



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

Maven Essentials

Get started with the essentials of Apache Maven and get your build automation system up and running quickly

Prabath Siriwardena

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Maven Essentials

Get started with the essentials of Apache Maven
and get your build automation system up and
running quickly

Prabath Siriwardena



BIRMINGHAM - MUMBAI

Maven Essentials

Copyright © 2015 Packt Publishing

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

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

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

First published: November 2015

Production reference: 1251115

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

ISBN 978-1-78398-676-7

www.packtpub.com

Credits

Author

Prabath Siriwardena

Project Coordinator

Kinjal Bari

Reviewer

Antonio Mendoza Pérez

Proofreader

Safis Editing

Commissioning Editor

Amarabha Banerjee

Indexer

Monica Ajmera Mehta

Acquisition Editor

Shaon Basu

Graphics

Disha Haria

Content Development Editor

Samantha Gonsalves

Production Coordinator

Nilesh Mohite

Technical Editor

Vivek Pala

Cover Work

Nilesh Mohite

Copy Editor

Pranjali Chury

About the Author

Prabath Siriwardena is the director of Security Architecture at WSO2 Inc., a company that produces a wide variety of open source software from data to screen. He is a member of OASIS Identity Metasystem Interoperability (IMI) TC, OASIS eXtensible Access Control Markup Language (XACML) TC, OASIS Security Services (SAML) TC, OASIS Identity in the Cloud TC, and OASIS Cloud Authorization (CloudAuthZ) TC. Prabath is also a member of PMC Apache Axis and has spoken at numerous international conferences, including OSCON, ApacheCon, WSO2Con, EIC, IDentity Next, and OSDC. He has more than 10 years of industry experience and has worked with many Fortune 100 companies.

Acknowledgments

I would first like to thank Shaon Basu, an acquisition editor at Packt Publishing, who came up with the idea of writing a book on Apache Maven; then, Samantha Gonsalves, a content development editor at Packt Publishing, who I worked with closely throughout the project—thank you very much, Samantha, for your patience and flexibility. Also, I would like to thank all the others at Packt Publishing who helped me to make this book a reality from the initial idea. Thank you very much for all your continuous support.

Dr. Sanjiva Weerawarana, the CEO of WSO2, and Paul Fremantle, the CTO of WSO2, have always been my mentors. I am truly grateful to both Dr. Sanjiva and Paul for everything they have done for me.

I would like to thank my beloved wife, Pavithra, and my loving little daughter, Dinadi. Pavithra wanted me to write this book even more than I wanted to. If I say she is the driving force behind this book, I am not exaggerating. She simply went beyond by not only feeding me with encouragement, but also by helping immensely in reviewing the book and developing samples. She was always the first reader. Thank you very much, Pavithra. Also, thanks to little Dinadi for your patience—it was your time that I spent writing the book.

I would also like to thank all the technical reviewers of the book. All your suggestions and thoughts are extremely valuable and much appreciated.

My parents and my sister have been the driving force behind me since my birth. If not for them, I wouldn't be who I am today. I am grateful to them for everything they have done for me. Last but not least, my wife's parents were amazingly helpful in making sure that the only thing I had to do was to write this book, taking care of almost all the other things that I was supposed to do.

Although this sounds like a one-man effort, it's actually a team effort. Thanks to everyone who supported me in different ways.

About the Reviewer

Antonio Mendoza Pérez is a senior software engineer with over 9 years of experience of developing Java EE applications. He also has an increasing interest in Scala, Groovy, and other JVM programming languages. He has contributed to several projects by developing the core element in both frontend and backend, such as customizing its construction and distribution with Maven.

He also has been a reviewer of *JBPM6 Developer Guide*, Packt Publishing.

You can get in touch with him through his blog at <http://antmendoza.com>.

I would like to thank my parents and sisters who have always been there for me.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	v
Chapter 1: Apache Maven Quick Start	1
Installing Apache Maven	1
Installing Apache Maven on Ubuntu	2
Installing Apache Maven on Mac OS X	3
Installing Apache Maven on Microsoft Windows	4
Configuring the heap size	5
Hello Maven!	6
Convention over configuration	7
Maven repositories	9
IDE integration	9
NetBeans integration	9
IntelliJ IDEA integration	10
Eclipse integration	10
Troubleshooting	10
Enabling Maven debug level logs	10
Building a dependency tree	10
Viewing all the environment variables and system properties	11
Viewing the effective POM file	12
Viewing the dependency classpath	13
Summary	13
Chapter 2: Understanding the Project Object Model (POM)	15
Project Object Model (POM)	15
POM hierarchy	17
Super POM	18
POM extending and overriding	23
Maven coordinates	25
The parent POM	27

Managing POM dependencies	29
Transitive dependencies	33
Dependency scopes	35
Optional dependencies	38
Dependency exclusion	39
Summary	42
Chapter 3: Maven Archetypes	43
Archetype quickstart	44
Batch mode	47
Archetype catalogues	47
Building an archetype catalogue	51
Public archetype catalogues	51
The anatomy of archetype – catalog.xml	53
The archetype plugin goals	54
Java EE web applications with the archetype plugin	55
Deploying web applications to a remote Apache Tomcat server	57
Android mobile applications with the archetype plugin	59
EJB archives with the archetype plugin	61
JIRA plugins with the archetype plugin	64
Spring MVC applications with the archetype plugin	65
Summary	66
Chapter 4: Maven Plugins	67
Common Maven plugins	69
The clean plugin	69
The compiler plugin	70
The install plugin	73
The deploy plugin	73
The surefire plugin	75
The site plugin	77
The jar plugin	80
The source plugin	81
The resources plugin	82
The release plugin	83
Plugin discovery and execution	84
Plugin management	87
Plugin repositories	87
Plugin as an extension	89
Summary	89

Chapter 5: Build Lifecycles	91
Standard lifecycles in Maven	92
The clean lifecycle	92
The default lifecycle	95
The site lifecycle	100
Lifecycle bindings	101
Lifecycle extensions	105
Summary	108
Chapter 6: Maven Assemblies	109
The assembly plugin	110
The assembly descriptor	112
Artifact/resource filtering	125
Assembly help	125
A runnable standalone Maven project	126
Summary	131
Chapter 7: Best Practices	133
Dependency management	134
Defining a parent module	136
POM properties	137
Avoiding repetitive groupIds and versions, and inheriting from the parent POM	141
Following naming conventions	141
Think twice before you write your own plugin. You may not need it!	143
The Maven release plugin	144
The Maven enforcer plugin	145
Avoiding the use of unversioned plugins	147
Descriptive parent POM files	149
Documentation is your friend	150
Avoid overriding the default directory structure	151
Using SNAPSHOT versioning during the development	152
Get rid of unused dependencies	152
Avoiding keeping credentials in application POM files	153
Avoiding using deprecated references	154
Avoiding repetition – use archetypes	155
Avoiding using maven.test.skip	155
Summary	157
Index	159

Preface

Maven is the number one build tool used by developers, and it has been there for more than a decade. Maven stands out among other build tools due to its extremely extensible architecture, which is built on top of the concept convention over configuration. This, in fact, has made Maven the de-facto tool to manage and build Java projects. It's being used widely by many open source Java projects under the Apache Software Foundation, SourceForge, Google Code, and many more.

This book provides a step-by-step guide, showing the readers how to use Apache Maven in an optimal way to address enterprise build requirements. Following the book, readers will be able to gain a thorough understanding of the following key areas:

- How to get started with Apache Maven, applying Maven best practices in order to design a build system to improve a developer's productivity
- How to customize the build process to suit it exactly to your enterprise's needs by using appropriate Maven plugins, lifecycles, and archetypes
- How to troubleshoot build issues with greater confidence
- How to design the build in a way that avoids any maintenance nightmares with proper dependency management
- How to optimize Maven configuration settings
- How to build your own distribution archive using Maven assemblies

What this book covers

Chapter 1, Apache Maven Quick Start, focuses on building a basic foundation around Maven to get started. It starts by explaining the basic steps to install and configure Maven on Ubuntu, Mac OS X, and Microsoft Windows operating systems. The latter part of the chapter covers some of the common useful Maven tips and tricks.

Chapter 2, Understanding the Project Object Model (POM), focuses on the Maven Project Object Model (POM) and how to adhere to the industry-wide accepted best practices to avoid maintenance nightmares. The key elements of a POM file, POM hierarchy and inheritance, managing dependencies, and related topics are covered here.

Chapter 3, Maven Archetypes, focuses on how Maven archetypes provide a way of reducing repetitive work in building Maven projects. There are thousands of archetypes out there that are available publicly to help you build different types of projects. This chapter covers a commonly used set of archetypes.

Chapter 4, Maven Plugins, covers some of the most commonly used Maven plugins and then explains how plugins are discovered and executed. Maven only provides a build framework, while the Maven plugins perform the actual tasks. Maven has a large, rich set of plugins, and the chances are very small that you will have to write your own custom plugin.

Chapter 5, Build Lifecycles, explains how the three standard lifecycles work and how we can customize them. Later in the chapter, we discuss how to develop our own lifecycle extensions.

Chapter 6, Maven Assemblies, covers real-world examples of how to use the Maven assembly plugin in detail, and finally concludes with an end-to-end sample Maven project.

Chapter 7, Best Practices, looks at and highlights some of the best practices to be followed in a large-scale development project with Maven. It is always recommended to follow the best practices since it will drastically improve developer productivity and reduce any maintenance nightmares.

What you need for this book

To follow the examples that are presented in this book, you will need the following software:

- Apache Maven 3.3.x, which you can find at <http://maven.apache.org/download.cgi>
- Java 1.7+ SDK, which you can find at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- Operating Systems: Windows, Linux, or Mac OS X.

Who this book is for

The book is ideal for experienced developers who are already familiar with build automation, but want to learn how to use Maven and apply its concepts to the most difficult scenarios in build automation.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"When you type `mvn clean install`, Maven will execute all the phases in the default lifecycle up to `install` (including the `install` phase)."

A block of code is set as follows:

```
<project>
[... ]
<build>
[... ]
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```



```
    </plugin>
  </plugins>
  [...]
</build>
[...]
</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:


```
<project>
  [...]
  <build>
    [...]
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
    [...]
  </build>
  [...]
</project>
```

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

```
$ mvn install:install
```

New terms and **important words** are shown in bold.

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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 could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

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

Piracy

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

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

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

Questions

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

1

Apache Maven Quick Start


Apache Maven is popular as a build tool. However, in reality, it goes beyond being just a build tool. It provides a comprehensive build management platform. Prior to Maven, developers had to spend a lot of time in building a build system. There was no common interface. It differed from project to project, and each time a developer moved from one project to another, there was a learning curve. Maven filled this gap by introducing a common interface. It merely ended the era of the build engineer.

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

- Installing and configuring Maven on Ubuntu, Mac OS X, and Microsoft Windows
- IDE integration
- Tips and tricks for using Maven effectively

Installing Apache Maven

Installing Maven on any platform is more than a straightforward task. At the time of writing this book, the latest version was 3.3.3, which is available for download at <http://maven.apache.org/download.cgi>. This version requires JDK 1.7.0 or above.

 You should keep a note on the Java requirement for version 3.3.3, if you are planning to upgrade from versions 3.0.*, 3.1.*, or 3.2.*. Prior to Maven 3.3.x, the only requirement was JDK 1.5.0. or JDK 1.6.0 (for 3.2.*).

Apache Maven is an extremely lightweight distribution. It does not have any hard requirements on memory, disk space, or CPU. Maven itself is built on top of Java, and it would work on any operating system that runs **Java virtual machine (JVM)**.

Installing Apache Maven on Ubuntu

Installing Maven on Ubuntu is a single line command. Proceed with the following steps:

1. Run the following `apt-get` command in the command prompt. You need to have the `sudo` privileges to execute this:

```
$ sudo apt-get install maven
```
2. This takes a few minutes to complete. Upon the completion of the installation, you can run the following command to verify the installation:

```
$ mvn -version
```
3. You should get an output similar to the following, if Apache Maven has been installed successfully:

```
$ mvn -version
Apache Maven 3.3.3
Maven home: /usr/share/maven
Java version: 1.7.0_60, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-7-oracle/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.13.0-24-generic", arch: "amd64",
family: "unix"
```
4. Maven is installed under the `/usr/share/maven` directory. To check the directory structure behind the Maven installation directory, use the following command:

```
$ ls /usr/share/maven
bin  boot  conf  lib  man
```
5. Maven configuration files can be found at `/etc/maven`, which can be listed using the following command:

```
$ ls /etc/maven
m2.conf  settings.xml
```

If you don't want to work with the `apt-get` command, there is another way of installing Maven under any Unix-based operating system. We will discuss this in the next section. Since Mac OS X has a kernel built at the top of the Unix kernel, installing Maven on Mac OS X would be the same as installing it on any Unix-based operating system.

Installing Apache Maven on Mac OS X

Most of the OS X distributions prior to OS X Mavericks had Apache Maven preinstalled. To verify that you've got Maven installed in your system, try out the following command. If it does not result in a version, then it means you do not have it installed:

```
$ mvn -version
```

The following steps will guide you through the Maven installation process on Max OS X Yosemite:

1. First, we need to download the latest version of Maven. Throughout this book, we will use Maven 3.3.3, which was the latest version at the time of writing this book. Maven 3.3.3 ZIP distribution can be downloaded from <http://maven.apache.org/download.cgi>.
2. Unzip the downloaded ZIP file into `/usr/share/java`. You need to have the `sudo` privileges to execute this:

```
$ sudo unzip apache-maven-3.3.3-bin.zip -d /usr/share/java/
```

3. In case you already have Maven installed in your system, use the following command to unlink. `/usr/share/maven` is only a symlink to the directory where Maven is installed:

```
$ sudo unlink /usr/share/maven
```

4. Use the following command to create a symlink to the latest Maven distribution that you just unzipped. You need to have the `sudo` privileges to execute this:

```
$ sudo ln -s /usr/share/java/apache-maven-3.3.3 /usr/share/maven
```

5. Use the following command to update the value of the `PATH` environment variable:

```
$ export PATH=$PATH:/usr/share/maven/bin
```

6. Use the following command to update (or set) the value of the `M2_HOME` environment variable:

```
$ export M2_HOME=/usr/share/maven
```

7. Verify the Maven installation with the following command:

```
$ mvn -version
```

```
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06;  
2015-04-22T04:57:37-07:00)
```

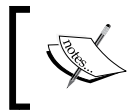
```
Maven home: /usr/share/maven
```

```
Java version: 1.7.0_75, vendor: Oracle Corporation
```

```
Java home:    /Library/Java/JavaVirtualMachines/jdk1.7.0_75.jdk/
Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.10.2", arch: "x86_64",
family: "mac"
```

8. If you get the following error while running the preceding command, it means you have another version of Maven running in your system, and the `PATH` system variable includes the path to its `bin` directory. If that is the case, you need to clean out the value of the `PATH` system variable by removing the path to the old Maven installation:

```
-Dmaven.multiModuleProjectDirectory system property is not
set. Check $M2_HOME environment variable and mvn script match.
```



Maven can also be installed on Mac OS X with Homebrew.
This video explains the installation process in detail –
<https://www.youtube.com/watch?v=xTzLGcqUf8k>

Installing Apache Maven on Microsoft Windows

First, we need to download the latest version of Maven. Apache Maven 3.3.3 ZIP distribution can be downloaded from <http://maven.apache.org/download.cgi>. Then, we need to perform the following steps:

1. Unzip the downloaded ZIP file into the `C:\Program Files\ASF` directory.
2. Set the `M2_HOME` environment variable and point it to `C:\Program Files\ASF\apache-maven-3.3.3`.
3. Verify the Maven installation with the following command on the command prompt:

```
mvn -version
```



To know more about how to set the environment variables on Microsoft Windows, refer to <http://www.computerhope.com/issues/ch000549.htm>.

Configuring the heap size

Once you have Maven installed in your system, the very next step is to fine-tune it for an optimal performance. By default, the maximum heap allocation is 512 MB, which starts from 256 MB (`-Xms256m` to `-Xmx512m`). This default limit is not good enough to build a large, complex Java project, and it is recommended that you have at least 1024 MB of the maximum heap.

If you encounter `java.lang.OutOfMemoryError` at any point during a Maven build, then it is mostly due to a lack of memory. You can use the `MAVEN_OPTS` environment variable to set the maximum allowed heap size for Maven at a global level. The following command will set the heap size in any Unix-based operating system, including Linux and Mac OS X. Make sure that the value set as the maximum heap size does not exceed your system memory of the machine, which runs Maven:

```
$ export MAVEN_OPTS="-Xmx1024m -XX:MaxPermSize=128m"
```

If you are on Microsoft Windows, use the following command:

```
$ set MAVEN_OPTS=-Xmx1024m -XX:MaxPermSize=128m
```

Here, `-Xmx` takes the maximum heap size and `-XX:MaxPermSize` takes the maximum **Permanent Generation (PermGen)** size.



Maven runs as a Java process on JVM. As it proceeds with a build, it keeps on creating Java objects. These objects are stored in the memory allocated to Maven. This area of memory where Java objects are stored is known as heap. Heap is created at the JVM start and it increases as more and more objects are created up to the defined maximum limit. The `-Xms` JVM flag is used to instruct JVM about the minimum value that it should set at the time of creating the heap. The `-Xmx` JVM flag sets the maximum heap size.

PermGen is an area of memory managed by JVM, which stores the internal representations of Java classes. The maximum size of PermGen can be set by the `-XX:MaxPermSize` JVM flag.

When the Java virtual machine cannot allocate enough memory to Maven, it could result in an `OutOfMemoryError`. To know more about the Maven `OutOfMemoryError`, refer to <https://cwiki.apache.org/confluence/display/MAVEN/OutOfMemoryError>.

Hello Maven!

The easiest way to get started with a Maven project is to use the `generate` goal of the `archetype` plugin to generate a simple Maven project. Maven archetypes are discussed in detail in *Chapter 3, Maven Archetypes*, and plugins are covered in *Chapter 4, Maven Plugins*.

Let's start with a simple example:

```
$ mvn archetype:generate
    -DgroupId=com.packt.samples
    -DartifactId=com.packt.samples.archetype
    -Dversion=1.0.0
    -DinteractiveMode=false
```

This command will invoke the `generate` goal of the Maven `archetype` plugin to create a simple Java project. You will see that the following project structure is created with a sample POM file. The name of the root or the base directory is derived from the value of the `artifactId` parameter:

```
com.packt.samples.archetype
| -pom.xml
| -src
| -main/java/com/packt/samples/App.java
| -test/java/com/packt/samples/AppTest.java
```

The sample POM file will only have a dependency to the `junit` JAR file with `test` as the scope:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.samples</groupId>
  <artifactId>com.packt.samples.archetype</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>com.packt.samples.archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The generated `App.java` class will have the following template code. The name of the package is derived from the provided `groupId` parameter. If you want to have a different value as the package name, then you need to pass this value in the command itself as `-Dpackage=com.packt.samples.application`:

```
package com.packt.samples;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

To build the sample project, run the following command from the `com.packt.samples.archetype` directory, where the `pom.xml` file exists:

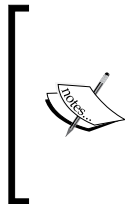
```
$ mvn clean install
```

Convention over configuration

Convention over configuration is one of the main design philosophies behind Apache Maven. Let's go through a few examples.

A complete Maven project can be created using the following configuration file (`pom.xml`):

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>sample-one</artifactId>
  <version>1.0.0</version>
</project>
```



The Maven POM file starts with the `<project>` element. Always define the `<project>` element with the schema. Some tools can't validate the file without it:

```
<project xmlns=http://maven.apache.org/POM/4.0.0
  xmlns:xsi=.....
  xsi:schemaLocation="...">
```

The `pom.xml` file is the heart of any Maven project and is discussed in detail in *Chapter 2, Understanding the Project Object Model (POM)*. Copy the previous configuration element and create a `pom.xml` file out of it. Then, place it in a directory called `chapter-01`, and then create the following child directories under it:

- `chapter-01/src/main/java`
- `chapter-01/src/test/java`

Now, you can place your Java code under `chapter-01/src/main/java` and test cases under `chapter-01/src/test/java`. Use the following command to run the Maven build from where the `pom.xml` is:

```
$ mvn clean install
```

This little configuration that you found in the sample `pom.xml` file is tied up with many conventions:

- Java source code is available at `{base-dir}/src/main/java`
- Test cases are available at `{base-dir}/src/test/java`
- The type of the artifact produced is a JAR file
- Compiled class files are copied to `{base-dir}/target/classes`
- The final artifact is copied to `{base-dir}/target`
- `http://repo.maven.apache.org/maven2`, is used as the repository URL.

If someone needs to override the default, conventional behavior of Maven, then it is possible too. The following sample `pom.xml` file shows how to override some of the preceding default values:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>sample-one</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <build>
    <sourceDirectory>${basedir}/src/main/java</sourceDirectory>
    <testSourceDirectory>${basedir}/src/test/java
                                </testSourceDirectory>
    <outputDirectory>${basedir}/target/classes
                                </outputDirectory>
  </build>
</project>
```

Maven repositories

The magic behind how Maven finds and loads dependent jars for a given Maven project is Maven repositories. In the corresponding `pom.xml` file of your Maven project, under the `<dependencies>` element, you can define references to all the dependent jar files required to build your project successfully. Each dependency defined in the `pom.xml` file is identified uniquely using Maven coordinates. Maven coordinates uniquely identify a project, a dependency, or a plugin defined in a POM. Each entity is uniquely identified by the combination of a group identifier, an artifact identifier, and version (and, of course, with the packaging and the classifier). Maven coordinates are discussed in detail in *Chapter 2, Understanding the Project Object Model (POM)*. Once Maven finds out all the required dependencies for a given project, it loads them to the local file system of Maven repositories, and adds them to the project classpath.

By convention, Maven uses `http://repo.maven.apache.org/maven2` as the repository. If all the artifacts required to build the project are present in this repository, then those will be loaded into the local file system or the local Maven repository, which is, by default, at `USER_HOME/.m2/repository`. You can add custom repositories at the project level under the `<repositories>` element of the `pom.xml` file or at the global level under the `MAVEN_HOME/conf/settings.xml` file.

IDE integration

Most of the hardcore developers never want to leave their IDE. Not just for coding, but for building, deploying, testing, and for everything if possible - they would happily do these from the IDE itself. Most of the popular IDEs support Maven integration, and they have developed their own plugins to support Maven.

NetBeans integration

NetBeans 6.7 or newer ships with in-built Maven integration, while NetBeans 7.0 has newer versions that bundle a complete copy of Maven 3 and runs it for builds just like you would from the command line. For version 6.9 or older, you have to download a Maven build and configure the IDE to run this. More information corresponding to Maven and NetBeans integration is available at <http://wiki.netbeans.org/MavenBestPractices>.

IntelliJ IDEA integration

IntelliJ IDEA has inbuilt support for Maven; hence, you don't need to perform any additional steps to install it. More information corresponding to Maven and IntelliJ IDEA integration is available at http://wiki.jetbrains.net/intellij/Creating_and_importing_Maven_projects.

Eclipse integration

The M2Eclipse project provides first class Maven support through the Eclipse IDE. More information corresponding to Maven and the Eclipse integration is available at <https://www.eclipse.org/m2e/>.



The book *Maven for Eclipse*, published by Packt Publishing, discusses in detail Maven and Eclipse integration at <https://www.packtpub.com/application-development/maven-eclipse>.

Troubleshooting

If everything works fine, we should never have to worry about troubleshooting. However, most of the time this is not the case. A Maven build can fail for many reasons, some of which are under your control and also out of your control. Knowing proper troubleshooting tips helps you to pinpoint the exact problem. The following section lists some of the most used troubleshooting tips. We will expand the list as we proceed in this book.

Enabling Maven debug level logs

Once the Maven debug level logging is enabled, it will print all the actions that it takes during the build process. To enable debug level logging, use the following command:

```
$ mvn clean install -X
```

Building a dependency tree

If you find any issue with any dependency in your Maven project, the first step is to build a dependency tree. This shows where each dependency comes from. To build the dependency tree, run the following command against your project POM file:

```
$ mvn dependency:tree
```

The following shows the truncated output of the previous command executed against the Apache Rampart project:

```
[INFO] -----
[INFO] Building Rampart - Trust 1.6.1-wso2v12
[INFO] -----
[INFO]
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ rampart-
trust ---
[INFO] org.apache.rampart:rampart-trust:jar:1.6.1-wso2v12
[INFO] +- org.apache.rampart:rampart-policy:jar:1.6.1-wso2v12:compile
[INFO] +- org.apache.axis2:axis2-kernel:jar:1.6.1-wso2v10:compile
[INFO] | +- org.apache.ws.commons.axiom:axiom-api:jar:1.2.11-
wso2v4:compile (version managed from 1.2.11)
[INFO] | | \- jaxen:jaxen:jar:1.1.1:compile
[INFO] | +- org.apache.ws.commons.axiom:axiom-impl:jar:1.2.11-
wso2v4:compile (version managed from 1.2.11)
[INFO] | +- org.apache.geronimo.specs:geronimo-ws-
metadata_2.0_spec:jar:1.1.2:compile
[INFO] | +- org.apache.geronimo.specs:geronimo-
jta_1.1_spec:jar:1.1:compile
[INFO] | +- javax.servlet:servlet-api:jar:2.3:compile
[INFO] | +- commons-httpclient:commons-httpclient:jar:3.1:compile
[INFO] | | \- commons-codec:commons-codec:jar:1.2:compile
[INFO] | +- commons-fileupload:commons-fileupload:jar:1.2:compile
```

Viewing all the environment variables and system properties

If you have multiple JDKs installed in your system, you may wonder what is being used by Maven. The following command will display all the environment variables and system properties set for a given Maven project:

```
$ mvn help:system
```

The following is the truncated output of the previous command:

```
=====Platform Properties Details=====

=====
System Properties
```

```
=====

java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path= /Library/Java/JavaVirtualMachines/jdk1.7.0_75.jdk/
Contents/Home/jre/lib
java.vm.version= 24.75-b04
awt.nativeDoubleBuffering=true
gopherProxySet=false
mrj.build=11M4609
java.vm.vendor=Apple Inc.
java.vendor.url=http://www.apple.com/
guice.disable.misplaced.annotation.check=true
path.separator=:
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
file.encoding.pkg=sun.io
sun.java.launcher=SUN_STANDARD
user.country=US
sun.os.patch.level=unknown

=====
Environment Variables
=====

JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/
CurrentJDK/Home
HOME=/Users/prabath
TERM_SESSION_ID=C2CEFB58-4705-4C67-BE1F-9E4179F96391
M2_HOME=/usr/share/maven/maven-3.3.3/
COMMAND_MODE=unix2003
Apple_PubSub_Socket_Render=/tmp/launch-w7NZbG/Render
LOGNAME=prabath
USER=prabath
```

Viewing the effective POM file

Maven uses default values for configuration parameters when they are not overridden in the configuration. This is exactly what we discussed under the *Convention over configuration* section. If we take the same sample POM file that we used before in this chapter, we can see how the effective POM file would look using the following command. This is also the best way to see what default values are being used by Maven:

```
$ mvn help:effective-pom
```



More details about the `effective-pom` command are discussed in *Chapter 2, Understanding the Project Object Model (POM)*.

Viewing the dependency classpath

The following command will list all the JAR files and directories in the build classpath:

```
$ mvn dependency:build-classpath
```

The following shows the truncated output of the previous command executed against the Apache Rampart project:

```
[INFO] -----
[INFO] Building Rampart - Trust 1.6.1-wso2v12
[INFO] -----
[INFO]
[INFO] --- maven-dependency-plugin:2.1:build-classpath (default-cli) @
rampart-trust ---
[INFO] Dependencies classpath:
/Users/prabath/.m2/repository/bouncycastle/bcprov-jdk14/140/bcprov-
jdk14-140.jar:/Users/prabath/.m2/repository/commons-cli/commons-cli/1.0/
commons-cli-1.0.jar:/Users/prabath/.m2/repository/commons-codec/commons-
codec/1.2/commons-codec-1.2.jar:/Users/prabath/.m2/repository/commons-
collections/commons-collections/3.1/commons-collections-3.1.jar
```

Summary

This chapter focused on building a basic foundation around Maven to bring all the readers into a common ground. It started with explaining the basic steps to install and configure Maven under Ubuntu, Mac OS X, and Microsoft Windows operating systems. The latter part of the chapter covered some of the common useful Maven tips and tricks. As we proceed with the book, some of the concepts touched in this chapter will be discussed in detail.

In the next chapter, we will discuss **Maven Project Object Model (POM)** in detail.

2

Understanding the Project Object Model (POM)

POM is at the heart of any Maven project. This chapter focuses on the core concepts and best practices related to POM in building a large-scale, multi-module Maven project.

As we proceed with this chapter, the following topics will be covered in detail:

- The POM hierarchy, super POM, and parent POM
- Extending and overriding POM files
- Maven coordinates
- Managing dependencies
- Transitive dependencies
- Dependency scopes and optional dependencies

Project Object Model (POM)

Any Maven project must have a `pom.xml` file. POM is the Maven project descriptor just like the `web.xml` file in your Java EE web application, or the `build.xml` file in your Ant project. The following code lists out all the key elements in a Maven `pom.xml` file. As we proceed with the book, we will discuss how to use each element in the most effective manner:

```
<project>

  <parent>...</parent>

  <modelVersion>4.0.0</modelVersion>
  <groupId>...</groupId>
```

```
<artifactId>...</artifactId>
<version>...</version>
<packaging>...</packaging>

<name>...</name>
<description>...</description>
<url>...</url>
<inceptionYear>...</inceptionYear>
<licenses>...</licenses>
<organization>...</organization>
<developers>...</developers>
<contributors>...</contributors>

<dependencies>...</dependencies>
<dependencyManagement>...</dependencyManagement>
<modules>...</modules>
<properties>...</properties>

<build>...</build>
<reporting>...</reporting>

<issueManagement>...</issueManagement>
<ciManagement>...</ciManagement>
<mailingLists>...</mailingLists>
<scm>...</scm>
<prerequisites>...</prerequisites>

<repositories>...</repositories>
<pluginRepositories>...</pluginRepositories>

<distributionManagement>...</distributionManagement>

<profiles>...</profiles>
</project>
```

The following code shows a sample `pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>jose</artifactId>
  <version>1.0.0</version>

  <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>

<dependencies>
  <dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>nimbus-jose-jwt</artifactId>
    <version>2.26</version>
  </dependency>
</dependencies>
</project>

```

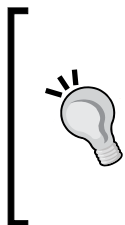
POM hierarchy

POM files maintain a parent-child relationship between them. A child POM file inherits all the configuration elements from its parent POM. Using this trick, Maven sticks to its design philosophy *convention over configuration*. The minimal POM configuration for any Maven project is extremely simple, which is as follows:

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>sample-one</artifactId>
  <version>1.0.0</version>
</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.

