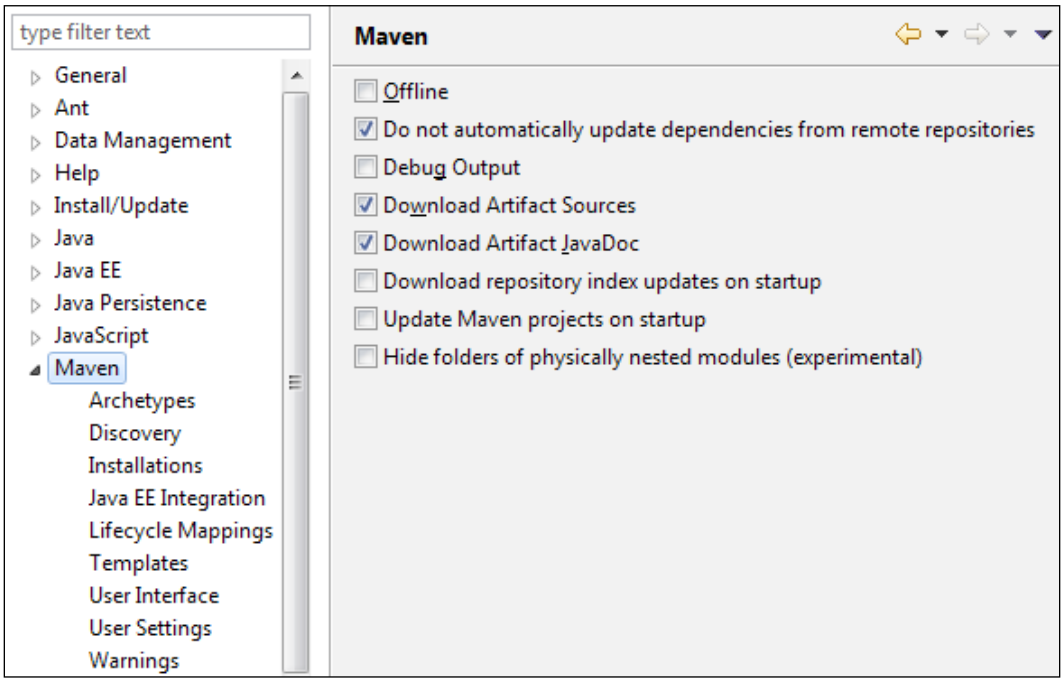
In all these cases, the Maven output is displayed in the **Console** view of the Eclipse IDE. From the **Console** view, we can click on the **Open Console** icon on the right-hand side of the toolbar and then on the **Maven Console** menu item. We will see the Maven commands that are launched by the m2e plugin, and the logs of all the activities executed in the background.

m2e plugin settings

We can customize the behavior of the m2e plugin through the Eclipse preferences. From the main Eclipse menu, navigate to **Windows** | **Preferences** to open the preference window and then click on the item regarding Maven (see the following screenshot). Here we can see some global settings for the m2e plugin. Among these, maybe the most useful are the **Download Artifact Sources** and **Download Artifact JavaDoc** checkboxes, to enable automatic downloads of dependency sources and dependency Javadocs. The dependency sources and Javadocs will be integrated in the Eclipse IDE so they will be available during all our developing and debugging activities. To download them from the command line, we should invoke the Maven Dependency Plugin from the project directory as follows:

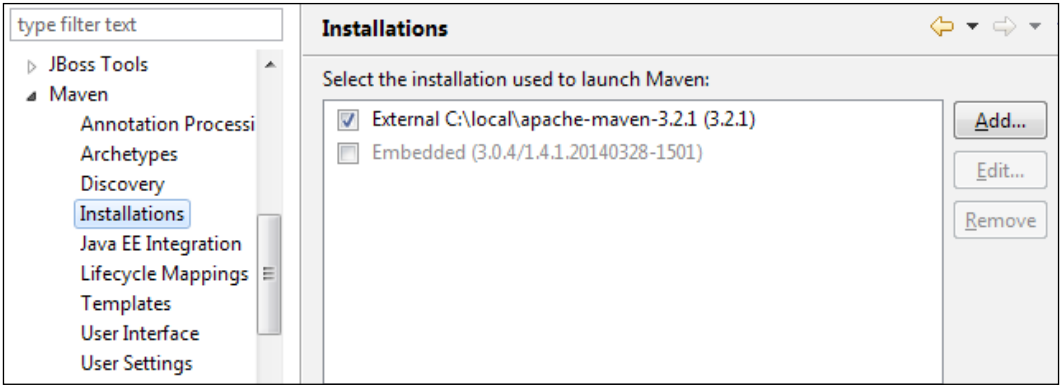
```
$ mvn dependency:sources
$ mvn dependency:resolve -Dclassifier=javadoc
```

We can see the preference window as shown in the following screenshot:



Maven general preferences

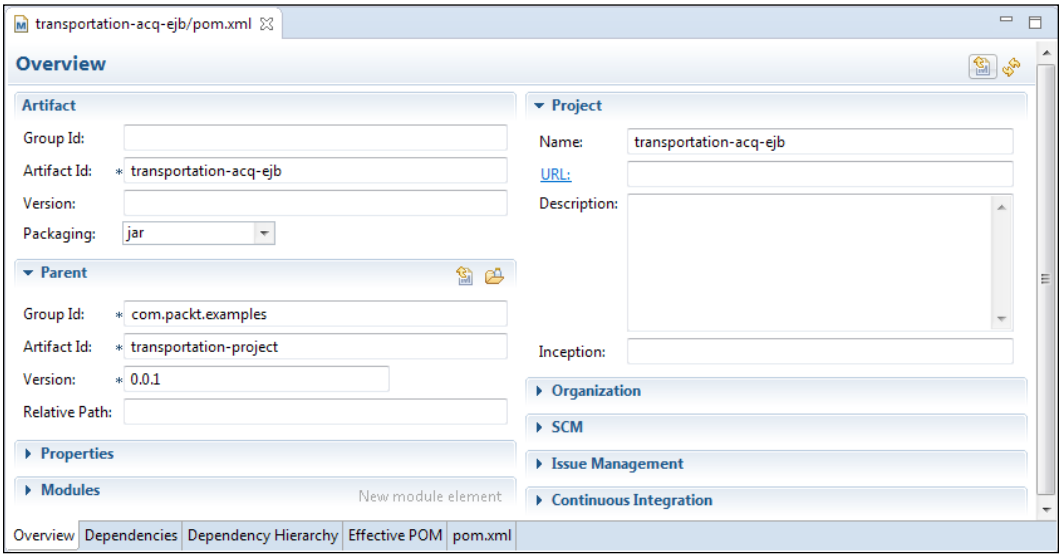
We can also edit the Maven installation used by the m2e plugin by clicking on the **Installations** subitem of the **Maven** preferences. In fact, the m2e plugin comes with an embedded Maven runtime that can be changed with an external installation, as shown in the following screenshot:



