Community Experience Distilled

Apache Maven Dependency Management

Manage your Java and JEE project dependencies with ease with this hands-on guide to Maven

Jonathan Lalou

[PACKT] PUBLISHING open source* community experience distilled

Apache Maven Dependency Management

Manage your Java and JEE project dependencies with ease with this hands-on guide to Maven

Jonathan Lalou

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Apache Maven Dependency Management

First published: October 2013

Production Reference: 1211013

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

ISBN 978-1-78328-301-9

www.packtpub.com

Cover Image by Jonathan Lalou

Credits

Author
Jonathan Lalou

Reviewer
Cedric Gatay

Acquisition Editor
Akram Hussain

Commissioning Editors
Manasi Pandire
Shreerang Deshpande

Technical Editors
Sharvari Baet
Nadeem N. Bagban
Kanhucharan Panda

Project Coordinators
Romal Karani
Ankita Goenka

Proofreaders
Paul Hindle
Stephen Swaney

Indexer
Monica Ajmera Mehta

Production Coordinators
Nilesh R. Mohite
Manu Joseph

Cover Work
Nilesh R. Mohite

About the Author

Jonathan Lalou is an engineer fascinated by new technologies, computer sciences, and the digital world since his childhood. Graduated from the Ecole des Mines—one of the French best polytechnic institutes—, Jonathan has more than 13 years of experience in Java and the JEE ecosystem.

Jonathan has worked for several global companies and financial institutions such as Syred, Philips, Sungard, Ixis CIB, BNP Paribas, and Amundi AM with strong ties, daily contacts, and frequent trips in Western Europe, Northern America, Judea, and emerging Asia. During his career, Jonathan has successfully climbed many levels: developer, architect, Scrum master, team leader, and project manager. Now, Jonathan has joined StepInfo (`http://www.stepinfo.com/`), a high-tech company focused on Java, and sponsor of local JUG and Devoxx, where he works as a project director, trainer, and leader of the expert task forces.

Jonathan's skills include a wide range of technologies and frameworks such as Spring, Hibernate, GWT, Mule ESB, Struts, JSF, Groovy, Android, EJB, JMS, application servers, agile methods, and of course Apache Maven.

Jonathan is available on the cloud. You can catch him on:

- Blog: `http://jonathan.lalou.free.fr`
- Twitter: `http://twitter.com/john_the_cowboy`
- LinkedIn: `http://www.linkedin.com/in/jonathanlalou`

About the Reviewer

Cedric Gatay has an engineering degree in Computer Science. He likes well-crafted and unit-tested code.

He has a very good understanding of Java languages (giving courses in Engineering schools and talking at local Java Users Groups).

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

He is also the founder of a collaborative blog for developers Bloggure: `http://www.bloggure.info`.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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

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

`http://PacktLib.PacktPub.com`

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface

In one decade, Apache Maven has successfully established itself in most of the companies dealing, coding, and releasing Java and JEE applications. Facing the still increasing problems of greedy, complex, and voluminous enterprise applications, architects, and developers have taken advantage of Maven's power and its ability to manage the plenty of dependencies, imports, and links. However, to be effective, and not to induce endless conflicts (any developer's nightmare), the management of dependencies should prove to be rigorous. Fortunately, Maven itself, and several plugins, do include both flexible and powerful features and tools.

This is what we propose to explore, develop, and illustrate in this book.

What this book covers

Chapter 1, *Basic Dependency Management*, covers nomenclature, long and short designations, transitive dependencies, parents and modules, and version ranges.

Chapter 2, *Dependency Mechanism and Scopes*, covers scope nomenclature, the `<dependencyManagement>` tag, modules and submodules, Maven Reactor, and management of dependencies in folders.

Chapter 3, *Dependency Designation (advanced)*, covers classic cases of <type>, how to create a new one, classifier, how to detect and fix dependency conflicts: Maven Dependency, enforce and Taggletatte plugins, and Dynamic POMs.

Chapter 4, *Migration of Dependencies to Apache Maven*, covers how to introduce Maven to non-Maven projects and even with non-Maven standard libraries.

Chapter 5, *Tools within Your IDE*, is specific to most spread IDEs: IntelliJ IDEA, Eclipse and NetBeans.

Chapter 6, *Release and Distribute*, deals with best practices before releases, how to fix conflicts with tier-parties, delivering artifacts and distributions, and distribution management.

Appendix, *Useful Public Repositories*, covers useful public repositories.

What you need for this book

In order to get the best out of this book, you need to have Maven 3.0.5 installed on your system, as well as a JDK (Java Development Kit), version 6 or above.

Access to the Internet is strongly advised.

You should also set environment variables such as `JAVA_HOME` and `MVN_HOME`, pointing respectively to JDK and Maven install folders.

Any text editor (PS-Pad, NotePad++, or even VIm and XEmacs) is sufficient to deal with Maven; anyways, you have interest then install an IDE like Eclipse, NetBeans, or IntelliJ IDEA. From a personal viewpoint, I advise the latter, but all of them are pretty good.

Who this book is for

This book is intended at developers, architects, and software urban planners, with a first experience of Apache Maven. More specifically, it suits teams who work on Java and JEE ecosystems, with wide spread frameworks, such as Spring, Hibernate, GWT, Groovy, and Apache Commons.

Conventions

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

Code words in text are shown as follows: " You can specify a different scope with the `<scope>` tag."

A block of code is set as follows:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.9</version>
</dependency>
```

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

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
  <version>${hibernate.version}</version>
</dependency>
```

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

```
$ mvn dependency:tree
```

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

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1
Basic Dependency Management

In this chapter, we will recollect and/or explain the basics behind dependency management.

Throughout this work, we will assume that you know the basic notions about Maven, that is, repository, POM, plugin, goal, build, lifecycles, and so on.

Nomenclature

Let's review some basic notions related to Maven and dependencies.

Reminders on Maven origins

In late 1970s, Dr Stuart Feldman's `make` and `makefiles` allowed developers to order a build process, and then to build automatically a project.

In 2000, Sun released an equivalent of `makefiles` for Java platform, which is now known as Apache Ant. Ant used XML files to order and script the build.

Apache Maven was released in 2005 based on different concepts, it is aimed at **automating** the build. Ant was build oriented, Maven is project oriented; all Java projects may be considered as nodes and leaves of a huge dependency graph.

Dependency

What is a dependency? A dependency is another archive—JAR, ZIP, and so on—which your current project needs in order to compile, build, test, and/or to run. In other terms, a dependency is a file or a group of files that contain(s) the classes pointed at in your `import` clauses, and even more (think of an object that would be dynamically created, thanks to reflexivity).

From Maven's viewpoint, a project depends on other projects, and is depended on by other projects. Anyway, a project only need know which projects it does depend on; there is no burden to spread or inform the projects that depend on it.

The dependencies are gathered in the `pom.xml` file, inside of a `<dependencies>` tag.

On launching Maven on a project, either for a build or another goal, the dependencies are resolved, and are then loaded from the local repository; if they are not present there, then Maven will download them from a remote repository and store them in the local repository. Anyway, a manual installation of an archive remains possible.

Usually, the local repository is located at the home directory, for example, `C:\Documents and Settings\myLogin\.m2\repository` under Windows XP, `C:\Users\myLogin\.m2\repository` under Windows Vista/7/8, or `~/.m2/repository` under Linux and Unix.

You can override the local repository folder. To perform that, you can add a tag `<localRepository>` within your `settings.xml`, for example:

```
<localRepository>/path/to/any/folder</localRepository>
```

Alternatively, you may dynamically specify a different folder, adding explicitly the destination folder as a runtime property on executing Maven, for example:

```
mvn clean \
-Dm2.localRepository=/path/to/another/folder/
somewhere/else
```

Long and short designations

Maven standardizes the way to identify a dependency. So, each dependency is described by the following data:

- `groupId`: A macro group or family of projects or archives to which a project belongs. Usually a same `groupId` gathers projects released by a same editor or that share a same functional domain. For example, `junit`, `org.hibernate`, and `org.richfaces.ui`.
- `artifactId`: The unique identifier of the project among the projects sharing the same `groupId`. For example, `junit`, `hibernate-annotations`, and `richfaces-components-ui`.
- `version`: This tag can have various values:
    - If no version is hinted, then Maven will use the last available artifact in the local repository.
    - The version of the project may be one of the following:
        1. A release version: A release version is "tagged", and can be considered as stable ; it shall not change anymore once published. For example,: 4.9, 3.3.1.GA, and 4.1.0.Final.
        2. A **Snapshot** version. Snapshots are artifacts still in development. They are expected to change and to be updated often. For example, `SNAPSHOT`, `1.0-SNAPSHOT` (assumed to be released later as `1.0`), and `1.2.3-SNAPSHOT` (assumed to be released later as 1.2.3).
- `classifier`: We will study this dependency in detail in *Chapter 3*, *Dependency Designation (advanced)*.
- `type`: We will study this dependency in detail in *Chapter 3*, *Dependency Designation (advanced)*.
- `scope`: We will study this dependency in detail in *Chapter 3*, *Dependency Designation (advanced)*.
- Other tags are available, we will review them later: `exclusion`, `optional`, and `systemPath`.

Long designation

So, basically, the preceding dependencies are equivalent to the following block in `pom.xml`:

```
<dependencies>
    <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.9</version>
     </dependency>
```

```
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
           <version>3.3.1.GA</version>
           <!-- ''GA'' stands for ''General Availability''-->
       </dependency>
       <dependency>
           <groupId>org.richfaces.ui</groupId>
           <artifactId>richfaces-components-ui</artifactId>
           <version>4.1.0.Final</version>
       </dependency>
</dependencies>
```

You can specify a range of values instead of a unique one. This will detailed later.

If your `pom.xml` points to many artifacts of the same `groupId` (which is common with frameworks such as Spring, and Hibernate), you would rather use properties in order to factorize the code and be sure to upgrade consistently the group when needed:

In other terms, you should add a `properties` block, and then reference them. Taking the same example, you should get something similar to the following:

```
<properties>
     <junit.version>4.9</junit.version>
     <hibernate.version>3.3.1.GA</hibernate.version>
     <richfaces.version>4.1.0.Final</richfaces.version>
 </properties>

<dependencies>
     <dependency>
         <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
  </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>${hibernate.version}</version>
   </dependency>
   <dependency>
       <groupId>org.richfaces.ui</groupId>
```

```
            <artifactId>richfaces-components-ui</artifactId>
            <version>${richfaces.version}</version>
        </dependency>
    </dependencies>
```

The classic XML block notation is called **long designation**.

Short designation

Long designation is verbose and fits only XML usage. This is why in other contexts, such as in logs, the **short designation** is preferred. It consists in collapsing the XML tree tags.

The preceding three dependencies are equivalent to the following:

- `junit:junit:jar:4.9:compile`
- `org.hibernate:hibernate-annotations:jar:3.3.1.GA:compile`
- `org.richfaces.ui:richfaces-components-ui:jar:4.1.0.Final:compile`

You may have noticed that `jar` and `compile` were introduced; in a few words, they correspond to the default values for the tags `type` and `scope`.

Visualizing dependencies

Modern IDEs include many features to help visualize dependencies.