Super POM

Any POM file can point to its parent POM. In case the parent POM element is missing, there is a system wide POM file that is automatically treated as the parent POM. This POM file is well known as the **super POM**. Ultimately, all the application POM files get extended from the super POM. The super POM file is at the top of the POM hierarchy and is bundled inside `MAVEN_HOME/lib/maven-model-builder-3.3.3.jar - org/apache/maven/model/pom-4.0.0.xml`. In Maven 2, this was bundled inside `maven-2.X.X-uber.jar`. All the default configurations are defined in the super POM file. Even the simplest form of a POM file will inherit all the configurations defined in the super POM file. Whatever configuration you need to override, you can do it by redefining the same section in your application POM file. The following lines of code show the super POM file configuration, which comes with Maven 3.3.3:

```
<project>
  <modelVersion>4.0.0</modelVersion>
```

The Maven central is the only repository defined under the *repositories* section. It will be inherited by all the Maven application modules. Maven uses these repositories defined under the *repositories* section to download all the dependent artifacts during a Maven build. The following code snippet shows the configuration block in `pom.xml`, which is used to define repositories:

```
<repositories>
  <repository>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```



There are two types of repositories in Maven: local and remote. The local repository is maintained in your local machine – by default at `USER_HOME/.m2/repository`. Anything that you build locally with `mvn install` will get deployed into the local repository. When you start with a fresh Maven repository, it will be empty. You need to download everything – from the simplest `maven-compiler-plugin` to all your project dependencies. A Maven build can either be an online or offline build. By default, it is an online build, unless you add `-o` into your Maven build command. If it's an offline build, Maven assumes that all the related artifacts are readily available in the local Maven repository; if not, it will complain. If it is an online build, Maven will download the artifacts from remote repositories and store them in the local repository. The Maven local repository location can be changed to a preferred location by editing `MAVEN_HOME/conf/settings.xml` to update the value of the `localRepository` element:

```
<localRepository>/path/to/local/repo</localRepository>
```

Plugin repositories define where to find Maven plugins. We'll be talking about Maven plugins in *Chapter 4, Maven Plugins*. The following code snippet shows the configuration related to the plugin repositories:

```
<pluginRepositories>
  <pluginRepository>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
    <releases>
      <updatePolicy>never</updatePolicy>
    </releases>
  </pluginRepository>
</pluginRepositories>
```

The `build` configuration section includes all the information required to build a project:

```
<build>
  <directory>${project.basedir}/target</directory>
  <outputDirectory>${project.build.directory}/classes</outputDirectory>
```

```
<finalName>${project.artifactId}-${project.version}
</finalName>
<testOutputDirectory>${project.build.directory}/test-classes
</testOutputDirectory>
<sourceDirectory>${project.basedir}/src/main/java
</sourceDirectory>
<scriptSourceDirectory>${project.basedir}/src/main/scripts
</scriptSourceDirectory>
<testSourceDirectory>${project.basedir}/src/test/java
</testSourceDirectory>

<resources>
  <resource>
    <directory>${project.basedir}/src/main/resources
    </directory>
  </resource>
</resources>
<testResources>
  <testResource>
    <directory>${project.basedir}/src/test/resources
    </directory>
  </testResource>
</testResources>

<pluginManagement>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.3</version>
    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2-beta-5</version>
    </plugin>
    <plugin>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.8</version>
    </plugin>
    <plugin>
      <artifactId>maven-release-plugin</artifactId>
      <version>2.3.2</version>
    </plugin>
  </plugins>
</pluginManagement>
</build>
```

The reporting section includes the details of report plugins, which are used to generate reports and are later displayed on the site generated by Maven. The super POM only provides a default value for the output directory:

```
<reporting>
  <outputDirectory>${project.build.directory}/site
</outputDirectory>
</reporting>
```

The following code snippet defines the default build profile. When no profiles are defined at the application level, the default build profile will get executed. We will be talking about profiles in *Chapter 7, Best Practices*:

```
<profiles>
  <profile>
    <id>release-profile</id>

    <activation>
      <property>
        <name>performRelease</name>
        <value>true</value>
      </property>
    </activation>

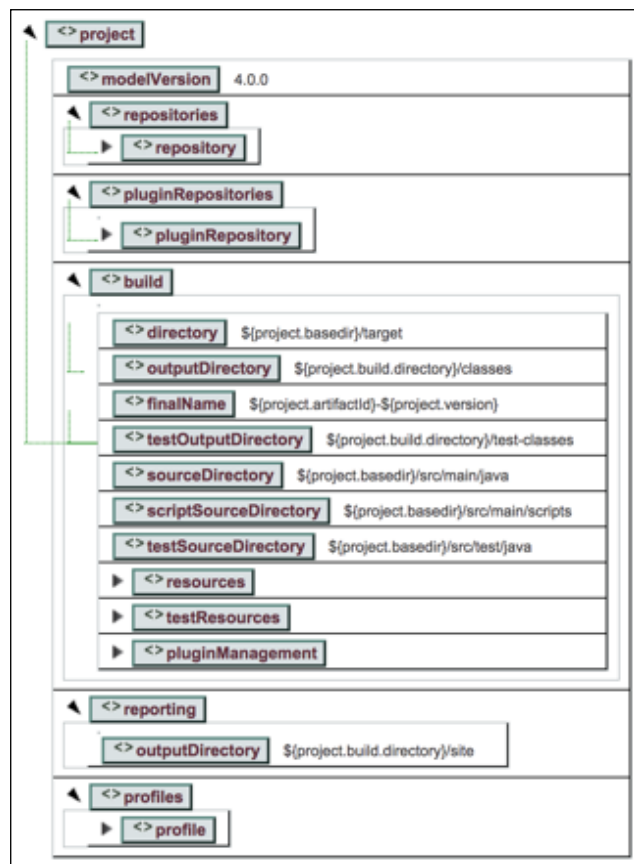
    <build>
      <plugins>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-source-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-sources</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-javadoc-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-javadocs</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```



```
        </execution>
      </executions>
    </plugin>
    <plugin>
      <inherited>true</inherited>
      <artifactId>maven-deploy-plugin</artifactId>
      <configuration>
        <updateReleaseInfo>true</updateReleaseInfo>
      </configuration>
    </plugin>
  </plugins>
</build>
</profile>
</profiles>

</project>
```

The following figure shows an abstract view of the super POM file with key configuration elements:



POM extending and overriding

Let's see how POM overriding works. In the following example, we extend the `repositories` section to add one more repository than what is defined in the Maven super POM:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>sample-one</artifactId>
  <version>1.0.0</version>

  <repositories>
    <repository>
      <id>wso2-nexus</id>
      <name>WSO2 internal Repository</name>
      <url>http://maven.wso2.org/nexus/content/
                                groups/wso2-public/
      </url>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>daily</updatePolicy>
        <checksumPolicy>ignore</checksumPolicy>
      </releases>
    </repository>
  </repositories>

</project>
```

Execute the following command from the directory where the above POM file is located:

```
$ mvn help:effective-pom
```

This will display the effective POM for the application, which combines all the default settings from the super POM file and the configuration defined in your application POM. In the following code snippet, you can see that the `<repositories>` section in the super POM file is being extended by your application-specific configuration. Now, the `<repositories>` section has the central repository defined in the super POM as well as your application-specific repository:

```
<repositories>
  <repository>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>daily</updatePolicy>
```

```
        <checksumPolicy>ignore</checksumPolicy>
    </releases>
    <id>wso2-nexus</id>
    <name>WSO2 internal Repository</name>
    <url>
        http://maven.wso2.org/nexus/content/groups/wso2-public/
    </url>
</repository>
<repository>
    <snapshots>
        <enabled>>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Central Repository</name>
    <url>https://repo.maven.apache.org/maven2</url>
</repository>
</repositories>
```

If you want to override any of the configuration elements corresponding to the Maven central repository inherited from the super POM file, then you have to define a repository in your application POM with the same repository id (as of the Maven central repository), and override the configuration element that you need.

One main advantage of the POM hierarchy in Maven is that you can extend as well as override the configuration inherited from the top. Say, for example, that you may need to keep all the plugins defined in the super POM, but just want to override the version of `maven-release-plugin`. The following configuration shows how to do it. By default, in the super POM, the `maven-release-plugin` version is 2.3.2, and here, we update it to 2.5 in our application POM. If you run `mvn help:effective-pom` again against the updated POM file, you will notice that the plugin version is updated, whereas the rest of the plugin configuration from the super POM remains unchanged:

```
<project>

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt</groupId>
    <artifactId>sample-one</artifactId>
    <version>1.0.0</version>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <artifactId>maven-release-plugin</artifactId>
```

```

        <version>2.5</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

</project>

```

To override the configuration of a given element or an artifact in the POM hierarchy, Maven should be able to uniquely identify the corresponding artifact. In the preceding scenario, the plugin was identified by its `artifactId`. In *Chapter 4, Maven Plugins* we will further discuss how Maven locates plugins.

Maven coordinates

Maven coordinates uniquely identify a project, dependency, or plugin defined in a POM. Each entity is uniquely identified by the combination of a group identifier, artifact identifier, and the version (and, of course, with the packaging and the classifier). The group identifier is a way of grouping different Maven artifacts. For example, a set of artifacts produced by a company can be grouped under the same group identifier. The artifact identifier is the way you identify an artifact, which could be a JAR, WAR, or any type of an artifact uniquely identified within a given group. The `version` element lets you keep the same artifact in different versions in the same repository.



A valid Maven POM file must have `groupId`, `artifactId`, and `version`. The `groupId` and `version` elements can also be inherited from the parent POM.

All the three coordinates of a given Maven artifact are used to define its path in the Maven repository. If we take the following example, the corresponding JAR file is installed into the local repository with the path `M2_REPO/repository/com/packt/sample-one/1.0.0/`:

```

<groupId>com.packt</groupId>
<artifactId>sample-one</artifactId>
<version>1.0.0</version>

```

If you have gone through the elements of the super POM file carefully, you might have noticed that it does not have any of the previously mentioned elements—no `groupId`, `artifactId`, or `version`. Does this mean that the super POM file is not a valid POM? The super POM file is similar to an abstract class in Java. It does not work by itself; it must be inherited by a child POM. Another way to look at the super POM file is that it's the Maven's way of sharing default configurations.

Once again, if you look at the `<pluginManagement>` section of the super POM, as shown in the following code snippet, you will notice that a given plugin artifact is only identified by its `artifactId` and `version` elements. This contradicts what was mentioned before: a given artifact is uniquely identified by the combination of `groupId`, `artifactId`, and `version`. How is this possible?

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.3</version>
</plugin>
```

There is an exception for plugins. You need not specify `groupId` for a plugin in the POM file—it is optional. By default, Maven uses `org.apache.maven.plugins` or `org.codehaus.mojo` as `groupId`. Have a look at the following section in `MAVEN_HOME/conf/settings.xml`. Everything that you define in this file will be globally applicable for all the Maven builds, which run in the corresponding machine. In case you want to keep the configuration at user level (in a multi-user environment), you can simply copy the `settings.xml` file from `MAVEN_HOME/conf` to `USER_HOME/.m2`. If you want to add the additional `groupId` elements for plugin lookup, you will have to uncomment the following section and add them there:

```
<!-- pluginGroups
| This is a list of additional group identifiers that
| will be searched when resolving plugins by their prefix, i.e.
| when invoking a command line like "mvn prefix:goal".
| Maven will automatically add the group identifiers
| "org.apache.maven.plugins" and "org.codehaus.mojo"
| if these are not already contained in the list.
|-->
<pluginGroups>
  <!-- pluginGroup
  | Specifies a further group identifier to use for plugin
  | lookup.
  <pluginGroup>com.your.plugins</pluginGroup>
  -->
</pluginGroups>
```



We will be discussing Maven plugins in detail in
Chapter 4, Maven Plugins.



The parent POM

When we deal with hundreds of Maven modules, we need to structure the project to avoid any redundancies or duplicate configurations. If not, it will lead to a huge maintenance nightmare. Let's have a look at some popular open source projects.

The WSO2 Carbon Turing project, available at <https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/>, has more than 1000 Maven modules. Anyone who downloads the source code from the root should be able to build it with all the components. The `pom.xml` file at the root acts as a module aggregating POM. It defines all the Maven modules that need to be built under the `<modules>` element. Each module element defines the relative path (from the root POM) to the corresponding Maven module. There needs to be another POM file under the defined relative path. The root POM in the WSO2 Carbon Turing project only acts as an aggregator module. It does not build any parent-child relationship with other Maven modules. The following code snippet shows the module configuration in the root `pom.xml`:

```
<modules>
  <module>parent</module>
  <module>dependencies</module>
  <module>service-stubs</module>
  <module>components</module>
  <module>platform-integration/clarity-framework</module>
  <module>features</module>
  <module>samples/shopping-cart</module>
  <module>samples/shopping-cart-global</module>
</modules>
```

Now, let's have a look at the POM file inside the `parent` module. This POM file defines plugin repositories, a distribution repository, plugins, and a set of properties. This does not have any dependencies, and this is the POM file that acts as the parent for all the other Maven submodules. The parent POM file has the following coordinates:

```
<groupId>org.wso2.carbon</groupId>
<artifactId>platform-parent</artifactId>
<version>4.2.0</version>
<packaging>pom</packaging>
```

If you look at the POM file inside the `components` module, it refers to `parent/pom.xml` as the parent Maven module. The value of the `relativePath` element, by default, refers to the `pom.xml` file a level above, that is, `../pom.xml`. However, in this case, it is not the parent POM; hence, the value of the element must be overridden and set to `../parent/pom.xml`, shown as follows:

```
<groupId>org.wso2.carbon</groupId>
<artifactId>carbon-components</artifactId>
<version>4.2.0</version>
<parent>
  <groupId>org.wso2.carbon</groupId>
  <artifactId>platform-parent</artifactId>
  <version>4.2.0</version>
  <relativePath>../parent/pom.xml</relativePath>
</parent>
```

If you go inside the `components` module and run `mvn help:effective-pom`, you will notice that an effective POM aggregates both the configurations defined in `parent/pom.xml` and `components/pom.xml`. Parent POM files help to propagate common configuration elements to downstream Maven modules, and it can go up to many levels. The `components/pom.xml` file acts as the parent POM for Maven modules below its level. For example, let's have a look at the following `components/identity/pom.xml` file. It has a reference to the `components/pom.xml` file as its parent. Note that here we do not need to use the `relativePath` element, since the corresponding parent POM is at the default location:

```
<groupId>org.wso2.carbon</groupId>
<artifactId>identity</artifactId>
<version>4.2.0</version>
<parent>
  <groupId>org.wso2.carbon</groupId>
  <artifactId>carbon-components</artifactId>
  <version>4.2.0</version>
</parent>
```

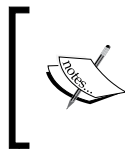


A complete list of elements in a POM file is explained in detail at <http://maven.apache.org/ref/3.3.3/maven-model/maven.html>.

Managing POM dependencies

In a large-scale development project with hundreds of Maven modules, managing dependencies could be a hazardous task. There are two effective ways to manage dependencies: POM inheritance and dependency grouping. With POM inheritance, the parent POM has to define all the common dependencies used by its child modules under the `dependencyManagement` section. In this way, we can avoid all the duplicate dependencies. Also, if we have to update the version of a given dependency, then we only have to make changes in one place. Let's take the same example we discussed before using the WSO2 Carbon Turing project. Let's have a look at the `dependencyManagement` section of `parent/pom.xml` (only a part of the POM file is shown here):

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.axis2</groupId>
      <artifactId>axis2-transport-mail</artifactId>
      <version>${axis2-transport.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.ws.commons.axiom.wso2</groupId>
      <artifactId>axiom</artifactId>
      <version>${axiom.wso2.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```



To know more about dependency management, refer to *Introduction to the Dependency Mechanism*, available at <http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>.

Let's have a look at the dependency section of `identity/org.wso2.carbon.identity.core/4.2.3/pom.xml`, which extends from `components/pom.xml`. Here, you will only see `groupId` and `artifactId` of a given dependency, and not `version`. The version of each dependency is managed through the `dependencyManagement` section of the parent POM. In case any child Maven module wants to override the version of an inherited dependency, it can simply add the `version` element:

```
<dependencies>
  <dependency>
    <groupId>org.apache.axis2.wso2</groupId>
    <artifactId>axis2</artifactId>
```



```
</dependency>
<dependency>
  <groupId>org.apache.ws.commons.axiom.wso2</groupId>
  <artifactId>axiom</artifactId>
</dependency>
</dependencies>
```

Another best practice to highlight here is the way dependency versions are specified in the parent POM file, which is as follows:

```
<version>${axiom.wso2.version}</version>
```

Instead of specifying the version number inside the dependency element itself, here, we have taken it out and represented the version as a property. The value of the property is defined under the `properties` section of the parent POM, as shown in the following line of code. This makes POM maintenance extremely easy:

```
<properties>
  <axis2.wso2.version>1.6.1.wso2v10</axis2.wso2.version>
</properties>
```

The second approach to manage dependencies is through dependency grouping. All the common dependencies can be grouped into a single POM file. This approach is much better than POM inheritance. Here, you do not need to add references to individual dependencies. Let's go through a simple example. First, we need to logically group all the dependencies into a single POM file.


Apache Axis2 is an open source SOAP engine. To build an Axis2 client, you need to have all the following dependencies added to your project:

```
<dependency>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-kernel</artifactId>
  <version>1.6.2</version>
</dependency>
<dependency>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-adb</artifactId>
  <version>1.6.2</version>
</dependency>
<dependency>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-transport-http</artifactId>
  <version>1.6.2</version>
</dependency>
<dependency>
```

```

    <groupId>org.apache.axis2</groupId>
    <artifactId>axis2-transport-local</artifactId>
    <version>1.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.axis2</groupId>
    <artifactId>axis2-xmlbeans</artifactId>
    <version>1.6.2</version>
  </dependency>

```


 If you have multiple Axis2 client modules, in each module, you need to duplicate all these dependencies. The complete source code of the Apache Axis2 project is available at <http://svn.apache.org/viewvc/axis/axis2/java/core/trunk/modules/>.

To avoid dependency duplication, we can create a Maven module with all the previously mentioned five dependencies, as shown in the following project. Make sure to set the value of the packaging element to pom:

```

<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>axis2-client</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>

  <dependencies>
    <dependency>
      <groupId>org.apache.axis2</groupId>
      <artifactId>axis2-kernel</artifactId>
      <version>1.6.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.axis2</groupId>
      <artifactId>axis2-adb</artifactId>
      <version>1.6.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.axis2</groupId>
      <artifactId>axis2-transport-http</artifactId>
      <version>1.6.2</version>
    </dependency>
  </dependencies>

```

```
<dependency>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-transport-local</artifactId>
  <version>1.6.2</version>
</dependency>
<dependency>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-xmlbeans</artifactId>
  <version>1.6.2</version>
</dependency>
</dependencies>

</project>
```

Now, in all of your Axis2 client projects, you only need to add a dependency to the `com.packt.axis2-client` module, shown as follows:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>my-axis2-client</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>com.packt</groupId>
      <artifactId>axis2-client</artifactId>
      <version>1.0.0</version>
      <type>pom<type>
    </dependency>
  </dependencies>

</project>
```



Make sure to set the value of the `type` element to `pom` under the dependency element, as we are referring to a dependency of `pom` packaging here. In case it is skipped, Maven, by default, will look for an artifact with the `jar` packaging:

Transitive dependencies

The transitive dependency feature was introduced in Maven 2.0, which automatically identifies the dependencies of your project dependencies and gets all of them into the build path of your project. Let's take the following POM as an example. It only has a single dependency:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>jose</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>com.nimbusds</groupId>
      <artifactId>nimbus-jose-jwt</artifactId>
      <version>2.26</version>
    </dependency>
  </dependencies>

</project>
```

If you try to create an Eclipse project from the previous POM file using the `mvn eclipse:eclipse` command, it will result in the following `.classpath` file. There you can see, in addition to the `nimbus-jose-jwt-2.26.jar` file, three more JARs have been added. These are the transitive dependencies of the `nimbus-jose-jwt` dependency:

```
<classpath>
  <classpathentry kind="src" path="src/main/java"
    including="**/*.java"/>
  <classpathentry kind="output" path="target/classes"/>
  <classpathentry kind="con"
    path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry kind="var" path="M2_REPO/com/nimbusds/nimbus-
    jose-jwt/2.26/nimbus-jose-jwt-2.26.jar"/>
  <classpathentry kind="var" path="M2_REPO/net/jcip/jcip-
    annotations/1.0/jcip-annotations-1.0.jar"/>
  <classpathentry kind="var" path="M2_REPO/net/minidev/json-
    smart/1.1.1/json-smart-1.1.1.jar"/>
  <classpathentry kind="var"
    path="M2_REPO/org/bouncycastle/bcprov-jdk15on/1.50/bcprov-
    jdk15on-1.50.jar"/>
</classpath>
```

If you look at the POM file of the `nimbus-jose-jwt` project, you will see that the previously mentioned transitive dependencies are defined there as dependencies. Maven does not define a limit for transitive dependencies. One transitive dependency may have a reference to another transitive dependency, and it can go on like that endlessly, given that there are no cyclic dependencies found.

Transitive dependencies can cause some pain too, if not used with care. If we take the same Maven module that we discussed before as an example, and have the following Java code inside the `src/main/java` directory, it will compile exquisitely with no errors. This only has a single dependency — `nimbus-jose-jwt-2.26.jar`. However, the `net.minidev.json.JSONArray` class comes from a transitive dependency, which is `json-smart-1.1.1.jar`. The build works fine, because Maven gets all the transitive dependencies into the project build path. Everything will work finely until, one fine day, you update the version of `nimbus-jose-jwt` and the new version has a reference to a new version of the `json-smart` JAR, which is not compatible with your code. This could easily break your build, or it may cause test cases to fail. This would create hazards, and it would be a nightmare to find out the root cause.

The following Java code uses the `JSONArray` class from `json-smart-1.1.1.jar`:

```
import net.minidev.json.JSONArray;
import com.nimbusds.jwt.JWTClaimsSet;

public class JOSEUtil {

    public static void main(String[] args) {

        JWTClaimsSet jwtClaims = new JWTClaimsSet();

        JSONArray jsonArray = new JSONArray();

        jsonArray.add("maven-book");

        jwtClaims.setIssuer("https://packt.com");

        jwtClaims.setSubject("john");

        jwtClaims.setCustomClaim("book", jsonArray);

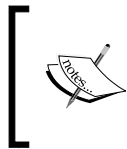
    }
}
```

To avoid such a nightmare, you need to follow a simple rule of thumb. If you have any `import` statement in a Java class, you need to make sure that the dependency JAR file corresponding to this is being added to the project POM file.

The Maven dependency plugin helps you to find such inconsistencies in your Maven module. Run the following command and observe its output:

```
$ mvn dependency:analyze
[INFO] --- maven-dependency-plugin:2.8:analyze (default-cli) @
jose ---
[WARNING] Used undeclared dependencies found:
[WARNING] net.minidev:json-smart:jar:1.1.1:compile
```

Note the two warnings in the previous output. It clearly says that we have an undeclared dependency for `json-smart jar`.



The Maven dependency plugin has several goals to find out inconsistencies and possible loopholes in how you manage dependencies. For more details on this, refer to <http://maven.apache.org/plugins/maven-dependency-plugin/>.

Dependency scopes

Maven defines the following six scope types. If there is no `scope` element defined for a given dependency, the default scope—`compile`—will be applied.