Maven installations

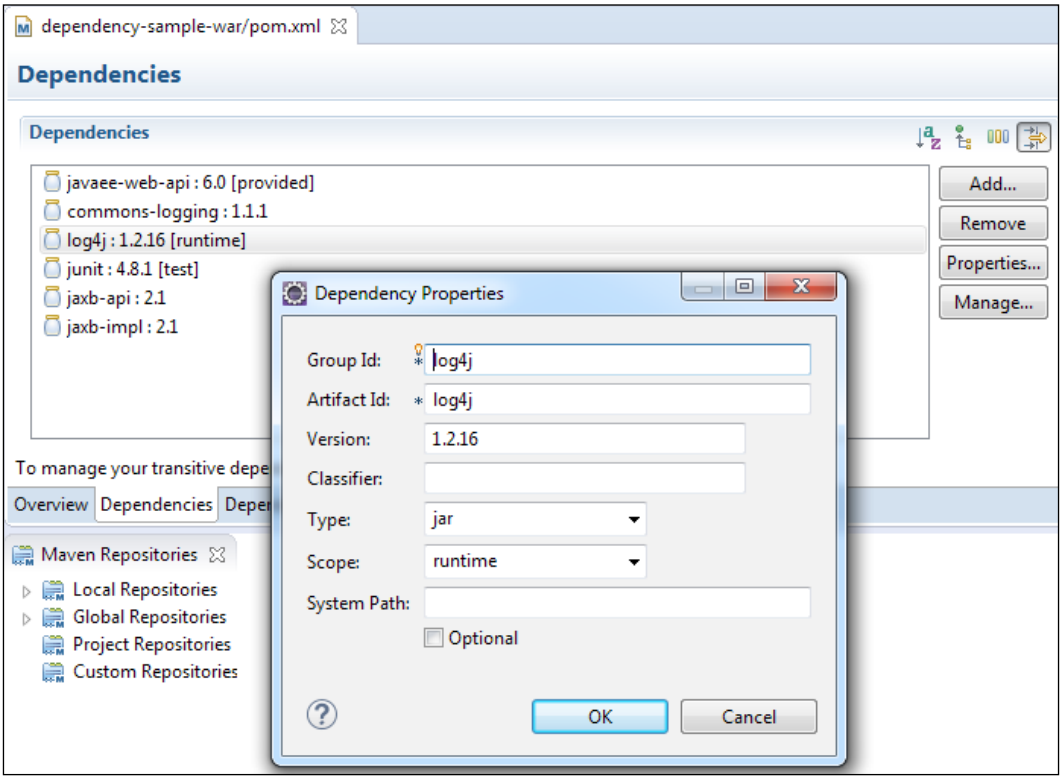
Managing the POM

We can edit our project POM within the Eclipse IDE by clicking on the `pom.xml` file. By default, it will be opened with the Maven POM editor that is shown in the following screenshot:



Maven POM editor

By clicking on the **Overview** tab, we can edit the Maven coordinates and also the main POM elements such as packaging, name, description, project properties, project parent, and project modules. In the last tab, **pom.xml**, we can make changes to the POM file with a text editor. We can also see a tab that displays the **Effective POM** (described in *Chapter 2*, *Core Maven Concepts*) and one named **Dependency Hierarchy**, which is a representation of the dependency tree (see again *Chapter 2*, *Core Maven Concepts*). Project dependencies can be edited by clicking on the **Dependencies** tab, as shown in the following screenshot:

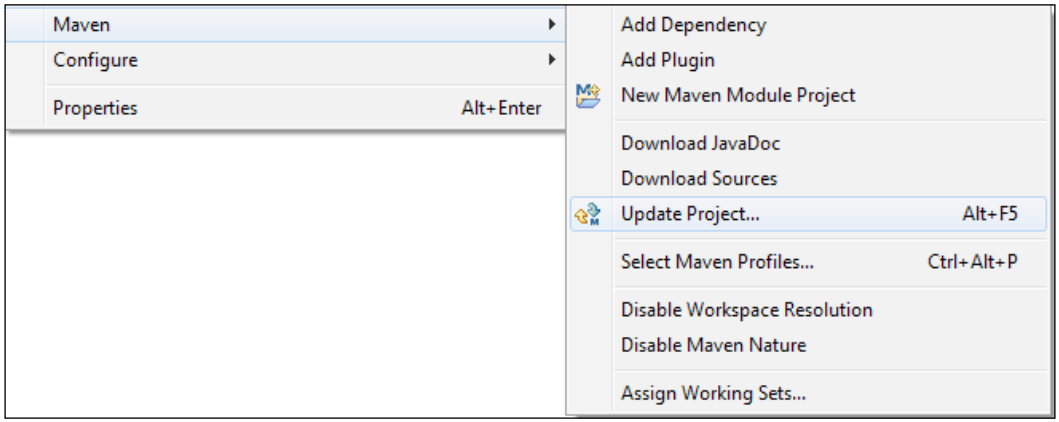


Editing Maven dependencies

All the changes that we make to the project POM through the Maven POM editor are automatically reflected by the Eclipse IDE and the Eclipse project configuration is consequently updated. For example, if we add a dependency in our POM, even through the text editor, it will be immediately visible in the **Package Explorer** view under the **Maven Dependencies** classpath folder.

Sometimes, the `pom.xml` file is marked with a red cross and in the **Problems** view appears the error message **Project configuration is not up to date with pom.xml. Run Maven->Update Project or use Quick Fix**.

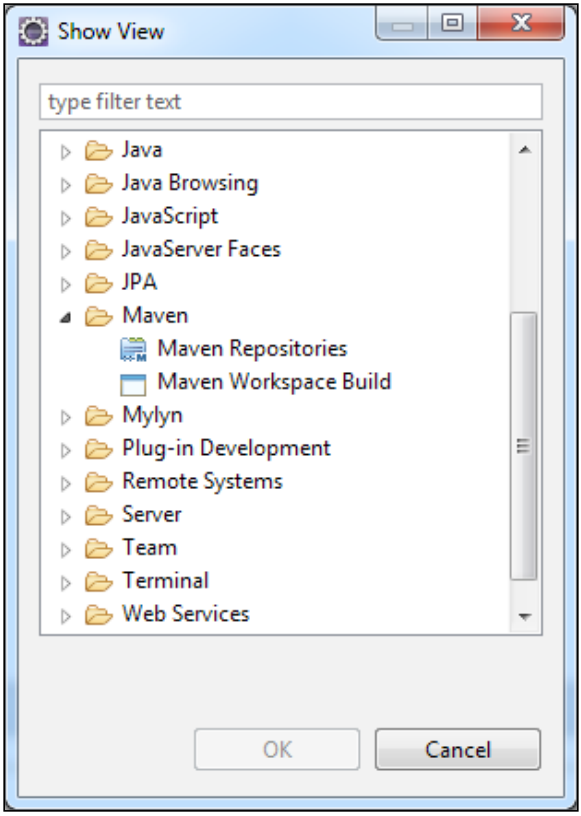
This means that a manual update of the project configuration is needed, and it can be done by right-clicking on the project and then on the **Update Maven Project** menu item, as shown in the following screenshot:



Updating the Maven project configuration

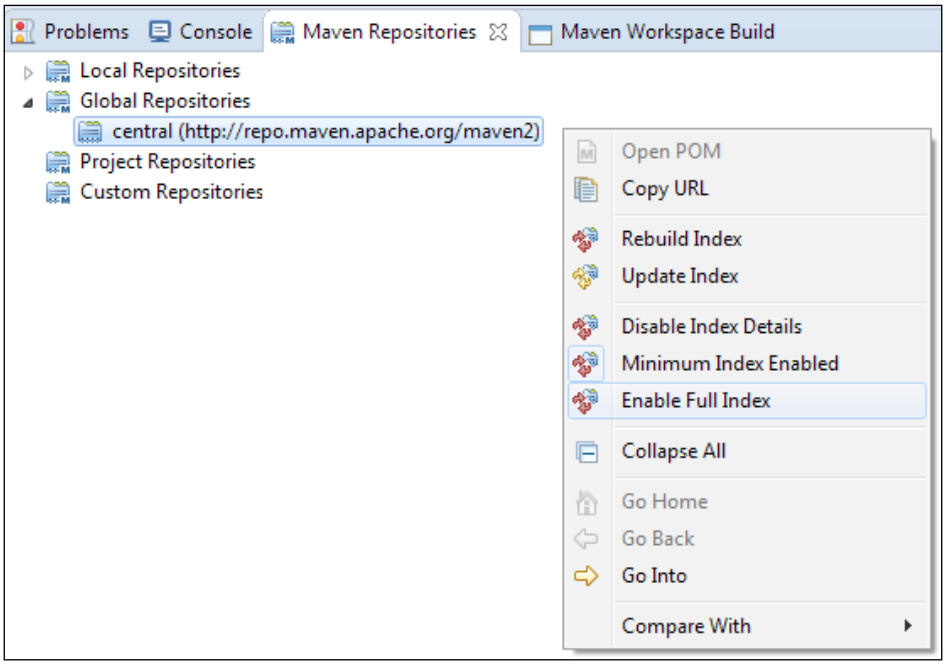
Managing repository indexes

Thanks to the m2e plugin, we can navigate the remote repositories declared in our projects and open the POMs contained in them with the Maven POM editor. This happens through repository indexes. In order to enable repository indexes, we have to open the **Maven Repositories** view by navigating to **Window | Show View | Maven**, as shown in the following screenshot:



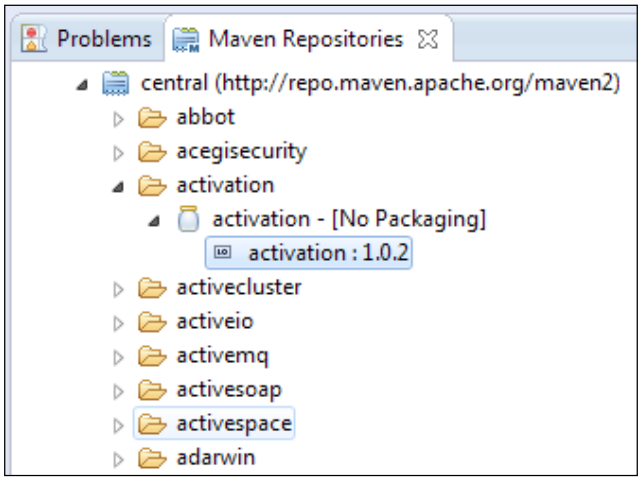
The Maven Repositories view

Then we have to choose a repository (for example, the Maven Central) and enable the repository index, as shown in the following screenshot:



Enable the Maven repository index

After waiting a while, we will able to browse the repository as shown in the following screenshot:



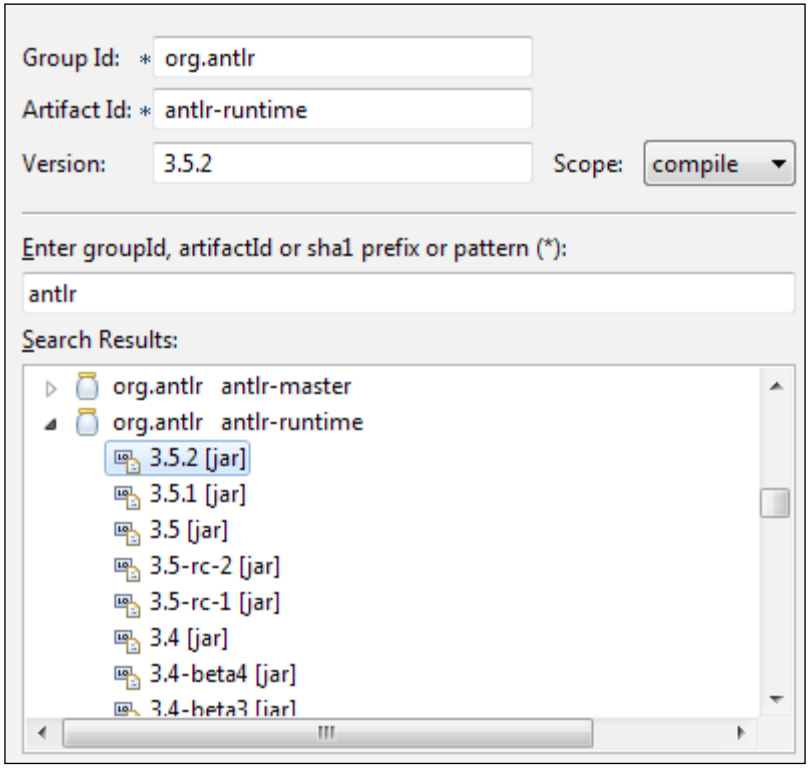
Browsing Maven repositories

We can keep a repository index up to date if we periodically right-click on the repository and then on the **Update Index** menu item.

Managing dependencies and plugins

Repository indexes are also useful for adding dependencies and plugins from the project pop-up menu or from the Maven POM editor.

For example, when we click on the **Add...** button of the **Dependencies** tab of the Maven POM editor, we can insert the Maven coordinates of the needed dependency and confirm, and a new dependency element will be created in our POM. If we need a certain dependency but we do not exactly know its Maven coordinates, we can input a search string and the m2e plugin will query the repository indexes. We can choose the correct dependency in the search results that appear at the bottom of the window, as shown in the following screenshot (in which we are looking for the ANTLR dependency):



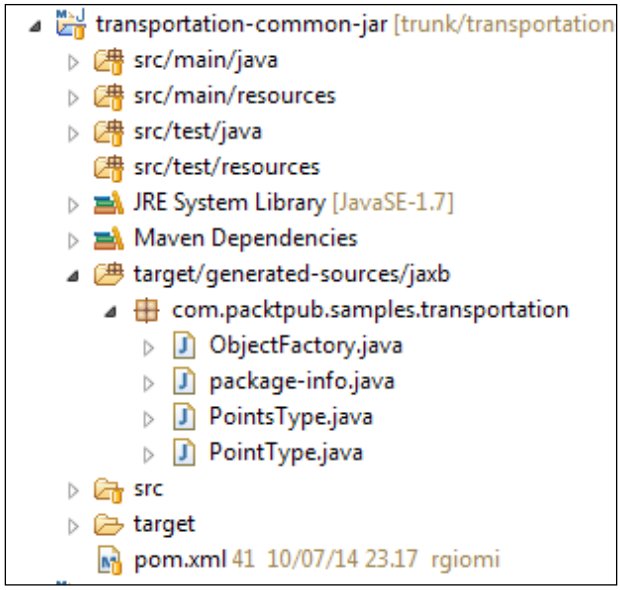
Searching Maven dependencies

m2e connectors and lifecycle mapping

At the beginning of this appendix, we spoke about the Maven plugin connectors, which work in the background to guarantee the execution of the Maven plugins during the Eclipse build process. For example, if we declare the JAXB-2 Maven Plugin in our project as follows (as we did in the `transportation-common-jar` module of our sample project; see *Chapter 2*, *Core Maven Concepts*):