With the dependency plugin, Maven includes a goal to print the tree of dependencies. The `dependency:tree`, for example, will be given as follows:

```
/workarea/development/cartography/$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
------
[INFO] Building Cartography Application 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------
------
[INFO]
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ cartography
---
```

```
[INFO] com.stepinfo.poc.cartography:cartography:jar:1.0-SNAPSHOT
[INFO] +- mockobjects:mockobjects-core:jar:0.09:test
[INFO] +- junit:junit:jar:4.9:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.1:test
[INFO] +- org.hibernate:hibernate-annotations:jar:3.3.1.GA:compile
[INFO] |  +- org.hibernate:hibernate:jar:3.2.6.ga:compile
[INFO] |  |  +- net.sf.ehcache:ehcache:jar:1.2.3:compile
[INFO] |  |  +- javax.transaction:jta:jar:1.0.1B:compile
[INFO] |  |  +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |  |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  |  +- antlr:antlr:jar:2.7.6:compile
[INFO] |  |  +- cglib:cglib:jar:2.1_3:compile
[INFO] |  |  +- asm:asm:jar:1.5.3:compile
[INFO] |  |  \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] |  +- org.hibernate:hibernate-commons-
annotations:jar:3.0.0.ga:compile
[INFO] |  +- org.hibernate:ejb3-persistence:jar:1.0.1.GA:compile
[INFO] |  \- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] +- org.springframework:spring-core:jar:3.1.0.RELEASE:compile
[INFO] |  \- org.springframework:spring-asm:jar:3.1.0.RELEASE:compile
[INFO] +- org.springframework:spring-orm:jar:3.1.0.RELEASE:compile
[INFO] |  +- org.springframework:spring-beans:jar:3.1.0.RELEASE:compile
[INFO] |  +- org.springframework:spring-jdbc:jar:3.1.0.RELEASE:compile
[INFO] |  \- org.springframework:spring-tx:jar:3.1.0.RELEASE:compile
[INFO] |     +- aopalliance:aopalliance:jar:1.0:compile
[INFO] |     +- org.springframework:spring-aop:jar:3.1.0.RELEASE:compile
[INFO] |     \- org.springframework:spring-context:jar:3.1.0.RELEASE:comp
ile
[INFO] |        \- org.springframework:spring-expression:jar:3.1.0.RELEAS
E:compile
[INFO] +- mysql:mysql-connector-java:jar:5.1.6:compile
[INFO] +- commons-lang:commons-lang:jar:2.3:compile
[INFO] \- log4j:log4j:jar:1.2.16:compile
[INFO] ------------------------------------------------------------------
------
[INFO] BUILD SUCCESS
```

```
[INFO] ------------------------------------------------------------------
------
[INFO] Total time: 2.485s
[INFO] Finished at: Mon Jun 24 19:08:20 CEST 2013
[INFO] Final Memory: 5M/15M
[INFO] ------------------------------------------------------------------
------
```

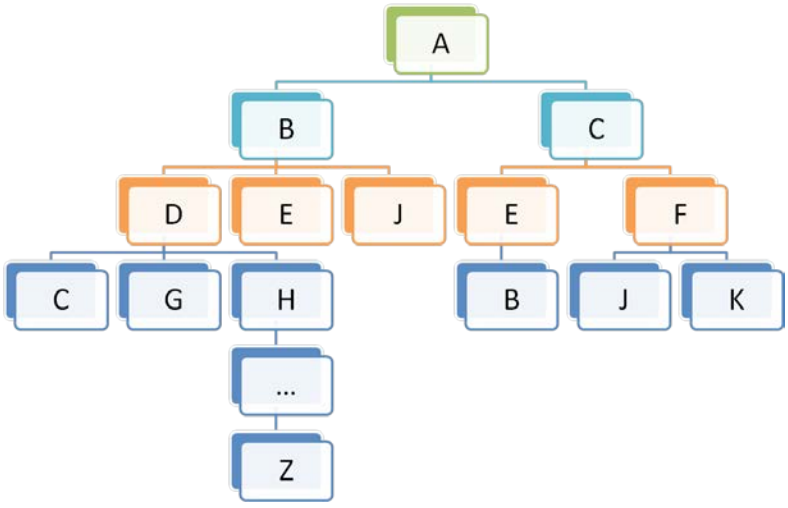
This tool will be very useful when we deal with conflicts of JARs.

Transitive dependencies

One of Maven's major contributions is the way it deals and manages not only *direct* dependencies, but also transitive ones.

The concept of transitivity

Dependencies are transitive. This means that if A depends on B and B depends on C, then A depends on both B and C. Theoretically, there is no limit to the depth of dependency. So, if you observe the following diagram of the tree of dependencies, you will notice that by transitivity, A depends on B, C, D, … until Z:



Even worse, we could have added a bit of complexity in mixing different versions of the same artifacts. In this very example with A as root project, B and C are **level 1** or direct dependencies, D, E, J, and F are **level 2** dependencies, C, G, H, and K are **level 3**, and so on.

You can imagine that the greater the level of dependencies, the more complex the situation is. The underlying issue of transitivity, you may guess, is when dependencies bear on the same `groupId/artifactId` but with different versions.

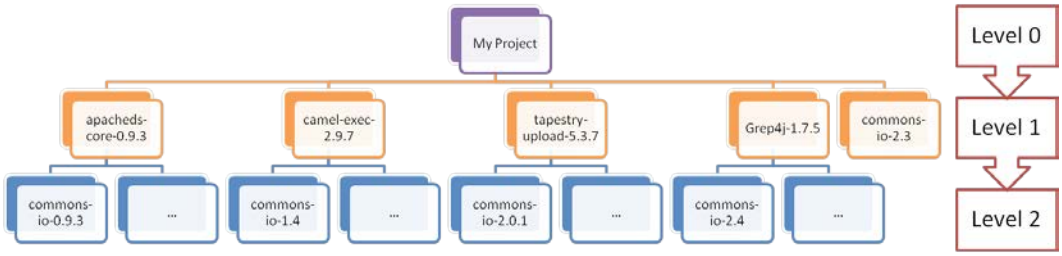
Resolution

Maven carries out the following algorithm to choose between two different versions:

- **Nearest first**: A dependency of *lower* level has priority over another of the *higher* depth. Hence, a direct dependency has priority over a transitive dependency.
- **First found**: At the same level, the first dependency that is found is taken.

This algorithm is known as **dependency mediation**.

Let's consider an example. The following diagram shows a dependency tree:



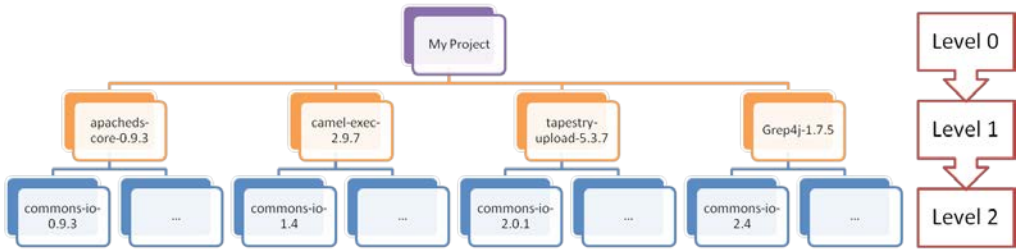
Here is the corresponding `dependencies` block in POM:

```
<dependencies>
     <dependency>
          <groupId>directory</groupId>
          <artifactId>apacheds-core</artifactId>
          <version>0.9.3</version>
          <!--implicit dependency to commons-io:
           commons-io:1.0-->
      </dependency>
      <dependency>
         <groupId>org.apache.camel</groupId>
         <artifactId>camel-exec</artifactId>
         <version>2.9.7</version>
         <!--implicit dependency to commons-io:commons-io:
          1.4-->
     </dependency>
```

```
<dependency>
    <groupId>org.apache.tapestry</groupId>
    <artifactId>tapestry-upload</artifactId>
    <version>5.3.7</version>
    <!--implicit dependency to commons-io:
     commons-io:2.0.1-->
</dependency>
<dependency>
    <groupId>com.googlecode.grep4j</groupId>
    <artifactId>grep4j</artifactId>
    <version>1.7.5</version>
    <!--implicit dependency to commons-io:commons-io:
     2.4-->
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.3</version>
</dependency>
```

The `commons-io-2.3` dependency is a dependency of **level 1**. So, even though it is declared after other artifacts and their transitive dependencies, then the dependency mediation will resolve `commons-io` to version 2.3. This case illustrates the concept of *nearest first*.

Now let's compare to a POM for which `commons-io-2.3` has been deleted from **level 1**. The dependency tree shown in the following diagram:

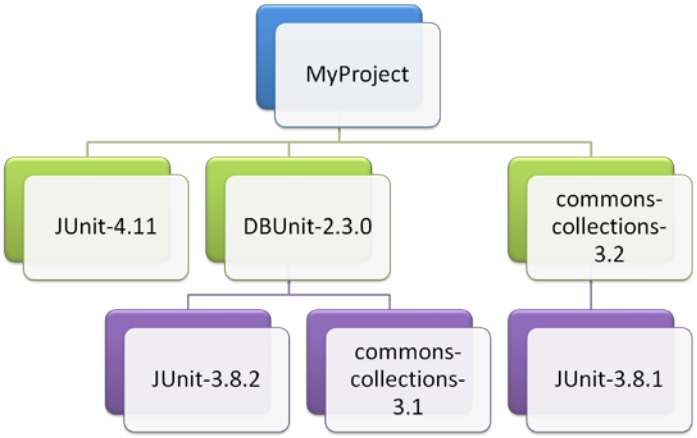


All dependencies to `commons-io` are of **level 2**, and differ on the versions: `0.9.3` (via `apacheds-core`), `1.4` (via `camel-exec`), `2.0.1` (via `tapestry-upload`), and `2.4` (via `grep4j`). Unlike a popular belief, the resolution will *not* lead to take the greatest version number (that is, `2.4`), but the first transitive version that appears in the dependency tree, in other terms `0.9.3`.

Had another dependency been declared before `apacheds-core`, its embed version of `commons-io` would have been resolved instead of version `0.9.3`. This case illustrates the concept of *first found*.

Exclusions

Let's consider the following example:



Our project needs `JUnit-4.11`, as well as `DBUnit-2.4.9` and `commons-collections-2.1`. But the two latter depend on other versions of JUnit, respectively `2.3.0` and `3.2`. Moreover, commons-collections depends on JUnit-3.8.1. Therefore, on building the project with goal test, we may encounter strange behaviors.

In this situation, you have to use an `<exclusion>` tag, in order to break the transitive dependency.

The POM will look similar to the following:

```
<dependencies>
     <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>${junit.version}</version>
         <scope>test</scope>
     </dependency>
     <dependency>
        <groupId>org.dbunit</groupId>
        <artifactId>dbunit</artifactId>
        <version>${dbunit.version}</version>
```

```
            <scope>test</scope>
            <exclusions>
                <!--Exclude transitive dependency to
                 JUnit-3.8.2 -->
                <exclusion>
                    <artifactId>junit</artifactId>
                    <groupId>junit</groupId>
                </exclusion>
               <!--Exclude transitive dependency to
                 Commons-Collections-3.1-->
                <exclusion>
                    <artifactId>commons-collections
                    </artifactId>
                    <groupId>commons-collections</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>commons-collections</groupId>
            <artifactId>commons-collections</artifactId>
            <version>${commons-collections.version}
            </version>
            <exclusions>
                <!--Exclude transitive dependency to
                 JUnit-3.8.1 -->
                <exclusion>
                    <artifactId>junit</artifactId>
                    <groupId>junit</groupId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
```

Most of the time, you will choose to exclude a transitive dependency for one of the following reasons:

- Conflicts between versions of the same artifact of your project, such as preceding version.
- Conflicts between artifacts of your project and artifacts from the platform of deployment, such as Tomcat or another server. For instance, if your project depends on `wsdl4j-1.5.3` and is deployed on JBoss AS 7.1.1, then a conflict may appear with JBoss's dependency to `wsdl4j-1.6.2`.

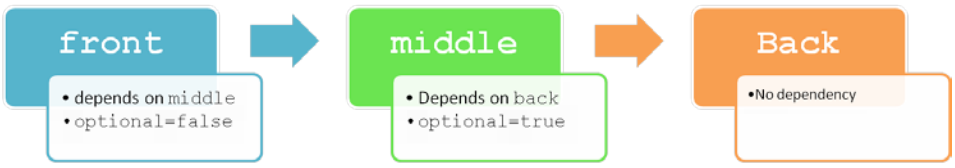
- In some cases, you do not want some of your dependencies to be exported within the archive you build (even though in this case, using a play on the dependency scope should be more elegant). The opposite case (when you need use your own dependencies and ignore the similar artifacts bundled with the server) will be exposed in *Chapter 6*, *Release and Distribute*.

Optional dependencies

The previous mechanism, based on `exclusion` tag, is in charge of the **depending** project to exclude unwanted dependencies.

Another mean exists to exclude transitive dependencies. This time, the charge lies on the project on which it is depended on. Maven provides the `optional` tag that takes a boolean value (`true`/`false`).