- `compile`: This is the default scope. Any dependency defined under the `compile` scope will be available in all the class paths. It will be packaged into the final artifact produced by the Maven project. If you are building a WAR type artifact, then the referred JAR files with the `compile` scope will be embedded into the WAR file itself.
- `provided`: This scope would expect that, the corresponding dependency would be provided either by the JDK or a container that runs the application. The best example is the servlet API. Any dependency with the `provided` scope will be available in the build time class path, but it won't be packaged into the final artifact. If it's a WAR file, the servlet API will be available in the class path during build time, but won't get packaged into the WAR file. See the following example of the `provided` scope:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.0.1</version>
```

```
<scope>provided</scope>
</dependency>
```

- **runtime:** Dependencies defined under the `runtime` scope will be available only during the runtime, not in the build time class path. These dependencies will be packaged into the final artifact. You may have a web-based app that talks to a MySQL database in runtime. Your code does not have any hard dependency to the MySQL database driver. The code is written against the Java JDBC API, and it does not need the MySQL database driver at build time. However, during runtime, it needs the driver to talk to the MySQL database. For this, the driver should be packaged into the final artifact.
- **test:** Dependencies are only needed for test compilation (for example, JUnit and TestNG), and execution must be defined under the `test` scope. These dependencies won't get packaged into the final artifact.
- **system:** This is very much similar to the scope `provided`. The only difference is that with the `system` scope, you need to tell Maven how to find it. System dependencies are useful when you do not have the referred dependency in a Maven repository. With this you need to make sure that all the system dependencies are available to download with the source code itself. It is always recommended to avoid using system dependencies. The following code snippet shows how to define a system dependency:

```
<dependency>
  <groupId>com.packt</groupId>
  <artifactId>jose</artifactId>
  <version>1.0.0</version>
  <scope>system</scope>
  <systemPath>${basedir}/lib/jose.jar</systemPath>
</dependency>
```



`basedir` is an inbuilt property defined in Maven to represent the directory, which has the corresponding POM file.

- **import:** This is only applicable for dependencies defined under the `dependencyManagement` section with the packaging type `pom`. Let's take the following POM file; it has the packaging type defined as `pom`:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>axis2-client</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
```

```

    <dependencyManagement>
      <dependencies>
        <dependency>
          <groupId>org.apache.axis2</groupId>
          <artifactId>axis2-kernel</artifactId>
          <version>1.6.2</version>
        </dependency>
        <dependency>
          <groupId>org.apache.axis2</groupId>
          <artifactId>axis2-adb</artifactId>
          <version>1.6.2</version>
        </dependency>
      </dependencies>
    </dependencyManagement>
  </project>

```

Now, from a different Maven module, we add a dependency under the `dependencyManagement` section to the previous module, with the scope value set to `import` and the value of type set to `pom`:

```

<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>my-axis2-client</artifactId>
  <version>1.0.0</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.packt</groupId>
        <artifactId>axis2-client</artifactId>
        <version>1.0.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>

```

Now, if we run `mvn help:effective-pom` against the above POM file, we will see that the dependencies from the first are being imported as follows:

```

<dependencyManagement>
  <dependencies>
    <dependency>

```



```
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-kernel</artifactId>
<version>1.6.2</version>
</dependency>
<dependency>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-adb</artifactId>
  <version>1.6.2</version>
</dependency>
</dependencies>
</dependencyManagement>
```

Optional dependencies

Let's say that we have a Java project that has to work with two different OSGi runtimes. We have written almost all the code to the OSGi API, but there are certain parts in the code that consumes the OSGi runtime-specific APIs. When the application is running, only the code path related to the underneath OSGi runtime will get executed, not both. This raises the need to have both the OSGi runtime JARs at the build time. However, in runtime, we do not need both code execution paths, only the one related to the corresponding OSGi runtime. We can meet these requirements by optional dependencies, shown as follows:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>osgi.client</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>org.eclipse.equinox</groupId>
      <artifactId>osgi</artifactId>
      <version>3.1.1</version>
      <scope>compile</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.apache.phoenix</groupId>
      <artifactId>phoenix-core</artifactId>
      <version>3.0.0-incubating</version>
      <scope>compile</scope>
      <optional>true</optional>
```

```
    </dependency>
  </dependencies>

</project>
```

For any client project that needs `com.packt.osgi.client` to work in an Equinox OSGi runtime, it must explicitly add a dependency to the Equinox JAR file, as shown in the following code:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>my.osgi.client</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>org.eclipse.equinox</groupId>
      <artifactId>osgi</artifactId>
      <version>3.1.1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.packt</groupId>
      <artifactId>osgi.client</artifactId>
      <version>1.0.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>

</project>
```

Dependency exclusion

Dependency exclusion helps to avoid getting a selected set of transitive dependencies. Say, for example, that we have the following POM file with two dependencies: one for `nimbus-jose-jwt` and the other for the `json-smart` artifact:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>jose</artifactId>
  <version>1.0.0</version>
```

```
<dependencies>
  <dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>nimbus-jose-jwt</artifactId>
    <version>2.26</version>
  </dependency>
  <dependency>
    <groupId>net.minidev</groupId>
    <artifactId>json-smart</artifactId>
    <version>1.0.9</version>
  </dependency>
</dependencies>

</project>
```

If you try to run `mvn eclipse:eclipse` against the previous POM file, you will see the following `.classpath` file having a dependency on the `json-smart` file version 1.0.9, as rightly expected:

```
<classpathentry kind="var" path="M2_REPO/net/minidev/json-smart/1.0.9/json-smart-1.0.9.jar"/>
```

Let's say that we have another project that refers the same `nimbus-jose-jwt` artifact, and a newer version of the `json-smart` JAR file:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>jose.ext</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>com.nimbusds</groupId>
      <artifactId>nimbus-jose-jwt</artifactId>
      <version>2.26</version>
    </dependency>
    <dependency>
      <groupId>net.minidev</groupId>
      <artifactId>json-smart</artifactId>
      <version>1.1.1</version>
    </dependency>
  </dependencies>

</project>
```

If you try to run `mvn eclipse:eclipse` against the previous POM file, you will see the following `.classpath` file having a dependency on the `json-smart` artifact version 1.1.1:

```
<classpathentry kind="var" path="M2_REPO/net/minidev/json-smart/1.1.1/json-smart-1.1.1.jar"/>
```

Still, we do not see a problem. Now, say that we build a WAR file having dependencies to both the previous Maven modules:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>jose.war</artifactId>
  <version>1.0.0</version>
  <version>war</version>

  <dependencies>
    <dependency>
      <groupId>com.packt</groupId>
      <artifactId>jose</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>com.packt</groupId>
      <artifactId>jose.ext</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>

</project>
```

Once the WAR file is created inside `WEB-INF/lib`, we can only see version 1.1.1 of the `json-smart` JAR file. This comes as a transitive dependency of the `com.packt.jose.ext` project. There can be a case where the WAR file does not need version 1.1.1 in its runtime, but version 1.0.9. To achieve this, we need to exclude the version 1.1.1 of the `json-smart` JAR file from the `com.packt.jose.ext` project, as shown in the following code:

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>jose.war</artifactId>
  <version>1.0.0</version>
```

```
<version>war</version>

<dependencies>
  <dependency>
    <groupId>com.packt</groupId>
    <artifactId>jose</artifactId>
    <version>1.0.0</version>
  </dependency>
  <dependency>
    <groupId>com.packt</groupId>
    <artifactId>jose.ext</artifactId>
    <version>1.0.0</version>
    <exclusions>
      <exclusion>
        <groupId>net.minidev</groupId>
        <artifactId>json-smart</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>

</project>
```

Now, if you look inside `WEB-INF/lib`, you will only see version 1.0.9 of the `json-smart` JAR file.

Summary

In this chapter, we focused our discussion around Maven POM, and how to adhere to industry-wide accepted best practices to avoid maintenance nightmares. The key elements of a POM file, POM hierarchy and inheritance, managing dependencies, and related topics were covered here.

In the next chapter, we will have a look at Maven archetypes.

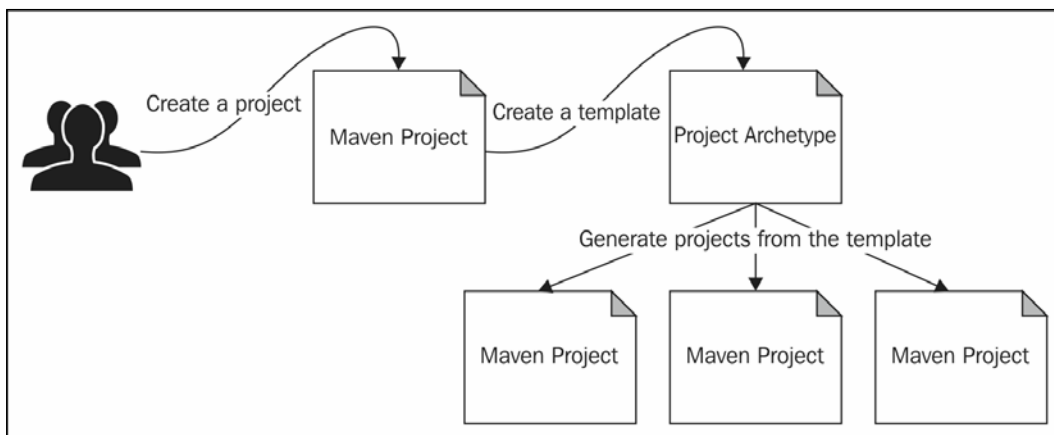
3

Maven Archetypes

The word **archetype** has its roots in Greek literature. It's derived from two Greek words, **archein** and **typos**. The word **archein** means original or old, while **typos** means patterns.

The word archetype means original patterns. The famous psychologist, Carl Gustav Jung introduced the archetype concept in psychology. Jung argued that there are 12 different archetypes that represent human motivation, and he further divided them into three categories: ego, soul, and self. The innocent, regular guy, hero, and caregiver fall under the ego type. The explorer, rebel, lover, and creator fall under the soul type. The self type includes jester, sage, magician, and ruler. The concept behind Maven archetypes does not deviate a lot from what Jung explained in psychology.

The following figure shows the relationship between a Maven project, a project archetype, and projects generated from the archetype:



When we create a Java project, we need to structure it in different ways based on the type of the project. If it's a Java EE web application, then we need to have a `WEB-INF` directory and a `web.xml` file. If it's a Maven plugin project, we need to have a `Mojo` class that extends from `org.apache.maven.plugin.AbstractMojo`. As each type of project has its own predefined structure, why would everyone have to build the same structure again and again? Why not start with a template? Each project can have its own template, and developers can extend the template to suite their requirements. Maven archetypes address this concern. Each archetype is a project template.



A list of Maven archetypes can be found at
<http://maven-repository.com/archetypes>.

In this chapter, we will discuss the following topics:

- The Maven archetype plugin
- The most used archetypes

Archetype quickstart

The Maven archetype is a plugin in itself. We will discuss plugins in detail in *Chapter 4, Maven Plugins*. The `generate` goal of the archetype plugin has been used to generate a Maven project from an archetype. Let's start with a simple example:

```
$ mvn archetype:generate
   -DgroupId=com.packt.samples
   -DartifactId=com.packt.samples.archetype
   -Dversion=1.0.0
   -DinteractiveMode=false
```

This command will invoke the `generate` goal of the Maven archetype plugin to create a simple Java project. You will see that the following project structure has been created with a sample POM file. The name of the root or the base directory is derived from the value of the `artifactId` parameter:

```
com.packt.samples.archetype
| -pom.xml
| -src
| -main/java/com/packt/samples/App.java
| -test/java/com/packt/samples/AppTest.java
```

The sample POM file will only have a dependency to the junit JAR file, with test as the scope:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.samples</groupId>
  <artifactId>com.packt.samples.archetype</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>com.packt.samples.archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The generated `App.java` class will have the following template code. The name of the package is derived from the provided `groupId` parameter. If we want a different value as the package name, then we need to pass this value in the command itself as `-Dpackage=com.packt.samples.application`:

```
package com.packt.samples;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

This is the simplest way to get started with a Maven project. In the previous example, we used the non-interactive mode by setting `interactiveMode=false`. This will force the plugin to use whatever values we passed in the command itself, along with the default values.

To invoke the plugin in the interactive mode, just type `mvn archetype:generate`. This will prompt for user inputs as the plugin proceeds with its execution. The very first one is to ask for a filter or a number for the type of the archetype. The filter can be specified in the format of `[groupId:]artifactId`, shown as follows:

```
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 471:
```

When you type the filter criteria, for example, `org.apache.maven.archetypes:maven-archetype-quickstart`, the plugin will display the number associated with it, shown as follows:

```
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 471: org.apache.maven.archetypes:maven-
archetype-quickstart
```

Choose archetype:

```
1: remote -> org.apache.maven.archetypes:maven-archetype-quickstart
(An archetype which contains a sample Maven project.)
```

```
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 1:
```

In this case, there is only one archetype which matches the filter, and the number associated with it is 1. If you press *Enter* against the last line in the previous output, or just type 1, the plugin will start to proceed with the `org.apache.maven.archetypes:maven-archetype-quickstart` archetype.

Something that you might have already noticed is that as soon as you type `mvn archetype:generate`, the plugin displays a long list of Maven archetypes supported by the plugin, and each archetype has a number associated with it. You can avoid this long list by specifying a filter criterion with the command itself, shown as follows:

```
$ mvn archetype:generate
-Dfilter=org.apache.maven.archetypes:maven-archetype-quickstart
```

Choose archetype:

```
1: remote -> org.apache.maven.archetypes:maven-archetype-quickstart
(An archetype which contains a sample Maven project.)
```

```
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 1:
```

Batch mode

The archetype plugin can operate in the batch mode either by setting the `interactiveMode` argument to `false` or passing `-B` as an argument. When operating in the batch mode, you need to clearly specify which archetype you are going to use with the arguments `archetypeGroupId`, `archetypeArtifactId`, and `archetypeVersion`. You also need to clearly identify the resultant artifact with the `groupId`, `artifactId`, `version`, and `package` arguments, shown as follows:

```
$ mvn archetype:generate -B
-DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.0
-DgroupId=com.packt.samples
-DartifactId=com.packt.samples.archetype
-Dversion=1.0.0
-Dpackage=1.5
```

Any inquisitive mind should be asking a very valid question by now.

In the non-interactive mode, we did not type any filter or provide any Maven coordinates for the archetype in the very first example. So, how does the plugin know about the archetype? When no archetype is specified, the plugin goes with the default one, which is `org.apache.maven.archetypes:maven-archetype-quickstart`.

Archetype catalogues

How does the plugin find all the archetypes available in the system? When you just type `mvn archetype:generate`, a list of archetypes is displayed by the plugin for the user selection. The complete list is around 1100, but only the first 10 are shown here:

```
1: remote -> br.com.ingenieux:elasticbeanstalk-service-webapp-
  archetype (A Maven Archetype Encompassing RestAssured, Jetty,
  Jackson, Guice and Jersey for Publishing JAX-RS-based Services on
  AWS' Elastic Beanstalk Service)
2: remote -> br.com.ingenieux:elasticbeanstalk-wrapper-webapp-
  archetype (A Maven Archetype Wrapping Existing war files on AWS'
  Elastic Beanstalk Service)
3: remote -> br.com.otavio.vraptor.archetypes:vraptor-archetype-blank
  (A simple project to start with VRaptor 4)
4: remote -> br.gov.frameworkdemoiselle.archetypes:demoiselle-html-
  rest (Archetype for web applications (HTML + REST) using Demoiselle
  Framework)
```

```
5: remote -> br.gov.frameworkdemoiselle.archetypes:demoiselle-jsf-jpa
  (Archetype for web applications (JSF + JPA) using Demoiselle
  Framework)
6: remote -> br.gov.frameworkdemoiselle.archetypes:demoiselle-minimal
  (Basic archetype for generic applications using Demoiselle
  Framework)
7: remote -> br.gov.frameworkdemoiselle.archetypes:demoiselle-vaadin-
  jpa (Archetype for Vaadin web applications)
8: remote -> ch.sbb.maven.archetypes:iib9-maven-projects (IBM
  Integration Bus 9 Maven Project Structure)
9: remote -> ch.sbb.maven.archetypes:wmb7-maven-projects (WebSphere
  Message Broker 7 Maven Project Structure)
10: remote -> co.ntier:spring-mvc-archetype (An extremely simple
  Spring MVC archetype, configured with NO XML.)
```

Going back to the original question, how does the plugin find these details about different archetypes?

The archetype plugin maintains the details about different archetypes in an internal catalogue, which comes with the plugin itself. The archetype catalogue is simply an XML file. The following shows the internal catalogue of the archetype plugin:

```
<archetype-catalog>

<!-- Internal archetype catalog listing archetypes from the Apache
Maven project. -->

<archetypes>
  <archetype>
    <groupId>org.apache.maven.archetypes</groupId>
    <artifactId>maven-archetype-archetype</artifactId>
    <version>1.0</version>
    <description>An archetype which contains a sample
      archetype.</description>
  </archetype>
  <archetype>
    <groupId>org.apache.maven.archetypes</groupId>
    <artifactId>maven-archetype-j2ee-simple</artifactId>
    <version>1.0</version>
    <description>An archetype which contains a simplified sample
      J2EE application.</description>
  </archetype>
  <archetype>
    <groupId>org.apache.maven.archetypes</groupId>
```

```
<artifactId>maven-archetype-plugin</artifactId>
<version>1.2</version>
<description>An archetype which contains a sample Maven
  plugin.</description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
  <artifactId>maven-archetype-plugin-site</artifactId>
  <version>1.1</version>
  <description>An archetype which contains a sample Maven plugin
    site.This archetype can be layered upon an existing Maven
    plugin project.</description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
  <artifactId>maven-archetype-portlet</artifactId>
  <version>1.0.1</version>
  <description>An archetype which contains a sample JSR-268
    Portlet.</description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
  <artifactId>maven-archetype-profiles</artifactId>
  <version>1.0-alpha-4</version>
  <description></description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
  <artifactId>maven-archetype-quickstart</artifactId>
  <version>1.1</version>
  <description>An archetype which contains a sample Maven
    project.</description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
  <artifactId>maven-archetype-site</artifactId>
  <version>1.1</version>
  <description>An archetype which contains a sample Maven site
    which demonstrates some of the supported document types like
    APT, XDoc, and FML and demonstrates how to i18n your site.
    This archetype can be layered upon an existing Maven
    project.</description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
```

```
<artifactId>maven-archetype-site-simple</artifactId>
<version>1.1</version>
<description>An archetype which contains a sample Maven
  site.</description>
</archetype>
<archetype>
  <groupId>org.apache.maven.archetypes</groupId>
  <artifactId>maven-archetype-webapp</artifactId>
  <version>1.0</version>
  <description>An archetype which contains a sample Maven Webapp
    project.</description>
</archetype>
</archetypes>
</archetype-catalog>
```



In addition to the internal catalogue, you can also maintain a local archetype catalogue. This is available at `USER_HOME/.m2/archetype-catalog.xml`, and by default, it's an empty file.



There is also a remote catalogue available at `http://repo1.maven.org/maven2/archetype-catalog.xml`.

By default, the archetype plugin will load all the available archetypes from the local and remote catalogues. If we go back to the archetype list displayed by the plugin and type `mvn archetype:generate`, then by looking at each entry, we can determine whether a given archetype is loaded from the internal, local, or remote catalogue.

For example, the following archetype is loaded from the remote catalogue:

```
1: remote -> br.com.ingenieux:elasticbeanstalk-service-webapp-
  archetype (A Maven Archetype Encompassing RestAssured, Jetty,
  Jackson, Guice and Jersey for Publishing JAX-RS-based Services on
  AWS' Elastic Beanstalk Service)
```

If you want to force the archetype plugin to list all the archetypes from the internal catalogue only, then you need to use the following command:

```
$ mvn archetype:generate -DarchetypeCatalog=internal
```

To list all the archetypes from the local catalogue only, you need to use the following command:

```
$ mvn archetype:generate -DarchetypeCatalog=local
```

To list all the archetypes from the `internal`, `local`, and `remote` catalogues, you need to use the following command:

```
$ mvn archetype:generate -DarchetypeCatalog=internal,local,remote
```

Building an archetype catalogue

In addition to the `internal`, `local`, and `remote` catalogues, you can also build your own catalogue. Say you have developed your own set of Maven archetypes and need to build a catalogue out of them, which can be shared with others by publicly hosting it. Once you have built the archetypes, they will be available in your `local` Maven repository. The following command will crawl through the `local` Maven repository and build an archetype catalogue from all the archetypes available there. Here, we use the `crawl` goal of the `archetype` plugin:

```
$ mvn archetype:crawl -Dcatalog=my-catalog.xml
```

Public archetype catalogues

People who develop archetypes for their projects will list them in publicly hosted archetype catalogues. The following list shows some of the publicly available Maven archetype catalogues:

- **Fuse:** The Fuse archetype catalogue can be found at <http://repo.fusesource.com/nexus/content/groups/public/archetype-catalog.xml>
- **Java.net:** The Java.net archetype catalogue can be found at <http://download.java.net/maven/2/archetype-catalog.xml>
- **Cocoon:** The Cocoon archetype catalogue can be found at <http://cocoon.apache.org/archetype-catalog.xml>
- **MyFaces:** The MyFaces archetype catalogue can be found at <http://myfaces.apache.org/archetype-catalog.xml>
- **Apache Synapse:** The Apache Synapse archetype catalogue can be found at <http://synapse.apache.org/archetype-catalog.xml>

Let's take Apache Synapse as an example. Synapse is an open source Apache project that builds an **enterprise service bus (ESB)**. The following command uses the Apache Synapse archetype to generate a Maven project:

```
$ mvn archetype:generate
    -DgroupId=com.packt.samples
    -DartifactId=com.packt.samples.synapse
    -Dversion=1.0.0
```

```
-Dpackage=com.packt.samples.synapse.application
-DarchetypeCatalog=http://synapse.apache.org
-DarchetypeGroupId=org.apache.synapse
-DarchetypeArtifactId=synapse-package-archetype
-DarchetypeVersion=2.0.0
-DinteractiveMode=false
```

The previous command will produce the following directory structure. If you look at the `pom.xml` file, you will notice that it contains all the necessary instructions along with the required dependencies to build the Synapse project:

```
com.packt.samples.synapse
| -pom.xml
| -src/main/assembly/bin.xml
| -conf/log4j.properties
| -repository/conf
|               | -axis2.xml
|               | -synapse.xml
```

Let's have a look at the previous Maven command that we had used to build the project with the Synapse archetype. The most important argument is `archetypeCatalog`. The value of the `archetypeCatalog` argument can point directly to the `archetype-catalog.xml` file or to a directory that contains the `archetype-catalog.xml` file. The following configuration shows the `archetype-catalog.xml` file corresponding to the Synapse archetype. It only has a single archetype, but with two different versions:

```
<archetype-catalog>
  <archetypes>
    <archetype>
      <groupId>org.apache.synapse</groupId>
      <artifactId>synapse-package-archetype</artifactId>
      <version>1.3</version>
      <repository>http://repo1.maven.org/maven2</repository>
      <description>Create a Synapse 1.3 custom package</description>
    </archetype>
    <archetype>
      <groupId>org.apache.synapse</groupId>
      <artifactId>synapse-package-archetype</artifactId>
      <version>2.0.0</version>
      <repository>
        http://people.apache.org/repo/m2-snapshot-repository
      </repository>
      <description>Create a Synapse 2.0.0 custom
        package</description>
    </archetype>
  </archetypes>
</archetype-catalog>
```

```

    </archetype>
  </archetypes>
</archetype-catalog>

```



The value of the `archetypeCatalog` parameter can be a comma-separated list, where each item points to an `archetype-catalog.xml` file or to a directory, which contains `archetype-catalog.xml`. The default values are `remote` and `local`, where the archetypes are loaded from the local repository and the remote repository. If you want to load an `archetype-catalog.xml` file from the local file system, then you need to prefix the absolute path to the file with `file:///`. The value `local` is just a shortcut for `file://~/.m2/archetype-catalog.xml`.

In the previous Maven command, we used the `archetype` plugin in the non-interactive mode, so we had to be very specific with the archetype that we needed to generate the Maven project. This was done with the following three arguments. The value of these three arguments must match the corresponding elements defined in the associated `archetype-catalog.xml` file:

```

-DarchetypeGroupId=org.apache.synapse
-DarchetypeArtifactId=synapse-package-archetype
-DarchetypeVersion=2.0.0

```

The anatomy of archetype – catalog.xml

We have already gone through a couple of sample `archetype-catalog.xml` files and their uses. The XML schema of the `archetype-catalog.xml` file is available at <http://maven.apache.org/xsd/archetype-catalog-1.0.0.xsd>. The following shows an `archetype-catalog.xml` file skeleton with all the key elements:

```

<archetype-catalog>
  <archetypes>
    <archetype>
      <groupId></groupId>
      <artifactId></artifactId>
      <version></version>
      <repository></repository>
      <description></description>
    </archetype>
    ...
  </archetypes>
</archetype-catalog>

```


The archetypes parent element can hold one or more archetype child elements. Each archetype element should uniquely identify the Maven artifact corresponding to it. This is done by combining the `groupId`, `artifactId`, and `version` elements of the artifact. These three elements carry the exact same meaning that we discussed under Maven coordinates. The `description` element can be used to describe the archetype. The value of the `description` element will appear against the archetype when it is listed by the archetype plugin. For example, the following output is generated according to the pattern—`groupId:artifactId (description)` from the `archetype-catalog.xml` file when you type `mvn archetype:generate`:

Choose archetype:

```
1: remote -> org.apache.maven.archetypes:maven-archetype-quickstart
   (An archetype which contains a sample Maven project.)
```

Each archetype child element can carry a value for the `repository` element. This instructs the archetype plugin where to find the corresponding artifact. When no value is specified, the artifact is loaded from the repository, where the catalogue file comes from.

The archetype plugin goals

So far in this chapter, we have only discussed the `generate` and `crawl` goals of the archetype plugin. All the useful functionalities in the Maven build process are developed as plugins. A given Maven plugin can have multiple goals, where each goal carries out a very specific task. We will discuss plugins in detail in *Chapter 4, Maven Plugins*.

The following goals are associated with the archetype plugin:

- `archetype:generate`: The `generate` goal creates a Maven project corresponding to the selected archetype. This accepts the `archetypeGroupId`, `archetypeArtifactId`, `archetypeVersion`, `filter`, `interactiveMode`, `archetypeCatalog`, and `baseDir` arguments. We have already discussed almost all of these arguments in detail.
- `archetype:update-local-catalog`: The `update-local-catalog` goal has to be executed against a Maven archetype project. This will update the `local` archetype catalog with the new archetype. The `local` archetype catalog is available at `~/ .m2/archetype-catalog.xml`.

- `archetype:jar`: The `jar` goal has to be executed against a Maven archetype project, which will create a JAR file out of it. This accepts the `archetypeDirectory` argument, which contains the classes; it also accepts the `finalName` argument, the name of the JAR file to be generated, and the `outputDirectory` argument, which is the location where the final output is copied.
- `archetype:crawl`: The `crawl` goal crawls through a local or a file system-based Maven repository (not remote or via HTTP) and creates an archetype catalogue file. This accepts `catalogFile` as an argument (which maps into the `catalog` system property), which is the name of the catalogue file to be created. By default, this crawls through the `local` Maven repository, and to override the location, we need to pass the corresponding repository URL with the `repository` argument.
- `archetype:create-from-project`: The `create-from-project` goal creates an archetype project from an existing project. If you compare this with the `generate` goal, then `generate`, in fact, creates a new Maven project from scratch corresponding to the selected archetype, while `create-from-project` creates a Maven archetype project from an existing project. In other words, `create-from-project` generates a template out of an existing Maven project.
- `archetype:integration-test`: The `integration-test` goal will execute the integration tests associated with the Maven archetype project.
- `archetype:help`: The `help` goal will display the manual associated with the archetype plugin, listing out all the available goals. If you want to get a detailed description of all the goals, then use the `-Ddetail=true` parameter along with the command. It is also possible to get help for a given goal. For example, the following command will display the help associated with the `generate` goal:

```
$ mvn archetype:help -Ddetail=true -Dgoal=generate
```

Java EE web applications with the archetype plugin

If you want to start with a Java EE web application, you can simply use the `maven-archetype-webapp` archetype to generate the Maven project skeleton, shown as follows:

```
$ mvn archetype:generate -B
-DgroupId=com.packt.samples
-DartifactId=my-webapp
```

```
-Dpackage=com.packt.samples.webapp
-Dversion=1.0.0
-DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-webapp
-DarchetypeVersion=1.0
```

The preceding command will produce the following directory structure. One issue here is that it does not have the `java` directory just after `src/main`. If you want to add any Java code, you need to make sure that you first create an `src/main/java` directory and create your Java package under it; otherwise, with the default configuration settings, Maven won't pick your classes for compilation. By default, Maven looks for the source code inside `src/main/java`:

```
my-webapp
| -pom.xml
| -src/main/webapp
|       | -index.jsp
|       | -WEB-INF/web.xml
| - src/main/resources
```

The `maven-archetype-webapp` archetype is not the only archetype to generate a Java EE project using the archetype plugin. Codehaus, a collaborative environment to build open source projects, also provides a few archetypes to generate web applications. The following example uses the `webapp-javaee6` archetype from Codehaus:

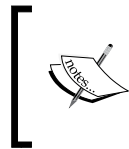
```
$ mvn archetype:generate -B
-DgroupId=com.packt.samples
-DartifactId=my-webapp
-Dpackage=com.packt.samples.webapp
-Dversion=1.0.0
-DarchetypeGroupId=org.codehaus.mojo.archetypes
-DarchetypeArtifactId=webapp-javaee6
-DarchetypeVersion=1.3
```

The preceding command will produce the following directory structure. This overcomes one of the issues in the `maven-archetype-webapp` archetype, and creates the `src/main/java` and `src/test/java` directories. The only issue here is that it does not create the `src/main/webapp/WEB-INF` directory, which you will have to create manually:

```
my-webapp
| -pom.xml
| -src/main/webapp/index.jsp
| -src/main/java/com/packt/samples/webapp/
| -src/test/java/com/packt/samples/webapp/
```

Deploying web applications to a remote Apache Tomcat server

Now, we have created a template web application either using the `maven-archetype-webapp` or `webapp-javaee6` archetype. Let's see how to deploy this web application into a remote Apache Tomcat application server from Maven itself. Most developers would love doing this rather than manual copying.



This assumes you have already installed Apache Tomcat in your environment. If not, you can download Tomcat 7.x distribution from <http://tomcat.apache.org/download-70.cgi> and set it up.

To deploy the web application, perform the following steps:

1. As we are going to deploy the web application to a remote Tomcat server, we need to have a valid user account that has the privilege to deploy a web application. Add the following entries to the `TOMCAT_HOME/conf/tomcat-users.xml` file under the `tomcat-users` root element. This will create a user with the name `admin` and the password `password`, and the `manager-gui` and `manager-script` roles:

```
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<user username="admin" password="password" roles="manager-
gui,manager-script" />
```

2. Now, we need to configure Maven to talk to the remote Tomcat server. Add the following configuration to `USER_HOME/.m2/settings.xml` under the `servers` element, shown as follows:

```
<server>
  <id>apache-tomcat</id>
  <username>admin</username>
  <password>password</password>
</server>
```

3. Go inside the root directory of the template web application that we generated before (`my-webapp`), and then add the `tomcat7-maven-plugin` to the `pom.xml` file available there. The complete `pom.xml` file will look like this:

```
<project >
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.samples</groupId>
  <artifactId>my-webapp</artifactId>
  <packaging>war</packaging>
```

```
<version>1.0.0</version>
<name>my-webapp Maven Webapp</name>
<url>http://maven.apache.org</url>

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

<build>
  <finalName>my-webapp</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <url>http://localhost:8080/manager/text</url>
        <server>apache-tomcat</server>
        <path>/my-webapp</path>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

4. Use the following Maven command to build and deploy the sample web application into the Tomcat server. Once it is deployed, you can access it via `http://localhost:8080/my-webapp/`:

```
$ mvn clean install tomcat7:deploy
```
5. To redeploy, use the following command:

```
$ mvn clean install tomcat7:redploy
```
6. To undeploy, use the following command:

```
$ mvn clean install tomcat7:undeploy
```

Android mobile applications with the archetype plugin

If you are an Android application developer who wants to start with a skeleton Android project, you can use the `android-quickstart` archetype developed by akquinet, shown as follows:

```
$ mvn archetype:generate -B
    -DarchetypeGroupId=de.akquinet.android.archetypes
    -DarchetypeArtifactId=android-quickstart
    -DarchetypeVersion=1.0.4
    -DgroupId=com.packt.samples
    -DartifactId=my-android-app
    -Dversion=1.0.0
```

This command produces the following skeleton project:

```
my-android-app
| -pom.xml
| -AndroidManifest.xml
| -android.properties
| -src/main/java/com/packt/samples/HelloAndroidActivity.java
| -res/drawable-hdpi/icon.png
| -res/drawable-ldpi/icon.png
| -res/drawable-mdpi/icon.png
| -res/layout/main.xml
| -res/values/strings.xml
| -assets
```

To build the Android skeleton project, run the following Maven command from the `my-android-app` directory:

```
$ mvn clean install -Dandroid.sdk.path=/path/to/android/sdk
```

The previous command looks straightforward, but is based on your Android SDK version; therefore, you might encounter certain issues. Some of the possible issues and solutions are as follows:

- You will see the following error if you pass an invalid value to the `android.sdk.path` argument:

```
[ERROR] Failed to execute goal
  com.jayway.maven.plugins.android.generation2:maven-android-
  plugin:2.8.3:generate-sources (default-generate-sources) on
  project my-android-app: Execution default-generate-sources
  of goal com.jayway.maven.plugins.android.generation2:maven-
  android-plugin:2.8.3:generate-sources failed: Path
  "/Users/prabath/Downloads/adt-bundle-mac-x86_64-
  20140702/platforms" is not a directory.
```

The path will point to the Android `sdk` directory, and right under this, you will find the `platforms` directory. By setting `android.sdk.path` to the correct path, you can avoid this error.