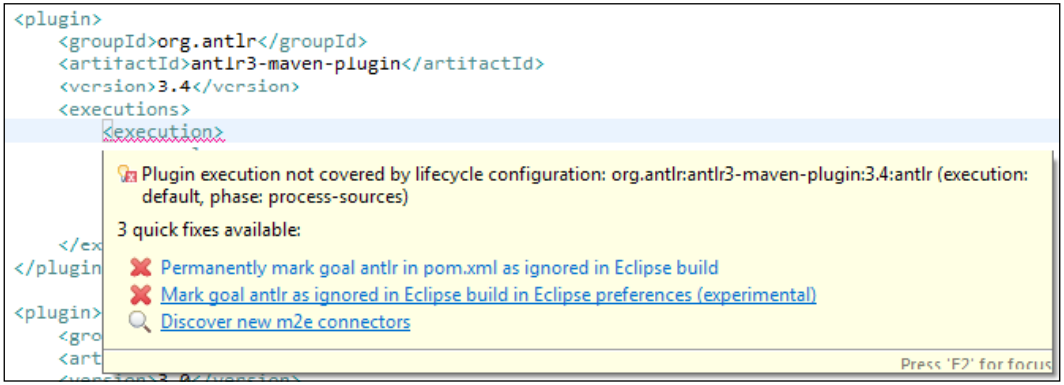
```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>1.6</version>
  <executions>
    <execution>
      <id>myExecution</id>
      <goals>
        <goal>xjc</goal>
      </goals>
      <configuration>
        <schemaDirectory>src/main/resources/schema/
        </schemaDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

When we put an XSD schema in the specified schema directory, the Eclipse IDE will automatically invoke the `xjc` goal of `jaxb2-maven-plugin`, which is bound to the `generate-sources` phase; the plugin will create an additional source folder `target/generated-sources/jaxb`, which is the default output directory of the plugin. Even without invoking Maven explicitly, the project will appear as shown in the following screenshot:



Structure of a project with generated sources

Of course m2e cannot support every Maven plugin out of the box. It could happen that while declaring a certain plugin, for example, `org.antlr:antlr3-maven-plugin`, we get the error **Plugin execution not covered by lifecycle configuration**, as shown in the following screenshot. Notice that we have to hover with the mouse on the underlined `<execution>` element of the plugin configuration to display this pop-up message:

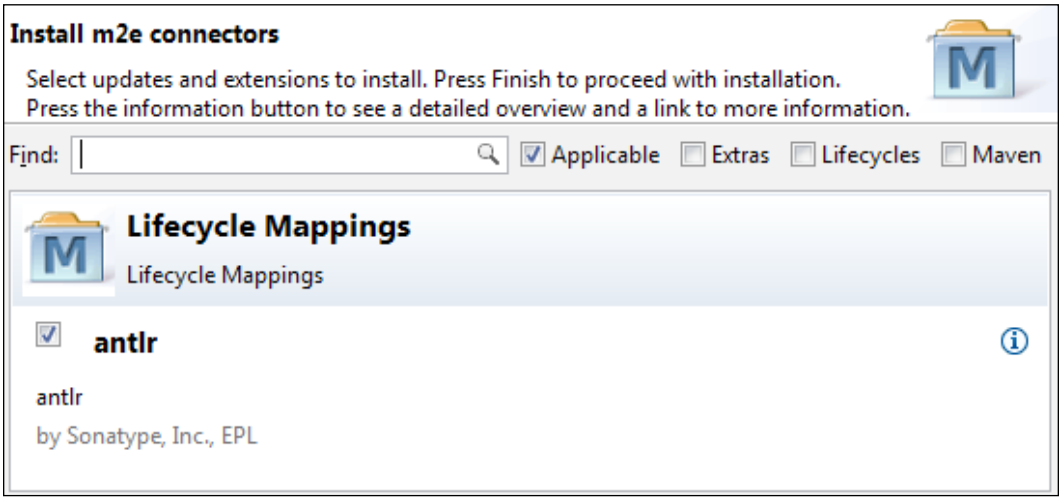


Plugin execution error for antl3-maven-plugin

Here we have three quick fixes available, and we will firstly try the third option Discover new m2e connectors. If we are lucky, we will find a suitable connector on the Eclipse Marketplace, as shown in the following screenshot, and we will proceed with installing it. Once the connector has been installed, our Eclipse IDE will manage the `antlr3` Maven plugin correctly.

The `antl3-maven-plugin` is able to generate the Java sources corresponding to the given grammar files, as in the case of the previous JAXB-2 plugin example.

The **Install m2e connectors** window is shown in the following screenshot:

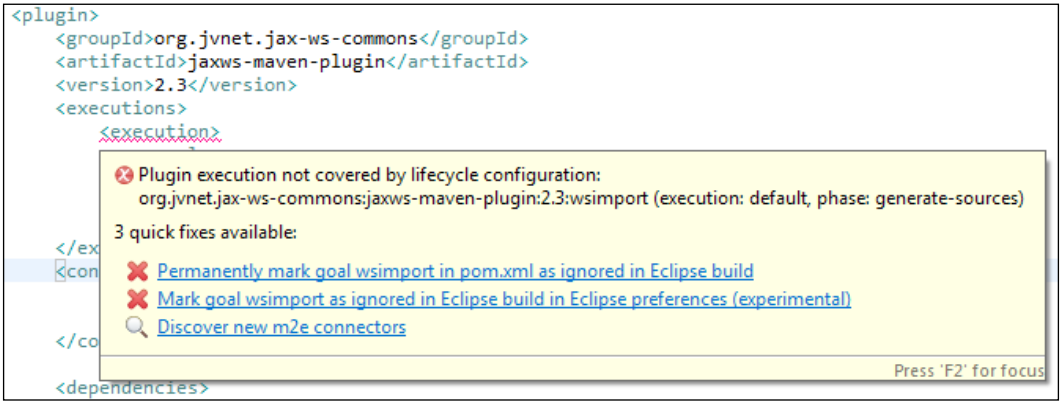


When we cannot find a suitable m2e connector, we have to choose between the first and the second quick fixes proposed by the pop-up error message. This is the case of `jaxws-maven-plugin`, for which, at the moment, a m2e connector is not available on the Eclipse Marketplace. Suppose that we have to generate the JAX-WS Java client for a web service with a given WSDL descriptor; we would have to declare the following Maven plugin in the `<build><plugins>` section of our POM:

```
<plugin>
  <groupId>org.jvnet.jax-ws-commons</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
  <version>2.3</version>
  <executions>
    <execution>
      <goals>
        <goal>wsimport</goal>
```

```
        </goals>
      </execution>
    </executions>
    <configuration>
      <wsdlDirectory>src/main/resources/wsdl/</wsdlDirectory>
    </configuration>
    <dependencies>
      <dependency>
        <groupId>com.sun.xml.ws</groupId>
        <artifactId>jaxws-tools</artifactId>
        <version>2.2.6</version>
      </dependency>
    </dependencies>
  </plugin>
```

We have to put our WSDL file in the directory specified with the `<wsdlDirectory>` element. This plugin, as usual, is bound to the `generate-sources` phase. The error message that we receive is shown in the following screenshot, and the search for a suitable connector does not give any results:



Plugin execution error for jaxws-maven-plugin

The second quick fix, **Mark goal <goal-name> as ignored in Eclipse build in Eclipse preferences**, simply disables this error message acting directly on the Eclipse project configuration and leaves the plugin ignored by the build process. As the POM is not interested in this setting, all the developers working on this project will encounter this error.

This happens if the Eclipse configuration files and directories are not added to the source version control (and we recommended adding them to `.svnignore` / `.cvsignore`).

On the contrary, if we choose the first quick fix **Permanently mark goal <goal-name> in pom.xml as ignored in Eclipse build**, a dummy plugin configuration will be automatically inserted in the POM, in the `<pluginManagement>` section:

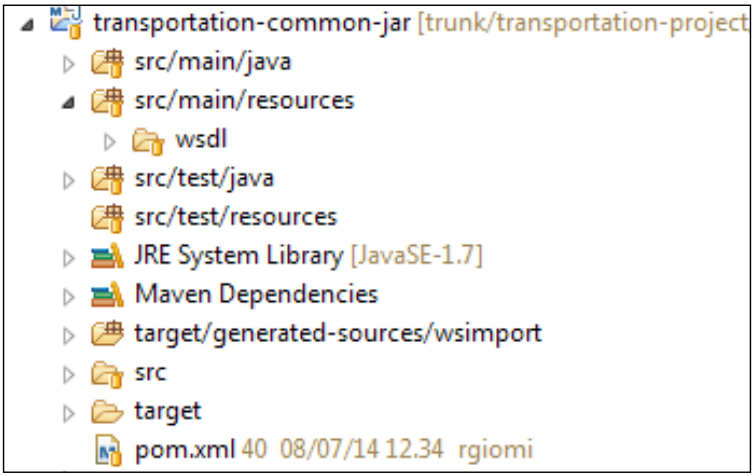
```
<pluginManagement>
  <plugins>
    <!--This plugin's configuration is used
        to store Eclipse m2e settings only.
        It has no influence on the Maven build itself.
    -->
    <plugin>
      <groupId>org.eclipse.m2e</groupId>
      <artifactId>lifecycle-mapping</artifactId>
      <version>1.0.0</version>
      <configuration>
        <lifecycleMappingMetadata>
          <pluginExecutions>
            <pluginExecution>
              <pluginExecutionFilter>
                <groupId>
                  org.jvnet.jax-ws-commons
                </groupId>
                <artifactId>
                  jaxws-maven-plugin
                </artifactId>
                <versionRange>
                  [2.3,)
                </versionRange>
                <goals>
                  <goal>wsimport</goal>
                </goals>
              </pluginExecutionFilter>
              <action>
                <ignore></ignore>
              </action>
            </pluginExecution>
          </pluginExecutions>
        </lifecycleMappingMetadata>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
```

The `lifecycle-mapping` plugin does not exist as a Maven plugin and it is not downloaded from any repositories. This is only a directive for the Eclipse IDE.

This way, `jaxws-maven-plugin` will be ignored permanently by all the developers who have to check out the project and open it in their Eclipse IDEs. None of them will encounter the previous error. Now that we have fixed this problem, we still have the original one: how can we execute the `wsimport` goal? In this case, we have to manually launch the Maven build from the **Debug As…** or **Run As…** menus. The Maven process will execute all the phases of the lifecycle with their bindings, as we ran it from the command line. We have to remember to launch the Maven execution every time we change the WSDL descriptor, in order to keep the Java sources up to date. Unfortunately, there is still another problem to be solved: the Eclipse IDE does not see the generated sources under `target/generated-sources/wsimport` and so we cannot use them in the project as we might get a lot of compilation errors in the IDE. Of course the Maven build process invoked manually will succeed, but we want to work on our project within Eclipse and we cannot accept a "broken" Java project. We can solve this issue using a workaround that consists of declaring `buid-helper-maven-plugin` in our POM as follows:

```
<build>
[…]
<plugins>
[…]
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.8</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>add-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>target/generated-sources/wsimport</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The Build Helper Maven plugin allows you to insert additional source directories in your projects. While we can define multiple directories for resources and web resources, the source directory is unique. There will not be a need for this plugin if we used Maven only from the command line, because the source directories added in the `generate-sources` phase are considered by the compiler plugin. As Eclipse is ignoring the `jaxws-maven-plugin`, it does not know anything about its output source directory, but it can consider the additional source directory defined through `build-helper-maven-plugin`. For its part, the Build Helper plugin needs a connector, which can be found on the Eclipse Marketplace. The final result is shown in the following screenshot:

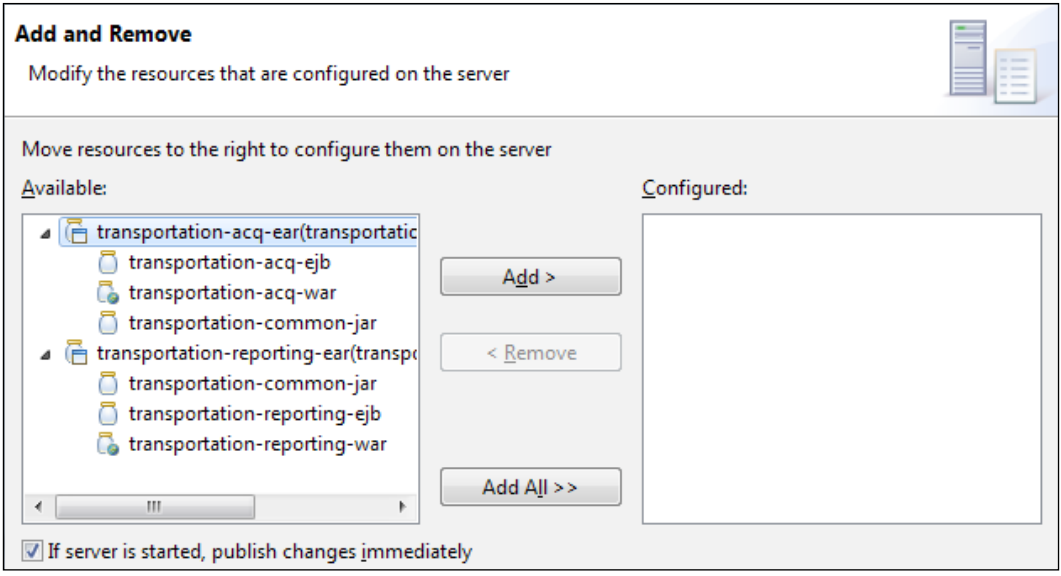


An additional source directory is recognized, thanks to the Build Helper Maven Plugin.

Managing Java EE projects

The Maven Integration for Eclipse provides a number of m2e connectors that configure our Maven Java EE projects in the **Web Tools Project** (**WTP**) environment. These integration features add the needed project facets to WAR, EJB, and EAR projects. Finally, the m2e plugin supports us in converting Eclipse WTP projects to Maven Java EE projects.

Ultimately, a Maven Java EE project can be treated as an Eclipse WTP project and can be deployed and debugged locally in the Eclipse IDE. All we have to do is click on the **Add** and **Remove…** menu item of any server defined in the workspace, and add the desired project to the server, as shown in the following screenshot:



Adding Maven projects to an application server

The consequence of this operation is that the moved resources will be deployed on the selected application server. This happens immediately if the server is running; otherwise, it will happen when the server is started.

C
Maven Global Settings

Maven's configuration can be easily customized by working with the `settings.xml` file. There are two locations where we can find the file:

- The Maven global settings: `$M2_HOME/conf/settings.xml`
- The user's settings: `${user.home}/.m2/settings.xml`

If both locations exist, the content will be merged, but the user's settings get the highest priority.

The settings.xml file

Generally, the `settings.xml` file holds the following elements:

- The location of the local repository
- The default user interaction policy
- The servers' configurations
- The profiles to use
- Other issues about plugins and mirrors not discussed in this book

The following code shows a simple user's `settings.xml` file:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <localRepository>
    ${user.home}/.m2/repository
  </localRepository>
  <interactiveMode>true</interactiveMode>
```

```
    <usePluginRegistry>false</usePluginRegistry>
    <offline>false</offline>

  </settings>
```

Sometimes, especially on a remote organization's build server, you might need to change the default local repository location since the partitioning for the current user is too small. The following code shows how to change our default `settings.xml` file:

```
[…]
<localRepository>
        /my-high-space-disk/maven-repo
</localRepository>

<interactiveMode>false</interactiveMode>
[…]
```

It is also a good idea to set the `interactiveMode` tag to `false` so as to prevent any request from Maven on a build server (check out the *Continuous integration and delivery with Hudson or Jenkins* section of *Chapter 5*, *Continuous Integration and Delivery with Maven*).

Servers

In the `servers` section, we can specify some important deployment settings such as the username and password of the remote repository (refer to *Chapter 5*, *Continuous Integration and Delivery with Maven*). The following lines define the username and password as per the `nexus` server:

```
[…]
<servers>
  <server>
    <id>nexus</id>
    <username>admin</username>
    <password>admin123</password>
  </server>
</servers>
[…]
```

Proxies

Maven needs an Internet connection to download dependent artifacts or plugins. If the network of your organization is controlled by a proxy, you need to define the proxies on `settings.xml`:

```
[…]
<proxies>
  <proxy>
    <active>true</active>
    <protocol>http</protocol>
    <host>myproxy</host>
    <port>8080</port>
    <username>mydomain\myuser</username>
    <password>mysecret</password>
    <nonProxyHosts>localhost,my-server</nonProxyHosts>
  </proxy>
  <proxy>
    <active>true</active>
    <protocol>https</protocol>
    <host>myproxy</host>
    <port>8081</port>
    <username>mydomain\myuser</username>
    <password>mysecret</password>
    <nonProxyHosts>localhost,my-server</nonProxyHosts>
  </proxy>

</proxies>
[…]
```

The actual version of Maven doesn't support automatic proxy through the PAC script.

Profiles

The `settings.xml` file provides a fine-grained mechanism to control profiles. Profiles are extensively described in *Chapter 4*, *Managing the Code*; a profile can be easily activated through the `-P` parameter:

```
$ mvn clean install -P myprofile
```

A profile can also be defined using the `activeByDefault` element. It is possible to activate a profile through the `settings.xml` file. The following code activates the `build-jdk5` profile when you run Maven under JDK 5:

```
[…]
<profiles>
  <profile>
    <id>build-jdk5</id>
    <activation>
      <activeByDefault>false</activeByDefault>
      <jdk>1.5</jdk>
    </activation>
  </profile>
</profiles>
[…]
```

The following code sets the `delivery-host` variable if a `target-env` property is `dev`:

```
[…]
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <property>
        <name>target-env</name>
        <value>dev</value>
      </property>
    </activation>
    <properties>
      <delivery-host>my-dev-host</delivery-host>
    </properties>
  </profile>
</profiles>
[…]
```

Prior to Maven's compile phase, you can test the active profiles launching with the following command line:

```
$ mvn help:active-profiles
```

We should get the following output:

```
Active Profiles for Project 'com.packt.examples:transp-acq-ear:1.0':
 Dev
 Prod


The following profiles are active:
 Dev
```

D
Maven Short References – Common Commands and Archetypes

"*Imagination is more important than knowledge...*"

Albert Einstein

This chapter summarizes the most important commands and concepts covered in the book, and it provides a textual mind map of Maven. The first section summarizes Maven's commands and related parameters covered during the book. The second section reports the complete list of Maven's variables. The last paragraph shows Maven's lifecycle.

Commands

Maven can be executed by the command line to build, check dependencies and code, deploy, and release artifacts. A complete explanation of these commands can be found at `http://maven.apache.org/`; the following subsections explain only the most common commands.

Build

Maven will clean the workspace, compile, and install on the local repository:

```
$ mvn clean install
```

Or, alternatively, you can specify the `pom.xml` file's name:

```
$ mvn clean install –f pom.xml
```

This command is the standard Maven call.