Let's consider the following example of dependencies:



Here are the corresponding POMs:

- For project `back`, the POM is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/
    4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>
         com.packt.maven.dependency.optional
     </groupId>
     <artifactId>back</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Example of POM which is depended on
      with 'optional' at true
    </name>
    <packaging>jar</packaging>

    <!-- no dependency at all -->

</project>
```

- For project `middle`, the POM is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/
    4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>
         com.packt.maven.dependency.optional
     </groupId>
     <artifactId>middle</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Example of POM with an optional dependency
    </name>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>
                com.packt.maven.dependency.optional
            </groupId>
            <artifactId>back</artifactId>
            <version>1.0-SNAPSHOT</version>
            <!-- The dependency to artifact 'back'
              is set at optional-->
            <optional>true</optional>
        </dependency>
    </dependencies>

</project>
```

- For project `front`, the POM is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/
    4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>
         com.packt.maven.dependency.optional
     </groupId>
     <artifactId>front</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Example of POM with scope import
```

```
            dependencyManagement of two artifacts
             with a version
            conflict because of transitive dependencies
        </name>
        <packaging>jar</packaging>
        <dependencies>
            <!-- Regular dependency ; 'front' depends
             on 'middle'-->
            <dependency>
                <groupId>
                    com.packt.maven.dependency.optional
                </groupId>
                <artifactId>middle</artifactId>
                <version>1.0-SNAPSHOT</version>
            </dependency>
        </dependencies>
    </project>
```

Now, we will see how to display the dependency trees. For `middle`, the tree is not different from what it would be, had the `optional` tag been set at `false`:

```
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ middle ---
[INFO] com.packt.maven.dependency.optional:middle:jar:1.0-SNAPSHOT
[INFO] \- com.packt.maven.dependency.optional:back:jar:1.0-
SNAPSHOT:compile
```

But for `front`, we get the following output:

```
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ front ---
[INFO] com.packt.maven.dependency.optional:front:jar:1.0-SNAPSHOT
[INFO] \- com.packt.maven.dependency.optional:middle:jar:1.0-
SNAPSHOT:compile
```

In other terms, `middle` has prevented its dependency to `back` to propagate transitively to other projects that depend on `middle` (among which `front`; but `middle` has no idea of `front`).

Had we *removed* the `optional` tag, we would have got that other trace:

```
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ front ---
[INFO] com.packt.maven.dependency.optional:front:jar:1.0-SNAPSHOT
[INFO] \- com.packt.maven.dependency.optional:middle:jar:1.0-
SNAPSHOT:compile
[INFO]    \- com.packt.maven.dependency.optional:back:jar:1.0-
SNAPSHOT:compile
```

As a conclusion, `exclusions` and `optional` allow to break the chain of transitivity. This may be driven either by the **depending** project or by the one on which it is **depended** on.

Parents/modules

When the number of your projects rises, the need of factorization and rationalization arises. Two tools exist for this purpose: parent POMs and modules. Although both kinds are often merged, they belong to different registers.

Parent POM

Parent POMs, aka super POMs, offer a mechanism of **inheritance**. They allow you to factorize some data and constants, among which are the following few:

- Common dependencies: In other terms, the artifacts that part or all of children POMs will depend on. Inscribing them in a parent POM has the same effect as writing them several times (and possibly having to update them manually on upgrading).
- Properties, such as:
  - Plugin
  - Declarations
  - Executions and IDs
- Configurations
- Common data: Developers' names, SCM address, distribution management, and so on.
- Resources

A parent POM, as a non archive target, will be declared with packaging `pom`; it is neither an archive nor expected to be distributed. It only references other projects.

So, a parent POM may look similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <!-- Either in a parent POM or in a independent POM,
    the triplet groupId/artifactId/version remain the unique
    way to identify a project-->
```

```
<groupId>com.packt.dependencyManagement.chapter1
</groupId>
<artifactId>superPom</artifactId>
<version>1.0</version>

<!-- The full name of the project-->
<name>Example of super POM: rationalizes properties,
    versions, plugins, etc.
</name>
<!-- The version of the parent POM ; notice it can or
  cannot be the same as the sons' versions -->
<packaging>pom</packaging>

<!-- These properties are common to all son POMs-->
<properties>
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
    <maven.compiler.source>1.6</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
    <gwt.version>2.0.3</gwt.version>
    <!-- etc. -->
    <dbunit.version>2.4.8</dbunit.version>
    <!-- sonar config -->
    <sonar.jdbc.url>
        jdbc:mysql://localhost:3306/sonar?useUnicode=true
    </sonar.jdbc.url>
    <sonar.jdbc.driver>com.mysql.jdbc.Driver
    </sonar.jdbc.driver>
    <!-- etc. -->
</properties>

<!-- The Source Code Management -->
<scm>
    <connection>
        scm:svn:http://localhost:8066/packt/branches/1.0
    </connection>
    <!-- etc. -->
</scm>

<!-- The distribution management -->
<distributionManagement>
    <repository>
        <id>repoDistrib</id>
        <name>repoDistrib</name>
```

```
            <!-- Here we use a local repository -->
            <url>file:///C:/artifacts</url>
        </repository>
    </distributionManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>sonar-maven-plugin</artifactId>
                <version>2.0</version>
            </plugin>
            <!-- etc. -->
        </plugins>
    </build>
</project>
```

As you can see, the data of the parent POM are not specific to one project; unlike, the data are shared by many other projects. Moreover, the scalability is swift: you can add any child project; this will not disturb the behavior of the parent POM.

The son project will refer to its parent with this header, that is, as usual by the triplet `groupId/artifactId/version` version.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/
          4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <!--The identifier of the parent POM-->
    <parent>
        <groupId>com.packt.dependencyManagement.chapter1
        </groupId>
        <artifactId>superPom</artifactId>
        <version>2.6.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>sonPom</artifactId>
    <name>The first son project of the parent POM</name>
    <packaging>jar</packaging>
</project>
```

Modules

Whereas parent POM provided a mechanism of *inheritance*, Maven **modules** provide a mechanism of **aggregation**, this means you can define groups of projects on which to run the same goal.

Let's consider the following `pom.xml`, and let's assume it is located in `/anywhere/multimodule/ folder`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.packt.dependencyManagement.chapter1</groupId>
   <artifactId>multimodule-simple</artifactId>
   <name>An example of multimodule POM</name>
   <packaging>pom</packaging>
   <version>1.0-SNAPSHOT</version>

   <modules>
       <module>multimodule-ear</module>
       <module>multimodule-war</module>
   </modules>
</project>
```

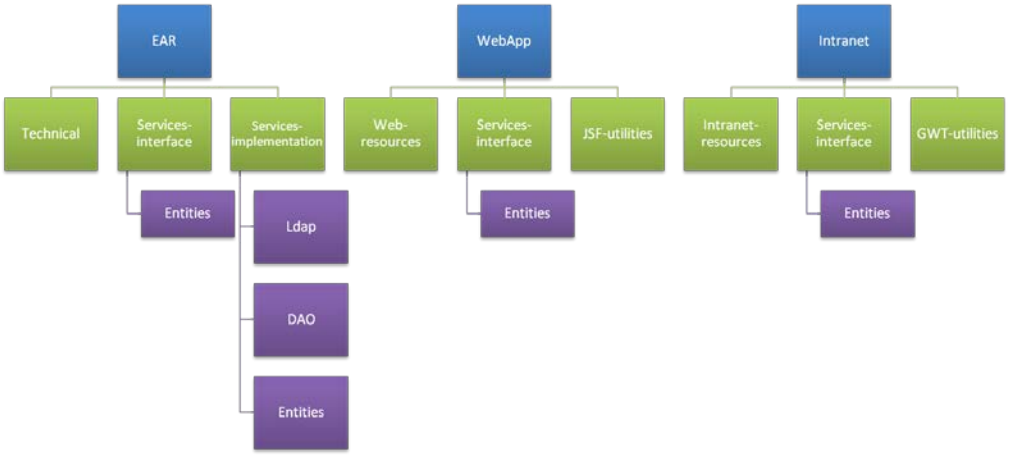
Two modules are declared as many n-tiers architectures: the first for an `EAR` module, and a second one for a `WAR` module. On each and every goal that is executed on `com.packt.maven.dependency:multimodule-simple:pom:1.0-SNAPSHOT`, the same goal will be executed on the `/anywhere/multimodule/multimodule-ear/pom.xml` and `/anywhere/multimodule/multimodule-war/pom.xml` files.

For instance, the call (`-f pom.xml` is redundant, we write it only to be explicit) `/anywhere/multimodule/ $ mvn clean install –f pom.xml` is equivalent to the following:

```
/anywhere/multimodule/ $ cd multimodule-ear
/anywhere/multimodule/multimodule-ear/ $ mvn clean install –f pom.xml
/anywhere/multimodule/multimodule-ear/ $ cd ../multimodule-war
/anywhere/multimodule/multimodule-war/ $ mvn clean install –f pom.xml
```

The interest of modules appears when you combine them with profiles. Then, only the modules declared in the linked profile will be run.

Let's consider the following dependency tree:



There are three archives (`EAR`, `WebApp`, and `Intranet`), which depend on JARs (technical, services interface, web resources, and so on).

Sometimes you work only on the EAR, sometimes only on the Intranet; and sometimes you work on the three. With modules, you can define profiles of compilation. So, the `pom.xml` will look similar to the following (the code is self-documented):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/
         4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt.dependencyManagement.chapter1
    </groupId>
    <artifactId>multimodule-withProfiles</artifactId>
    <name>Example multimodule POM using profiles</name>
   <packaging>pom</packaging>
   <version>1.2.3-SNAPSHOT</version>

   <profiles>
       <profile>
           <!-- In this profile, all submodules will be built ;
```

```
        please note the order of build: from the module with
        the least dependencies, to the module with the
        most ones -->
      <id>all</id>
      <activation>
          <!-- This profile is active by default,
           if no other is specified-->
          <activeByDefault>true</activeByDefault>
      </activation>
      <modules>
          <!--Submodules needed to build EAR-->
          <module>technical</module>
          <module>entities</module>
          <module>services-interface</module>
          <module>Ldap</module>
          <module>DAO</module>
          <module>Services-implementation</module>
          <!--Now EAR can be built-->
          <module>EAR</module>

          <!--Submodules needed to build WebApp-->
          <module>Web-resources</module>
          <!-- Services-implementation and Entities
            have already been built,
            we do not mention them here-->
          <module>JSF-utilities</module>
          <module>WebApp</module>

          <!--Submodules needed to build Intranet-->
          <module>Intranet-resources</module>
          <!-- Services-implementation and Entities
            have already been built,
            we do not mention them here-->
          <module>GWT-utilities</module>
          <module>Intranet</module>
      </modules>
  </profile>
  <profile>
      <!--In this profile, we build only artifacts
        which EAR depends on, then EAR-->
      <id>EAR</id>
      <modules>
          <module>technical</module>
          <module>entities</module>
```

```
                <module>services-interface</module>
                <module>Ldap</module>
                <module>DAO</module>
                <module>Services-implementation</module>
                <module>EAR</module>
            </modules>
            <activation>
                <!-- this profile is not active by
                 default : therefore, it will not be called
                 unless explicitly specified -->
                <activeByDefault>false</activeByDefault>
            </activation>
        </profile>
        <profile>
            <!--In this profile, we build only artifacts
             which WebApp depends on, then WebApp -->
            <id>WebApp</id>
            <modules>
                <!-- We need add Entities and Services-interface,
                 because unlike above they have not been build-->
                <module>entities</module>
                <module>services-interface</module>
                <module>Web-resources</module>
                <module>JSF-utilities</module>
                <module>WebApp</module>
            </modules>
        </profile>
        <profile>
            <!--In this profile, we build only artifacts
             which Intranet depends on, then Intranet -->
            <id>Intranet</id>
            <modules>
                <!-- We need add Entities and Services-interface,
                 because unlike above they have not been build-->
                <module>entities</module>
                <module>services-interface</module>
                <module>Intranet-resources</module>
                <module>GWT-utilities</module>
                <module>Intranet</module>
            </modules>
        </profile>
    </profiles>

</project>
```

A call to `/anywhere/multimodule-withProfiles/ $ mvn clean install –f pom.xml` is equivalent to the following:

```
/anywhere/multimodule-withProfiles/ $ mvn clean install –f pom.xml -Pall
```

But you can decide to build only EAR using the following:

```
/anywhere/multimodule-withProfiles/ $ mvn clean install –f pom.xml -PEAR
```

You can build only the Intranet using the following:

```
/anywhere/multimodule-withProfiles/ $ mvn clean install –f pom.xml –
Pintranet
```

You can also build with two profiles, for instance EAR and WebApp:

```
/anywhere/multimodule-withProfiles/ $ mvn clean install \
                                      –f pom.xml \
                                      –PEAR,WebApp
```

In a further chapter, we will dive into module mechanisms.

Most of the time, module descriptions are gathered within the parent POM.

Version ranges

Earlier, we have seen an artifact is described by the `groupId/artifactId/version` triplet. Actually, you can specify not only a version number, but also a range of versions.

The grammatical meaning of the mathematical signs is as follows:

- Parenthesis signs `(` and `)` hint an including range
- Brackets signs `[` and `]` hint an excluding range
- Commas separate subsets

The following table explains the grammatical meaning of a few ranges:

Range	Meaning
`1.2`	Version equals to `1.2` or is starting with `1.2`
`[1.2]`	Version strictly equal to `1.2`
`(,1.2]`	Anything less than `1.2`, included

Range	Meaning
`(,1.2)`	Anything less than `1.2`, excluded
`[1.2, )`	Anything greater than `1.2`, included
`(1.2,)`	Anything greater than `1.2`, excluded
`(1.2,3.4)`	Anything between `1.2` and `3.4`, both excluded
`(1.2,3.4]`	Anything between `1.2` (excluded) and `3.4` (included)
`[1.2,3.4)`	Anything between `1.2` (included) and `3.4` (excluded)
`[1.2,3.4]`	Anything between `1.2` and `3.4`, both included
`(,1.2],[3.4,)`	Anything less that `1.2` (included) or greater than `3.4` (included)
`(,1.2),(1.2,)`	Anything except `1.2`

The following are few meaningful examples, which will help you understand the ranges in a better way:

Range	Versions that will be accepted	Versions that will not be accepted
`1.2`	`1.2, 1.2.0, 1.2.3-GA, 1.2.3-Final`	`1.1, 3.4, Foo-1.2`
`[1.2]`	`1.2`	`1.2.0, 1.2.3-GA, 1.2.3-Final, 1.1, 3.4, Foo-1.2`
`(,1.2]`	`0.1, 1.0, 1.1.1, 1.2`	`1.3, 3.4`
`(,1.2)`	`0.1, 1.0, 1.1.1`	`1.2, 1.3, 3.4`
`[1.2, )`	`1.2, 1.3, 3.4`	`0.1, 1.0, 1.1.1`
`(1.2,)`	`1.3, 3.4`	`0.1, 1.0, 1.1.1, 1.2`
`(1.2,3.4)`	`1.2.1, 1.3, 3.3.9`	`1.1.8, 1.2, 3.4, 3.4.1`
`(1.2,3.4]`	`1.2.1, 1.3, 3.3.9, 3.4`	`1.1.8, 1.2, 3.4.1`
`[1.2,3.4)`	`1.2, 1.2.1, 1.3, 3.3.9`	`1.1.8, 3.4, 3.4.1`
`[1.2,3.4]`	`1.2, 1.2.1, 1.3, 3.3.9, 3.4`	`1.1.8, 3.4.1`
`(,1.2],[3.4,)`	`1.1.9, 1.2, 3.4, 3.5`	`1.3, 3.0`
`(,1.2),(1.2,)`	`1.1, 1.3, 3.4`	`1.2`

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.

Summary

In this opening chapter, we learned or revised different concepts. We are henceforth able to:

- Define a dependency and designations, whether short or long
- Describe the mechanism of dependency mediation
- Define parent POMs with different submodules
- Define different version ranges

2
Dependency Mechanism and Scopes

In the course of this second chapter, we will study the different scopes. Then, we will see the `dependencyManagement` tag.

We will dive deeper in the dependency management seen for multimodule projects. In the end, we will describe the content and meaning of files in the local repository.

Scopes

In the previous chapter, we have seen the `groupId`, `artifactId`, and `version` tags, used to determine in a deterministic way a project.

The `dependency` tag owns another subtag named `scope`.

Nomenclature of scope

The scope hints at the visibility of a dependency, relatively to the different life phases (build, test, runtime, and so on). Maven provides six scopes: `compile`, `provided`, `runtime`, `test`, `system`, and `import`.

Let's review them a bit more.

Compile

This is the default scope. Dependencies with `<scope>compile</scope>` are needed to build, test, and run, and are propagated to dependent projects.

Scope `compile` is to be used in most of the cases, for instance, when a class of your `src/` folder uses imports of classes.

So, as an example, consider that your code holds the following:

```
import org.apache.log4j.Logger;
import org.springframework.util.Assert;
```

If your code holds the preceding lines then, your `pom.xml` will contain the following:

```
<dependencies>
     <dependency>
         <groupId>log4j</groupId>
         <artifactId>log4j</artifactId>
         <version>1.2.14</version>
         <!-- may be omitted -->
         <scope>compile</scope>
     </dependency>
     <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>3.0.3.RELEASE</version>
        <!-- may be omitted -->
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Provided

Dependencies with `<scope>provided</scope>` are required to build and test. They are also required to run, but should not exported, because the dependence will be provided at runtime, for instance, by a servlet container or an application server. As a corollary, they are *not* propagated to dependent projects.

So, as an example, consider that your code holds the following:

```
import javax.jms.TextMessage;
import javax.ejb.EJBContext;
```

If your code holds the preceding lines then, your pom.xml will contain the following:

```
<groupId>milyn</groupId>
<!--
  also exists with:
 <groupId>j2ee</groupId>
 -->
<artifactId>j2ee</artifactId>
<version>1.4</version>
<scope>provided</scope>
```

Runtime

Dependencies with `<scope>runtime</scope>` *are not* needed to build, but *are* part of the classpath to test and run, and are propagated to dependent projects.

To illustrate a use case, let's consider a very simple project that includes a unique class, without any import:

```
package com.packt.maven.dependency.scopeRuntime;

public class LazyGuy {
   public static void main(String[] args) {
       System.out.println("I am lazy");
   }
}
```

The corresponding `pom.xml` is simple, too, and depends on no other artifact:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>
          com.packt.dependencyManagement.chapter2
      </groupId>
      <artifactId>scopeRuntime</artifactId>
      <packaging>jar</packaging>
      <version>1.0-SNAPSHOT</version>
      <name>ScopeRuntimeIllustrator</name>
 </project>
```

As given, the project compiles and runs.

Let's make the case a bit more complex. We instantiate `XStream`, but as done earlier, absolutely no import is declared and no dependence is added to `pom.xml`:

```
package com.packt.dependencyManagement.chapter2.scopeRuntime;

  public class LessLazyGuy {
      public static void main(String[] args) {
          try {
              final Object xstream;
              // We create an instance of XStream.
              // Beware that there is absolutely *no* import
```

```
                 // of the class
                xstream = ClassLoader.getSystemClassLoader().
                loadClass("com.thoughtworks.xstream.XStream").
                        newInstance();
                System.out.println("Success: " + xstream.toString());
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }
```

The project compiles as follows:

```
/workarea/development/projects/ScopeRuntime $ mvn clean install --quiet
[debug] execute contextualize
[debug] execute contextualize

-------------------------------------------------------
 T E S T S
-------------------------------------------------------

Results :

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

You can launch Maven in quiet mode with the `-q` or `--quiet` option then only errors are displayed in output console.

Let's get the execution classpath:

```
/workarea/development/projects/ScopeRuntime $ mvn dependency:build-
classpath
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building ScopeRuntimeIllustrator 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
```

```
[INFO]
[INFO] --- maven-dependency-plugin:2.1:build-classpath (default-cli) @
ScopeRuntime ---
[INFO] No dependencies found.
[INFO] Dependencies classpath:

[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 1.266s
[INFO] Finished at: Wed Jun 26 18:14:56 CEST 2013
[INFO] Final Memory: 5M/15M
[INFO] ------------------------------------------------------------
```

You can get the classpath used by a project in executing the `dependency:build-classpath` goal. We will deal with the dependency plugin in detail in a further chapter

Now, let's run the application:

```
/workarea/development/projects/ScopeRuntime $  java -cp target/
ScopeRuntime-1.0-SNAPSHOT.jar com.packt.maven.dependency.scopeRuntime.
LessLazyGuy
java.lang.ClassNotFoundException: com.thoughtworks.xstream.XStream
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
        at sun.misc.Launcher$AppClassLoader
         .loadClass(Launcher.java:301)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        at com.packt.maven.dependency.scopeRuntime.LessLazyGuy
        .main(LessLazyGuy.java:8)
```

As expected, the `XStream` object could not be instantiated. Let's take the `pom.xml` file. We add a dependency with the `runtime` scope:

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
```

```
            <version>1.4.4</version>
            <scope>runtime</scope>
        </dependency>
```

It will not affect the build path, only the runtime `classpath`:

```
/workarea/development/projects/ScopeRuntime $ mvn dependency:build-
classpath
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building ScopeRuntimeIllustrator 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.1:build-classpath (default-cli) @
ScopeRuntime ---
[INFO] Dependencies classpath:
/.m2/repository/com/thoughtworks/xstream/xstream/1.4.4/xstream-
1.4.4.jar:/.m2/repository/xmlpull/xmlpull/1.1.3.1/xmlpull-1.1.3.1.jar:/.
m2/repository/xpp3/xpp3_min/1.1.4c/xpp3_min-1.1.4c.jar [INFO] -----------
------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 1.266s
[INFO] Finished at: Wed Jun 26 18:20:35 CEST 2013
[INFO] Final Memory: 5M/15M
[INFO] ------------------------------------------------------------
```

Now, let's run the application:

```
/workarea/development/projects/ScopeRuntime $ java -cp /target/
ScopeRuntime-1.0-SNAPSHOT.jar:/.m2/repository/com/thoughtworks/xstream/
xstream/1.4.4/xstream-1.4.4.jar:/.m2/repository/xmlpull/xmlpull/1.1.3.1/
xmlpull-1.1.3.1.jar:/.m2/repository/xpp3/xpp3_min/1.1.4c/xpp3_min-1.1.4c.
jar
Success: com.thoughtworks.xstream.XStream@93dee9
```

Test

Dependencies with `<scope>test</scope>` are not needed to build and run the project, and are not propagated to dependent projects. But they are needed to compile and run the unit tests.

Dependencies with scope `test` are related to testing, among which we can quote the most famous: JUnit, DBUnit, various mock frameworks, and so on, for example,:

```
<dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.11</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.easymock</groupId>
     <artifactId>easymockclassextension</artifactId>
     <version>2.5.2</version>
     <scope>test</scope>
 </dependency>
```

Obviously, if some of your dependencies are imported only in tests (`cglib, javax.mail`, or anything else), then include them with `test` scope rather than `compile` scope. The main interest is to slim the archived created of unneeded dependencies.

System

Dependencies with `<scope>system</scope>` are similar to ones with scope provided. As such, they are required to build, test, and run, and are not propagated. The main difference is that `system` dependencies are not retrieved from a repository but from a hard written address on the filesystem.

When scope value is `system`, then an additional tag is mandatory: `systemPath`, which points at the location of the needed archive.

For instance, here is a `system` dependency:

```
<dependency>
    <groupId>com.sun</groupId>
    <artifactId>tools</artifactId>
    <version>1.6.0</version>
    <scope>system</scope>
    <systemPath>
        ${java.home}/../lib/tools.jar
    </systemPath>
</dependency>
```

As in the preceding example, most of the time `system` dependencies are needed to process or generate sources. Among the use cases is the generation of code thanks to annotations.

Import

Scope `import` is available only as a subtag of `<dependencyManagement>`. We will deal with it at the same time as `<dependencyManagement>`.

Scope overlay rules (via transitive dependencies)

Let's assume your project, let's say "Blue", depends on an artifact, say "Green", with scope `compile` that depends on a third artifact, say "Yellow", with scope `runtime`. Via transitivity, your project "Blue" will depend on "Yellow". The question is: what scope will "Yellow" be in your actual dependency tree?

The answer is: `runtime`. Indeed:

- In order to compile, Blue needs everything Green needs to compile; but Green does not need Yellow to compile; so Yellow is not needed by Blue to compile
- Yellow is needed by Green to run, and Green is needed by Blue to run; so Yellow is needed by Blue to run

More generally, here is the matrix of scope transitivity:

	Scope of project "Yellow" within "Green"			
Scope of project "Green" within "Blue"	**Compile**	**Provided**	**Runtime**	**Test**
Compile	Compile	(none)	Runtime	(none)
Provided	Provided	(none)	Provided	(none)
Runtime	Runtime	(none)	Runtime	(none)
Test	Test	(none)	Test	(none)

Scope of project "Yellow" within "Blue"

The dependencyManagement tag

The `dependencyManagement` tag is used in parent POMs (but not only). Basically, it can be seen as a way to factorize implicit and default `scope`, `exclusion`, and `version` within the son POMs.

In a further chapter, we will deal of the means of displaying the **effective POM**, that is, the POM that is actually read and run by Maven, and which differs from the POM written by the user.

First case study

For instance, let's consider the following parent POM. A `dependencyManagement` tag is declared, as well as a scope (`provided`) and an exclusion; this information will be considered by inheriting projects:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
           4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>
          com.packt.dependencyManagement.chapter2
      </groupId>
      <artifactId>ParentWithDependencyManagement</artifactId>
     <version>1.0-SNAPSHOT</version>
     <name>Example of multimodule parent POM with
         dependencyManagement tag
     </name>
     <packaging>pom</packaging>

     <dependencyManagement>
         <dependencies>
             <dependency>
                 <groupId>com.google.gwt</groupId>
                 <artifactId>gwt-servlet</artifactId>
                 <version>2.5.0</version>
                 <scope>provided</scope>
                 <exclusions>
                     <exclusion>
                         <groupId>org.json</groupId>
                         <artifactId>json</artifactId>
                     </exclusion>
                 </exclusions>
             </dependency>
         </dependencies>
     </dependencyManagement>
     <modules>
         <module>DependencyManagementSon</module>
     </modules>
 </project>
```