- By default, the `android-quickstart` archetype assumes the Android platform to be 7. You will see the following error if the Android platform installed in your local machine is different from this:

```
[ERROR] Failed to execute goal com.jayway.maven.plugins.android.
generation2:maven-android-
  plugin:2.8.3:generate-sources (default-generate-sources) on
  project my-android-app: Execution default-generate-sources
  of goal com.jayway.maven.plugins.android.generation2:maven-
  android-plugin:2.8.3:generate-sources failed: Invalid SDK:
  Platform/API level 7 not available.
```

To fix this, open the `pom.xml` file and set the right platform version with `<sdk><platform>20</platform></sdk>`.

- By default, the `android-quickstart` archetype assumes that the `aapt` tool is available under `sdk/platform-tools`. However, with the latest `sdk`s, it's being moved to `sdk/build-tools/android-4.4W`; you will get the following error:

```
[ERROR] Failed to execute goal
  com.jayway.maven.plugins.android.generation2:maven-android-
  plugin:2.8.3:generate-sources (default-generate-sources) on
  project my-android-app: Execution default-generate-sources
  of goal com.jayway.maven.plugins.android.generation2:maven-
  android-plugin:2.8.3:generate-sources failed: Could not find
  tool 'aapt'.
```

To fix the error, you need to update the `maven-android-plugin` version and `artifactId`.

Open up the `pom.xml` file inside the `my-android-app` directory and find the following plugin configuration. Change `artifactId` to `android-maven-plugin` and `version` to `4.0.0-rc.1`, shown as follows:

```
<plugin>
  <groupId>
    com.jayway.maven.plugins.android.generation2
  </groupId>
  <artifactId>android-maven-plugin</artifactId>
  <version>4.0.0-rc.1</version>
  <configuration></configuration>
  <extensions>true</extensions>
</plugin>
```

Once the build is complete, `android-maven-plugin` will produce the `my-android-app-1.0.0.apk` and `my-android-app-1.0.0.jar` artifacts inside the target directory.

To deploy the skeleton Android application (apk) to the connected device, use the following Maven command:

```
$ mvn android:deploy -Dandroid.sdk.path=/path/to/android/sdk
```

EJB archives with the archetype plugin

Here, we will discuss how to create a Maven **Enterprise JavaBeans (EJB)** project using the `ejb-javaee6` archetype developed by Codehaus, which is a collaborative environment to build open source projects:

```
$ mvn archetype:generate -B
    -DgroupId=com.packt.samples
    -DartifactId=my-ejbapp
    -Dpackage=com.packt.samples.ejbapp
    -Dversion=1.0.0
    -DarchetypeGroupId=org.codehaus.mojo.archetypes
    -DarchetypeArtifactId=ejb-javaee6
    -DarchetypeVersion=1.5
```

The previous command produces the following skeleton project. You can create your EJB classes inside `src/main/java/com/packt/samples/ejbapp/`:

```
my-ejbapp
| -pom.xml
| -src/main/java/com/packt/samples/ejbapp/
| -src/main/resources/META-INF/MANIFEST.MF
```


If you look at the following `pom.xml` file inside `my-ejbapp` directory, you will notice that `maven-ejb-plugin` is used internally to produce the EJB artifact:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-ejb-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <ejbVersion>3.1</ejbVersion>
  </configuration>
</plugin>
```

Even though we have highlighted `ejb-javaee6`, it is not the best out there to generate a Maven EJB project. The template produced by the `ejb-javaee6` archetype is very basic. Oracle WebLogic has developed a better EJB archetype, `-basic-webapp-ejb`. The following example shows how to use the `basic-webapp-ejb` archetype:

```
$ mvn archetype:generate -B
    -DarchetypeGroupId=com.oracle.weblogic.archetype
    -DarchetypeArtifactId=basic-webapp-ejb
    -DarchetypeVersion=12.1.3-0-0
    -DgroupId=com.packt.samples
    -DartifactId=my-ejbapp
    -Dpackage=com.packt.samples.ejbapp
    -Dversion=1.0.0
```

Prior to executing the previous command, there is more homework to be done. The `basic-webapp-ejb` archetype is not available in any public Maven repositories. First, you need to download the WebLogic distribution from http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/wls_12c_netbeans_install/wls_12c_netbeans_install.html, and then install it locally by performing the instructions given in the `README.txt` file. Once the installation is complete, the `basic-webapp-ejb` archetype and `weblogic-maven-plugin` can be installed into the local Maven repository, shown as follows:

1. Go to `wls12130/wlserver/server/lib` and execute the following command. This will build the plugin JAR file using the WebLogic JarBuilder tool:

```
$ java -jar wljarbuilder.jar -profile weblogic-maven-plugin
```
2. The previous command will create the `weblogic-maven-plugin.jar` file. Now, we need to extract it out to get the `pom.xml` file. From `wls12130/wlserver/server/lib`, execute the following command:

```
$ jar xvf weblogic-maven-plugin.jar
```

3. Now, we need to copy the `pom.xml` file to `wls12130/wlserver/server/lib`. From `wls12130/wlserver/server/lib`, execute the following command:


```
$ cp META-INF/maven/com.oracle.weblogic/weblogic-maven-plugin/pom.xml .
```
4. Now, we can install `weblogic-maven-plugin.jar` into the local Maven repository. From `wls12130/wlserver/server/lib`, execute the following command:


```
$ mvn install:install-file -Dfile=weblogic-maven-plugin.jar -DpomFile=pom.xml
```
5. In addition to the plugin, we also need to install the `basic-webapp-ejb` archetype. To do this, go to `wls12130/oracle_common/plugins/maven/com/oracle/maven/oracle-maven-sync/12.1.3` and execute the following two commands. Note that `oracle_common` is a hidden directory. If you are using a different version of WebLogic instead of 12.1.3, use the number associated with your version:


```
$ mvn install:install-file -DpomFile=oracle-maven-sync-12.1.3.pom -Dfile=oracle-maven-sync-12.1.3.jar
$ mvn com.oracle.maven:oracle-maven-sync:push -Doracle-maven-sync.oracleHome=/Users/prabath/Downloads/wls12130 -Doracle-maven-sync.testingOnly=false
```

Once you are done with these steps, you can execute the following command to generate the EJB template project using the WebLogic `basic-webapp-ejb` archetype. Make sure that you have the right version of `archetypeVersion`; this should match the archetype version that comes with your WebLogic distribution:

```
$ mvn archetype:generate -B
-DarchetypeGroupId=com.oracle.weblogic.archetype
-DarchetypeArtifactId=basic-webapp-ejb
-DarchetypeVersion=12.1.3-0-0
-DgroupId=com.packt.samples
-DartifactId=my-ejbapp
-Dpackage=com.packt.samples.ejbapp
-Dversion=1.0.0
```

This command produces the following skeleton project:

```
my-ejbapp
| -pom.xml
| -src/main/java/com/packt/samples/ejbapp
|   | -entity/Account.java
```

```
|-service/AccountBean.java
|-service/AccountManager.java
|-service/AccountManagerImpl.java
|-interceptor/LogInterceptor.java
|-interceptor/OnDeposit.java
|-src/main/resources/META-INF/persistence.xml
|-src/main/scripts
|-src/main/webapp/WEB-INF/web.xml
|-src/main/webapp/WEB-INF/beans.xml
|-src/main/webapp/css/bootstrap.css
|-src/main/webapp/index.xhtml
|-src/main/webapp/template.xhtml
```

To package the EJB archive, execute the following command from the `my-ejbapp` directory. This will produce `basicWebappEjb.war` inside the `target` directory. Now, you can deploy this WAR file into your Java EE application server, which supports EJB:

```
$ mvn package
```

JIRA plugins with the archetype plugin

JIRA is an issue-tracking system developed by Atlassian. It is quite popular among many open source projects. One of the extension points in JIRA is its plugins. Here, we will see how to generate a skeleton JIRA plugin using `jira-plugin-archetype` developed by Atlassian:

```
$ mvn archetype:generate -B
-DarchetypeGroupId=com.atlassian.maven.archetypes
-DarchetypeArtifactId=jira-plugin-archetype
-DarchetypeVersion=3.0.6
-DgroupId=com.packt.samples
-DartifactId=my-jira-plugin
-Dpackage=com.packt.samples.jira
-Dversion=1.0.0
-DarchetypeRepository=
http://repo.jfrog.org/artifactory/libs-releases/
```

This command will produce the following project template:

```
my-jira-plugin
|-pom.xml
|-README
```

```
| -LICENSE
| -src/main/java/com/packt/samples/jira/MyPlugin.java
| -src/main/resources/atlassian-plugin.xml
| -src/test/java/com/packt/samples/jira/MyPluginTest.java
| -src/test/java/it/MyPluginTest.java
| -src/test/resources/TEST_RESOURCES_README
| -src/test/xml/TEST_XML_RESOURCES_README
```

Spring MVC applications with the archetype plugin

Spring **model view controller** (MVC) is a web application framework developed under the Spring framework, which is an open source application framework and an inversion of the control container. Here, we will see how to generate a template Spring MVC application using the `spring-mvc-quickstart` archetype.



To know more about the Spring MVC framework, refer to <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html>.

Currently, the `spring-mvc-quickstart` archetype is not available in any of the public Maven repositories, so we have to download it from GitHub and build from the source, shown as follows:

```
$ git clone https://github.com/kolorobot/spring-mvc-quickstart-archetype.git
$ cd spring-mvc-quickstart-archetype
$ mvn clean install
```

Once the archetype is built from the source and is available in the local Maven repository, you can execute the following command to generate the template Spring MVC application:

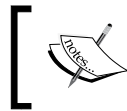
```
$ mvn archetype:generate -B
-DarchetypeGroupId=com.github.spring-mvc-archetypes
-DarchetypeArtifactId=spring-mvc-quickstart
-DarchetypeVersion=1.0.0-SNAPSHOT
-DgroupId=com.packt.samples
-DartifactId=my-spring-app
-Dpackage=com.packt.samples.spring
-Dversion=1.0.0
```

This will produce the following project template:

```
my-spring-app
| -pom.xml
| -src/main/java/com/packt/samples/spring/Application.java
| -src/main/webapp/WEB-INF/views
| -src/main/webapp/resources
| -src/main/resources
| -src/test/java/com/packt/samples/spring
| -src/test/resources
```

Let's see how to run the template Spring MVC application with the embedded Tomcat via Maven itself. Once the server is up, you can browse through the web application via `http://localhost:8080/my-spring-app`. The embedded Tomcat can be launched via the run goal of the tomcat7 plugin, shown as follows:

```
$ mvn test tomcat7:run
```



More details about the tomcat7 plugin are available at <http://tomcat.apache.org/maven-plugin-trunk/tomcat7-maven-plugin/>.

Summary

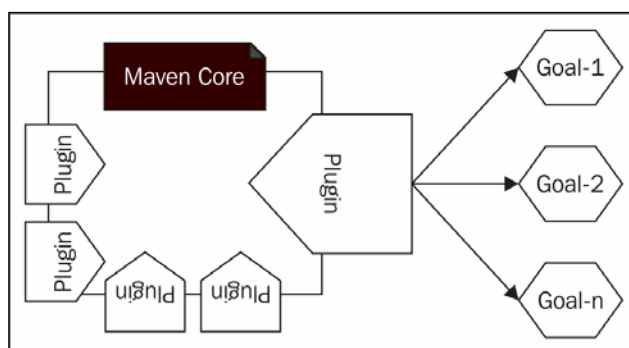
In this chapter, we focused on Maven archetypes. Maven archetypes provide a way of reducing repetitive work in building Maven projects. There are thousands of archetypes out there available publicly to assist you when building different types of projects. This chapter covered a commonly used set of archetypes.

In the next chapter, we will look into Maven plugins.


4

Maven Plugins

The roots of Maven go back to the *Jakarta Turbine* project, which was started as an attempt to simplify the build process of Jakarta Turbine. The beauty of Maven is its design. It does not try to do everything by itself, but rather delegates to a plugin framework. When you download Maven from its website, it's only the core framework, and the plugins are downloaded on demand. All the useful functionalities in the build process are developed as Maven plugins. You can also call Maven a plugin execution framework. The following figure shows the Maven plugins:




A Maven plugin can be executed on its own or can be executed as a part of a Maven lifecycle. We will discuss Maven lifecycles in *Chapter 5, Build Lifecycles*.

 A Maven build lifecycle consists of a set of well-defined phases. Each phase groups a set of goals defined by Maven plugins and the lifecycle defines the order of execution. Maven comes with three standard lifecycles: **default**, **clean**, and **site**. Each lifecycle defines its own set of phases.

Each plugin has its own set of goals, and each goal is responsible for performing a specific action. Let's see how to execute the `clean` goal of the Maven `clean` plugin. The `clean` goal will attempt to clean the working directory and the associated files created during the build:

```
$ mvn clean:clean
```

[ Maven plugins can be self-executed as `mvn plugin-prefix-name:goal-name`.]

The same `clean` plugin can be executed via the `clean` lifecycle, as shown in the following command:

```
$ mvn clean
```

The `clean` goal of the Maven `clean` plugin is associated with the `clean` phase of the `clean` lifecycle. The `clean` lifecycle defines three phases: `pre-clean`, `clean`, and `post-clean`. A phase in a lifecycle is just an ordered placeholder in the build execution path. For example, the `clean` phase in the `clean` lifecycle cannot do anything on its own. In the Maven architecture, it has two key elements: nouns and verbs. Both nouns and verbs, which are related to a given project, are defined in the POM file. The name of the project, the name of the parent project, the dependencies, and the type of packaging are nouns. Plugins bring verbs into the Maven build system, and they define what needs to be done during the build execution via its goals. A plugin is a group of goals. Each goal of a plugin can be executed on its own or can be registered as part of a phase in a Maven build lifecycle. One difference here is that when you execute a Maven plugin on its own, it only runs the goal specified in the command; however, when you run it as a part of a lifecycle, then Maven executes all the plugin goals associated with the corresponding lifecycle up until the specified phase (including that phase).

When you type `mvn clean`, it executes all the phases defined in the `clean` lifecycle up to and including the `clean` phase. Don't be confused; in this command, `clean` is not the name of the lifecycle, it's the name of a phase. It's only a coincidence that the name of the phase happens to be the name of the lifecycle. In Maven, you cannot simply execute a lifecycle by its name—it has to be the name of a phase. Maven will find the corresponding lifecycle and will execute all phases in it up to the given phase (including that phase).

In this chapter, we will be talking about the following topics:

- Commonly used Maven plugins and their usage
- Plugin discovery and execution process

Common Maven plugins

Maven plugins are mostly developed under the Apache Maven project itself, as well as under the Codehaus and Google Code projects. The following sections list out a set of commonly used Maven plugins and their usages.

The clean plugin

As discussed earlier, the `clean` plugin executes the `clean` goal of the Maven `clean` plugin to remove any of the working directories and other resources created during the build, as follows:

```
$ mvn clean:clean
```

The Maven `clean` plugin is also associated with the `clean` lifecycle. If you just execute `mvn clean`, the `clean` goal of the `clean` plugin will get executed.

You do not need to explicitly define the Maven `clean` plugin in your project POM file. Your project inherits it from the Maven super POM file. In *Chapter 2, Understanding the Project Object Model (POM)*, we discussed the Maven super POM file in detail. The following configuration in the super POM file associates the Maven `clean` plugin with all the Maven projects:

```
<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>
```



The Maven default lifecycle includes the phases: validate, initialize, generate-sources, process-sources, generateresources, process-resources, compile, process-classes, generate-test-sources, process-test-sources, generate-testresources, process-test-resources, test-compile, process-testclasses, test, prepare-package, package, pre-integration-test, integration-test, post-integration-test, verify, install, deploy.

By default, the `clean` goal of the `clean` plugin runs under the `clean` phase of the Maven `clean` lifecycle. If your project wants the `clean` plugin to run by default, then you can associate it with the `initialize` phase of the Maven default lifecycle. You can add the following configuration to your application POM file:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>2.5</version>
        <executions>
          <execution>
            <id>auto-clean</id>
            <phase>initialize</phase>
            <goals>
              <goal>clean</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Now, the `clean` goal of the `clean` plugin will get executed when you execute any of the phases in the Maven default lifecycle; there is no need to explicitly execute the `clean` phase of the `clean` lifecycle. For example, `mvn install` will run the `clean` goal in its `initialize` phase. This way, you can override the default behavior of the Maven `clean` plugin. A complete Maven sample project with the previous plugin configuration is available at <https://svn.wso2.org/repos/wso2/people/prabath/maven-mini/chapter04/jose>.

The compiler plugin

The compiler plugin is used to compile the source code. This has two goals: `compile` and `testCompile`. The `compile` goal is bound to the `compile` phase of the Maven default lifecycle. When you type `mvn clean install`, Maven will execute all the phases in the default lifecycle up to the `install` phase, which also includes the `compile` phase. This, in turn, will run the `compile` goal of the compiler plugin.

The following command shows how to execute the `compile` goal of the `compiler` plugin by itself. This will simply compile your source code:

```
$ mvn compiler:compile
```

All the Maven projects inherit the `compiler` plugin from the super POM file. As shown in the following configuration, the super POM defines the `compiler` plugin. It associates the `testCompile` and `compile` goals with the `test-compile` and `compile` phases of the Maven default lifecycle:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</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>
```

By default, the Maven `compiler` plugin assumes JDK 1.5 for both the `source` and `target` elements. JVM identifies the Java version of the source code via the `source` configuration parameter and the version of the compiled code via the `target` configuration parameter. If you want to break the assumption made by Maven and specify your own `source` and `target` versions, you need to override the `compiler` plugin configuration in your application POM file, as follows:

```
<project>
  [...]
  <build>
    [...]
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
            <source>1.7</source>
            <target>1.7</target>
        </configuration>
    </plugin>
</plugins>
[...]
```

```
</build>
[...]
```

You can pass any argument to the `compiler` plugin under the `compilerArgument` element, not just the `source` and `target` elements. This is more useful when the Maven `compiler` plugin does not have an element defined for the corresponding JVM argument. For example, the same `source` and `target` values can also be passed in the following manner:

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

```
<build>
[...]
```

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
      <compilerArgument>-source 1.7 -target 1.7</compilerArgument>
    </configuration>
  </plugin>
</plugins>
[...]
```

```
</build>
[...]
```

The install plugin

The `install` plugin will deploy the final project artifacts into the local Maven repository defined under the `localRepository` element of `MAVEN_HOME/conf/settings.xml`, where the default location is `USER_HOME/.m2/repository`. The `install` goal of the `install` plugin is bound to the `install` phase of the Maven default lifecycle. When you type `mvn clean install`, Maven will execute all the phases in the default lifecycle up to and including the `install` phase.

The following command shows how to execute the `install` goal of the `install` plugin by itself:

```
$ mvn install:install
```

All the Maven projects inherit the `install` plugin from the super POM file. As shown in the following configuration, the super POM defines the `install` plugin. It associates the `install` goal with the `install` phase of the Maven default lifecycle:

```
<plugin>
  <artifactId>maven-install-plugin</artifactId>
  <version>2.4</version>
  <executions>
    <execution>
      <id>default-install</id>
      <phase>install</phase>
      <goals>
        <goal>install</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The `install` goal of the `install` plugin does not have any configurations to be overridden at the project level.

The deploy plugin

The `deploy` plugin will deploy the final project artifacts into a remote Maven repository. The `deploy` goal of the `deploy` plugin is associated with the `deploy` phase of the default Maven lifecycle. To deploy an artifact via the default lifecycle, `mvn clean install` is not sufficient. It has to be `mvn clean deploy`. Any guesses why?

The `deploy` phase of the default Maven lifecycle comes after the `install` phase. Executing `mvn clean deploy` will execute all the phases of the default Maven lifecycle up to and including the `deploy` phase, which also includes the `install` phase. The following command shows how to execute the `deploy` goal of the `deploy` plugin by itself:

```
$ mvn deploy:deploy
```

All the Maven projects inherit the `deploy` plugin from the super POM file. As shown in the following configuration, super POM defines the `deploy` plugin. It associates the `deploy` goal with the `deploy` phase of the Maven default lifecycle:

```
<plugin>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.7</version>
  <executions>
    <execution>
      <id>default-deploy</id>
      <phase>deploy</phase>
      <goals>
        <goal>deploy</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Before executing either `mvn deploy:deploy` or `mvn deploy`, you need to set up the remote Maven repository details in your project POM file, under the `distributionManagement` section, as follows:

```
[...]
<distributionManagement>
  <repository>
    <id>wso2-maven2-repository</id>
    <name>WSO2 Maven2 Repository</name>
    <url>scp://dist.wso2.org/home/httpd/dist.wso2.org/
      maven2/</url>
  </repository>
</distributionManagement>
[...]
```

In this example, Maven connects to the remote repository via `scp`. **Secure Copy (scp)** defines a way of securely transferring files between two nodes in a computer network, which is built on top of popular SSH. To authenticate to the remote server, Maven provides two ways; one is based on a username and password, and the other one is based on SSH authentication keys. To configure username/password credentials against the Maven repository, we need to add the following `<server>` configuration element to `USER_HOME/.m2/settings.xml`. The value of the `id` element must carry the value of the remote repository hostname:

```
<server>
  <id>dist.wso2.org</id>
  <username>my_username</username>
  <password>my_password</password>
</server>
```

If the remote repository only supports SSH authentication keys, then we need to specify the location of the private key, as follows:

```
<server>
  <id>dist.wso2.org</id>
  <username>my_username</username>
  <privateKey>/path/to/private/key</privateKey>
</server>
```

The `deploy` goal of the `deploy` plugin does not have any configurations to be overridden at the project level.

The surefire plugin

The `surefire` plugin will run the unit tests associated with the project. The `test` goal of the `surefire` plugin is bound to the `test` phase of the default Maven lifecycle. When you type `mvn clean install`, Maven will execute all the phases in the default lifecycle up to and including the `install` phase, which also includes the `test` phase.

The following command shows how to execute the `test` goal of the `surefire` plugin:

```
$ mvn surefire:test
```

All the Maven projects inherit the `surefire` plugin from the super POM file. As shown in the following configuration, the super POM defines the `surefire` plugin. It associates the `test` goal with the `test` phase of the Maven default lifecycle:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
```

```
<version>2.12.4</version>
<executions>
  <execution>
    <id>default-test</id>
    <phase>test</phase>
    <goals>
      <goal>test</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

Since the surefire plugin is defined in the super POM file, you do not need to add it explicitly to your application POM file. However, you need to add a dependency to junit, as follows:

```
<dependencies>
  [...]
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
  </dependency>
  [...]
</dependencies>
```

The surefire plugin is not just coupled to JUnit, it can be used with other testing frameworks as well. If you are using TestNG, then you need to add a dependency to testng, as follows:

```
<dependencies>
  [...]
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.3.1</version>
    <scope>test</scope>
  </dependency>
  [...]
</dependencies>
```

The `surefire` plugin introduces a concept called test providers. You can specify a test provider within the plugin itself; if not, it will be derived from the dependency JAR file. For example, if you want to use the `junit47` provider, then within the plugin configuration, you can specify it as shown here. The `surefire` plugin supports, by default, four test providers: `surefire-junit3`, `surefire-junit4`, `surefire-junit47`, and `surefire-testng`:

```
<plugins>
[... ]
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.17</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven.surefire</groupId>
      <artifactId>surefire-junit47</artifactId>
      <version>2.17</version>
    </dependency>
  </dependencies>
</plugin>
[... ]
</plugins>
```

Since all the Maven projects inherit the `surefire` plugin from the super POM file, you do not need to override its configuration in the application POM file unless it's an absolute necessity. One of the reasons to override the parent configuration is to override the default test provider selection algorithm.

The site plugin

The `site` plugin generates static HTML web content for a Maven project, including the reports configured in the project. This defines eight goals, where each goal runs in one of the four phases defined in the Maven `site` lifecycle: `pre-site`, `site`, `post-site`, and `site-deploy`, which can be described as follows:

- `site:site`: This goal generates a site for a single Maven project
- `site:deploy`: This goal deploys the generated site via a Wagon supported protocol to the site URL specified in the `<distributionManagement>` section of the POM file
- `site:run`: This goal opens the site with the Jetty web server

- `site:stage`: This goal generates a site in a local staging or mock directory based on the site URL specified in the `<distributionManagement>` section of the POM file
- `site:stage-deploy`: This goal deploys the generated site to a staging or mock directory to the site URL specified in the `<distributionManagement>` section of the POM file
- `site:attach-descriptor`: This goal adds the site descriptor (`site.xml`) to the list of files to be installed/deployed
- `site:jar`: This goal bundles the site output into a JAR file so that it can be deployed to a repository
- `site:effective-site`: This goal calculates the effective site descriptor after inheritance and interpolation of `site.xml`

All the Maven projects inherit the `site` plugin from the super POM file. As shown in the following configuration, the super POM defines the `site` plugin. It associates the `site` and `deploy` goals with the `site` and `site-deploy` phases of the Maven default lifecycle:

```
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.3</version>
  <executions>
    <execution>
      <id>default-site</id>
      <phase>site</phase>
      <goals>
        <goal>site</goal>
      </goals>
      <configuration>
        <outputDirectory>
          PROJECT_HOME/target/site</outputDirectory>
        <reportPlugins>
          <reportPlugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>
              maven-project-info-reports-plugin
            </artifactId>
          </reportPlugin>
        </reportPlugins>
      </configuration>
    </execution>
    <execution>
      <id>default-deploy</id>
```

```

    <phase>site-deploy</phase>
    <goals>
      <goal>deploy</goal>
    </goals>
    <configuration>
      <outputDirectory>
        PROJECT_HOME/target/site</outputDirectory>
      <reportPlugins>
        <reportPlugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>
            maven-project-info-reports-plugin
          </artifactId>
        </reportPlugin>
      </reportPlugins>
    </configuration>
  </execution>
</executions>
<configuration>
  <outputDirectory>
    PROJECT_HOME/target/site</outputDirectory>
  <reportPlugins>
    <reportPlugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>
        maven-project-info-reports-plugin
      </artifactId>
    </reportPlugin>
  </reportPlugins>
</configuration>
</plugin>

```

As defined in the previous configuration, when you run `mvn site` or `mvn site:site`, the resultant HTML web content will be created inside the `target/site` directory under the project home. The `site` goal of the `site` plugin only generates the HTML web content; to deploy it, you need to use the `deploy` goal. To deploy `site` to a remote application server, you need to specify the remote machine details under the `distributionManagement` section of your application POM file, as follows:

```

<project>
  ...
  <distributionManagement>
    <site>
      <id>mycompany.com</id>
      <url>scp://mycompany/www/docs/project/</url>
    </site>
  </distributionManagement>

```

```
    </site>
  </distributionManagement>
  ...
</project>
```

To configure credentials to connect to the remote computer, you need to add the following `<server>` configuration element under the `<servers>` parent element of `USER_HOME/.m2/settings.xml`:

```
<server>
  <id>mycompany.com</id>
  <username>my_username</username>
  <password>my_password</password>
</server>
```

The generated site or the web content can be deployed to the remote location by executing the `deploy` goal of the Maven `site` plugin, as follows:

```
$ mvn site:deploy
```

In most of the cases, you do not need to override the `site` plugin configuration.

The jar plugin

The `jar` plugin creates a JAR file from your Maven project. The `jar` goal of the `jar` plugin is bound to the `package` phase of the Maven default lifecycle. When you type `mvn clean install`, Maven will execute all the phases in the default lifecycle up to and including the `install` phase, which also includes the `package` phase.

The following command shows how to execute the `jar` goal of the `jar` plugin:

```
$ mvn jar:jar
```

All the Maven projects inherit the `jar` plugin from the super POM file. As shown in the following configuration, the super POM defines the `jar` plugin. It associates the `jar` goal with the `package` phase of the Maven default lifecycle.

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.4</version>
  <executions>
    <execution>
      <id>default-jar</id>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```

    </execution>
  </executions>
</plugin>

```

In most of the cases, you do not need to override the `jar` plugin configuration, except in a case where you need to create a self-executable JAR file.