```
$ mvn clean install –f my_pom.xml
```

Given the specified POM, Maven will clean the workspace, compile, and install on the local repository. Also refer to *Chapter 2*, *Core Maven Concepts*; *Chapter 3*, *Writing Plugins*; and *Chapter 5*, *Continuous Integration and Delivery with Maven*.

```
$ mvn clean install -DskipTests
```

Maven will skip all tests (**Surefire Plugin** and **Failsafe Plugin**). Also refer to *Chapter 3*, *Writing Plugins* and *Chapter 5*, *Continuous Integration and Delivery with Maven*.

```
$ mvn --non-recursive clean compile
```

Or you can use a short notation:

```
$ mvn -N clean compile
```

Maven will clean the workspace and will compile, but it does not recurse into subprojects. It is useful to install the parent POM avoiding submodules compiling:

```
$ mvn -U clean compile
```

Maven will clean the workspace, compile, and update the local repository:

```
$ mvn clean install –pl my_artifact -am
```

Given the artifact ID, Maven will clean the workspace, compile, and install the module specified and all snapshot dependencies on the local repository. Also refer to *Chapter 3*, *Writing Plugins*, and *Chapter 5*, *Continuous Integration and Delivery with Maven*. It is useful to install a specific artifact avoiding the aggregator POM.

Deploy and release

Deploy the artifact on the remote repository configured (see the `distributionManagement` tag) into the given POM or parent POM or passed as parameters:

```
$ mvn deploy
```

Or, alternatively, you can specify to deploy the artifacts at the end of the multimodule build:

```
$ mvn deploy –DdeployAtEnd=true
```

You can also specify the final destination:

```
$ mvn deploy:deploy –DaltDeploymentRepository=http://myhost
```

Also refer to *Chapter 2*, *Core Maven Concepts*, and *Chapter 5*, *Continuous Integration and Delivery with Maven*. If the repository requires user access, configure the `settings.xml` file. The `deployAtEnd` parameter is useful for multimodules since Maven will deploy all project reactors at the end of the build phase:

```
$ mvn release:clean release:prepare
```

Rollback the action:

```
$ mvn release:rollback
```

The preceding command performs the release preparation (Maven release plugin) or the rollback (also refer to *Chapter 5*, *Continuous Integration and Delivery with Maven*).

```
$ mvn –-batch-mode -f MyMultiModule_pom.xml \
-DallowTimestampedSnapshots=true -DignoreSnapshots=true \
-DreleaseVersion=0.0.2 \
release:clean release:prepare
```

The preceding command performs the release preparation (Maven release plugin) setting the current release version to 0.0.2 of the given multimodule POM (refer to *Chapter 5*, *Continuous Integration and Delivery with Maven*):

```
$ mvn scm:checkin
```

Or you can update the current workspace with:

```
$ mvn scm:update
```

The preceding command performs the commit or update from the current SCM (SVN and GIT) repository (refer to *Chapter 5*, *Continuous Integration and Delivery with Maven*).

Android

Maven shows the available devices, deploys the application, and launches the Android emulator:

```
$ mvn android:devices
$ mvn android:deploy
$ mvn android:run
```

Execute the following command to show all the available options:

```
$ mvn android:help
```

Miscellaneous

The following command shows the plugin's details (refer to *Chapter 2*, *Core Maven Concepts*):

```
$ mvn help:describe -DgroupId=MyGroupId \
                    -DartifactId=MyPlugin \
                    -Dversion=0.0.0
```

Generate a text reporting the project's dependency tree:

```
$ mvn dependency:tree
```

You can customize the verbosity through the following command:

```
$ mvn dependency:tree -Dverbose -Dincludes=MyLibrary
```

Through the `includes` parameter, we can filter only the specified packages (refer to *Chapter 2*, *Core Maven Concepts*).

```
$ mvn dependency:purge-local-repository
```

This goal is meant to delete all of the dependencies for the current project from the local repository.

```
$ mvn help:active-profiles
```

This goal shows the active profiles (refer to *Chapter 4*, *Managing the Code*).

```
$ mvn site
```

This goal generates the reports (site or FindBugs or PMD) on the target directory.

```
$ mvn assembly:single
```

This goal is executed to package the project in conjunction with the Assembly Plugin.

```
$ mvn package
```

This goal executes a custom assembly described within the descriptor file (refer to *Chapter 4*, *Managing the Code*).

Archetypes

Archetypes are templates for generating projects. The following list reports the most common archetypes used in this book:

```
$ mvn archetype:create \
  -DgroupId=com.packt.myexamples \
  -DartifactId=MyProject
```

Maven generates a simple project:

```
$ mvn archetype:generate \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-plugin \
-DarchetypeVersion=1.2
```

Maven generates a simple project within a class called `MyMojo.java` with the default method implemented. The generated structure is:

```
project
|-- pom.xml
`-- src
    `-- main
        `-- java
            `-- MyMojo.java
```

```
$ mvn archetype:generate \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-web
```

The archetype generates a web application project:

```
$ mvn archetype:generate \
-DarchetypeArtifactId=android-quickstart \
-DarchetypeGroupId=de.akquinet.android.archetypes \
-DarchetypeVersion=0.1.0 -DgroupId=MyAndroidProject \
-DartifactId=MyAndroidArtifact
```

Maven generates an Android archetype.

Maven variables

Using Maven's variables is an easy way to customize your `pom.xml` file automatically. The following table reports a complete list of these variables:

Variables	Description
`${project.name}`	This is the built-in property that contains the name of the project.
`${project.artifactId}`	This is the built-in property that contains the unique identifier of the artifact.
`${project.description}`	This is the built-in property that contains the description of the project.
`${project.groupId}`	This is the built-in property that contains the unique identifier of the group.
`${project.baseUri}`	This is the built-in property that contains the URI of the project.
`${project.version}`	This is the built-in property, equivalent to `${version}`, containing the version of the project.
`${project.parent.version}` `${project.parent.groupId}`	These are the built-in properties that contain the version or the group ID of the parent POM.
`${basedir}`	This is the built-in property representing the directory in which the `pom.xml` file is stored.
`${project.build.directory}`	This is a property, defined in the central Maven's POM, containing the path of the build directory. The default value is `target`.

`${project.build.sourceDirectory}`
`${project.build.scriptSourceDirectory}`
`${project.build.testSourceDirectory}`
	This is a set of properties, containing the path of Java/script sources.

`${project.build.outputDirectory}` `${project.build.testOutputDirectory}`	This is a set of properties, defined in the central Maven's POM, containing the directory in which class files are stored during the build process. The default value is `target/classes`.
`${project.build.finalName}`	This is the built-in property containing the final name of the file created when the built project is packaged.

Variables	Description
`${settings.localRepository}`	This is an environment variable, containing the Maven2 installation folder.
`${env.M2_HOME}`	This is an environment variable containing the Maven2 installation folder.
`${env.HOME}`	This is the built-in property containing the user's home directory.
`${env.PATH}`	This is the built-in property containing the current path in which Maven is running.
`${env.JAVA_HOME}`	This is an environment variable specifying the path to the current `JRE_HOME` folder.
`${ENV.*}`	Through this suffix, we can access the OSes environment variables.
`${settings.*}`	Through this suffix, we can access the `settings.xml` variables.
`${java.home}` or `${java.version}` or `${os.version}` or `${user.home}` or `${user.name}`	The Java environment variables are accessible by Maven. Here, we report the most common Java environment's variables.

The default and clean Maven lifecycle

Maven's lifecycle is responsible for the build process. The default phases are described in the following table:

Phase	Actions
`validate`	Validate the project and the directives provided
`initialize`	Read and set properties or create directories
`generate-sources`	Generate sources for the compilation
`process-sources`	Process source code; for example, filter values
`generate-resources`	Generate resources for the compilation
`process-resources`	Filter the resource files and copy them in the output directory
`compile`	Compile the source code
`process-classes`	Process classes just compiled; for example, bytecode instrumentation
`generate-test-sources`	Generate sources for the test

Phase	Actions
`process-test-sources`	Process source code for the test; for example, filter values
`generate-test-resources`	Generate resources for the test
`process-test-resources`	Filter the test resource files and copy them in the test output directory
`test-compile`	Compile the test source code
`process-test-classes`	Process classes just compiled for test; for example, bytecode instrumentation
`test`	Run the unit tests
`prepare-package`	Execute operations before packaging
`package`	Produce the packaged artifact (JAR, WAR, and EAR)
`pre-integration-test`	Perform operations before the integration tests; for example, start a server
`integration-test`	Launch integration tests
`post-integration-test`	Perform operations after the integration tests; for example, stop a server
`verify`	Verify the correctness of the package just created
`install`	Install the package in the local repository so that other projects can use it as a dependency
`deploy`	Install the package in a remote repository

The clean phases are described in the following table:

Phase	Actions
`preclean`	Preclean phase
`clean`	Remove files generated from the previous build
`postclean`	Finalize the clean phase

Index

Symbols

${basedir}, Maven variables 240
${descriptorDir} variable 110
${env.HOME}, Maven variables 241
${env.JAVA_HOME}, Maven variables 241
${env.M2_HOME}, Maven variables 241
${ENV.*}, Maven variables 241
${env.PATH}, Maven variables 241
${java.home}, Maven variables 241
${java.version}, Maven variables 241
${project.artifactId}, Maven variables 240
${project.build.directory}, Maven variables 240
${project.build.finalName}, Maven variables 240
${project.build.outputDirectory}, Maven variables 240
${project.build.scriptSourceDirectory}, Maven variables 240
${project.build.sourceDirectory}, Maven variables 240
${project.build.testOutputDirectory}, Maven variables 240
${project.build.testSourceDirectory}, Maven variables 240
${project.description}, Maven variables 240
${project.parent.groupId}, Maven variables 240
${project.parent.version}, Maven variables 240
${project.version}, Maven variables 240
${property-name} syntax 51
${settings.localRepository}, Maven variables 241
${settings.*}, Maven variables 241
${user.home}, Maven variables 241
${user.name}, Maven variables 241
-am parameter 63
-Ddetail option 34
-Ddetail parameter 28
@Mojo annotation
  about 76
  requiresOnline 76
  requiresProject 76
  threadSafe 76
@Parameter annotation 77
-pl parameter 63
<profiles> element 50
<proxies> element 50
<servers> element 50

A

activations element 101
aggregate directive 122
aggregate POMs 62, 63
Almost Plain Text (APT) 126
Android application
  AndroidManifest file, creating 179, 180
  AndroidManifest file, modifying 179, 180
  creating 178
  creating, with archetype 178
  Maven plugin goals, building with 185
  simple Maven POM file, defining 181, 182
Android connector, Eclipse integration
  installing 196
Android Developer Tools (ADT) 196
Android emulator
  launching 237, 238

AndroidManifest file
creating 179, 180
modifying 179, 180
Android Maven Plugin
prerequisites 177
Ant
about 138, 139
Ant-Maven integration 140
custom tasks 139
installing 138
Maven-Ant integration 139, 140
tasks, URL 139
Ant-Maven integration 140, 141
Ant tasks
else 139
if 139
maven-ant 139
Svnant 139
then 139
Apache Ant. *See* **Ant**
Apache Maven 199
API v4
library, compatibility 186
archetype
about 239
used, for creating Android application 178
archive
building, through Assembly plugin 107, 108

B

basedir property 77
BatchHandler 106
best practices, POMs
about 62
aggregate POMs 62, 63
dependency management 63-65
plugin management 65, 66
binaries element 109
bin descriptor 108
bug detector 194, 195
bug fixing
about 162
MantisBT 163-165

C

central POM, properties
${basedir} 79
${env.M2_HOME} 79
${java.home} 79
${project.build.directory} 79
${project.build.finalName} 79
${project.build.outputDirectory} 79
${project.name} 79
${settings.localRepository} 79
${version} 79
CID
about 130
Build Phase 131
Hudson, configuring 148, 149
Hudson, installing 147
Hudson, working with 150-153
key concepts 130
with Hudson 147
with Jenkins 147
clean 19
clean lifecycle, phases
clean 26
postclean 26
preclean 26
commands, Maven
Android emulator, launching 237, 238
build 235
deploy 236
miscellaneous 238
release 236
URL 235
compatibility library
for API v4 186
compile phase, Maven 241
compile scope 38
component, transportation project
selecting, to build 166-169
containerDescriptorHandler 108
Continuous Delivery 129
Continuous Integration 129
Continuous Integration and Delivery. *See* **CID**

Continuous Integration Management (CIM) 118
custom plugin
implementations 93-97
mantis-maven-plugin 93

D

default 19
default bindings 31, 32
default lifecycle 20-26
default phases, Maven lifecycle
compile 241
deploy 242
generate-resources 241
generate-sources 241
generate-test-resources 242
generate-test-sources 241
initialize 241
install 242
integration-test 242
package 242
post-integration-test 242
pre-integration-test 242
prepare-package 242
process-classes 241
process-resources 241
process-sources 241
process-test-classes 242
process-test-resources 242
process-test-sources 242
test 242
test-compile 242
validate 241
verify 242
dependencies
declaring 185
dependency inheritance 46
dependency scopes 38-41
dependency tree 43, 44
effective POM 46-49
managing 37, 221
POM file, used with 187
super POM 46-49
transitive dependencies 43-45
version ranges 42
dependency inheritance 46
dependency management 63-65
dependency scopes
about 38-41
compile 38
provided 38
runtime 38
system 38
test 38
dependency tree 43, 44
deployAtEnd parameter 237
deploy phase, Maven 242
description tags, Maven POM file 183
descriptor file 107, 108
descriptor file, Maven Assembly Plugin 108, 109
directed acyclic graph (DAG) 199
domain-specific language (DSL) 199

E

EAR, packaging type
about 31
default bindings 31
Eclipse
URL 11
Eclipse integration
about 196
Android connector, installing 196
Mavenized Android Project 196
EE applications
building 54
enterprise applications, building 55-60
WEB applications, building 54, 55
effective POM 46-49