The following is a son POM:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
            4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <name>Example of son POM whose parent has
          dependencyManagement tag
      </name>
      <parent>
          <groupId>
             com.packt.dependencyManagement.chapter2
         </groupId>
         <artifactId>ParentWithDependencyManagement
         </artifactId>
         <version>1.0-SNAPSHOT</version>
     </parent>
     <modelVersion>4.0.0</modelVersion>
     <artifactId>DependencyManagementSon</artifactId>
     <version>1.0-SNAPSHOT</version>
     <dependencies>
         <dependency>
             <!-- Neither scope nor version are written-->
             <groupId>com.google.gwt</groupId>
             <artifactId>gwt-servlet</artifactId>
         </dependency>
     </dependencies>
 </project>
```

Then the dependency in the actual POM will be:

```
         <dependency>
              <groupId>com.google.gwt</groupId>
              <artifactId>gwt-servlet</artifactId>
              <version>2.5.0</version>
              <scope>provided</scope>
              <exclusions>
                  <exclusion>
                      <artifactId>json</artifactId>
                      <groupId>org.json</groupId>
                 </exclusion>
             </exclusions>
         </dependency>
```

In other words, the `scope`, `version`, and `exclusions` provided by the parent POM within the `dependencyManagement` tag were propagated to the son POM.

Yet, if the son POM did *not* declare a dependency of which `groupId` and `artifactId` are referenced in `dependencyManagement`, then this dependency would *not* be added to the son POM dependencies.

Second case study

Anyway, a son POM can override the default scope and version, which is inherited from parent POM, and by declaring it explicitly. Therefore, let's consider this second son POM:

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/
          4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <name>Example of son POM whose parent has
         dependencyManagement tag
     </name>
     <parent>
         <groupId>
            com.packt.dependencyManagement.chapter2
        </groupId>
        <artifactId>ParentWithDependencyManagement
        </artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>DependencyManagementOverridingSon
    </artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>com.google.gwt</groupId>
            <artifactId>gwt-servlet</artifactId>
            <!-- We explicitly hint at values of scope and
            version that are different from those in
            parent POM:
            * compile instead of provided
            * 2.1.0 instead of 2.5.0-->
            <version>2.1.0</version>
            <scope>compile</scope>
```

```
            </dependency>
        </dependencies>
    </project>
```

The corresponding effective POM will contain this dependency:

```
        <dependency>
           <groupId>com.google.gwt</groupId>
           <artifactId>gwt-servlet</artifactId>
           <version>2.1.0</version>
           <scope>compile</scope>
           <exclusions>
             <exclusion>
               <artifactId>json</artifactId>
               <groupId>org.json</groupId>
            </exclusion>
          </exclusions>
        </dependency>
```

Notice that, unlike `scope` and `version`, there is **no way** to remove or update an `exclusion` tag.

The import scope

The `dependencyManagement` tag can be used out of a hierarchy parent/son POM, but also as an easy way to factorize dependencies, for instance in a large organization with plenty of projects.

Let's consider the following project, let's say `head`, the code is self-documented:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
           4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>
          com.packt.dependencyManagement.chapter2.scopeImport
      </groupId>
      <artifactId>head</artifactId>
      <version>1.0-SNAPSHOT</version>
      <name>Example of POM with scope import
          dependencyManagement of two artifacts with a version
          conflict because of transitive dependencies
      </name>
```

```
    <packaging>pom</packaging>

    <dependencyManagement>
    <dependencies>
            <dependency>
                <groupId>org.apache.hadoop</groupId>
                <artifactId>hadoop-core</artifactId>
                        <!-- Explicit declaration of version-->
                <version>1.2.0</version>
            </dependency>

            <dependency>
                <!-- The artifact induces a transitive
                dependency to:
                'org.apache.hadoop:hadoop-tools:jar:1.0.4'
                Therefore, there is a potential conflict
                with the version of the artifact declared above
                -->
                <groupId>org.springframework.data</groupId>
                <artifactId>spring-data-hadoop</artifactId>
                <version>1.0.0.RELEASE</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <!-- spring-data-hadoop is explicitly declared as a
          dependency. Then its scope and version will be
          retrieved from the <dependencyManagement> data:
          1.0.0.RELEASE Besides, the induced dependency
          ('org.apache.hadoop:hadoop-tools:jar:1.0.4') should be
          inherited. Unlike, org.apache.hadoop:hadoop-
          tools:jar:1.2.0, that is *not* declared, will *not*
          be retrieved, even though it appears in the
         <dependencyManagement> block -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-hadoop</artifactId>
        </dependency>
    </dependencies>
</project>
```

Let's consider another project, let's say `legWithoutImport`, of which POM is:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>
            com.packt.dependencyManagement.chapter2.scopeImport
        </groupId>
        <artifactId>legWithoutImport</artifactId>
       <version>1.0-SNAPSHOT</version>
       <name>Example of POM without scope 'import'</name>
       <packaging>jar</packaging>
       <dependencyManagement>
           <dependencies>
               <!--Set a dependency to project Head-->
               <dependency>
                   <groupId>
                     com.packt.dependencyManagement.chapter2
                    .scopeImport
                   </groupId>
                   <artifactId>head</artifactId>
                   <version>1.0-SNAPSHOT</version>
                   <type>pom</type>
                   <!--no scope is indicated ; therefore it is
                    the default 'compile'-->
               </dependency>
           </dependencies>
       </dependencyManagement>
       <dependencies>
           <dependency>
               <groupId>
                 com.packt.dependencyManagement.chapter2
                 .scopeImport
               </groupId>
               <artifactId>head</artifactId>
               <type>pom</type>
               <!-- No need to hint at a specific version:
                the right one will ne got from the
                dependencyManagement tag above-->
           </dependency>
       </dependencies>
   </project>
```

This POM is a regular one. `legWithoutImport` depends on head, which is of type `pom`; hence, `legWithoutImport` will be transmitted by the dependencies schemes of `head`.

Among the dependency tree, we will get theses artifacts:

```
    org.springframework.data:spring-data-
  hadoop:jar:1.0.0.RELEASE:compile
    org.apache.hadoop:hadoop-core:jar:1.0.4:compile
```

As expected, the project depends on `spring-data-hadoop:jar:1.0.0.RELEASE` and `org.apache.hadoop:hadoop-core:jar:1.0.4` as a transitive dependency.

Now, here is another project, let's say `legWithImport`. In the `dependencyManagement` tag, an `import` scope was added:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
           4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>
         com.packt.dependencyManagement.chapter2.scopeImport
     </groupId>
     <artifactId>legWithImport</artifactId>
     <version>1.0-SNAPSHOT</version>
     <name>Example of POM with scope 'import'</name>
     <packaging>jar</packaging>
     <dependencyManagement>
         <dependencies>
             <dependency>
                 <groupId>
                     com.packt.dependencyManagement.chapter2
                     .scopeImport
                 </groupId>
                 <artifactId>head</artifactId>
                 <version>1.0-SNAPSHOT</version>
                 <type>pom</type>
                 <scope>import</scope>
             </dependency>
         </dependencies>
     </dependencyManagement>

     <dependencies>
         <dependency>
             <groupId>
                 com.packt.dependencyManagement.chapter2
                 .scopeImport
             </groupId>
             <artifactId>head</artifactId>
```

```
            <type>pom</type>
            <!-- The right version should not be written
             explicitly ; but Maven 3 denies to build if
             the tag is omitted.-->
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
    </project>
```

On printing the dependency tree, the following lines appear:

```
  org.springframework.data:spring-data-
hadoop:jar:1.0.0.RELEASE:compile (version managed from 1.0.0.RELEASE)
  org.apache.hadoop:hadoop-core:jar:1.2.0:compile (version managed
from 1.0.4)
```

The dependency that was retrieved was of version `org.apache.hadoop:hadoop-core:jar:1.2.0`, and not the implicit and transitive `1.0.4`. Here, it is the effect of scope `import`.

The output `(version managed from XXX)` indicates a transitive dependency was shortcut because of an explicit one.

So, as a summary, what are the role and interest of scope `import`? Scope `import` allows forcing the version in the transitive dependencies. It offers consistency of multiprojects organizations with a common base set, and reduces the risk of anarchy on dependencies, among which the situation of **diamond dependencies**, that is, `A` depends on `B` and `C`; both `B` and `C` depend on `D`.

Modules and submodules (advanced)

In this section, we present Maven Reactor, which is behind Maven's ability to deal with multimodule projects.

Maven Reactor

Maven Reactor is the mechanism that Maven uses to manage multimodule projects. There also exists a plugin, called **Reactor**, which will be reviewed a bit later. Maven Reactor serves three primary functions given as follows:

- To list the modules to build
- To determine the best order of construction
- To perform the actual build on the right projects

Reactor sorting

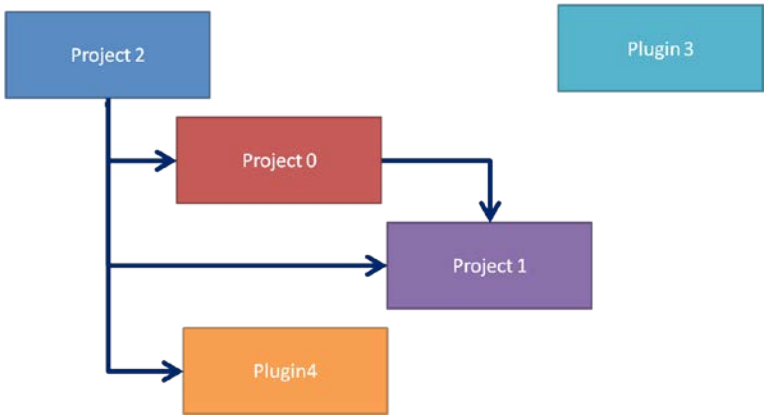
In a multimodule project, the submodules often depend on each other: either as a project dependency, or as a plugin dependency. This is why the order of the submodules within the `modules` tag is not necessarily honored. Maven Reactor will sort the submodules in such a way that a needed dependency is built before another project that does need it. The actual rules are the following to sort the submodules:

- **Project dependency first**: If project `A` depends on project `B`, then `B` is built before `A`.
- **Plugin dependency first**: If project `A` depends on plugin `C`, then `C` is built before `A`.
- **Build extension first**: If `F` is a plugin declaring a build extension (refer the *Creating new packaging/type* section in *Chapter 3*, *Advanced Dependency Designation*) and `G` is a module of any other type (either another type of plugin or a project), then `F` is built before `G`.
- **First declared, first built**: Out of these rules, at same topologic level, the submodules are built in the same order as they are declared.

Let's illustrate these rules with the following case. Consider a POM having five submodules:

- Three projects:
  - `project0`: It depends on `project1`
  - `project1`: It depends on no other project nor plugin
  - `project2`: It depends on both `project0` and `project1`; uses `plugin4`
- Two basic plugins, only print "hello world" or "hi world" on standard output:
  - `plugin3`: It is not used by any other module
  - `plugin4`: It depends on no other module (but is used by `project2`)

- The tree of dependencies is shown in the following diagram:



- Let's assume the dependencies are declared in the following order:

```
<modules>
  <!--project0 depends on project1-->
  <module>project0</module>
  <!--project1 depends on no other project nor plugin
   -->
  <module>project1</module>
  <!--project2 depends on both project0 and project1;
   uses plugin4-->
  <module>project2</module>
  <!--Plugin3 is not used by any other module-->
  <module>plugin3</module>
  <!--Plugin4 is used by project2-->
  <module>plugin4</module>
</modules>
```

- The question is: in which order shall Maven Reactor build these submodules?
  - First of all, `Project2` and `Plugin3` have no ties between them; but `Plugin3` is declared after `Project2`, therefore, `Project2` will be built before `Plugin3`.
  - Secondly, `Project2` depends on `Project0`, `Project1`, and `Plugin4`, therefore, `Project2` will be built after these three modules.
  - Thirdly, `Project0` and `Plugin4` have no ties, but `Plugin4` is declared after `Project0`, therefore, `Project0` will be built before `Plugin4`.
  - Lastly, `Project0` depends on `Project1` hence `Project1` will be built before `Project0`.

- So, we get the order of build that is confirmed by Maven execution:

```
[INFO] Reactor Summary:
[INFO]
[INFO] project1 ..................... SUCCESS [26.734s]
[INFO] project0 ..................... SUCCESS [0.797s]
[INFO] plugin4 ..................... SUCCESS [13.015s]
[INFO] project2 ..................... SUCCESS [0.594s]
[INFO] plugin3 ..................... SUCCESS [1.766s]
```

  If we had set the submodules order declaration to this one:

```
<modules>
    <module>plugin3</module>
    <module>project2</module>
    <module>plugin4</module>
    <module>project1</module>
    <module>project0</module>
</modules>
```

  Then, honoring the same rules, we would have got the following order:

```
[INFO] Reactor Summary:
[INFO]
[INFO] plugin3 ..................... SUCCESS [3.000s]
[INFO] project1 ..................... SUCCESS [0.500s]
[INFO] project0 ..................... SUCCESS [0.359s]
[INFO] plugin4 ..................... SUCCESS [0.984s]
[INFO] project2 ..................... SUCCESS [0.422s]
```

Parallel build

Since Maven 3.0, build can be performed in parallel. For instance, `mvn -T 8 <goals>` will build on eight parallel threads, and `mvn -T 2C <goals>` will build with two threads per CPU core. Anyway, this feature is still experimental and you should beware of Thread unsafe-plugins.

Reactor options and the Reactor plugin for Maven 2

Maven Reactor is available from any project, that is, you need not declare any dependency to call it. Maven Reactor has several convenient options. Until Maven 2.0.11, those available options were only through an explicit call to Reactor plugin.

- `--also-make` or `-am`: This option is used to build the projects given in arguments (including their dependencies).
    - Use case: Your build is launched with a `-P` (profile) argument, hinting that only a part of submodules should be built; this option allows you to add arbitrary modules to build.
    - Equivalent goal in Maven 2: This option make folders and projects, for example, `mvn reactor:make –Dmake.folders=project0, ~/workarea/project1`, or `mvn reactor:make –Dmake.projects=project2,plugin4`.
- `--also-make-dependents` or `-amd`: This options is used to build the projects given in arguments, as well as the projects that depend on them.
    - Use case: You rebuild a module among a sequence of many; using this argument you can force the dependent projects to rebuild, too, without rebuilding the complete set of modules.
    - Equivalent goal in Maven 2: This option is used to make dependents, for example, `mvn reactor:make-dependents –Dmake.folders=project0, ~/workarea/project1`, or `mvn reactor:make-dependents –Dmake.projects=project2,plugin4`.
- `--fail-fast` or `-ff`: (behavior activated by default) This option makes the whole-set build fail as soon as one module fails to build.
- `--fail-at-end` or `-fae`: If a module fails to build, the build keeps on running on the other modules. Anyway, at the end the build will be marked as failed.
- `--non-recursive`: This option does not build the submodules on all projects but only the very project on which the build was executed.
    - Use case: When your project is not only a `pom`-packaged one, but contains useful code that need be compiled, independently of its submodules.
- `-r`: This option is used to build the list of projects given in arguments, ignoring the modules declared in the POM.

- `--resume-from` or `-rf`: This option starts or resumes the build since the module given as parameter, including the dependent projects.
    - Use case: Your project contains several submodules, and one of them makes the build fail. When you have identified and fixed the issue, building the former is not worthwhile; but you are interested in resuming the build at the last point where it was broken.
    - Equivalent goal in Maven 2: Is used to resume the reactor, foe example, `mvn reactor:resume –Dfrom=plugin3`.
- A last goal is available for Maven 2 but has no native equivalent in Maven 3: `make-scm-changes`. It allows building the projects that have been changed locally, as well as the depended on projects. To enable this feature, an `<scm>` block is required in your POM.

Management of dependencies in folders

Let's take a look at how to manage the project dependencies in folders in the following sections.

The dependencies in their folders

Dependencies that are downloaded from a remote repository, or are manually installed, are stored in the local repository folder, let's say `~/.m2/repository`. Each artifact is copied into a specific location, which depends on its `groupId/artifactId/version`, and so on. To determine the actual folder, the following algorithm is applied:

- `groupId` is split on dots; each piece of the result will result in a folder.
- `artifactId` and `version` are transposed to unique folders, even if their values contains dots.

As an example, let's consider a dependency to `org.hibernate:hibernate-annotations:jar:3.3.1.GA:compile`. The matching folder is `.m2/repository/org/hibernate/hibernate-annotations/3.3.1.GA`. The `groupId` is split, whereas `artifactId` and `version` are preserved. The folder will contain several files:

- Three files related to the JAR:
    - `hibernate-annotations-3.3.1.GA.jar`: The very artifact
    - `hibernate-annotations-3.3.1.GA.jar.tmp.sha1.tmp` and `hibernate-annotations-3.3.1.GA.jar.sha1`: Secure hash files

- Three parallel files related to the POM:
    - `hibernate-annotations-3.3.1.GA.pom`
    - `hibernate-annotations-3.3.1.GA.pom.tmp.sha1.tmp`
    - `hibernate-annotations-3.3.1.GA.pom.sha1`

Nonarchive files

For snapshot versions, an additional file should appear: `maven-metadata.xml` (for a remote repository), or `maven-metadata-${repoId}.xml` for a local repository. Most of the time, this last file is called `maven-metadata-local.xml`. It contains the timestamps of the times when the considered snapshot was downloaded. When two snapshots share the same version, such as `1.2.3-SNAPSHOT`, this mechanism allows to re-install a snapshot if, and only if, the snapshot has changed since the last update date.

```
<snapshotVersion>
  <extension>jar</extension>
  <value>1.0-SNAPSHOT</value>
  <updated>20130711201754</updated>
</snapshotVersion>
```

Other files that may appear:

- `_maven.repositories`: This file records the repository from which the artifact was downloaded. Theoretically, at least for release artifacts, there should not be a difference between an artifact `anyGroup:anyArtifact:anyVersion` downloaded from a repository `RepoA` and the very same artifact downloaded from `RepoB`. The following piece of code says that `commons-beanutils-1.7.0.jar` was downloaded from the repository of which the ID is `central`.

  ```
  commons-beanutils-1.7.0.pom>central=
  commons-beanutils-1.7.0.jar>central=
  ```

- `*.*.lastUpdated` and `resolver-status.properties`: These files indicate Maven attempted and failed to download the archive. In order to save bandwidth, Maven will not try to download the same archive for a given time period. Anyway, you can force the attempt to download with the `-U` or `--update-snapshots` options.

Most of the time, you run Maven on an existing POM file. Anyway, sometimes you may need to run Maven independently of any project. This is the case, for instance, when you run `mvn --help`. In this situation, an implicit POM is called. Technically, it is referred to as `standalone-pom`.

You can download a given artifact and install it to local repository with a standalone command, thanks the plugin and `dependency:get` goal. You can point at the desired artifact by its short designation, for example:

```
mvn dependency:get \
          -Dartifact=org.springframework:spring-core:3.1.0.RELEASE \
          -DrepoUrl=http://repo1.maven.apache.org/maven2
```

The needed artifact will be quietly downloaded and copied at the right place on local file system.

For instance if you attempt to download the artifact `unexistingGroup:unexistingArtifact:1.0` on Maven central in standalone, then a `~/.m2/repository/unexistingGroup/unexistingArtifact/resolver-status.properties` folder and other files will be created, with a content similar to the following:

```
  #NOTE: This is an internal implementation file, its format can be
changed without prior notice.
  #Fri Jul 12 12:20:57 CEST 2013
   maven-metadata-central.xml.lastUpdated=1373624457406
   maven-metadata-central.xml.error=
```

You can notice the date given as a number of milliseconds since 1970 (here it is `1373624457406`) corresponds to the date given in human-readable format (here it is `Fri Jul 12 12:20:57 CEST 2013`).

In the same way, if your POM depends on the same unresolvable dependency (`unexistingGroup:unexistingArtifact:1.0`):

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
           4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>
          com.packt.maven.dependency.chapter2
      </groupId>
      <artifactId>unresolvableArtifact</artifactId>
     <version>1.0-SNAPSHOT</version>
     <name>Example of POM with unresolvable artifacts ; files
         *.*.lastUpdated should be created in local repo
         dependencies
     </name>

     <dependencies>
```

```
        <dependency>
            <groupId>unexistingGroup</groupId>
            <artifactId>unexistingArtifact</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

</project>
```

Then the build will fail with an explicit error message:

```
Failed to execute goal on project unresolvableArtifact: Could not resolve
dependencies for project com.packt.maven.dependency.chapter2:unresolvab
leArtifact:jar:1.0-SNAPSHOT: Failure to find unexistingGroup:unexistingA
rtifact:jar:1.0 in http://repo.maven.apache.org/maven2 was cached in the
local repository, resolution will not be reattempted until the update
interval of central has elapsed or updates are forced
```

And corollary, a folder will be created (`~/.m2/repository/unexistingGroup/unexistingArtifact/1.0`; notice that this is one level under the previous one, corresponding to the version) with two files `unexistingArtifact-1.0.jar.lastUpdated` and `unexistingArtifact-1.0.pom.lastUpdated`, of which content is similar:

```
#NOTE: This is an internal implementation file, its format
  can be changed without prior notice.
#Fri Jul 12 12:28:10 CEST 2013
http\://repo.maven.apache.org/maven2/.lastUpdated=1373624890218
http\://repo.maven.apache.org/maven2/.error=
```

The conveyed information includes the last update date and possibly the error returned by the server.

Summary

Thus, at the end of this second chapter, we are able to:

- Select and set the scope to the best value, among those available
- Use the `dependencyManagement` tag to rationalize the versions and scope with multi-module projects
- Use features and understand the underlying rules guiding Maven Reactor
- Identify, read, and understand the content of files within the local repository

3
Dependency Designation (advanced)

In this chapter, we will see the last tags that define an artifact: `type/packaging` and `classifier`. Then, we will study some plugins that will help us identify and fix conflicts between dependencies of a project or a group of projects. In the end, we will present the dynamic POMs that allow developers to increase the flexibility and strictness of their POMs.

The type tag

The `type` tag in dependencies allows distinguishing the type of archive, that is, most often the file extension. In contrast with `groupId` and `artifactId`, which do not differ from an artifact self-declaration to a dependency reference, the type of archive will resort to two different tags: `packaging` for self-declaration and `type` for dependency reference.

The classic cases

Mostly, the `type` tag hints at the kind of archive or the packaging format. Here are the available types by default: `pom`, `jar` (default value), `maven-plugin`, `ejb`, `war`, `ear`, `rar` (Resource ARchive), `par` (Persistence ARchive, basically a JAR with a `META-INF/persistence.xml` file), and `ejb3`.

Plugins can create their own packaging, and therefore their own packaging types, so this list can be extended, for example, Android's **Application PacKage** (**APK**) file.

Most of the time, the `type` tag you depend on is the `packaging` tag that is declared on dependency POM. To illustrate this, let's consider a classic case; if you are familiar with WAR deployment on WebSphere you will know that IBM's Application Server can deploy EAR but no WAR. This is why people are used to bundling a WAR within an EAR. So, for instance, your EAR's POM will be declared of `packaging ear`, and depend on an artifact of `type war`, refer to the code highlighted in bold:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
           4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
       <parent>
           <groupId>com.packt.dependencyManagement.chapter4
           </groupId>
           <artifactId>exampleWithType</artifactId>
           <version>1.0</version>
      </parent>
      <modelVersion>4.0.0</modelVersion>
      <artifactId>exampleWithTypeEar</artifactId>
      <version>${project.parent.version}</version>
      <packaging>ear</packaging>
      <dependencies>
          <dependency>
              <groupId>${project.groupId}</groupId>
              <artifactId>exampleWithTypeWar</artifactId>
              <version>${project.version}</version>
               <type>war</type>
          </dependency>
      </dependencies>
  </project>
```

Likewise, the WAR's POM will declare a packaging war as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/
         4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.packt.dependencyManagement.chapter4
        </groupId>
        <artifactId>exampleWithType</artifactId>
```

```
        <version>1.0</version>
    </parent>
    <artifactId>exampleWithTypeWar</artifactId>
    <version>${project.parent.version}</version>
    <packaging>war</packaging>
</project>
```

The types available in the `type` and `packaging` tags are more restrictive than those allowed in the `assembly.xml` files; they extend the default types to ZIP, TAR (as well as `tar.bz2` and `tar.gz`), GZIP, and so on.