Creating a self-executable JAR file with `maven-jar-plugin` can be found at <http://maven.apache.org/shared/maven-archiver/examples/classpath.html>.

The source plugin

The `source` plugin creates a JAR file with the project source code. It defines five goals: `aggregate`, `jar`, `test-jar`, `jar-no-fork`, and `test-jar-no-fork`. All these five goals of the `source` plugin run under the `package` phase of the default lifecycle.

Unlike any of the plugins we discussed earlier, if you want to execute the `source` plugin with the Maven default lifecycle, it has to be defined in the project POM file, as shown here. The super POM file does not define the `source` plugin; it has to be defined within your Maven project itself:

```

<project>
...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <outputDirectory>
            /absolute/path/to/the/output/directory
          </outputDirectory>
          <finalName>filename-of-generated-jar-file</finalName>
          <attach>false</attach>
        </configuration>
      </plugin>
    </plugins>
  </build>
...
</project>

```

What is the difference between the `jar` plugin and the `source` plugin? Both create JAR files; however, the `jar` plugin creates a JAR file from the binary artifact, while the `source` plugin creates a JAR file from the source code. Small-scale open source projects use this approach to distribute the corresponding source code along with the binary artifacts.

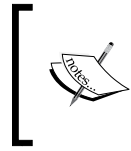
The resources plugin

The `resources` plugin copies the resources associated with the main project as well as the tests to the project output directory. The `resources` goal of the `resources` plugin copies the main resources into the main output directory, and it runs under the `process-resources` phase of the Maven default lifecycle. The `testResources` goal copies all the resources associated with the tests to the test output directory, and it runs under the `process-test-resources` phase of the Maven default lifecycle. The `copyResources` goal can be configured to copy any resource to the project output directory, and this is not bound to any of the phases in the Maven default lifecycle.

All the Maven projects inherit the `resources` plugin from the super POM file. As shown in the following configuration, super POM defines the `resources` plugin. It associates `resources` and `testResources` goals with the `process-resources` and `process-test-resources` phases of the Maven default lifecycle. When you type `mvn clean install`, Maven will execute all the phases in the default lifecycle up to and including the `install` phase, which also includes the `process-resources` and `process-test-resources` phases.

```
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <id>default-resources</id>
      <phase>process-resources</phase>
      <goals>
        <goal>resources</goal>
      </goals>
    </execution>
    <execution>
      <id>default-testResources</id>
      <phase>process-test-resources</phase>
      <goals>
        <goal>testResources</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

In most of the cases, you do not need to override the `resources` plugin configuration, unless you have a specific need to filter resources.



More details about resource filtering with `maven-resources-plugin` can be found at <http://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html>.

The release plugin

Releasing a project requires a lot of repetitive tasks. The objective of the Maven release plugin is to automate them. The `release` plugin defines the following eight goals, which are executed in two stages: preparing the release and performing the release:

- `release:clean`: This goal cleans up after a release preparation
- `release:prepare`: This goal prepares for a release in **Software Configuration Management (SCM)**
- `release:prepare-with-pom`: This goal prepares for a release in SCM, and it generates release POMs by fully resolving the dependencies
- `release:rollback`: This goal rolls back to a previous release
- `release:perform`: This goal performs a release from SCM
- `release:stage`: This goal performs a release from SCM into a staging folder/repository
- `release:branch`: This goal creates a branch of the current project with all the versions updated
- `release:update-versions`: This goal updates the versions in POM(s)

The preparation stage will complete the following tasks with the `release:prepare` goal:

- It verifies that all the changes in the source code are committed.
- It ensures that there are no SNAPSHOT dependencies. During the project development phase, we use SNAPSHOT dependencies; however, at the time of the release, all the dependencies should be changed to the latest released version of each dependency.
- The version of the project POM file will be changed from SNAPSHOT to a concrete version number.

- The SCM information in the project POM file will be changed to include the final destination of the tag.
- It execute all the tests against the modified POM files.
- It commits the modified POM files to SCM and tag the code with the version name.
- It changes the version in POM files in the trunk to a SNAPSHOT version and commits the modified POM files to the trunk.

Finally, the release will be performed with the `release:perform` goal. This will check out the code from the `release` tag in the SCM, and run a set of predefined goals: `site` and `deploy-site`.

The `maven-release-plugin` is not defined in the super POM file; it should be explicitly defined in your application POM file. The `releaseProfiles` configuration element defines the profiles to be released, and the `goals` configuration element defines the plugin goals to be executed during `release:perform`, as follows:

```
<plugin>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.5</version>
  <configuration>
    <releaseProfiles>release</releaseProfiles>
    <goals>deploy assembly:single</goals>
  </configuration>
</plugin>
```

Plugin discovery and execution

To associate a plugin with your Maven project, you have to either define it explicitly in your application POM file, or you should inherit it from a parent POM or the super POM file. Let's take a look at the Maven `jar` plugin. The `jar` plugin is defined by the super POM file, and all the Maven projects inherit it. To define a plugin (which is not inherited from the POM hierarchy), or associate a plugin with your Maven project, you must add the plugin configuration under the `build/plugins/plugin` element of your application `pom.xml`. In this way, you can associate any number of plugins with your project, as shown here:

```
<project>
...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
```

```

        <version>2.4</version>
        <executions>
            <execution>
                <id>default-jar</id>
                <phase>package</phase>
                <goals>
                    <goal>jar</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
...
</project>

```

In the Maven execution environment, what matters is not just your application POM file but the effective POM file. The effective POM file is constructed by the project POM file, any parent POM files, and the super POM file.

A Maven plugin can be executed in two ways:

- Using a lifecycle
- Directly invoking a plugin goal

If it is executed via a lifecycle, then there are plugin goals associated with different phases of the lifecycle. When each phase gets executed, all the plugin goals will also get executed only if the effective POM file of the project has defined the corresponding plugins under its plugins configuration. The same applies even when you try to invoke a plugin goal directly (for example, `mvn jar:jar`), the goal will be executed only if the corresponding plugin is associated with the project.

In either way, how does Maven find the plugin corresponding to the provided plugin goal?

Similar to any other dependency in Maven, a plugin is also uniquely identified by three coordinates: `groupId`, `artifactId`, and `version`. For plugins, however, you do not need to explicitly specify `groupId`. Maven assumes two `groupId` elements by default: `org.apache.maven.plugins` and `org.codehaus.mojo`. First, it will try to locate the plugin from `USER_HOME/.m2/repository/org/apache/maven/plugins`, and if that fails, it will locate it from `USER_HOME/.m2/repository/org/codehaus/mojo`.

In the previous sample plugin configuration, you may not find `groupId`. The `jar` plugin is available at `USER_HOME/.m2/repository/org/apache/maven/plugins/maven-jar-plugin`.

Maven also lets you add your own plugin groups, and they can be included in the plugin discovery. You can do it by updating `USER_HOME/.m2/settings.xml` or `MAVEN_HOME/conf/settings.xml`, as shown in the following manner:

```
<pluginGroups>
  <pluginGroup>com.packt.plugins</pluginGroup>
</pluginGroups>
```

Maven will always give priority to the previous configuration and then start looking for the well-known `groupId` elements: `org.apache.maven.plugins` and `org.codehaus.mojo`.

Let's take a look at some of the sample plugin configurations used in some popular open source projects.

Apache Felix provides a bundle plugin for Maven, which creates an OSGi bundle out of a Maven project. Another open source project, WSO2 Carbon, uses this bundle plugin in its development. You can find a sample POM file, which consumes the plugin at <https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/service-stubs/org.wso2.carbon.qpid.stub/4.2.0/pom.xml>. This is a custom plugin, which does not fall into any of the `groupId` elements known to Maven by default. In that case, anyone who uses the plugin must qualify the plugin with `groupId`, or they must add the corresponding `groupId` element to the `pluginGroups` configuration element, as discussed earlier.

The following code shows the plugin configuration from the WSO2 Carbon project:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-SymbolicName>
        ${project.artifactId}</Bundle-SymbolicName>
      <Bundle-Name>${project.artifactId}</Bundle-Name>
      <Carbon-Component>UIBundle</Carbon-Component>
      <Import-Package>
        org.apache.axis2.*;
        version="${axis2.osgi.version.range}",
        org.apache.axiom.*;
        version="${axiom.osgi.version.range}",
        *;resolution:=optional
      </Import-Package>
      <Export-Package>
```

```

        org.wso2.carbon.qpid.stub.*;
        version="${carbon.platform.package.export.version}",
    </Export-Package>
</instructions>
</configuration>
</plugin>

```

Plugin management

If you take a look at the previous configuration carefully, you do not see a version for the bundle plugin. This is where the `pluginManagement` element comes into play. With the `pluginManagement` configuration element, you can avoid repetitive usage of the plugin version. Once you define a plugin under `pluginManagement`, all the child POM files will inherit that configuration.

The WSO2 Carbon project defines all the plugins used by its child projects under the `pluginManagement` section of <https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/parent/pom.xml>, and all the projects inherit it. A truncated part of the configuration is as follows:

```

<pluginManagement>
  <plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>2.3.5</version>
    <extensions>true</extensions>
  </plugin>
</pluginManagement>

```



We'll discuss plugin management in detail in *Chapter 7, Best Practices*.

Plugin repositories

Maven downloads plugins on demand when it cannot find a plugin in its local repository. By default, Maven looks for any plugin that is not available locally in the Maven plugin repository defined by the super POM file (this is the default behavior; you can also define plugin repositories in your application POM file). The following code snippets shows how to define plugin repositories:


```

<pluginRepositories>
  <pluginRepository>
    <id>central</id>

```

```
<name>Maven Plugin Repository</name>
<url>http://repo1.maven.org/maven2</url>
<layout>default</layout>
<snapshots>
  <enabled>false</enabled>
</snapshots>
<releases>
  <updatePolicy>never</updatePolicy>
</releases>
</pluginRepository>
</pluginRepositories>
```

If you develop a custom plugin, just like the Apache Felix bundle plugin, you must make it available for the rest via a plugin repository, and any other consumer of that plugin, such as the WSO2 Carbon project, must define the corresponding plugin repository in its POM file or in a parent POM file.



The WSO2 Carbon project defines two plugin repositories in its parent POM file at <https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/parent/pom.xml>.

The Apache Felix bundle plugin is available at <http://dist.wso2.org/maven2/org/apache/felix/maven-bundle-plugin/>.

The following configuration is a part of the WSO2 Carbon project `parent/pom.xml`, which defines the two plugin repositories:

```
<pluginRepositories>
  <pluginRepository>
    <id>wso2-maven2-repository-1</id>
    <url>http://dist.wso2.org/maven2</url>
  </pluginRepository>
  <pluginRepository>
    <id>wso2-maven2-repository-2</id>
    <url>http://dist.wso2.org/snapshots/maven2</url>
  </pluginRepository>
</pluginRepositories>
```

Plugin as an extension

If you look at the definition of the Apache Felix bundle plugin, you might have noticed the `extensions` configuration element, which is set to `true`:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
</plugin>
```

As we discussed earlier, the goal of the `bundle` plugin is to build an OSGi bundle from a Maven project. In other words, the Apache Felix `bundle` plugin introduces a new packaging type with an existing file extension, `.jar`. If you look at the POM file of the WSO2 Carbon project, which consumes the `bundle` plugin, you can see the packaging of the project is set to `bundle` (<https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/service-stubs/org.wso2.carbon.qpid.stub/4.2.0/pom.xml>), as follows:

```
<packaging>bundle</packaging>
```

If you are associating a plugin with your project, which introduces a new packaging type or a customized lifecycle, then you must set the value of the `extensions` configuration element to `true`. Once this is done, the Maven engine will go further and will look for the `components.xml` file inside `META-INF/plexus` of the corresponding `jar` plugin.

Summary

In this chapter, we focused on Maven plugins. Maven only provides a build framework while the Maven plugins perform the actual tasks. Maven has a large, rich set of plugins, and the chances that you have to write your own custom plugin are very slim. This chapter covered some of the most commonly used Maven plugins, and later explained how plugins are discovered and executed. If you would like to know about custom plugin development, refer to the book, *Mastering Apache Maven 3* by Packt Publishing.

In the next chapter, we will focus on Maven build lifecycle.

5

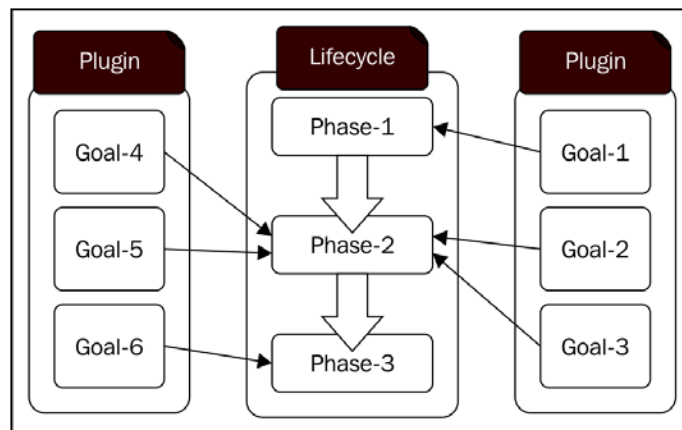
Build Lifecycles

A Maven build lifecycle consists of a set of well-defined phases. Each phase groups a set of goals defined by Maven plugins, and the lifecycle defines the order of execution. A Maven plugin is a collection of goals where each goal is responsible for performing a specific action. We discussed Maven plugins in detail in *Chapter 4, Maven Plugins*.

In this chapter, the following topics will be covered:

- Standard lifecycles in Maven
- Lifecycle bindings
- Building custom lifecycle extensions

The following figure shows the relationship between Maven plugin goals and lifecycle phases:



Let's take the simplest Maven build command that every Java developer is familiar with:

```
$ mvn clean install
```

What will this do? As a developer, how many times have you executed the previous command? Have you ever thought of what happens inside? If not, it's time to explore it now.

Standard lifecycles in Maven

Maven comes with three standard lifecycles:

- `clean`
- `default`
- `site`

Each lifecycle defines its own set of phases.

The clean lifecycle

The `clean` lifecycle defines three phases: `pre-clean`, `clean`, and `post-clean`. A phase in a lifecycle is just an ordered placeholder in the build execution path. For example, the `clean` phase in the `clean` lifecycle cannot do anything on its own. In the Maven architecture, it has two key elements: nouns and verbs. Both nouns and verbs, which are related to a given project, are defined in the POM file. The name of the project, the name of the parent project, the dependencies, and the type of packaging are nouns. Plugins bring verbs into the Maven build system, and they define what needs to be done during the build execution via its goals. A plugin is a group of goals. Each goal of a plugin can be executed on its own or can be registered as part of a phase in a Maven build lifecycle.

When you type `mvn clean`, it executes all the phases defined in the `clean` lifecycle up to and including the `clean` phase. Don't be confused; in this command, `clean` is not the name of the lifecycle it's the name of a phase. It's only a coincidence that the name of the phase happens to be the name of the lifecycle. In Maven, you cannot simply execute a lifecycle by its name—it has to be the name of a phase. Maven will find the corresponding lifecycle and will execute all phases in it up to the given phase (including that phase).

When you type `mvn clean`, it cleans out project's working directory (by default, it's the target directory). This is done via the Maven `clean` plugin. To find more details about the Maven `clean` plugin, type the following command. It describes all the goals defined inside the `clean` plugin:

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

Name: Maven Clean Plugin

Description: The Maven Clean Plugin is a plugin that removes files generated at build-time in a project's directory.

Group Id: org.apache.maven.plugins

Artifact Id: maven-clean-plugin

Version: 2.5

Goal Prefix: clean

This plugin has 2 goals.

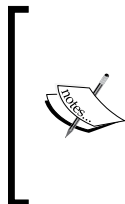
`clean:clean`

Description: Goal, which cleans the build. This attempts to clean a project's working directory of the files that were generated at build-time. By default, it discovers and deletes the directories configured in `project.build.directory`, `project.build.outputDirectory`, `project.build.testOutputDirectory`, and `project.reporting.outputDirectory`. Files outside the default may also be included in the deletion by configuring the `filesets` tag.

`clean:help`

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

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



Everything in Maven is a plugin. Even the command we executed previously to get goal details of the `clean` plugin executes another plugin—the `help` plugin. The following command will describe the `help` plugin:

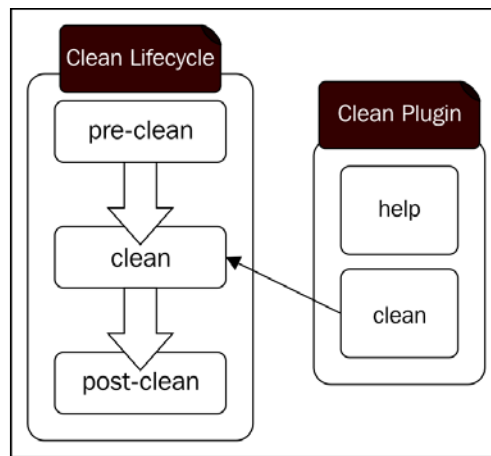
```
$ mvn help:describe -Dplugin=help
```

`describe` is a goal defined inside the `help` plugin.

The `clean` plugin has two goals defined in it: `clean` and `help`. As mentioned previously, each goal of a plugin can be executed on its own or can be registered as part of a phase in a Maven build lifecycle. The `clean` goal of the `clean` plugin can be executed on its own with the following command:

```
$ mvn clean:clean
```

The following figure shows the relationship between the Maven `clean` plugin goals and the `clean` lifecycle phases:



The first instance of the `clean` word in the previous command is the prefix of the `clean` plugin, while the second one is the name of the goal. When you type `mvn clean`, it's the same `clean` goal that gets executed. However, this time, it gets executed through the `clean` phase of the `clean` lifecycle, and it also executes all the phases in the corresponding lifecycle up to, and including, the `clean` phase—not just the `clean` phase. The `clean` goal of the `clean` plugin is configured by default to get executed during the `clean` phase of the `clean` lifecycle. The plugin goal to lifecycle phase mapping can be provided through the application POM file; if not, it will be inherited from the super POM file. The super POM file, which defines the `clean` plugin by default, adds the plugin to the `clean` phase of the `clean` lifecycle. You cannot define a phase with the same name in two different lifecycles.

The following code snippet shows how the `clean` goal of the Maven `clean` plugin is associated with the `clean` phase of the `clean` lifecycle:

```
<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>
```

The `pre-clean` and `post-clean` phases of the `clean` lifecycle do not have any plugin bindings. The objective of the `pre-clean` phase is to perform any operations prior to the cleaning task and the objective of the `post-clean` phase is to perform any operations after the cleaning task. If you need to associate any plugins with these two phases, you simply need to add them to the corresponding plugin configuration.


The default lifecycle

The default lifecycle in Maven defines 23 phases. When you run the `mvn clean install` command, it will execute all the phases from the default lifecycle up to, and including, the `install` phase. To be precise, Maven will first execute all the phases in the `clean` lifecycle up to, and including, the `clean` phase, and it will then execute the default lifecycle up to, and including, the `install` phase.

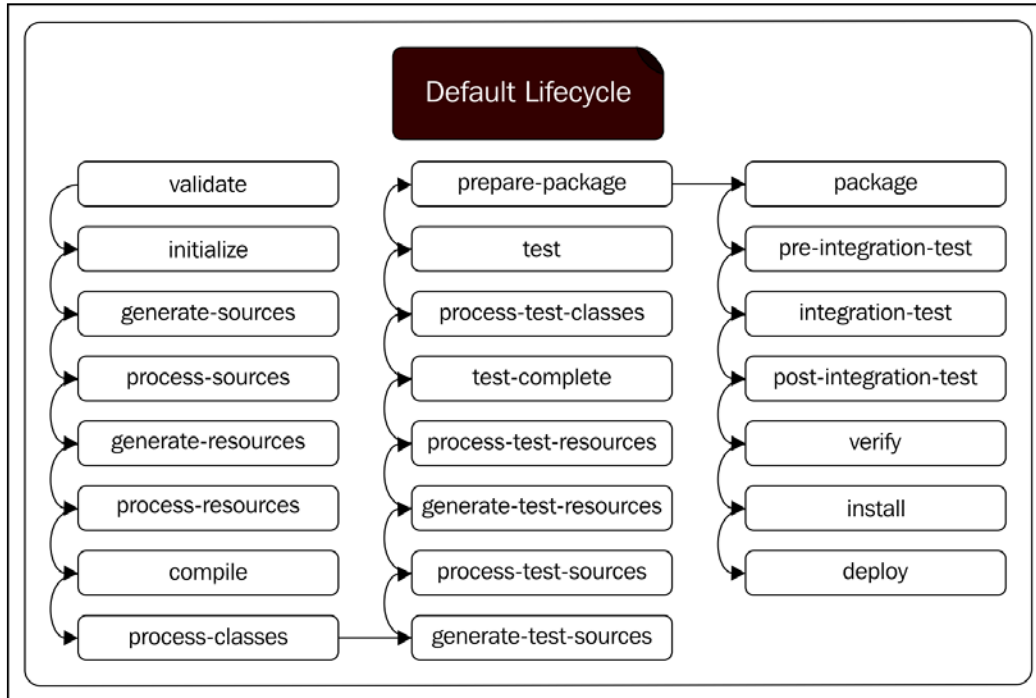
The phases in the default lifecycle do not have any associated plugin goals. The plugin bindings for each phase are defined by the corresponding packaging (that is, `jar` or `war`). If the type of packaging of your Maven project is `jar`, then it will define its own set of plugins for each phase. If the packaging type is `war`, then it will have its own set of plugins. The following points summarize all the phases defined under the default lifecycle in their order of execution:


- `validate`: This phase validates the project POM file and ensures that all the necessary information related to carrying out the build is available.
- `initialize`: This phase initializes the build by setting up the right directory structure and initializing properties.
- `generate-sources`: This phase generates any required source code.
- `process-sources`: This phase processes the generated source code; for example, there can be a plugin running in this phase to filter the source code based on some defined criteria.
- `generate-resources`: This phase generates any resources that need to be packaged with the final artifact.

- `process-resources`: This phase processes the generated resources. It copies the resources to their destination directories and makes them ready for packaging.
- `compile`: This phase compiles the source code.
- `process-classes`: This phase can be used to carry out any bytecode enhancements after the `compile` phase.
- `generate-test-sources`: This phase generates the required source code for tests.
- `process-test-sources`: This phase processes the generated test source code; for example, there can be a plugin running in this phase to filter the source code based on some defined criteria.
- `generate-test-resources`: This phase generates all the resources required to run tests.
- `process-test-resources`: This phase processes the generated test resources. It copies the resources to their destination directories and makes them ready for testing.
- `test-compile`: This phase compiles the source code for tests.
- `process-test-classes`: This phase can be used to carry out any bytecode enhancements after the `test-compile` phase.
- `test`: This phase executes tests using the appropriate unit test framework.
- `prepare-package`: This phase is useful in organizing the artifacts to be packaged.
- `package`: This phase packs the artifacts into a distributable format, for example, JAR or WAR.
- `pre-integration-test`: This phase performs the actions required (if any) before running integration tests. This may be used to start any external application servers and deploy the artifacts into different test environments.
- `integration-test`: This phase runs integration tests.
- `post-integration-test`: This phase can be used to perform any cleanup tasks after running the integration tests.
- `verify`: This phase verifies the validity of the package. The criteria to check the validity needs to be defined by the respective plugins.
- `install`: This phase installs the final artifact in the local repository.
- `deploy`: This phase deploys the final artifact to a remote repository.

 The packaging type of a given Maven project is defined under the `<packaging>` element in the `pom.xml` file. If the element is omitted, then Maven assumes it as a `jar` packaging.

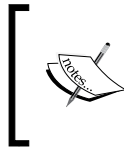
The following figure shows all the phases defined under the Maven default lifecycle and their order of execution:



 More details about Maven lifecycles can be found at <http://maven.apache.org/ref/3.3.3/maven-core/lifecycles.html>.

Let's take a look at a concrete example. Run the following command against a Maven project having the `jar` packaging:

```
$ mvn help:describe -Dcmd=deploy
```



If you do not have such a project, you can download a sample Maven project from <https://svn.wso2.org/repos/wso2/people/prabath/maven-mini/chapter05/jose/>.

Here, we are using the Maven help plugin to find more details about the deploy phase corresponding to the jar packaging, and it will produce the following output:

It is a part of the lifecycle for the POM packaging 'jar'. This lifecycle includes the following phases:

```
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: org.apache.maven.plugins:maven-resources-
  plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-
  plugin:2.5.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: org.apache.maven.plugins:maven-
  resources-plugin:2.6:testResources
* test-compile: org.apache.maven.plugins:maven-compiler-
  plugin:2.5.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar
* pre-integration-test: Not defined
* integration-test: Not defined
* post-integration-test: Not defined
* verify: Not defined
* install: org.apache.maven.plugins:maven-install-
  plugin:2.4:install
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy
```

The output lists out all the Maven plugins registered against different phases of the default lifecycle for the jar packaging. The jar goal of maven-jar-plugin is registered against the package phase, while the install goal of maven-install-plugin is registered in the install phase.

Let's run the previous command against a POM file having the war packaging. It produces the following output:

It is a part of the lifecycle for the POM packaging 'war'. This life includes the following phases:

```
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: org.apache.maven.plugins:maven-resources-
  plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-
  plugin:2.5.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: org.apache.maven.plugins:maven-resources-
  plugin:2.6:testResources
* test-compile: org.apache.maven.plugins:maven-compiler-
  plugin:2.5.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-war-plugin:2.2:war
* pre-integration-test: Not defined
* integration-test: Not defined
* post-integration-test: Not defined
* verify: Not defined
* install: org.apache.maven.plugins:maven-install-plugin:2.4:install
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy
```

Now, if you look at the package phase, you will notice that we have a different plugin goal: the war goal of maven-war-plugin.

Similar to the `jar` and `war` packaging, each of the other packaging type defines its own binding for the default lifecycle.

The site lifecycle

The `site` lifecycle is defined with four phases: `pre-site`, `site`, `post-site`, and `site-deploy`. The `site` lifecycle has no value without the Maven `site` plugin. The `site` plugin is used to generate static HTML content for a project. The generated HTML content will also include appropriate reports corresponding to the project. The `site` plugin defines eight goals, and two of them are directly associated with the phases in the `site` lifecycle.

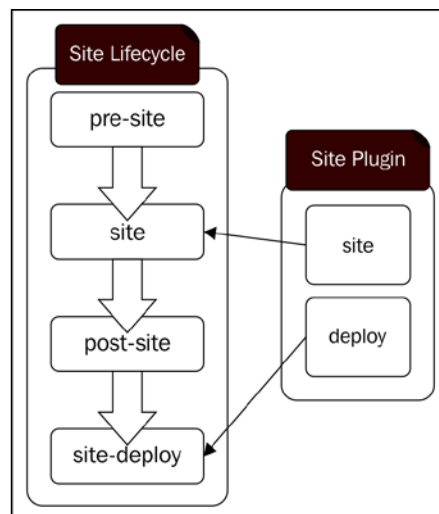
Let's run the following command against a POM file to describe the site goal:

```
$ mvn help:describe -Dcmd=site
```

As shown in the following output, the `site` goal of the `site` plugin is associated with the `site` phase, while the `deploy` goal of the `site` plugin is associated with the `site-deploy` phase:

```
[INFO] 'site' is a lifecycle with the following phases:  
* pre-site: Not defined  
* site: org.apache.maven.plugins:maven-site-plugin:3.3:site  
* post-site: Not defined  
* site-deploy: org.apache.maven.plugins:maven-site-plugin:3.3:deploy
```

The following figure shows the relationship between the Maven `site` plugin goals and the `site` lifecycle phases:



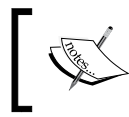
Lifecycle bindings

Under the discussion of the default lifecycle, we briefly touched upon the concept of lifecycle bindings. The default lifecycle is defined without any associated lifecycle bindings, while both the `clean` and `site` lifecycles are defined with bindings. The standard Maven lifecycles and their associated bindings are defined under the file `META-INF/plex/components.xml` of `MAVEN_HOME/lib/maven-core-3.3.3.jar`.

Here is the configuration for the default lifecycle without any associated plugin bindings:

```
<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>
```


The `components.xml` file, which is also known as the **component descriptor**, describes the properties required by Maven to manage the lifecycle of a Maven project. The `role` element specifies the Java interface exposed by this lifecycle component and defines the type of the component. All the lifecycle components must have `org.apache.maven.lifecycle.Lifecycle` as the role. The `implementation` tag specifies the concrete implementation of the interface. The identity of a component is defined by the combination of the role and the `role-hint` elements. The `role-hint` element is not a mandatory element; however, if we have multiple elements of the same type, then we must define a `role-hint` element. Corresponding to Maven lifecycles, the name of the lifecycle is set as the value of the `role-hint` element.



Maven uses `components.xml` to define more other components than Maven lifecycles. Based on the type of the component, the value of the `role` element is set.

The `clean` lifecycle is defined with an associated plugin binding to the `clean` goal of `maven-clean-plugin`. The plugin binding is defined under the element `default-phases`. The configuration is as follows:

```
<component>
  <role>org.apache.maven.lifecycle.Lifecycle</role>
  <implementation>
    org.apache.maven.lifecycle.Lifecycle
  </implementation>
  <role-hint>clean</role-hint>
  <configuration>
    <id>clean</id>
    <phases>
      <phase>pre-clean</phase>
      <phase>clean</phase>
      <phase>post-clean</phase>
    </phases>
    <default-phases>
      <clean>
        org.apache.maven.plugins:maven-clean-plugin:2.4.1:clean
      </clean>
    </default-phases>
  </configuration>
</component>
```

The site lifecycle is defined with associated plugin bindings to the site and site-deploy goals of maven-site-plugin. The plugin bindings are defined under the default-phases element, with the following configuration:

```
<component>
  <role>org.apache.maven.lifecycle.Lifecycle</role>
  <implementation>
    org.apache.maven.lifecycle.Lifecycle
  </implementation>
  <role-hint>site</role-hint>
  <configuration>
    <id>site</id>
    <phases>
      <phase>pre-site</phase>
      <phase>site</phase>
      <phase>post-site</phase>
      <phase>site-deploy</phase>
    </phases>
    <default-phases>
      <site>
        org.apache.maven.plugins:maven-site-plugin:2.0.1:site
      </site>
      <site-deploy>
        org.apache.maven.plugins:maven-site-plugin:2.0.1:deploy
      </site-deploy>
    </default-phases>
  </configuration>
</component>
```

Finally, let's take a look at how the jar plugin binding for the default lifecycle is defined. The following component element defines a plugin binding to an existing lifecycle. The associated lifecycle is defined under the configuration/lifecycles/lifecycle/id element:

```
<component>
  <role>
    org.apache.maven.lifecycle.mapping.LifecycleMapping
  </role>
  <role-hint>jar</role-hint>
  <implementation>org.apache.maven.lifecycle.mapping.
DefaultLifecycleMapping
  </implementation>
  <configuration>
```

```
    <lifecycles>
      <lifecycle>
        <id>default</id>
        <phases>
          <process-resources>
            org.apache.maven.plugins:maven-resources-
plugin:2.4.3:resources
          </process-resources>
          <compile>
            org.apache.maven.plugins:maven-compiler-
plugin:2.3.2:compile
          </compile>
          <process-test-resources>
            org.apache.maven.plugins:maven-resources-
plugin:2.4.3:testResources
          </process-test-resources>
          <test-compile>
            org.apache.maven.plugins:maven-compiler-
plugin:2.3.2:testCompile
          </test-compile>
          <test>
            org.apache.maven.plugins:maven-surefire-plugin:2.5:test
          </test>
          <package>
            org.apache.maven.plugins:maven-jar-plugin:2.3.1:jar
          </package>
          <install>
            org.apache.maven.plugins:maven-install-
plugin:2.3.1:install
          </install>
          <deploy>
            org.apache.maven.plugins:maven-deploy-plugin:2.5:deploy
          </deploy>
        </phases>
      </lifecycle>
    </lifecycles>
  </configuration>
</component>
```

Lifecycle extensions

The lifecycle extensions in Maven allow you to customize the standard build behavior. Let's take a look at the `org.apache.maven.AbstractMavenLifecycleParticipant` class. A custom lifecycle extension should extend from the `AbstractMavenLifecycleParticipant` class, which provides the following three methods that you can override:

- `afterProjectsRead(MavenSession session)`: This method is invoked after all the Maven project instances have been created. There will be one project instance for each POM file. In a large-scale build system, you have one parent POM and it points to multiple child POM files.
- `afterSessionEnd(MavenSession session)`: This method is invoked after all Maven projects are built.
- `afterSessionStart(MavenSession session)`: This method is invoked after the `MavenSession` instance is created.

Let's try out the following example:

```
package com.packt.lifecycle.ext;

import org.apache.maven.AbstractMavenLifecycleParticipant;
import org.apache.maven.MavenExecutionException;
import org.apache.maven.execution.MavenSession;
import org.codehaus.plexus.component.annotations.Component;

@Component(role = AbstractMavenLifecycleParticipant.class, hint
    ="packt")
public class PACKTLifeCycleExtension extends
    AbstractMavenLifecycleParticipant {

    @Override
    public void afterProjectsRead(MavenSession session) {

        System.out.println("All Maven project instances are created.");
        System.out.println("Offline building: " + session.isOffline());
    }

    @Override
    public void afterSessionEnd(MavenSession session)
        throws MavenExecutionException {
        System.out.println("All Maven projects are built.");
    }
}
```

The previous code can be built with the following application POM file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>com.packt.lifecycle.ext</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-compat</artifactId>
      <version>3.2.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-core</artifactId>
      <version>3.2.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.plexus</groupId>
        <artifactId>plexus-component-metadata</artifactId>
        <version>1.5.5</version>
        <executions>
          <execution>
            <goals>
              <goal>generate-metadata</goal>
              <goal>generate-test-metadata</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Here, in the POM file, we use the `plexus-component-metadata` plugin to generate the Plexus descriptor from the source tags and class annotations.

Once the extension project is built successfully with `mvn clean install`, we need to incorporate the extension to other Maven builds. You can do it in two ways; one is by adding it to the project POM as an extension, as shown in the following code:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>
    com.packt.lifecycle.ext.sample.project
  </artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <name>Custom Lifecycle Extension Project</name>

  <build>
    <extensions>
      <extension>
        <groupId>com.packt</groupId>
        <artifactId>com.packt.lifecycle.ext</artifactId>
        <version>1.0.0</version>
      </extension>
    </extensions>
  </build>
</project>
```

Now, you can build the sample project with `mvn clean install`. It will produce the following output:

```
[INFO] Scanning for projects...
```

```
All Maven project instances are created.
```

```
Offline building: false
```

```
[INFO] -----
```

```
[INFO] BUILD SUCCESS
```

```
[INFO] -----
```

```
[INFO] Total time: 1.328 s
```

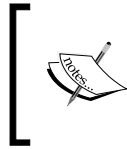
```
[INFO] Finished at: 2014-07-29T11:29:52+05:30
```

```
[INFO] Final Memory: 6M/81M
```

```
[INFO] -----
```

```
All Maven projects are built.
```

If you want to execute this extension for all your Maven projects without changing each and every POM file, then you need to add the lifecycle extension JAR file to `MAVEN_HOME/lib/ext`.



The complete source code corresponding to the lifecycle extension project can be downloaded from <https://svn.wso2.org/repos/wso2/people/prabath/maven-mini/chapter05/>.

Summary

In this chapter, we focused on Maven lifecycles and explained how the three standard lifecycles work and how we can customize them. Later in the chapter, we discussed how to develop our own lifecycle extensions.

In the next chapter, we will discuss how to build Maven assemblies.

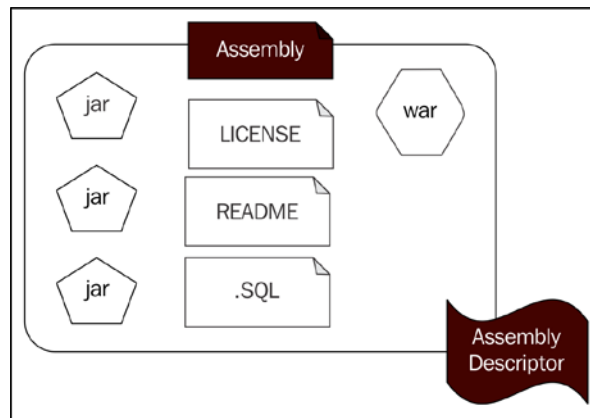
6

Maven Assemblies

Maven provides an extensible architecture via plugins and lifecycles. Archive types such as `.jar`, `.war`, `.ear`, and many more are supported by plugins and associated lifecycles. The JAR plugin creates an artifact with the `.jar` extension and the relevant metadata, according to the JAR specification. The JAR file is, in fact, a ZIP file with the optional `META-INF` directory. You can find more details about the JAR specification from <http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html>.

The JAR file aggregates a set of class files to build a single distribution unit. The WAR file aggregates a set of JAR files, Java classes, JSPs, images, and many more resources into a single distribution unit that can be deployed in a Java EE application server. However, when you build a product, you may need to aggregate many JAR files from different places, WAR files, README files, LICENSE files, and many more into a single ZIP file. To build such an archive, we can use the Maven assembly plugin.

The following figure shows the possible contents of a Maven assembly:



In this chapter, we will discuss the following topics:

- Maven assembly plugin
- Assembly descriptor
- Artifact/resource filters
- End-to-end example to build a custom distribution archive

The Maven assembly plugin produces a custom archive, which adheres to a user-defined layout. This custom archive is also known as the Maven assembly. In other words, a Maven assembly is a distribution unit, which is built according to a custom layout.

The assembly plugin

Let's take a quick look at a real-world example, which uses the assembly plugin.

WSO2 Identity Server (WSO2 IS) is an open source identity and entitlement management product distributed under the Apache 2.0 license as a ZIP file. The ZIP distribution is assembled using the Maven assembly plugin. Let's take a look at the root POM file of the distribution module of WSO2 IS, which builds the Identity Server distribution, available at <https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/products/is/5.0.0/modules/distribution/pom.xml>.

First, pay attention to the plugins section of the POM file. Here, you can see that `maven-assembly-plugin` is associated with the project. Inside the plugin configuration, you can define any number of executions with the `execution` element, which is a child element of the `executions` element, which has the following configuration:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <id>copy_components</id>
      <phase>test</phase>
      <goals>
        <goal>attached</goal>
      </goals>
      <configuration>
        <filters>
          <filter>
            ${basedir}/src/assembly/filter.properties
          </filter>
        </filters>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```

        </filter>
      </filters>
    <descriptors>
      <descriptor>src/assembly/dist.xml</descriptor>
    </descriptors>
  </configuration>
</execution>
<execution>
  <id>dist</id>
  <phase>package</phase>
  <goals>
    <goal>attached</goal>
  </goals>
  <configuration>
    <filters>
      <filter>
        ${basedir}/src/assembly/filter.properties
      </filter>
    </filters>
    <descriptors>
      <descriptor>src/assembly/bin.xml</descriptor>
      <descriptor>src/assembly/src.xml</descriptor>
      <descriptor>src/assembly/docs.xml</descriptor>
    </descriptors>
  </configuration>
</execution>
</executions>
</plugin>

```

If you look at the first execution element, it associates the attached goal of the assembly plugin with the test phase of the default lifecycle. In the same manner, the second execution element associates the attached goal with the package phase of the default lifecycle.

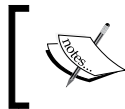


The Maven default lifecycle includes: validate, initialize, generate-sources, process-sources, generate-resources, process-resources, compile, process-classes, generate-test-sources, process-test-sources, generate-test-resources, process-test-resources, test-compile, process-test-classes, test, prepare-package, package, pre-integration-test, integration-test, post-integration-test, verify, install, deploy.

Everything inside the configuration element is plugin specific. In this case, the Maven assembly plugin knows how to process the filters and descriptors elements.

In this particular example, only the attached goal of the assembly plugin is used. The assembly plugin introduces eight goals; however, six of them are deprecated, including the attached goal. It is not recommended to use any of the deprecated goals. Later, we'll see how to use the single goal of the assembly plugin instead of the deprecated attached goal. The following lists out the six deprecated goals of the assembly plugin. In case you are using any of them, you should migrate your project to use the single goal, except for the fourth one, the unpack goal. For that, you need to use the unpack goal of the Maven dependency plugin. The following lists out the six deprecated goals of the assembly plugin:

- assembly:assembly
- assembly:attached
- assembly:directory
- assembly:unpack
- assembly:directory-single
- assembly:directory-inline



More details about the Maven assembly plugin and its goals can be found at <http://maven.apache.org/plugins/maven-assembly-plugin/plugin-info.html>.

The assembly descriptor

The assembly descriptor is an XML-based configuration, which defines how to build an assembly and how its content should be structured.

If we go back to our previous example, the attached goal of the assembly plugin creates a binary distribution according to the assembly descriptor, both in the test and the package phases of the default Maven lifecycle. The assembly descriptors for each phase can be specified under the descriptors element. As in the case of this particular example, there are multiple descriptor elements defined under the descriptors parent element. For the package phase, it has three assembly descriptors, as shown in the following configuration:

```
<descriptors>
  <descriptor>src/assembly/bin.xml</descriptor>
  <descriptor>src/assembly/src.xml</descriptor>
  <descriptor>src/assembly/docs.xml</descriptor>
</descriptors>
```

Each descriptor element instructs the assembly plugin from where to load the descriptor, and each descriptor file will be executed sequentially in the defined order.

Let's take a look at the `src/assembly/bin.xml` file, which is shown here:

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



The file path in the descriptor element is given relative to the root POM file under the distribution module. You can find the complete `bin.xml` file at <https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/products/is/5.0.0/modules/distribution/src/assembly/bin.xml>.

The value of the `format` element specifies the ultimate type of the artifact to be produced. It can be `zip`, `tar`, `tar.gz`, `tar.bz2`, `jar`, `dir`, or `war`. You can use the same assembly descriptor to create multiple formats. In that case, you need to include multiple `format` elements under the `formats` parent element.



Even though you can specify the format of the assembly in the assembly descriptor, it is recommended to do it via the plugin configuration. In the plugin configuration, you can define different formats for your assembly, as shown in the following block of code. The benefit here is that you can have multiple Maven profiles to build different archive types. We will discuss Maven profiles in *Chapter 7, Best Practices*:

```
<plugin>
  <executions>
    <execution>
      <configuration>
        <formats>
          <format>zip</format>
        </formats>
      </configuration>
    </execution>
  </executions>
</plugin>
```

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

When the value of the `includeBaseDirectory` element is set to `false`, the artifact will be created with no base directory. If this is set to `true`, which is the default value, the artifact will be created under the base directory. You can specify a value for the base directory under the `baseDirectory` element. In most cases, the value of `includeBaseDirectory` is set to `false` so that the final distribution unit directly packs all the artifacts right under it, without having another root directory:

```
<fileSets>
  <fileSet>
    <directory>target/wso2carbon-core-4.2.0</directory>
    <outputDirectory>wso2is-${pom.version}</outputDirectory>
    <excludes>
      <exclude>**/*.sh</exclude>
```

Each `fileSet` element under the `fileSets` parent element specifies the set of files to be assembled to build the final archive. The first `fileSet` element instructs to copy all the content from `directory` (which is `target/wso2carbon-core-4.2.0`) to the `outputDirectory`, excluding all the files defined under each `exclude` element. If no exclusions are defined, then all the content inside the `directory` will be copied into the `outputDirectory`. In this particular case, the value of `${pom.version}` will be replaced by the version of the artifact, defined in the `pom.xml` file under the distribution module.

The first `exclude` element instructs not to copy any file having the extension `.sh` from anywhere inside `target/wso2carbon-core-4.2.0` to the `outputDirectory` element:

```
<exclude>**/wso2server.bat</exclude>
<exclude>**/axis2services/sample01.aar</exclude>
```

The second `exclude` element instructs not to copy any file having the name `wso2server.bat` from anywhere inside `target/wso2carbon-core-4.2.0` to `outputDirectory`.

The third `exclude` element instructs not to copy the file `axis2services/sample01.aar` from anywhere inside `target/wso2carbon-core-4.2.0` to `outputDirectory`:

```
<exclude>**/axis2services/Echo.aar</exclude>
<exclude>**/axis2services/Version.aar</exclude>
<exclude>**/pom.xml</exclude>
<exclude>**/version.txt</exclude>
<exclude>**/README*</exclude>
<exclude>**/carbon.xml</exclude>
<exclude>**/axis2/*</exclude>
<exclude>**/LICENSE.txt</exclude>
<exclude>**/INSTALL.txt</exclude>
```

```

        <exclude>**/release-notes.html</exclude>
        <exclude>**/claim-config.xml</exclude>
        <exclude>**/log4j.properties</exclude>
        <exclude>**/registry.xml</exclude>
    </excludes>
</fileSet>

<fileSet>
    <directory>
        ../p2-profile-gen/target/wso2carbon-core-4.2.0/repository/conf/identity
    </directory>
    <outputDirectory>wso2is-${pom.version}/repository/conf/identity
    </outputDirectory>
    <includes>
        <include>**/*.xml</include>
    </includes>

```

The include element instructs to copy only the files having the .xml extension from anywhere inside the `../p2-profile-gen/target/wso2carbon-core-4.2.0/repository/conf/identity` directory to outputDirectory. If no include element is defined, everything will be included:

```

</fileSet>
<fileSet>
    <directory>
        ../p2-profile-gen/target/wso2carbon-core-4.2.0/repository/resources/security/ldif
    </directory>

    <outputDirectory>wso2is-${pom.version}/repository/resources/security/ldif
    </outputDirectory>
    <includes>
        <include>identityPerson.ldif</include>
        <include>scimPerson.ldif</include>
        <include>wso2Person.ldif</include>
    </includes>

```

The three include elements mentioned in the preceding code instruct to copy only the files having specific names from anywhere inside the `../p2-profile-gen/target/wso2carbon-core/4.2.0/repository/resources/security/ldif` directory to the outputDirectory:

```

</fileSet>

<fileSet>

```

```
    <directory>
      ../p2-profile-gen/target/wso2carbon-core-4.2.0/repository/
deployment/server/webapps
    </directory>
    <outputDirectory>${pom.artifactId}-${pom.version}/repository/
deployment/server/webapps
    </outputDirectory>
    <includes>
      <include>oauth2.war</include>
    </includes>
```

The include element instructs to copy only the WAR file with the name oauth2.war from anywhere inside the `../p2-profile-gen/target/wso2carbon-core/4.2.0/repository/resources/deployment/server/webapps` directory to the outputDirectory:

```
    </fileSet>

    <fileSet>
      <directory>
        ../p2-profile-gen/target/wso2carbon-core-4.2.0/repository/
deployment/server/webapps
      </directory>
      <outputDirectory>${pom.artifactId}-${pom.version}/repository/
deployment/server/webapps
      </outputDirectory>
      <includes>
        <include>authenticationendpoint.war</include>
      </includes>
    </fileSet>

    <fileSet>
      <directory>
        ../styles/service/src/main/resources/web/styles/css
      </directory>
      <outputDirectory>${pom.artifactId}-${pom.version}/resources/
allthemes/Default/admin
      </outputDirectory>
      <includes>
        <include>/**/*.css</include>
      </includes>
```

The `include` element instructs to copy any file with the extension `.css` from anywhere inside the `../styles/service/src/main/resources/web/styles/css` directory to the `outputDirectory`:

```
</fileSet>


<fileSet>
  <directory>
    ../p2-profile-gen/target/WSO2-CARBON-PATCH-4.2.0-0006
  </directory>
  <outputDirectory>
    wso2is-${pom.version}/repository/components/patches/
  </outputDirectory>
  <includes>
    <include>**/patch0006/*.*</include>
  </includes>
```

The `include` element instructs to copy all the files inside the `patch006` directory from anywhere inside the `../p2-profile-gen/target/WSO2-CARBON-PATCH-4.2.0-0006` directory to `outputDirectory`:

```
</fileSet>
</fileSets>

<files>
```

The `files` element is very much similar to the `fileSets` element in terms of the key functionality. Both can be used to control the contents of the assembly.

 The `files/file` element should be used when you are fully aware of the exact source file location while the `fileSets/fileSet` element is much flexible in picking files from a source based on a defined pattern.

The `fileMode` element in the following snippet defines a set of permissions to be attached to the copied file. The permissions are defined as per the four digit octal notation. You can read more about the four digit octal notation from http://en.wikipedia.org/wiki/File_system_permissions#Octal_notation_and_additional_permissions:

```
<file>
  <source>../p2-profile-gen/target/WSO2-CARBON-PATCH-${carbon.
kernel.version}-0006/lib/org.wso2.ciphertool-1.0.0-wso2v2.jar
  </source>
  <outputDirectory>
```



```
        ${pom.artifactId}-${pom.version}/lib/  
    </outputDirectory>  
    <filtered>true</filtered>  
    <fileMode>644</fileMode>  
  </file>  
</files>  
</assembly>
```

There are three descriptor elements defined under the assembly plugin for the package phase. The one we discussed earlier will create the binary distribution, while the `src/assembly/src.xml` and `src/assembly/docs.xml` files will create the source distribution and the documentation distribution respectively.

Let's also look at the assembly descriptor defined for the test phase:

```
<descriptors>  
  <descriptor>src/assembly/dist.xml</descriptor>  
</descriptors>
```

This is quite short and only includes the configuration required to build the initial distribution of the WSO2 Identity Server. Even though this project does this at the test phase, it seems like it has no value. In this case, it seems like `maven-antrun-plugin`, which is also associated with the package phase, but prior to the definition of the assembly plugin, needs the ZIP file distribution. Ideally, you should not have the assembly plugin run at the test phase unless there is a very strong reason to do so. You may need the distribution ready to run the integration tests; however, the integration tests should be executed in the `integration-test` phase, which comes after the package phase. In most cases, the assembly plugin is associated with the package phase of the Maven default lifecycle.

The following code shows the assembly descriptor defined in `src/assembly/dist.xml` for the test phase:

```
<assembly>  
  <formats>  
    <format>zip</format>  
  </formats>  
  <includeBaseDirectory>false</includeBaseDirectory>  
  <fileSets>  
    <!-- Copying p2 profile and osgi bundles-->  
    <fileSet>  
      <directory>  
        ../p2-profile-gen/target/wso2carbon-core-  
          4.2.0/repository/components  
      </directory>  
    </fileSet>  
  </fileSets>  
</assembly>
```

```

    <outputDirectory>wso2is-${pom.version}/repository/components
  </outputDirectory>
  <excludes>
    <exclude>**/eclipse.ini</exclude>
    <exclude>**/*.lock</exclude>
    <exclude>**/.data</exclude>
    <exclude>**/.settings</exclude>
  </excludes>
</fileSet>
</fileSets>
<dependencySets>
  <dependencySet>
    <outputDirectory>
      wso2is-${pom.version}/repository/deployment/client/modules
    </outputDirectory>
    <includes>
      <include>org.apache.rampart:rampart:mar</include>
    </includes>
  </dependencySet>
</dependencySets>
</assembly>

```

This configuration introduces a new element that we have not seen before, that is, `dependencySet`. The `dependencySet` element lets you include or exclude project dependencies to/from the final assembly that we are building. In the previous example, it adds the `rampart` module into the `outputDirectory` element. The value of the `include` element should be in the format of `groupId:artifactId:type[:classifier][:version]`. Maven will first look for this artifact with the defined coordinates in its local Maven repository, and if found, it will copy it to the location defined under the `outputDirectory` element.

Unlike the `fileSet` and `file` elements, the `dependencySet` element does not define a concrete path to pick and copy the dependency from. Maven finds the artifacts via the defined coordinates. If you want to include a dependency just by its `groupId` element and the `artifactId` coordinates, then you can follow the pattern `groupId:artifactId`. The particular artifact should be declared under the `dependencies` section of the POM file, which has the `assembly` plugin defined. You can find the following dependency definition for the `rampart` module in the POM file under the `distribution` module. If two versions of the same dependency are being defined in the same POM file (which is rather unlikely), then the last in the order will be copied:

```

<dependency>
  <groupId>org.apache.rampart</groupId>
  <artifactId>rampart</artifactId>

```

```
<type>mar</type>
<version>1.6.1-wso2v12</version>
</dependency>
```

You can also include a dependency by its `groupId`, `artifactId`, and `type`, as shown in the following configuration. Then, you can follow the pattern `groupId:artifactId:type[:classifier]`. This is the exact pattern followed in the previous example:

```
<includes>
  <include>org.apache.rampart:rampart:mar</include>
</includes>
```

If you want to be more precise, you can also include the version into the pattern. In this case, it will look like this:

```
<includes>
  <include>
    org.apache.rampart:rampart:mar:1.6.1-wso2v12
  </include>
</includes>
```



Most of the time we talk about four Maven coordinates; however, to be precise, there are five. A Maven artifact can be uniquely identified by these five coordinates: `groupId:artifactId:type[:classifier]:version`. We have already discussed the four main coordinates, but not the `classifier`. This is very rarely used; it can be quite useful in a scenario where we build an artifact out of the same POM file but with multiple target environments. We will be talking about `classifiers` in detail in *Chapter 7, Best Practices*.

The previous example only covered a very small subset of the assembly descriptor. You can find all available configuration options at <http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html>, which is quite an exhaustive list.



It is a best practice or a convention to include all the assembly descriptor files inside a directory called `assembly`, though it is not mandatory.

Let's take a look at another real-world example with **Apache Axis2**. Axis2 is an open source project released under the Apache 2.0 license. Axis2 has three types of distributions: a binary distribution as a ZIP file, a WAR file distribution, and a source distribution as a ZIP file. The binary ZIP distribution of Axis2 can be run on its own, while the WAR distribution must be deployed in a Java EE application server.

All three Axis2 distributions are created from the POM file inside the `distribution` module, which can be found at <http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/modules/distribution/pom.xml>.

This POM file associates the single goal of the Maven assembly plugin with the project, which initiates the process of creating the final distribution artifacts. The assembly configuration points to three different assembly descriptors—one for the ZIP distribution, the second for the WAR distribution, and the third for the source code distribution. The following code snippet shows the assembly plugin configuration:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <id>distribution-package</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>axis2-${project.version}</finalName>
        <descriptors>
          <descriptor>
            src/main/assembly/war-assembly.xml
          </descriptor>
          <descriptor>
            src/main/assembly/src-assembly.xml
          </descriptor>
          <descriptor>
            src/main/assembly/bin-assembly.xml
          </descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Let's take a look at the `bin-assembly.xml` file—the assembly descriptor, which builds the ZIP distribution:

```
<assembly>
  <id>bin</id>
```

```
<includeBaseDirectory>true</includeBaseDirectory>
<baseDirectory>axis2-${version}</baseDirectory>
<formats>
  <!--<format>tar.gz</format>
  //uncomment,if tar.gz archive needed-->
  <format>zip</format>
</formats>
```

This is exactly what we discussed earlier, and exactly what we wanted to avoid due to the same reason as in the comment, in the preceding code snippet. If we want to build a `tar.gz` distribution, then we need to modify the file. Instead of doing that, we can move the `format` configuration element out of the assembly descriptor to the plugin configuration defined in the `pom.xml` file. Then, you can define multiple profiles and configure the archive type based on the profile:

```
<fileSets>
  .....
</fileSets>

<dependencySets>
  <dependencySet>
    <useProjectArtifact>false</useProjectArtifact>
```

The `useProjectArtifact` configuration element instructs the plugin whether to include the artifact produced in this project build into the `dependencySet` element. By setting the value to `false`, we avoid it:

```
<outputDirectory>lib</outputDirectory>
<includes>
  <include>*:*:jar</include>
</includes>
<excludes>
  <exclude>
    org.apache.geronimo.specs:geronimo-activation_1.1_spec:jar
  </exclude>
</excludes>
</dependencySet>
<dependencySet>
  <useProjectArtifact>false</useProjectArtifact>
  <outputDirectory>lib/endorsed</outputDirectory>
  <includes>
    <include>javax.xml.bind:jaxb-api:jar</include>
  </includes>
</dependencySet>
<dependencySet>
```

```

<useProjectArtifact>false</useProjectArtifact>
<includes>
  <include>org.apache.axis2:axis2-webapp</include>
</includes>
<unpack>true</unpack>

```

The `includes` and `excludes` configuration elements will ensure that all the artifacts defined under the `dependencies` section of the `distribution/pom.xml` file will be included in the assembly, except the artifacts defined under the `excludes` configuration element. If you do not have any `include` elements, all the dependencies defined in the POM file will be included in the assembly, except what is defined under the `excludes` section.

Once the `unpack` configuration element is set to `true`, all the dependencies defined under the `include` elements will be unpacked into `outputDirectory`. The plugin is capable of unpacking `jar`, `zip`, `tar.gz`, and `tar.bz` archives. The `unpackOptions` configuration element, shown in the following configuration, can be used to filter out the content of the dependencies getting unpacked. According to the following configuration, only the files defined under the `include` elements under the `unpackOptions` element will be included; the rest will be ignored and won't be included in the assembly. In this particular case, `axis2-webapp` is a WAR file (which is defined under the `include` element of the previous configuration) and the `distributions/pom.xml` file has a dependency to it. This web app will be exploded (extracted), and then all the files inside the `WEB-INF/classes` and `axis2-web` directories will be copied into the `webapp` directory of the ZIP distribution, along with the `WEB-INF/web.xml` file:

```

<outputDirectory>webapp</outputDirectory>
<unpackOptions>
  <includes>
    <include>WEB-INF/classes/**/*</include>
    <include>WEB-INF/web.xml</include>
    <include>axis2-web/**/*</include>
  </includes>
</unpackOptions>
</dependencySet>
</dependencySets>
</assembly>

```

Now, let's take a look at `war-assembly.xml` – the assembly descriptor, which builds the WAR distribution. There is nothing new in this configuration, except the `outputFileNameMapping` configuration element. Since the value of the `format` element is set to `zip`, this assembly descriptor will produce an archive file conforming to the ZIP file specification. The value of the `outputFileNameMapping` configuration element gets applied to all the dependencies. The default value is parameterized: `${artifactId}-${version}${classifier?}.${extension}`. In this case, it's hardcoded to `axis2.war`, so the `axis2-webapp` artifact will be copied to the location defined under the `outputDirectory` element as `axis2.war`. Since there is no value defined for the `outputDirectory` element, the files will be copied to the root location, as shown here:

```
<assembly>
  <id>war</id>
  <includeBaseDirectory>false</includeBaseDirectory>
  <formats>
    <format>zip</format>
  </formats>
  <dependencySets>
    <dependencySet>
      <useProjectArtifact>false</useProjectArtifact>
      <includes>
        <include>org.apache.axis2:axis2-webapp</include>
      </includes>
      <outputFileNameMapping>axis2.war</outputFileNameMapping>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>../..</directory>
      <outputDirectory></outputDirectory>
      <includes>
        <include>LICENSE.txt</include>
        <include>NOTICE.txt</include>
        <include>README.txt</include>
        <include>release-notes.html</include>
      </includes>
      <filtered>true</filtered>
    </fileSet>
  </fileSets>
</assembly>
```

Artifact/resource filtering

We had a `filters` configuration, defined for the `assembly` plugin in the first example with the WSO2 Identity Server. This instructs the `assembly` plugin to apply the filter criteria defined in the provided filter or the set of filters for the files that are being copied to the final archive file. If you want to apply a filter to a given file, then you should set the value of the `filtered` element to `true`.

The following configuration shows how to define a filter:

```
<filters>
  <filter>${basedir}/src/assembly/filter.properties</filter>
</filters>
```

Let's take a look at the file `${basedir}/src/assembly/filter.properties`. This file defines a set of name/value pairs. The name is a special placeholder, which should be enclosed between `${` and `}` in the file to be filtered, and it will be replaced by the value during the filtering process:

```
product.name=WSO2 Identity Server
product.key=IS
product.version=5.0.0
hotdeployment=true
hotupdate=true
default.server.role=IdentityServer
```

Assembly help

As we discussed earlier, the `assembly` plugin currently only has two active goals: `single` and `help`; all the others are deprecated. As we witnessed in the previous example, the `single` goal is responsible for creating the archive with all sorts of other configurations.

The following command shows how to execute the `help` goal of the `assembly` plugin. This has to be executed from a directory with a POM file:

```
$ mvn assembly:help -Ddetail=true
```

If you see the following error when you run this command, you may not have the latest version. In that case, update the plugin version to 2.4.1 or later:

```
[ERROR] Could not find goal 'help' in plugin
org.apache.maven.plugins:maven-assembly-plugin:2.2-beta-2 among
available goals assembly, attach-assembly-descriptor, attach-
component-descriptor, attached, directory-inline, directory,
directory-single, single, unpack -> [Help 1]
```


A runnable standalone Maven project

Since we have covered a lot of ground-related information of the Maven assembly plugin, let's see how to build a complete end-to-end runnable standalone project with the assembly plugin. You can find the complete sample at <https://svn.wso2.org/repos/wso2/people/prabath/maven-mini/chapter06>. Proceed with the following steps to create a runnable standalone Maven project:

1. First, create a directory structure, as shown here:

```
| -pom.xml
| -modules
| - json-parser
|   | - src/main/java/com/packt/json/JSONParser.java
|   | - pom.xml
| - distribution
|   | - src/main/assembly/dist.xml
|   | - pom.xml
```

2. The `JSONParser.java` file is a simple Java class, which reads a JSON file and prints to the console, as shown here:

```
package com.packt.json;

import java.io.File;
import java.io.FileReader;
import org.json.simple.JSONObject;

public class JSONParser {

    public static void main(String[] args) {

        FileReader fileReader;
        JSONObject json;

        org.json.simple.parser.JSONParser parser;
        parser = new org.json.simple.parser.JSONParser();

        try {

            if (args == null || args.length == 0 || args[0] ==
                null || !new File(args[0]).exists())
            {
                System.out.println("No valid JSON file provided");
            } else {
                fileReader = new FileReader(new File(args[0]));
            }
        }
    }
}
```

```

        json = (JSONObject) parser.parse(fileReader);

        if (json != null) {
            System.out.println(json.toJSONString());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

3. Now, we can create a POM file under `modules/json-parser` to build our JAR file, as follows:

```

<project>

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt</groupId>
    <artifactId>json-parser</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
    <name>PACKT JSON Parser</name>

    <dependencies>
        <dependency>
            <groupId>com.googlecode.json-simple
            </groupId>
            <artifactId>json-simple</artifactId>
            <version>1.1</version>
        </dependency>
    </dependencies>

</project>

```

4. Once we are done with the `json-parser` module, the next step is to create the distribution module. The distribution module will have a POM file and an assembly descriptor. Let's first create the POM file under `modules/distribution`, which is shown here. This will associate two plugins with the project: `maven-assembly-plugin` and `maven-jar-plugin`. Both the plugins get executed in the `package` phase of the Maven default lifecycle. Since the `maven-assembly-plugin` is defined prior to the `maven-jar-plugin`, it will get executed first:

```

<project>
    <modelVersion>4.0.0</modelVersion>

```

```
<groupId>com.packt</groupId>
<artifactId>json-parser-dist</artifactId>
<version>1.0.0</version>
<packaging>jar</packaging>
<name>PACKT JSON Parser Distribution</name>

<dependencies>
<!--
Under the dependencies section we have to specify all the
dependent jars that must be assembled into the final artifact.
In this case we have two jar files. The first one is the external
dependency that we used to parse the JSON file and the second one
includes the class we wrote.
-->
    <dependency>
        <groupId>com.googlecode.json-simple</groupId>
        <artifactId>json-simple</artifactId>
        <version>1.1</version>
    </dependency>
    <dependency>
        <groupId>com.packt</groupId>
        <artifactId>json-parser</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <id>distribution-package</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <finalName>json-parser</finalName>
                        <descriptors>
                            <descriptor>
                                src/main/assembly/dist.xml
                            </descriptor>
                        </descriptors>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

```

        </configuration>
      </execution>
    </executions>
  </plugin>

```

```
<!--
```

Even though the maven-jar-plugin is inherited from the super pom, here we have redefined it because we need to add some extra configurations. Since we need to make our final archive executable, we need to define the class to be executable in the jar manifest. Here we have set `com.packt.json.JSONParser` as our main class. Also - the classpath is set to the lib directory. If you look at the assembly descriptor used in the assembly plugin, you will notice that, the dependent jar files are copied into the lib directory. The manifest configuration in the maven-jar-plugin will result in the following manifest file (META-INF/MANIFEST.MF).

```

Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: prabath
Build-Jdk: 1.6.0_65
Main-Class: com.packt.json.JSONParser
Class-Path: lib/json-simple-1.1.jar lib/json-parser-
            1.0.0.jar
-->

```

```

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.3.1</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <classpathPrefix>lib/</classpathPrefix>
            <mainClass>com.packt.json.JSONParser
            </mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>

  </plugins>
</build>
</project>

```

5. The following configuration shows the assembly descriptor (`module/distribution/src/main/assembly/dist.xml`), corresponding to the assembly plugin defined in the previous step:

```
<assembly>
  <id>bin</id>
  <formats>
    <format>zip</format>
  </formats>

  <dependencySets>
    <dependencySet>
      <useProjectArtifact>false</useProjectArtifact>
      <outputDirectory>lib</outputDirectory>
      <unpack>false</unpack>
    </dependencySet>
  </dependencySets>

  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory></outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

6. Now, we are done with the distribution module too. Next, we will create the root POM file, which aggregates both the json-parser and distribution modules, as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>json-parser-aggregator</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <name>PACKT JSON Parser Aggregator</name>
  <modules>
    <module>modules/json-parser</module>
    <module>modules/distribution</module>
  </modules>
</project>
```

7. We are all set to build the project. From the root directory, type `mvn clean install`. This will produce the `json-parser-bin.zip` archive inside the `modules/distribution/target` directory. The output will be as follows:

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] PACKT JSON Parser..... SUCCESS [ 1.790 s]
[INFO] PACKT JSON Parser Distribution.. SUCCESS [ 0.986 s]
[INFO] PACKT JSON Parser Aggregator.... SUCCESS [ 0.014 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

8. Go to `modules/distribution/target` and unzip `json-parser-bin.zip`.
9. To run the parser, type the following command, which will produce No valid JSON file provided as the output:
10. Once again, run the parser with a valid JSON file. You need to pass the path to the JSON file as an argument:

```
$ java -jar json-parser/json-parser-dist-1.0.0.jar
```

```
$ java -jar json-parser/json-parser-dist-1.0.0.jar
myjsonfile.json
```

The following is the output produced by the preceding command:

```
{
  "bookName" : "Mastering Maven", "publisher" : "PACKT"
}
```

Summary

In this chapter, we focused on the Maven `assembly` plugin. The `assembly` plugin provides a way of building custom archive files, aggregating many other custom configurations and resources. Most of the Java based products use the `assembly` plugin to build the final distribution artifacts. These can be a binary distribution, a source code distribution, or even a documentation distribution. This chapter covered real-world examples on how to use the Maven `assembly` plugin in detail, and finally, concluded with an end-to-end sample Maven project.

In the next chapter, we will discuss best practices in Maven.

7

Best Practices

In this book, so far, we discussed most of the key concepts related to Maven. In this chapter, we will focus on best practices associated with all of these core concepts. The following best practices constitute an essential ingredient in creating a successful and productive build environment. The criteria listed here will help you to evaluate the efficiency of your Maven project, mostly if you are dealing with a large-scale multi-module project:

- The time taken by a developer to get started with a new project and add it to the build system
- The effort required to upgrade a version of a dependency across all the project modules
- The time taken to build the complete project with a fresh local Maven repository
- The time taken to do a complete offline build
- The time taken to update the versions of Maven artifacts produced by the project; for example, from 1.0.0-SNAPSHOT to 1.0.0
- The effort required for a completely new developer to understand what your Maven build does
- The effort required to introduce a new Maven repository
- The time taken to execute unit tests and integration tests

The rest of the chapter talks about 25 industry-accepted best practices that would help you to improve developer productivity and reduce any maintenance nightmares.

Dependency management

In the following example, you will notice that the dependency versions are added to each and every dependency defined in the application POM file:

```
<dependencies>
  <dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>nimbus-jose-jwt</artifactId>
    <version>2.26</version>
  </dependency>
  <dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.2</version>
  </dependency>
</dependencies>
```

Imagine you have a set of application POM files in a multi-module project having the same set of dependencies. If you have duplicated the artifact version with each and every dependency, then to upgrade to the latest dependency you need to update all the POM files, which can easily lead to a mess.

Not just that, if you have different versions of the same dependency used in different modules of the same project, then it's going to be a debugging nightmare in the case of an issue.

With `dependencyManagement`, we can overcome both these issues. If it's a multi-module Maven project, you need to introduce `dependencyManagement` in the parent POM, so it will be inherited by all the other child modules:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.nimbusds</groupId>
      <artifactId>nimbus-jose-jwt</artifactId>
      <version>2.26</version>
    </dependency>
    <dependency>
      <groupId>commons-codec</groupId>
      <artifactId>commons-codec</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Once you define dependencies under the `dependencyManagement` section, as shown in the preceding code, you only need to refer a dependency from its `groupId` and the `artifactId` elements. The version element is picked from the appropriate `dependencyManagement` section:

```
<dependencies>
  <dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>nimbus-jose-jwt</artifactId>
  </dependency>
  <dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
  </dependency>
</dependencies>
```

With this, if you want to upgrade or downgrade a dependency, you only need to change the dependency version under the `dependencyManagement` section.

The same principle applies to plugins as well. If you have a set of plugins, which are used across multiple modules, you should define them under the `pluginManagement` section of the parent module. In this way, you can downgrade or upgrade plugin versions seamlessly just by changing the `pluginManagement` section of the parent POM, as shown in the following code:

```
<pluginManagement>
  <plugins>
    <plugin>
      <artifactId>maven-resources-plugin</artifactId>
      <version>2.4.2</version>
    </plugin>
    <plugin>
      <artifactId>maven-site-plugin</artifactId>
      <version>2.0-beta-6</version>
    </plugin>
    <plugin>
      <artifactId>maven-source-plugin</artifactId>
      <version>2.0.4</version>
    </plugin>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.13</version>
    </plugin>
  </plugins>
</pluginManagement>
```

Once you define the plugins in the plugin management section, as shown in the preceding code, you only need to refer a plugin from its `groupId` (optional) and `artifactId` elements. The version is picked from the appropriate `pluginManagement` section:

```
<plugins>
  <plugin>
    <artifactId>maven-resources-plugin</artifactId>
    <executions>.....</executions>
  </plugin>
  <plugin>
    <artifactId>maven-site-plugin</artifactId>
    <executions>.....</executions>
  </plugin>
  <plugin>
    <artifactId>maven-source-plugin</artifactId>
    <executions>.....</executions>
  </plugin>
  <plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <executions>.....</executions>
  </plugin>
</plugins>
```

Maven plugins were discussed in detail in *Chapter 4, Maven Plugins*.

Defining a parent module

In most of the multi-module Maven projects, there are many things that are shared across multiple modules. Dependency versions, plugin versions, properties, and repositories are only some of them. It is a common (and a best) practice to create a separate module called `parent` and define everything in common in its POM file. The packaging type of this POM file is `pom`. The artifact generated by the `pom` packaging type is itself a POM file.

The following are few examples of Maven parent modules:

- **Apache Axis2 project:** <http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/modules/parent/>
- **WSO2 Carbon project:** <https://svn.wso2.org/repos/wso2/carbon/platform/trunk/parent/>

Not all the projects follow this approach. Some just keep the parent POM file under the root directory (not under the parent module). The following are a couple of examples:

- **Apache Synapse project:** <http://svn.apache.org/repos/asf/synapse/trunk/java/pom.xml>
- **Apache HBase project:** <http://svn.apache.org/repos/asf/hbase/trunk/pom.xml>

Both the approaches deliver the same results, yet the first one is much preferred. With the first approach, the parent POM file only defines the shared resources across different Maven modules in the project while there is another POM file at the root of the project, which defines all the modules to be included in the project build. With the second approach, you define all the shared resources as well as all the modules to be included in the project build in the same POM file, which is under the project root directory. The first approach is better than the second one, based on the *separation of concerns* principle.

POM properties

There are six types of properties that you can use within a Maven application POM file:

- Built-in properties
- Project properties
- Local settings
- Environmental variables
- Java system properties
- Custom properties

It is always recommended that you use properties instead of hardcoding values in application POM files. Let's see a few examples.

Let's consider the example of the application POM file inside the Apache Axis2 distribution module, which is available at <http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/modules/distribution/pom.xml>. This defines all the artifacts created in the Axis2 project that need to be included in the final distribution. All the artifacts share the same `groupId` element as well as the `version` elements of the distribution module. This is a common scenario in most of the multimodule Maven projects. Most of the modules (if not all) share the same `groupId` and the `version` elements:

```
<dependencies>
  <dependency>
    <groupId>org.apache.axis2</groupId>
    <artifactId>axis2-java2wsdl</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.axis2</groupId>
    <artifactId>axis2-kernel</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.axis2</groupId>
    <artifactId>axis2-adb</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

In the preceding configuration, instead of duplicating the `version` element, Axis2 uses the project property `${project.version}`. When Maven finds this project property, it reads the value from the project POM `version` element. If the project POM file does not have a `version` element, then Maven will try to read it from the immediate parent POM file. The benefit here is, when you upgrade your project version some day, you only need to upgrade the `version` element of the distribution POM file (or its parent).

The preceding configuration is not perfect; it can be further improved as follows:

```
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>axis2-java2wsdl</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
```

```
<artifactId>axis2-kernel</artifactId>
<version>${project.version}</version>
</dependency>
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>axis2-adb</artifactId>
  <version>${project.version}</version>
</dependency>
</dependencies>
```

Here, we also replace the hardcoded value of the `groupId` element in all the dependencies with the project property `${project.groupId}`. When Maven finds this project property, it reads the value from the project POM `groupId` element. In case the project POM file does not have a `groupId` element, then Maven will try to read it from the immediate parent POM file.

Here is a list of some of the built-in properties and project properties of Maven:

- `project.version`: This refers to the value of the `version` element of the project POM file
- `project.groupId`: This refers to the value of the `groupId` element of the project POM file
- `project.artifactId`: This refers to the value of the `artifactId` element of the project POM file
- `project.name`: This refers to the value of the `name` element of the project POM file
- `project.description`: This refers to the value of the `description` element of the project POM file
- `project.basedir`: This refers to the path of the project's base directory

The following is an example, which shows the usage of this project property. Here, we have a system dependency, which needs to be referred from a file system path:

```
<dependency>
  <groupId>org.apache.axis2.wso2</groupId>
  <artifactId>axis2</artifactId>
  <version>1.6.0.wso2v2</version>
  <scope>system</scope>
  <systemPath>${project.basedir}/
    lib/axis2-1.6.jar</systemPath>
</dependency>
```

In addition to the project properties, you can also read properties from the `USER_HOME/.m2/settings.xml` file. For example, if you want to read the path to the local Maven repository, you can use the property, `${settings.localRepository}`. In the same way, with the same pattern, you can read any of the configuration elements, which are defined in the `settings.xml` file.

The environment variables defined in the system can be read using the `env` prefix within an application POM file. The `${env.M2_HOME}` property will return the path to Maven home, while `${env.java_home}` returns the path to the Java home directory. These properties will be quite useful within certain Maven plugins.

Maven also lets you define your own set of custom properties. Custom properties are mostly used when defining dependency versions.

You should not scatter custom properties all over the places. An ideal place to define them is the parent POM file in a multimodule Maven project, which will then be inherited by all the other child modules.

If you look at the parent POM file of the WSO2 Carbon project, you will find a large set of custom properties is defined (<https://svn.wso2.org/repos/wso2/carbon/platform/branches/turing/parent/pom.xml>). The following block of code contains some of those custom properties:

```
<properties>
  <rampart.version>1.6.1-wso2v10</rampart.version>
  <rampart.mar.version>1.6.1-wso2v10</rampart.mar.version>
  <rampart.osgi.version>1.6.1.wso2v10</rampart.osgi.version>
</properties>
```

When you add a dependency to the Rampart jar, you do not need to specify the version there. Just refer it by the `${rampart.version}` property name. Also, keep in mind that all the custom-defined properties are inherited and can be overridden in any child POM file:

```
<dependency>
  <groupId>org.apache.rampart.wso2</groupId>
  <artifactId>rampart-core</artifactId>
  <version>${rampart.version}</version>
</dependency>
```

Avoiding repetitive groupId and versions, and inheriting from the parent POM

In a multimodule Maven project, most of the modules (if not all) share the same `groupId` and the `version` elements. In that case, you can avoid adding `version` and `groupId` elements to your application POM file, as those will be automatically inherited from the corresponding parent POM.

If you look at `axis2-kernel` (which is a module of the Apache Axis2 project), you will find that no `groupId` or a `version` element is defined: (<http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/modules/kernel/pom.xml>). Maven reads them from the parent POM file:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.apache.axis2</groupId>
    <artifactId>axis2-parent</artifactId>
    <version>1.7.0-SNAPSHOT</version>
    <relativePath>../parent/pom.xml</relativePath>
  </parent>

  <artifactId>axis2-kernel</artifactId>
  <name>Apache Axis2 - Kernel</name>

</project>
```

Following naming conventions

When defining coordinates for your Maven project, you must always follow the naming conventions.

The value of the `groupId` element should follow the same naming convention you use in Java package names. It has to be a domain name (the reverse of the domain name) that you own—or at least that your project is developed under.

The following list covers some of the `groupId` naming conventions:

- The name of the `groupId` element has to be in lower case.
- Use the reverse of a domain name that can be used to uniquely identify your project. This will also help to avoid collisions between artifacts produced by different projects.

- Avoid using digits or special characters (for example, `org.wso2.carbon.identity-core`).
- Do not try to group two words into a single word by camel casing (for example, `org.wso2.carbon.identityCore`).
- Ensure that all the subprojects developed under different teams in the same company finally inherit from the same `groupId` and extend the name of the parent `groupId` rather than defining their own.

Let's go through some examples. You will notice that all the open source projects developed under **Apache Software Foundation (ASF)** use the same parent `groupId` (`org.apache`) and define their own `groupId`, which extends from the parent:

- **Apache Axis2 project:** `org.apache.axis2`, which inherits from the `org.apache` parent `groupId`
- **Apache Synapse project:** `org.apache.synapse`, which inherits from the `org.apache` parent `groupId`
- **Apache ServiceMix project:** `org.apache.servicemix`, which inherits from the `org.apache` parent `groupId`
- **WSO2 Carbon project:** `org.wso2.carbon`

Apart from the `groupId`, you should also follow the naming conventions while defining `artifactIds`.

The following lists out some of the `artifactId` naming conventions:

- The name of the `artifactId` has to be in lower case.
- Avoid duplicating the value of `groupId` inside the `artifactId`. If you find a need to start your `artifactId` element with the `groupId` element and add something to the end, then you need to revisit the structure of your project. You may need to add more module groups.
- Avoid using special characters (for example, `#`, `$`, `&`, `%`).
- Do not try to group two words into a single word by camel casing (for example, `identityCore`).

The following naming conventions for `version` are also equally important. The `version` of a given Maven artifact can be divided into four parts:

```
<Major version>.<Minor version>.<Incremental version>-<Build number or the qualifier>
```

The major version reflects the introduction of a new major feature. A change in the major version of a given artifact could also mean that the new changes are not necessarily backward compatible with the previously released artifact. The minor version reflects an introduction of a new feature to the previously released version in a backward compatible manner. The incremental version reflects a bug fixed release of the artifact. The build number can be the revision number from the source code repository.

This versioning convention is not just for Maven artifacts. Apple did a major release of its iOS mobile operating system in September 2014: iOS 8.0.0. Soon after the release, they discovered a critical bug in it that had an impact on cellular network connectivity and the TouchID on the iPhone. Then they released iOS 8.0.1 as a patch release to fix the issues.

Let's go through some examples:

- **Apache Axis2 1.6.0 release:** <http://svn.apache.org/repos/asf/axis/axis2/java/core/tags/v1.6.0/pom.xml>.
- **Apache Axis2 1.6.2 release:** <http://svn.apache.org/repos/asf/axis/axis2/java/core/tags/v1.6.2/pom.xml>.
- **Apache Axis2 1.7.0-SNAPSHOT:** <http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/pom.xml>.
- **Apache Synapse 2.1.0-wso2v5:** <http://svn.wso2.org/repos/wso2/tags/carbon/3.2.3/dependencies/synapse/2.1.0-wso2v5/pom.xml>. Here the synapse code is maintained under the WSO2 source repository, not under Apache. In this case, we use the wso2v5 classifier to make it different from the same artifact produced by Apache Synapse.

Think twice before you write your own plugin. You may not need it!

Maven is all about plugins! There is a plugin out there for almost everything you need to do. If you find a need to write a plugin, spend some time doing some research on the web to see whether you can find something similar – the chances are very high. You can also find a list of available Maven plugins at <http://maven.apache.org/plugins>.

The Maven release plugin

Releasing a project requires a lot of repetitive tasks. The objective of the Maven release plugin is to automate them. The release plugin defines following eight goals, which are executed in two stages – preparing the release and performing the release:

- `release:clean`: This cleans up after a release preparation
- `release:prepare`: This prepares for a release in SCM (Software Configuration Management)
- `release:prepare-with-pom`: This prepares for a release in SCM, and generates release POMs by fully resolving the dependencies
- `release:rollback`: This rolls back to a previous release
- `release:perform`: This performs a release from SCM
- `release:stage`: This performs a release from SCM into a staging folder or repository
- `release:branch`: This creates a branch of the current project with all versions updated
- `release:update-versions`: This updates the versions in the POM(s)

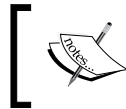
The preparation stage will complete the following tasks with the `release:prepare` goal:

- Verify that all the changes in the source code are committed.
- Make sure that there are no SNAPSHOT dependencies. During the project development phase we use SNAPSHOT dependencies, but, at the time of release, all the dependencies should be changed to a released version.
- The version of project POM files will be changed from SNAPSHOT to a concrete version number.
- The SCM information in the project POM will be changed to include the final destination of the tag.
- Execute all the tests against the modified POM files.
- Commit the modified POM files to SCM and tag the code with a version name.
- Change the version of POM files in the trunk to a SNAPSHOT version and then commit the modified POM files to the trunk.

Finally, the release will be performed with the `release:perform` goal. This will check out the code from the release tag in the SCM and run a set of predefined goals: `site` and `deploy-site`.

The `maven-release-plugin` is not defined in the super POM, and should be explicitly defined in your project POM file. The `releaseProfiles` configuration element defines the profiles to be released, and the `goals` configuration element defines the plugin goals to be executed during `release:perform`. In the following configuration, the `deploy` goal of the `maven-deploy-plugin` and the `single` goal of the `maven-assembly-plugin` will get executed:

```
<plugin>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.5</version>
  <configuration>
    <releaseProfiles>release</releaseProfiles>
    <goals>deploy assembly:single</goals>
  </configuration>
</plugin>
```



More details about the Maven Release plugin are available at <http://maven.apache.org/maven-release/maven-release-plugin/>.

The Maven enforcer plugin

The Maven Enforce plugin lets you control or enforce constraints in your build environment. These could be the Maven version, Java version, operating system parameters, and even user-defined rules.

The plugin defines two goals: `enforce` and `displayInfo`. The `enforcer:enforce` goal will execute all the defined rules against all the modules in a multimodule Maven project, while `enforcer:displayInfo` will display the project compliance details with respect to the standard rule set.

The `maven-enforcer-plugin` is not defined in the super POM, and should be explicitly defined in your project POM file:

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-enforcer-plugin</artifactId>
    <version>1.3.1</version>
    <executions>
      <execution>
        <id>enforce-versions</id>
        <goals>
          <goal>enforce</goal>
```

```
</goals>
<configuration>
  <rules>
    <requireMavenVersion>
      <version>3.2.1</version>
    </requireMavenVersion>
    <requireJavaVersion>
      <version>1.6</version>
    </requireJavaVersion>
    <requireOS>
      <family>mac</family>
    </requireOS>
  </rules>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
```

The preceding plugin configuration enforces the Maven version to be 3.2.1, Java version to be 1.6, and the operating system to be in the Mac family.