EJB, packaging type
about 30
default bindings 30
enterprise applications
building 55-60
environment, Maven Assembly Plugin
fitting to 106, 107
environment properties 51
excludes tag 112
execution-level configuration 34, 35

existing Maven projects
  importing 210, 211
extreme programming (XP) 129

F

Failsafe Plugin 236
file element 103, 109
fileSet element 108
fileSets tag 110
filters 53
FindBugs 160-162

G

generate-resources phase, Maven 241
generate-sources phase, Maven 241
generate-test-resources phase, Maven 242
generate-test-sources phase, Maven 241
getPluginContext() method 80
getTestFile method 84
GIT 132
goal 19, 26
Gradle
  about 199, 200
  integration plugins 200
  overview 199
  plugins, for other languages 200
  programming languages, building 200
  project configuration 202-204
  URL 201
  used, for creating project 201
  working 201
groupVersionAlignment element 109

H

Hudson
  about 132
  CID 147
  configuring 148, 149
  installing 147
  Maven-Hudson integration 153, 154
  URL 147
  working with 149-153
Hudson, transportation project
  configuring 171

I

includes directive 112
inheritance, POM file 8
initialize phase, Maven 241
install command 115
install phase, Maven 242
instrumentations 189
integration plugins, Gradle
  application 200
  ear 200
  jetty 200
  war 200
integration testing 84-87, 159, 160
Inversion of Control (IoC) 81

J

JAR, packaging type 29
  about 29
  default bindings 29
jar-with-dependencies descriptor 107
Javadoc
  reporting 121-123
Java EE projects
  managing 228, 229
jboss-as-maven-plugin 159
jdk element 103
Jenkins
  about 132
  CID 147
  configuring 148, 149
  URL 147
JIRA 118, 132
junit 72

L

lifecycle-mapping plugin 222-228
lifecycle references 31, 32
lifecycles
  about 19
  building 19
  clean lifecycle 26
  default lifecycle 20-26
lint 194, 195
local repository 24
lookupMojo method 84

M

m2e connectors 222-228
m2e plugin
about 209
settings 215, 216
Mantis
URL 68
MantisBT 118, 132-165
Mantis bug tracker 68
mantis-maven-plugin 93
Maven
about 7, 76
commands 235
variables 240
Maven 2 repository
URL 203
Maven-Ant integration 139
Maven archetype 69
Maven Assembly Plugin
about 105
descriptor file 108, 109
environment, fitting 106, 107
own archive, building through 107, 108
project configuration 109-114
Maven behavior 63
Maven build profiles
about 99, 100
profile activation 102, 103
profile, defining 100
profile structure 101
sample build profiles 103, 104
Maven central repository
URL 47
Maven Failsafe Plugin 159, 160
maven-glassfish-plugin 159
Maven Global Settings 50
Maven goals
about 26, 27
Maven help plugin 27, 28
parameters 27, 28
Maven help plugin 27, 28
Maven-Hudson integration 153
Mavenized Android Project, Eclipse integration 196
Maven lifecycle
clean phases 242
default phases 241
Maven Local Settings 50
Maven plain Old Java Object. *See* **Mojo**
maven-plugin-annotation
about 72
URL 72
maven-plugin-api 72
maven-plugin-plugin 87-93
Maven plugins
adding 33
configuring 33
execution-level configuration 34, 35
goals 184, 185
plugin-level configuration 33, 34
Maven POM file
defining 181, 182
description tags 183
Maven plugin goals, building with 184
Maven projects
building 213, 214
checking, from SCM repositories 211, 212
Maven properties
about 51, 52
environment properties 51
project properties 51
settings properties 51
system properties 51
Maven Release Plugin 143-145
Maven repository
deploying 204, 205
Maven SCM Plugin
about 142
goals 143
Maven settings
about 50
URL 50
Maven site
skinning 124, 125
URL 31, 37

Maven Site Plugin
about 114
project site creation, manually 114-119
simple site, creating 114
maven-surefire-plugin 14
maven-testing-plugin-harness 72
Maven variables
${basedir} 240
${ENV.*} 241
${env.HOME} 241
${env.JAVA_HOME} 241
${env.M2_HOME} 241
${env.PATH} 241
${java.home} 241
${java.version} 241
${os.version} 241
${project.artifactId} 240
${project.baseUri} 240
${project.build.directory} 240
${project.build.finalName} 240
${project.build.outputDirectory} 240
${project.build.scriptSourceDirectory} 240
${project.build.sourceDirectory} 240
${project.build.testOutputDirectory} 240
${project.build.testSourceDirectory} 240
${project.description} 240
${project.groupId} 240
${project.name} 240
${project.parent.groupId} 240
${project.parent.version} 240
${project.version} 240
${settings.*} 241
${settings.localRepository} 241
${user.name} 241
about 240
modularity, POM file 9
moduleSet 108
Mojo
about 76
implementing 76-80
testing 80-82
multimodule component, transportation project
preparing 169-171
multimodule version, transportation project
preparing, with flat structure 172-174

N

Nexus(Nexus Open Source)
about 132
access level 138
customizing 134
installation, testing 134
installing 133
installing, on Unix-based OS 133
installing, on Windows 133
server, configuring 134
server, testing 134
URL 133

O

offline flag 50
overriding, POM file 8

P

package
signing 191-194
zipaligning 191-194
package phase, Maven 242
packaging types
about 28
default bindings 31, 32
EAR 31
EJB 30
JAR 29
lifecycle references 31, 32
POM 30
WAR 29
phases, default lifecycle
compile 20
install 20
package 20
process-resources 20
process-test-resources 20
test 20
test-compile 20
plugin documentation
URL 36
plugin-level configuration 33, 34
plugin management 65, 66

plugins
about 67
developing 68-75
managing 221
POJO (Plain Old Java Object) 76
POM
managing 216-218
POM file
about 8
inheritance 8
modularity 9
overriding 8
repository 9
with dependencies 187
POM, packaging type
about 30
default bindings 30
POM, project
creating 206
pom.xml files 86
post-integration-test phase, Maven 242
postclean phase, Maven 242
pre-integration-test phase, Maven 242
preclean phase, Maven 242
prepare-package phase, Maven 242
process-classes phase, Maven 241
process-resources phase, Maven 241
process-sources phase, Maven 241
process-test-classes phase, Maven 242
process-test-resources phase, Maven 242
process-test-sources phase, Maven 242
profile activation 102, 103
profiles, settings.xml file 234
profile structure 101
project. *See* **transportation project**
project configuration, Gradle 202-204
project configuration, Maven Assembly Plugin 109-114
project descriptor 108
Project Object Model (POM) 8, 19
project properties 51
project site
configuring, for submodule 120
content 126, 127
creating 114-119
Javadoc, reporting 121-123
Maven sites, skinning 124, 125
provided scope 38
proxies, settings.xml file 233

R

release policies
solving 68
releases
enabling 61
release, transportation project
finalizing 174, 175
remote repository
deploying on 145, 146
repositories
configuring 60, 61
managing 135
official repositories, configuring 135, 136
releases, enabling 61
snapshots, enabling 61
User Managed Repository 136, 137
repository element 109
repository indexes
managing 219-221
repository management server
about 132, 135
advantages 133
Nexus, customizing 134
Nexus installation, testing 134
Nexus, installing 133
Nexus server, configuring 134
Nexus server, testing 135
repository, POM file 9
resource filtering 51-53
RoboGuice 2 185
runtime scope 38

S

sample build profiles 103, 104
SCM
about 119, 132
integration 141
Maven Release Plugin 143-145
Maven SCM Plugin 142
Maven SCM Plugin, goals 143
SCM repositories
Maven projects, checking 211, 212

scopes, Maven
compile 203
runtime 203
testCompile 203
testRuntime 203
servers, settings.xml file 232
settings properties 51
settings.xml file
about 86, 231, 232
elements 231
locations 231
profiles 233, 234
proxies, defining 233
servers 232
setUp method 81
shared development environment
aligning 159
site
about 19
creating 114
lifecycle 26
snapshots
enabling 61
software automation
testing 154-156
Software Configuration Management. *See* **SCM**
sources element 108
src descriptor 108
static code analysis tools
about 160-162
cobertura-maven-plugin 161
findbugs-maven-plugin 160
maven-checkstyle-plugin 160
maven-pmd-plugin 161
Subclipse
URL 211
submodule
site, configuring for 120
super POM 46-49
Surefire Plugin 236
system properties 51
system scope 38

T

test-compile phase, Maven 242
testing
best practices 82-84
test phase, Maven 242
test profile 189, 190
test report
scheduling 156-158
test scope 38
tomcat7-maven-plugin 159
transitive dependencies 43-45
transportation-acq-ear 10
transportation-acq-ejb 10
transportation-acq-war 10
transportation-android-apk 10
transportation-common-jar 10
transportation project
about 9-11, 165
creating 11-13
creating, with Gradle 202
component, selecting to build 166-169
functional architecture 9, 10
Hudson, configuring 171
issues 166
multimodule component, version preparing 169-171
multimodule version, preparing with flat structure 172, 173
organization 166
POM, creating 206
release, finalizing 174, 175
structuring 13-17
transportation-acq-ear 10
transportation-acq-ejb 10
transportation-acq-war 10
transportation-android-apk 10
transportation-common-jar 10
transportation-reporting-ear 10
transportation-reporting-ejb 10
transportation-reporting-war 10
transportation-statistics-batch-jar 10
transportation-reporting-ear 10
transportation-reporting-ejb 10
transportation-reporting-war 10
transportation-statistics-batch-jar 10

U

unit tests 154
Unix-based OS
  Nexus, installing on 133
unpackOptions element 109
url tag 116
User Managed Repository 136, 137
user repositories, Nexus
  3rd party 137
  releases 137
  snapshots 137

V

validate phase, Maven 241
verify phase, Maven 242
version ranges 42

W

wagon-ssh-external plugin 159
WAR, packaging type
  about 29
  default bindings 29
WEB applications
  building 54, 55
weblogic-maven-plugin 159
Web Tools Project (WTP) 228
Windows
  Nexus, installing on 133

Z

ZIP file
  backup directory 106
  bin directory 106
  conf directory 106
  etc directory 106
  lib directory 106
  libRun directory 106
  log directory 106