Creating a new packaging/type

Generally speaking, default packages and types fit your needs. Sometimes, you must have a specific packaging; the cases vary from when only the output artifact extension is original, to when a complete new format (including extension, archive compression, and added descriptors) must be created.

Case study

You need to define a specific packaging format, let's say "Hello World ARchive" with `hwar` as extension. Basically, the `hwar` format is only a ZIP compression with a different name, for instance to discriminate the `hwar` files from other archives and make them follow a particular workflow.

Our study case will rely on a parent POM of identifiers: `com.packt.dependencyManagement.chapter4:specificArchiveExample:pom:1.0`.

The first step – Maven plugin

Create a new project. This project is intended to describe the packaging, the format, and the produced file extension.

The POM will be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <parent>
          <groupId>com.packt.dependencyManagement.chapter4
          </groupId>
```

```
            <artifactId>specificArchiveExample</artifactId>
            <version>1.0</version>
        </parent>
        <modelVersion>4.0.0</modelVersion>

        <artifactId>helloWorldARchivePlugin</artifactId>
        <version>${project.parent.version}</version>
        <!-- We should use maven-plugin ; but since we do not declare
mojos we have to declare packaging jar-->
        <packaging>jar</packaging>

    </project>
```

Contrary to most of plugins, ours will be of packaging `jar` and not `maven-plugin` since it declares no Mojo. In short, a Maven Mojo is an implementation of a plugin goal. In our situation, the plugin shall not be called as a different step of build, but rather as an "over rider" of an existing goal, in order to change the extension of the built file; therefore no Mojo need be associated, and also, the plugin does not fit with a `maven-plugin` packaging declaration.

Let's make a small digression and introduce Codehaus' Plexus project, container and concepts. At first glance, Plexus can be seen as a software stack to write and run Java applications. Like Spring, Plexus is based on **Inversion of Control** (**IoC**) and **Dependency Injection** (**DI**) patterns, and allows to build modular components to be easily combined and reused. Maven is executed within a Plexus container.

In Plexus, a component descriptor gives a map of *roles* and *role implementations*. This map is written as a `component.xml` file.

Actually, Plexus, with plenty of features, is a large and well-documented project, with a complete tutorial on its website, available at `http://plexus.codehaus.org/`.

Then, create a `src/main/resources/META-INF/plexus/components.xml` Plexus file with following content (the code is self-documented):

```
<?xml version="1.0" encoding="UTF-8"?>
  <component-set>
      <!-- An XSD is supposed to be located at
        http://plexus.codehaus.org/xsd/components-1.3.0.xsd,
        but the link is dead. A JIRA ticket has been open, cf
        https://jira.codehaus.org/browse/PLX-469 -->
      <components>
          <component>
```

```
        <role>
            org.apache.maven.lifecycle.mapping.
             LifecycleMapping
        </role>
        <!-- The role-hint: basic role-hints are default
         packaging values, such as pom, jar, ejb, etc.-->
        <role-hint>helloWorldARchive</role-hint>
        <implementation>
            org.apache.maven.lifecycle.mapping.
            DefaultLifecycleMapping
        </implementation>
        <configuration>
            <phases>
                <!--name the usual jar lifecycle bindings,
                  since format hwar is similar to jar. If
                  needed, you can add others-->
                <process-resources>
                    org.apache.maven.plugins:maven-resources-
                    plugin:resources
                </process-resources>
                <package>
                    org.apache.maven.plugins:maven-jar-
                    plugin:jar
                </package>
                <install>
                    org.apache.maven.plugins:maven-install-
                    plugin:install
                </install>
                <deploy>
                    org.apache.maven.plugins:maven-deploy-
                    plugin:deploy
                </deploy>
            </phases>
        </configuration>
    </component>
    <component>
        <!-- Role and role-hint define the identity of the
         component-->
        <role>
            org.apache.maven.artifact.handler.ArtifactHandler
        </role>
        <role-hint>helloWorldARchive</role-hint>
        <implementation>
```

```
                org.apache.maven.artifact.handler.
                DefaultArtifactHandler
            </implementation>
            <configuration>
                <!-- Built files will hold this extension.
                  Besides, dependency will be resolved in
                  considering this extension in Maven local
                  repository.-->
                <extension>hwar</extension>
                <!-- The type to hint when an artifact
                 is depended on -->
                <type>helloWorldARchive</type>
                <!--The packaging in artifacts
                 self-declaration-->
                <packaging>helloWorldARchive</packaging>
            </configuration>
        </component>
    </components>
</component-set>
```

This `component.xml` file can be used almost as is in a majority of situations. Build this artifact and install it in your local repository.

The second step – call the plugin

Create a project, specify the expected packaging (here it is `helloWorldARchive`) and add a tag for the plugin you have just created (`com.packt.dependencyManagement.chapter4:helloWorldARchivePlugin:1.0`):

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
            4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <parent>
          <groupId>com.packt.dependencyManagement.chapter4
          </groupId>
          <artifactId>specificArchiveExample</artifactId>
          <version>1.0</version>
      </parent>

      <modelVersion>4.0.0</modelVersion>
      <artifactId>helloWorldProject</artifactId>
      <version>${project.parent.version}</version>
      <packaging>helloWorldARchive</packaging>
```

```
    <build>
        <plugins>
            <!--Call the plugin that has just been created-->
            <plugin>
                <groupId>
                    com.packt.dependencyManagement.chapter4
                </groupId>
                <artifactId>helloWorldARchivePlugin
                </artifactId>
                <version>1.0</version>
                <!-- These tag and value mean that the plugin
                 will produce a file with a non-standard
                 extension-->
                <extensions>true</extensions>
            </plugin>
        </plugins>
    </build>
</project>
```

Then build this second project. If you have not previously built the preceding plugin at least once, then `helloWorldProject` will fail, because Maven will try to resolve the needed plugin before building it.

You should get an output that contains the following:

```
[INFO] Installing /workarea/development/projects/chapter4/
specificArchiveExample/helloWorldProject/target/helloWorldProject-1.0.jar
to /users/jlalou/.m2/repository/com/packt/dependencyManagement/chapter4/
helloWorldProject/1.0/helloWorldProject-1.0.hwar
```

The preceding output means that a `helloWorldProject-1.0.hwar` artifact with the expected extension `.hwar` has successfully been created and installed in the local repository".

The Classifier

In the previous chapters, we dealt with the main ways to identify a dependency: `groupId`, `artifactId`, `version`, and `scope`. Let's see the final tags that are available.

The `classifier` tag is used to distinguish between *different artifacts that were built from the same POM and source code*. The `classifier` can be any string.

Common use cases when classifiers are specified are as follows:

- Artifacts generated owing to an environment (development, integration, QA, production, and so on)
- Artifacts related to a system, calling native functions, for example, one JAR for Windows, and another of Linux
- Artifacts related to different JDK and JRE levels, for example, one JAR for JDK-1.4 (that is, without annotations, generics, and so on), another for JDK 5, and a last for JDK 8 (with lambda features)
- Artifacts containing different outputs from the same project, for example, archives that are built with Maven Assemblies can be a JAR (with binaries), sources (with Java code), Javadoc (with generated HTML), and so on

When type does not match the packaging

The `type` tag usually represents the archive filename extension; nonetheless, that is not an absolute rule. A `type` tag can be mapped to a different extension, based on a `classifier`. Therefore, sometimes the analysis must be thinner, for instance, how to consider a artifact `ejb-client`? Pieces of clue are in Plexus's `component.xml` file; such an artifact is of *extension* `jar`, *packaging* `ejb`, and *classifier* `client`.

The dependency plugin

The dependency plugin gathers many features that are worth diving into them. `dependency` can be either through this short name, or through its complete designation: `org.apache.maven.plugins:maven-dependency-plugin`.

We have already seen the `tree` goal that allows displaying on standard output the hierarchy of dependencies, as well as the `get` goal that installs manually an archive to the local repository. Let's study a bit more the other features of the `dependency` plugin.

The analyze goal

The `analyze` goal (and `analyze-only`) will analyze the dependencies and determine which of them are used/unused or declared/undeclared.

The `analyze` goal is to be run standalone, whereas `analyze-only` is to be used in the build lifecycle.

Let's consider a POM with such dependencies:

```
<dependencies>
     <dependency>
         <!-- declared and unused -->
         <groupId>directory</groupId>
         <artifactId>apacheds-core</artifactId>
         <version>0.9.3</version>
         <!--implicit dependency to commons-io:
          commons-io:1.0-->
     </dependency>
     <dependency>
        <!-- declared and used -->
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-exec</artifactId>
        <version>2.9.7</version>
        <!--implicit dependency to commons-io:
         commons-io:1.4-->
    </dependency>
    <dependency>
        <!-- declared and unused -->
        <groupId>org.apache.tapestry</groupId>
        <artifactId>tapestry-upload</artifactId>
        <version>5.3.7</version>
        <!--implicit dependency to commons-io:
         commons-io:2.0.1-->
    </dependency>
    <dependency>
        <groupId>com.googlecode.grep4j</groupId>
        <artifactId>grep4j</artifactId>
        <version>1.7.5</version>
        <!--implicit dependency to commons-io:
         commons-io:2.4-->
    </dependency>
```

Let's create a class with the following content:

```
package com.packt.maven.dependency.massiveConflicts;

  import org.apache.camel.component.exec.ExecCommand;
  import org.apache.camel.impl.DefaultCamelContext;
  import org.grep4j.core.GrepExpression;

  public class MassiveConflictsFoo {
      private ExecCommand execCommand;
```

```
        // from camel-core-2.9.7, that is induced by camel-exec
        private DefaultCamelContext defaultCamelContext;
        private GrepExpression grepExpression;
    }
```

Let's run the `dependency:analyze` goal. Among the output, we get the following lines:

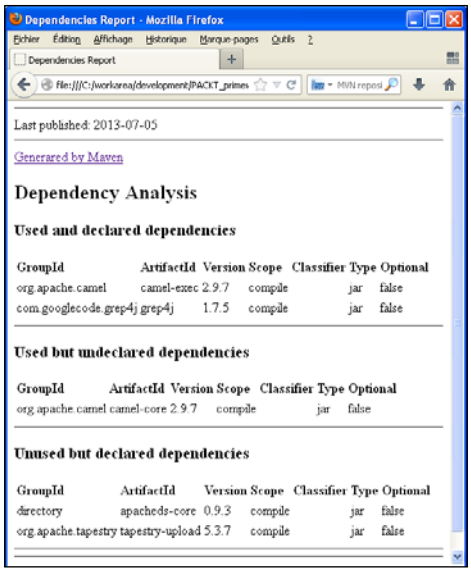
```
[INFO] --- maven-dependency-plugin:2.6:analyze (default-cli)
@ massiveConflicts ---
[WARNING] Used undeclared dependencies found:
[WARNING]    org.apache.camel:camel-core:jar:2.9.7:compile
[WARNING] Unused declared dependencies found:
[WARNING]    directory:apacheds-core:jar:0.9.3:compile
[WARNING]    org.apache.tapestry:tapestry-upload:jar:5.3.7:compile
```

As expected, dependencies to `camel-exec` and `grep4j`, which are used (via the imports of `org.apache.camel.component.exec.ExecCommand` and `org.grep4j.core.GrepExpression`) do not raise any alert. Unlike the import of `org.apache.camel.impl.DefaultCamelContext` from `camel-core-2.9.7` raises a warning. Likewise, dependencies to `apacheds-core` and `tapestry-upload` that are unused although declared, raise a log of level `WARN`.