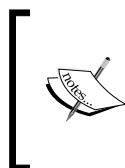
The Apache Axis2 project uses the enforcer plugin to make sure that no application POM file defines Maven repositories. All the artifacts required by Axis2 are expected to be in the Maven central repository. The following configuration element is extracted from <http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/modules/parent/pom.xml>. Here, it bans all the repositories and plugin repositories, except snapshot repositories:

```
<plugin>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <phase>validate</phase>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <requireNoRepositories>
            <banRepositories>true</banRepositories>
            <banPluginRepositories>true</banPluginRepositories>
            <allowSnapshotRepositories>true
          </allowSnapshotRepositories>
        </rules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```

        <allowSnapshotPluginRepositories>true
        </allowSnapshotPluginRepositories>
    </requireNoRepositories>
</rules>
</configuration>
</execution>
</executions>
</plugin>

```



In addition to the standard rule set shipped with the enforcer plugin, you can also define your own rules. More details about how to write custom rules are available at <http://maven.apache.org/enforcer/enforcer-api/writing-a-custom-rule.html>.

Avoiding the use of unversioned plugins

If you have associated a plugin with your application POM, without a version, then Maven will download the corresponding `maven-metadata.xml` file and store it locally. Only the latest released version of the plugin will be downloaded and used in the project. This can easily create uncertainties. Your project may work fine with the current version of a plugin, but later, if there is a new release of the same plugin, your Maven project will start to use the latest one automatically. This can result in an unpredictable behavior and lead to a debugging mess.



It is always recommended that you specify the plugin version along with the plugin configuration.

You can enforce this as a rule with the Maven `enforcer` plugin, shown as follows:

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.3.1</version>
  <executions>
    <execution>
      <id>enforce-plugin-versions</id>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>

```

```
<requirePluginVersions>
  <message>..... <message>
  <banLatest>true</banLatest>
  <banRelease>true</banRelease>
  <banSnapshots>true</banSnapshots>
  <phases>clean,deploy,site</phases>
  <additionalPlugins>
    <additionalPlugin>
      org.apache.maven.plugins:maven-eclipse-plugin
    </additionalPlugin>
    <additionalPlugin>
      org.apache.maven.plugins:maven-reactor-plugin
    </additionalPlugin>
  </additionalPlugins>
  <unCheckedPluginList>
    org.apache.maven.plugins:maven-enforcer-plugin,
    org.apache.maven.plugins:maven-idea-plugin
  </unCheckedPluginList>
</requirePluginVersions>
</rules>
</configuration>
</execution>
</executions>
</plugin>
```

The following explains each of the key configuration elements defined in the preceding code:

- **message:** This is used to define an optional message to the user, in case the rule execution fails.
- **banLatest:** This is used to restrict the use of "LATEST" as a version for any plugin.
- **banRelease:** This is used to restrict the use of "RELEASE" as a version for any plugin.
- **banSnapshots:** This is used to restrict the use of SNAPSHOT plugins.
- **banTimestamps:** This is used to restrict the use of SNAPSHOT plugins with timestamp versions.
- **phases:** This is a comma separated list of phases that should be used to find lifecycle plugin bindings. The default value is "clean,deploy,site".

- `additionalPlugins`: This is a list of additional plugins to enforce having versions. These plugins may not be defined in application POM files, but are used anyway, like `help`, `eclipse`, and so on. The plugins should be specified in the form `groupId:artifactId`.
- `unCheckedPluginList`: This is a comma separated list of plugins to skip version checking.



You can read more about the `requirePluginVersions` rule from <http://maven.apache.org/enforcer/enforcer-rules/requirePluginVersions.html>.

Descriptive parent POM files

Make sure your project's parent POM file is descriptive enough to list out what the project does, who the developers and contributors are, their contact details, the license under which the project artifacts are released, where to report issues, and so on. Here is a good example of a descriptive POM file, which is available at <http://svn.apache.org/repos/asf/axis/axis2/java/core/trunk/modules/parent/pom.xml>:

```
<project>
  <name>Apache Axis2 - Parent</name>
  <inceptionYear>2004</inceptionYear>
  <description>Axis2 is an effort to re-design and totally re-
    implement both Axis/Java.....</description>
  <url>http://axis.apache.org/axis2/java/core/</url>
  <licenses>
    <license>http://www.apache.org/licenses/
      LICENSE-2.0.html</license>
  </licenses>
  <issueManagement>
    <system>jira</system>
    <url>http://issues.apache.org/jira/browse/AXIS2</url>
  </issueManagement>
  <mailingLists>
    <mailingList>
      <name>Axis2 Developer List</name>
      <subscribe>java-dev-subscribe@axis.apache.org</subscribe>
      <unsubscribe>java-dev-unsubscribe@
        axis.apache.org</unsubscribe>
      <post>java-dev@axis.apache.org</post>
      <archive>http://mail-archives.apache.org/
        mod_mbox/axis-java-dev/</archive>
    </mailingList>
  </mailingLists>
  <otherArchives>
```



```
        <otherArchive>http://markmail.org/search/list:
                                org.apache.ws.axis-dev</otherArchive>
    </otherArchives>
</mailingList>
<developers>
    <developer>
        <name>Sanjiva Weerawarana</name>
        <id>sanjiva</id>
        <email>sanjiva AT wso2.com</email>
        <organization>WSO2</organization>
    </developer>
</developers>
<contributors>
    <contributor>
        <name>Dobri Kitipov</name>
        <email>kdobrik AT gmail.com</email>
        <organization>Software AG</organization>
    </contributor>
</contributors>
</project>
```

Documentation is your friend

If you are a good developer you know the value of documentation. Anything you write should not be cryptic or only be understood by you. Let it be a Java, .NET, C++ project, or a Maven project—the documentation is your friend. A code with a good documentation is extremely readable. If any configuration you add into an application POM file is not self-descriptive, make sure you add at least a single line comment explaining what it does.

Here to follow some good examples from the Apache Axis2 project:

```
<profile>
    <id>java16</id>
    <activation>
        <jdk>1.6</jdk>
    </activation>
    <!-- JDK 1.6 build still use JAX-WS 2.1 because integrating
        Java endorsed mechanism with Maven is bit of complex -
    -->
    <properties>
        <jaxb.api.version>2.1</jaxb.api.version>
        <jaxbri.version>2.1.7</jaxbri.version>
        <jaxws.tools.version>2.1.3</jaxws.tools.version>
```

```

    <jaxws.rt.version>2.1.3</jaxws.rt.version>
  </properties>
</profile>

```

```

<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <!-- Minimum required version here is 2.2-beta-4 because
        org.apache:apache:7 uses the runOnlyAtExecutionRoot
        parameter, which is not supported in earlier
        versions. -->
  <version>2.2-beta-5</version>
  <configuration>
    <!-- Workaround for MASSEMBLY-422 / MASSEMBLY-449 -->
    <archiverConfig>
      <fileMode>420</fileMode><!-- 420(dec)=644(oct) -->
      <directoryMode>493</directoryMode><!-- 493(dec)=755(oct) -->
      <defaultDirectoryMode>493</defaultDirectoryMode>
    </archiverConfig>
  </configuration>
</plugin>

```

```

<!-- No chicken and egg problem here because the plugin doesn't expose
any extension. We can always use the version from the current build.
-->
<plugin>
  <groupId>org.apache.axis2</groupId>
  <artifactId>axis2-repo-maven-plugin</artifactId>
  <version>${project.version}</version>
</plugin>

```

Avoid overriding the default directory structure

Maven follows the design philosophy *Convention over Configuration*. Without any configuration changes, Maven assumes the location of the source code is `${basedir}/src/main/java`, the location of tests is `${basedir}/src/test/java`, and the resources are available at `${basedir}/src/main/resources`. After a successful build, Maven knows where to place the compiled classes (`${basedir}/target/classes`) and where to copy the final artifact (`${basedir}/target/`). It is possible to change this directory structure, but it's recommended not to do so. Why?

Keeping the default structure improves the readability of the project. Even a fresh developer knows where to look, if he is familiar with Maven. Also, if you have associated plugins and other Maven extensions with your project, you will be able to use them with minimal changes if you have not altered the default Maven directory structure. Most of these plugins and other extensions assume the Maven convention by default.

Using SNAPSHOT versioning during the development

You should use the `SNAPSHOT` qualifier for the artifacts produced by your project if those are still under development and deployed regularly to a Maven snapshot repository. If the version to be released is `1.7.0`, then you should use the version `1.7.0-SNAPSHOT` while it's under development. Maven treats the version `SNAPSHOT` in a special manner. If you try to deploy `1.7.0-SNAPSHOT` into a repository, Maven will first expand the `SNAPSHOT` qualifier into a date and time value in UTC (Coordinated Universal Time). If the date/time at the time of deployment is 10.30 AM, November 10th, 2014, then the `SNAPSHOT` qualifier will be replaced with `20141110-103005-1`, and the artifact will be deployed with the version `1.7.0-20141110-103005-1`.

Get rid of unused dependencies

Always ensure that you maintain a clean application POM file. You should not have any unused dependencies defined or used undeclared dependencies. The Maven dependency plugin helps you in identifying such discrepancies.

The `maven-dependency-plugin` is not defined in the super POM and should be explicitly defined in your project POM file:

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.0</version>
</plugin>
```

Once the preceding configuration is added into your application POM file, you need to run the `analyze` goal of the dependency plugin against your Maven project:

```
$ mvn dependency:analyze
```

Here, you can see a sample output, which complains about an unused declared dependency:

```
[WARNING] Unused declared dependencies found:
[WARNING] org.apache.axis2:axis2-kernel:jar:1.6.2:compile
```



More details about the Maven dependency plugin are available at <http://maven.apache.org/plugins/maven-dependency-plugin/>.

Avoiding keeping credentials in application POM files

During a Maven build you need to connect to external repositories outside your firewall. In a tightly secured environment, any outbound connection has to go through an internal proxy server. The following configuration in `MAVEN_HOME/conf/settings.xml` shows how to connect to an external repository via a secured proxy server:

```
<proxy>
  <id>internal_proxy</id>
  <active>true</active>
  <protocol>http</protocol>
  <username>proxyuser</username>
  <password>proxypass</password>
  <host>proxy.host.net</host>
  <port>80</port>
  <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
</proxy>
```

Also, the Maven repositories can be protected for legitimate access. If a given repository is protected with HTTP Basic Authentication, the corresponding credentials should be defined as follows, under the `server` element of `MAVEN_HOME/conf/settings.xml`:

```
<server>
  <id>central</id>
  <username>my_username</username>
  <password>my_password</password>
</server>
```

Keeping confidential data in configuration files in clear text is a security threat that must be avoided. Maven provides a way of encrypting configuration data in `settings.xml`.

First, we need to create a master encryption key, shown as follows:

```
$ mvn -emp mymasterpassword  
{1J1MrCQRnngHIpSadxoyEKyt2zIGbm3Yl0ClKdTtRR6TleNaEfGOEoJaxNcdMr+G}
```

With the output from the above command, we need to create a file called `settings-security.xml` under `USER_HOME/.m2/` and add the encrypted master password there, as follows:

```
<settingsSecurity>  
  <master>  
    {1J1MrCQRnngHIpSadxoyEKyt2zIGbm3Yl0ClKdTtRR6TleNaEfGOEoJaxNcdMr+G}  
  </master>  
</settingsSecurity>
```

Once the master password is configured properly, we can start encrypting rest of the confidential data in `settings.xml`. Let's see how to encrypt the server password. First, we need to generate the encrypted password for the cleartext one using the following command. Note that earlier we used `emp` (encrypt master password) and now we are using `ep` (encrypt password):

```
$ mvn -ep my_password  
{PbYw8YaLb3cHA34/5EdHzoUsmmw/u/nWOwb9e+x6Hbs=}
```

Copy the value of the encrypted password and replace the corresponding value in `settings.xml`:

```
<server>  
  <id>central</id>  
  <username>my_username</username>  
  <password>  
    {PbYw8YaLb3cHA34/5EdHzoUsmmw/u/nWOwb9e+x6Hbs=}  
  </password>  
</server>
```

Avoiding using deprecated references

Since Maven 3.0 onwards, all the properties starting with `pom.*` are deprecated. Avoid using any of the deprecated Maven properties and, if you have used them already, ensure that you migrate to the equivalent ones.

Avoiding repetition – use archetypes

When we create a Java project, we need to structure it in different ways based on the type of the project. If it's a Java EE web application then we need to have a WEB-INF directory and a web.xml file. If it's a Maven plugin project, we need to have a Mojo class, which extends from `org.apache.maven.plugin.AbstractMojo`. Since each type of project has its own predefined structure, why would everyone have to build the same structure again and again? Why don't we start with a template? Each project can have its own template and the developers can extend the template to suit their requirements. Maven archetypes address this concern. Each archetype is a project template.

We discussed Maven archetypes in detail, in *Chapter 3, Maven Archetypes*.

Avoiding using `maven.test.skip`

You may manage an extremely small project that does not evolve a lot without unit tests. But, any large-scale project cannot exist without unit tests. Unit tests provide the first level of guarantee that you do not break any existing functionality with a newly introduced code change. In an ideal scenario, you should not commit any code to a source repository without building the complete project with unit tests.

Maven uses the `surefire` plugin to run tests and as a malpractice developers are used to skip the execution of unit tests by setting the `maven.test.skip` property to `true`, as follows:

```
$ mvn clean install -Dmaven.test.skip=true
```

This can lead to serious repercussions in the later stage of the project, and you must ensure that all your developers do not skip tests while building.

Using the `requireProperty` rule of the Maven `enforcer` plugin, you can ban developers from using the `maven.test.skip` property.

The following shows the `enforcer` plugin configuration that you need to add to your application POM:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.3.1</version>
  <executions>
    <execution>
      <id>enforce-property</id>
      <goals>
        <goal>enforce</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
</goals>
<configuration>
  <rules>
    <requireProperty>
      <property>maven.test.skip</property>
      <message>maven.test.skip must be specified</message>
      <regex>false</regex>
      <regexMessage>You cannot skip tests</regexMessage>
    </requireProperty>
  </rules>
  <fail>true</fail>
</configuration>
</execution>
</executions>
</plugin>
```

Now, if you run `mvn clean install` against your project, you will see the following error message:

maven.test.skip must be specified

This means you need to specify `Dmaven.test.skip=false` every time you run `mvn clean install`:

```
$ mvn clean install -Dmaven.test.skip=false
```

But if you set `-Dmaven.test.skip=true`, then you will see the following error:

You cannot skip tests

Still, you will find it a bit annoying to type `-Dmaven.test.skip=false` whenever you run a build. To avoid that, into your application POM file, add the property `maven.test.skip` and set its value to `false`:

```
<project>

  <properties>
    <maven.test.skip>false</maven.test.skip>
  </properties>

</project>
```



More details about `requireProperty` rule are available at <http://maven.apache.org/enforcer/enforcer-rules/requireProperty.html>.

Summary

In this chapter, we looked at and highlighted some of the best practices to be followed in a large-scale development project with Maven. Most of the points highlighted here were discussed in detail in previous chapters throughout the book. It is always recommended to follow best practices since it will drastically improve developer productivity and will reduce any maintenance nightmares.

Overall, the book covered Apache Maven 3 and its core concepts with examples, including Maven archetypes, plugins, assemblies, and lifecycles.

Index

A

Android mobile applications

with archetype plugin 59, 60

Apache Axis2 120

Apache Maven

about 1

installing, on Mac OS X 3, 4

installing, on Microsoft Windows 4

installing, on Ubuntu 2

URL 1

Apache Software Foundation (ASF) 142

Apache Synapse

URL 51

archetype

about 43-46

catalogues 47-51

plugin goals 54, 55

URL 44

archetype catalogue

about 47-51

building 51

catalog.xml file 53, 54

public archetype catalogues 51, 53

archetype plugin

Android mobile applications 59-61

EJB archives 61-64

Java EE web applications 55, 56

JIRA plugins 64

Spring MVC applications 65, 66

artifact/resource filtering 125

assembly descriptor

about 112-118

defining 118-124

URL 120

assembly plugin

about 110-112

help goal 125

single goal 125

URL 112

B

batch mode 47

best practices

application POM files, credentials

avoiding 153, 154

default directory structure override,

avoiding 151, 152

dependency management 134-136

deprecated references, usage avoiding 154

descriptive parent POM files 149

documentation 150

Maven enforcer plugin 145, 146

Maven release plugin 144, 145

maven.test.skip, usage avoiding 155, 156

naming conventions, following 141-143

parent module, defining 136, 137

parent POM, inheriting from 141

plugins 143

POM, properties 137-140

repetition, avoiding 155

repetitive groupIds and versions,

avoiding 141

SNAPSHOT versioning, using on

development 152

unused dependencies, avoiding 152, 153

unversioned plugins, usage

avoiding 147, 148

bundle plugin

URL 88

C

clean lifecycle 92-95

clean plugin 69, 70

Cocoon

URL 51

compiler plugin 70, 71

component descriptor 102

convention

versus configuration 7, 8

D

default lifecycle 95-99

default lifecycle, phases

compile 96

deploy 96

generate-resources 95

generate-sources 95

generate-test-resources 96

generate-test-sources 96

initialize 95

install 96

integration-test 96

package 96

post-integration-test 96

pre-integration-test 96

prepare-package 96

process-classes 96

process-resources 96

process-sources 95

process-test-classes 96

process-test-resources 96

process-test-sources 96

test 96

test-compile 96

validate 95

verify 96

dependency management

about 134-136

URL 29

deploy plugin 73-75

descriptive parent POM files 149

digit octal notation

URL 117

distribution module, Axis2

URL 121

E

Eclipse integration

about 10

reference link 10

EJB archives

with archetype plugin 61-63

enterprise service bus (ESB) 51

environment variables, Microsoft Windows

URL 4

F

Fuse

URL 51

H

heap size

configuring 5

Hello Maven! 6, 7

I

IDE integration

about 9

Eclipse integration 10

NetBeans integration 9

installation, Apache Maven

about 1

on Mac OS X 3, 4

on Microsoft Windows 4

on Ubuntu 2

URL 4

install plugin 73

IntelliJ IDEA integration

about 10

reference link 10

J

JAR file

about 109

URL 109

- jar plugin**
 - about 80
 - URL 81
- Java EE web applications**
 - with archetype plugin 55, 56
- Java.net**
 - URL 51
- Java virtual machine (JVM) 1**
- JIRA plugins**
 - with archetype plugin 64

L

- lifecycle bindings**
 - about 101
 - default lifecycle, coding 101-103
- lifecycle extensions**
 - about 105
 - example 105-107
 - reference link 108

M

- Mac OS X**
 - Apache Maven, installing on 3, 4
- Maven coordinates 25, 26**
- Maven dependency plugin**
 - URL 35
- Maven enforcer plugin 145-147**
- Maven plugins**
 - about 69
 - as extension 89
 - clean plugin 69, 70
 - compiler plugin 70, 71
 - deploy plugin 73, 75
 - discovering 84-86
 - executing 84-86
 - install plugin 73
 - jar plugin 80
 - management 87
 - reference link 70
 - release plugin 83, 84
 - repositories 87
 - resources plugin 82
 - site plugin 77-79
 - source plugin 81
 - surefire plugin 75-77
 - URL 143

- Maven release plugin 144, 145**
- Maven repositories**
 - about 9
 - reference link 9
- Microsoft Windows**
 - Apache Maven, installing on 4
- MyFaces**
 - URL 51

N

- naming conventions 141-143**
- NetBeans integration**
 - about 9
 - reference link 9

O

- OutOfMemoryError**
 - URL 5

P

- parent module**
 - defining 136, 137
- parent POM 27, 28**
- Permanent Generation (PermGen) 5**
- plugin management 87**
- plugin repositories**
 - about 87, 88
 - URL 88
- Project Object Model (POM)**
 - about 15, 16
 - dependencies, managing 29-32
 - elements, URL 28
 - extending 23-25
 - hierarchy 17
 - overriding 23-25
 - parent POM 27, 28
 - properties 137-140
 - super POM 18-22
- Project Object Model (POM), dependencies**
 - exclusion 39-42
 - optional 38, 39
 - scopes 35-37
 - transitive dependencies 33-35

public archetype catalogues

- Apache Synapse 51
- Cocoon 51
- Fuse 51
- Java.net 51
- MyFaces 51

R

release plugin 83, 84

remote catalogue

- URL 50

requirePluginVersions rule

- URL 149

requireProperty rule

- URL 156

resources plugin

- about 82
- URL 83

runnable standalone Maven project

- building 126-131
- URL 126

S

Secure Copy (scp) 75

site plugin 77, 79

Software Configuration Management (SCM) 83

source plugin 81

Spring MVC framework

- URL 65

standard lifecycles

- about 92
- clean lifecycle 92-95
- default lifecycle 95-99
- site lifecycle 100
- URL 97

super POM 18

surefire plugin 75-77

T

tomcat7 plugin

- URL 66

Tomcat 7.x distribution

- URL 57

troubleshooting

- about 10
- dependency classpath, viewing 13
- dependency tree, building 10, 11
- effective POM file, viewing 12
- environment variables, viewing 11
- Maven debug level logs, enabling 10
- system properties, viewing 11

U

Ubuntu

- Apache Maven, installing on 2

unversioned plugins

- usage, avoiding 149

W

web applications

- deploying, to remote Apache Tomcat server 57, 58

WebLogic distribution

- URL 62

WSO2 Carbon project

- URL 86, 140

WSO2 Identity Server (WSO2 IS)

- about 110
- URL 110



Thank you for buying **Maven Essentials**

About Packt Publishing

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

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

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

About Packt Open Source

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

Writing for Packt

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

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



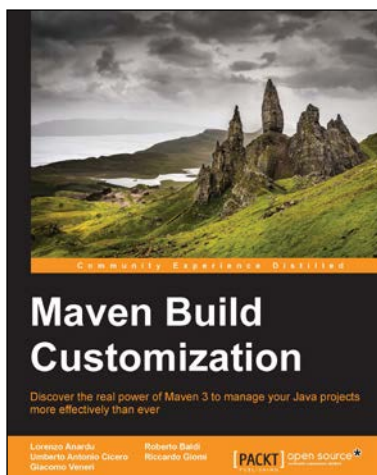
Android Application Development with Maven

ISBN: 978-1-78398-610-1

Paperback: 192 pages

Learn how to use and configure Maven to support all phases of the development of an Android application

1. Learn how to effectively use Maven to create, test, and release Android applications.
2. Customize Maven using a variety of suggested plugins for the most popular Android tools.
3. Discover new ways of accelerating the implementation, testing, and maintenance using this step-by-step simple tutorial approach.



Maven Build Customization

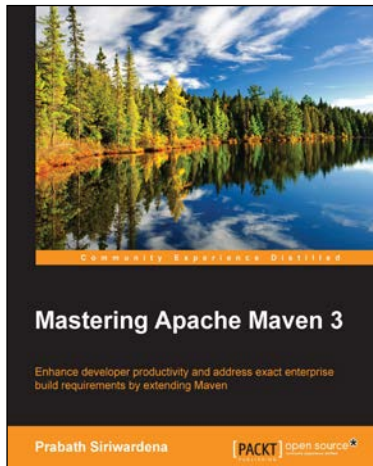
ISBN: 978-1-78398-722-1

Paperback: 270 pages

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

1. Administer complex projects customizing the Maven framework and improving the software lifecycle of your organization with "Maven friend technologies".
2. Automate your delivery process and make it fast and easy.
3. An easy-to-follow tutorial on Maven customization and integration with a real project and practical examples.

Please check www.PacktPub.com for information on our titles



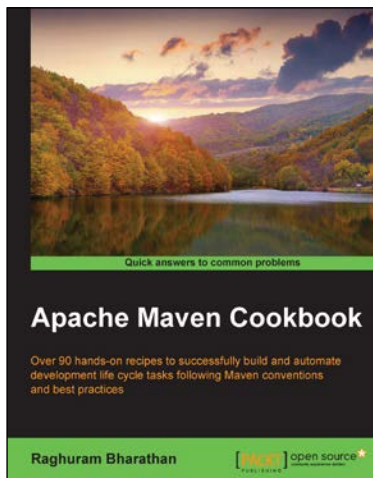
Mastering Apache Maven 3

ISBN: 978-1-78398-386-5

Paperback: 298 pages

Enhance developer productivity and address exact enterprise build requirements by extending Maven

1. Develop and manage large, complex projects with confidence.
2. Extend the default behavior of Maven with custom plugins, lifecycles, and archetypes.
3. Explore the internals of Maven to arm yourself with knowledge to troubleshoot build issues.



Apache Maven Cookbook

ISBN: 978-1-78528-612-4

Paperback: 272 pages

Over 90 hands-on recipes to successfully build and automate development life cycle tasks following Maven conventions and best practices

1. Understand the features of Apache Maven that makes it a powerful tool for build automation.
2. Full of real-world scenarios covering multi-module builds and best practices to make the most out of Maven projects.
3. A step-by-step tutorial guide full of pragmatic examples.

Please check www.PacktPub.com for information on our titles