A close goal is `analyze-report`. In order to use it, the following block must appear in the POM:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>
        maven-dependency-plugin
    </artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <id>copy</id>
            <phase>package</phase>
        </execution>
    </executions>
</plugin>
```

Then, running `mvn dependency:analyze-report` will generate an HTML report, available at the `target/dependency-analysis.html` location. In addition to previous data, this report will add the dependencies which are both declared and used, such as `camel-exec` and `grep4j` in our case:



Report generated by analyze-report

Let's quote two other goals:

- `analyze-dep-mgt`: Check mismatches in the `dependencyManagement` block
- `analyze-duplicate`: Checks and reports the duplicate declared dependencies

Classpath

The `dependency:build-classpath` goal will display the complete classpath of your project, with the concrete locations, that is, most of the time, the path in the Maven local repository. This is very useful if you need to launch your application in standalone.

For example:

```
$ mvn dependency:build-classpath
(...)
[INFO] --- maven-dependency-plugin:2.6:build-classpath (default-cli)
@ massiveConflicts ---
[INFO] Dependencies classpath:
/users/jonathan/.m2/repository/directory/apacheds-core/0.9.3/
apacheds-core-0.9.3.jar;/users/jonathan/.m2/repository/commons-lang/
commons-lang/2.0/commons-lang-2.0.jar;/users/jonathan/.m2/repository/
commons-collections/commons-collections/3.0/commons-collections-
3.0.jar;/users/jonathan/.m2/repository/commons-primitives/commons-
primitives/20041207.202534/commons-primitives-20041207.202534.jar;/users/
jonathan/.m2/repository/regexp/regexp/1.2/regexp-1.2.jar;(...)
```

Other goals of dependency

- Running `mvn dependency:list` will display the same information as `tree`, but removing the hierarchy, for example:

```
$ mvn dependency:list
(...)
[INFO] The following files have been resolved:
[INFO]    com.google.guava:guava:jar:12.0:compile
[INFO]    directory-shared:ldap-common:jar:0.9.3:compile
[INFO]    commons-codec:commons-codec:jar:1.5:compile
(...)
```

- Running `mvn dependency:list-repositories` will display the repositories used for the current build, for example:

```
$ mvn dependency:list-repositories
(...)
 [INFO] Repositories Used by this build:
[INFO]        id: central
      url: http://repo.maven.apache.org/maven2
   layout: default
snapshots: [enabled => false, update => daily]
 releases: [enabled => true, update => daily]
(...)
```

- Running `mvn dependency:copy-dependencies` will copy all the dependencies to a folder (by default, `target/dependency`). A use case is when you need create an archive or a distribution. Plugin `assembly` can do it, too, and in a better way.
- Running `mvn dependency:sources` will download the sources of dependencies when available, for example:

```
$ mvn dependency:sources
(...)
[INFO] --- maven-dependency-plugin:2.6:sources (default-cli) @
massiveConflicts ---
Downloading: http://repo.maven.apache.org/maven2/emma/
emma/2.0.5312/emma-2.0.5312-sources.jar
Downloading: http://repo.maven.apache.org/maven2/regexp/
regexp/1.2/regexp-1.2-sources.jar
(...)
```

```
[INFO]
[INFO] The following files have been resolved:
[INFO]    com.googlecode.grep4j:grep4j:jar:sources:1.7.5:compile
[INFO]    org.hamcrest:hamcrest-all:jar:sources:1.1:compile
(...)
[INFO] The following files have NOT been resolved:
[INFO]    emma:emma:jar:sources:2.0.5312
[INFO]    regexp:regexp:jar:sources:1.2
(...)
```

- Running `purge-local-repository` will clean the local repository. This operation is required in some cases:
    - As part of a continuous integration system (such as Apache Hudson/ Jenkins, and Bamboo)
    - As a periodic cleanse
    - In order to force the update of all dependencies from a "clean" remote repository

Other goals are available but their interest is lesser. These goals are `copy`, `get`, `go-offline`, `properties`, `resolve`, `resolve-plugins`, `unpack`, and `unpack-dependencies`.

Other miscellaneous plugins

Maven's ecosystem is rich, and many plugins allow you to improve your dependency management quality, and to detect potential and actual conflicts. Among them, let's overview two: **Maven Enforce** and **JBoss Tattletale**.

The Enforce plugin

Enforce, or in long form `org.apache.maven.plugins:maven-enforcer-plugin` likes to think it is "The Loving Iron Fist of Maven". Enforcer allows to set up rules that your project and dependencies must abide by. These rules affect the dependencies, classes, packages, Java version, and so on. Let's see a first example, with Maven Enforcer 1.2.

The dependency convergence

Let's consider the following POM (the code is self-documented):

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
            4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
          <groupId>com.packt.dependencyManagement.chapter4
          </groupId>
          <artifactId>head</artifactId>
         <version>1.0</version>
     </parent>
     <artifactId>enforcerDependencyConvergence</artifactId>
     <version>${project.parent.version}</version>
     <name>Example
         * POM with many conflicts because of JAR versions
         * use of plugin 'enforcer' with goal 'enforce' in
         order to detect them, thanks to rule
         'dependencyConvergence'
     </name>

     <dependencies>
         <dependency>
             <groupId>org.apache.pluto</groupId>
             <artifactId>pluto-portal-driver</artifactId>
             <version>1.1.7</version>
             <!--implicit dependency to 'taglibs:standard:1.0.6'
              ->
         </dependency>
         <dependency>
             <groupId>org.grails</groupId>
             <artifactId>grails-web</artifactId>
             <version>2.2.3</version>
             <!-- implicit dependency to 'taglibs:standard:1.1.2'
              -->
         </dependency>

     </dependencies>
     <build>
```

```
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>
                        maven-enforcer-plugin
                    </artifactId>
                    <version>1.2</version>
                    <configuration>
                        <rules>
                            <!-- We will apply the rule of dependency
                             convergence: this rule makes the build
                             fail if two dependencies (either direct
                             or transitive) diverge on the version
                             -->
                            <dependencyConvergence/>
                        </rules>
                    </configuration>
                    <executions>
                        <execution>
                            <goals>
                                <!--We run the goal 'enforce'-->
                                <goal>enforce</goal>
                            </goals>
                            <!-- The goal is called on phase
                             'verify'-->
                            <phase>verify</phase>
                        </execution>
                    </executions>
                </plugin>

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

As specified in the POM, the two dependencies have conflicts of JAR on `taglib:standard`.

The Enforce plugin has been set to be called on each build and makes it fail when two dependencies diverge on their versions. In our case, the output will contain the following piece of text:

```
[WARNING] Rule 0:
 org.apache.maven.plugins.enforcer.DependencyConvergence failed
 with message:
Failed while enforcing releasability the error(s) are [
```

```
Dependency convergence error for taglibs:standard:1.0.6 paths
to dependency are:
+-com.packt.dependencyManagement.chapter4:enforcerUseCase:1.0
  +-org.apache.pluto:pluto-portal-driver:1.1.7
    +-taglibs:standard:1.0.6
and
+-com.packt.dependencyManagement.chapter4:enforcerUseCase:1.0
  +-org.grails:grails-web:2.2.3
   +-taglibs:standard:1.1.2
```

The Maven Enforce allows to prevent a build on which dependency conflicts are detected.

Banned dependencies

A classic case you encounter is when you depend on a certain artifact, of which a unique version raises problems, while all other are accepted. At a given time of your project, the problematic version is not used, but by the game a natural upgrade of versions and transitive dependencies, you can fear the "bad" version may emerge again. In the same way, sometimes all versions are problematic; meanwhile a unique version is acceptable. In this situations, you can resort to the Maven Enforcer's rule of `<bannedDependencies/>`.

Let's consider the previous example. There are transitive dependencies to `jstl` and `commons-io`. Let's assume `javax.servlet:jstl:1.0.6` is the unique version of `jstl` that raises problem, and that only `commons-io:commons-io:2.1` does work, excluding all other versions. Then the `<rule>` block will be (the rest of the POM does not differ from the previous one):

```
<rules>
    <bannedDependencies>
        <excludes>
            <!-- All versions of commons-io are banned-->
            <exclude>commons-io:commons-io</exclude>
    <!-- Ban only version 1.0.6 of JSTL. The brackets *are*
    needed ; otherwise, Maven would ban versions
    "1.0.6 and above" -->
            <exclude>javax.servlet:jstl:[1.0.6]</exclude>
       </excludes>
       <includes>
           <!--Version 2.0.1 of commons-io
           is explicitly excluded from ban-->
           <include>commons-io:commons-io:2.0.1</include>
```

```
    <!--no use to add explicitly the allowed version of JSTL-->
            </includes>
        </bannedDependencies>
     </rules>
```

And the build will fail with the following error:

```
[WARNING] Rule 0: org.apache.maven.plugins.enforcer.BannedDependencies
failed with message:
Found Banned Dependency: commons-io:commons-io:jar:1.3.1
Found Banned Dependency: javax.servlet:jstl:jar:1.0.6
```

Fixing the issue is very simple; identify the branch and add exclusions. In our case, only add a block of exclusions within dependency to `org.apache.pluto:pluto-portal-driver:1.1.7`:

```
     <exclusions>
         <exclusion>
             <groupId>commons-io</groupId>
             <artifactId>commons-io</artifactId>
         </exclusion>
         <exclusion>
             <groupId>javax.servlet</groupId>
             <artifactId>jstl</artifactId>
         </exclusion>
     </exclusions>
```

Banned dependencies can be specified with wildcards, and therefore groups of exclusions can be set: not only by `groupId`, `artifactId`, `version`, and so on, but also by `type` and `scope`. This way, you can ban "all artifacts of type JAR" (`*:*:jar`) or "all artifacts with scope `runtime` or `provided`" (`*:*:runtime:provided`).

Other rules

The Enforcer plugin defines other rules. Considering the same POM as earlier, the following rules illustrate the features offered by Maven Enforcer:

- Always passes; useful to test plugin configuration. The opposite rule is `<alwaysFail/>`:

```
<alwaysPass/>
```

- Equivalent to `<bannedDependencies>` for plugins:

```
<bannedPlugins>
  <excludes>
    <!-- Here we ban the dependency to JBoss plugin-->
```

```
        <exclude>
         org.codehaus.mojo:jboss-maven-plugin:1.5.0
        </exclude>
    </excludes>
</bannedPlugins>
```

- Bans transitive dependencies:

```
<banTransitiveDependencies>
  <excludes>
    <!-- Here we ban the dependencies on SNAPSHOTs -->
       <exclude>
         *:*:*SNAPSHOT
       </exclude>
  </excludes>
</banTransitiveDependencies>
```

- Evaluates a beanshell script (simple Boolean expression or complete script with a Boolean returned value).

  For more details on BeanShell syntax and commands, refer to `http://www.beanshell.org/`.

```
<evaluateBeanshell>
  <!-- Here we ensure both this project and its parent
    POM share the same groupId and version-->
     <condition>
        ("${project.groupId}".
         equals("${project.parent.groupId}"))
            @and
             (${project.version}==
             ${project.parent.version})
     </condition>
        <message>Build failed because
            the beanshell returned
           'false'
       </message>
   </evaluateBeanshell>
```

- Checks the existence of an environment variable:

```
<requireEnvironmentVariable>
    <!--Here we check JAVA_HOME and M2_HOME
     exist and have been set-->
    <variableName>JAVA_HOME</variableName >
    <variableName>M2_HOME</variableName >
</requireEnvironmentVariable>
```

- Checks whether the files in the list of files do not exist:

```
<requireFilesDontExist>
  <files>
    <!-- Here we check the file ./helloWorld.txt does
     *not* exist -->
     <file>
        ${project.basedir}/helloWorld.txt
     </file>
  </files>
</requireFilesDontExist>
```

- Checks whether the files in the list of files do exist:

```
<requireFilesExist>
  <files>
    <!-- Here we check the file pom.xml *does* exist
     -->
      <file>
        ${project.basedir}/pom.xml
       </file>
  </files>
</requireFilesExist>
```

- Enforces that the list of files exist and are within a certain size range:

```
 <requireFilesSize>
     <maxsize>1000000</maxsize>
     <minsize>1</minsize>
     <files>
         <file>
             ${project.basedir}/pom.xml
         </file>
     </files>
 </requireFilesSize>
```

- Enforces the JDK version:

```
<requireJavaVersion>
  <!--Here we require JDK 5, 6 or 7
   -->
  <version>[1.5,)</version>
</requireJavaVersion>
```

- Enforces the Maven version:

```
<requireMavenVersion>
  <!--Here we require Maven version to be between 2.2 (included)
   and 3.1 (excluded)-->
  <version>[2.2,3.1)</version>
</requireMavenVersion>
```

- Bans repositories inclusion. Defining repositories in `pom.xml` is not advised; best practices consist in using a repository manager:

```
<requireNoRepositories/>
```

- Enforces the OS and/or CPU architecture. Available parameters include name, family, arch, and version:

```
<requireOS>
   <!-- Here we ensure we run on an 86 platform-->
   <arch>x86</arch>
</requireOS>
```

- Checks a property is set:

```
<requireProperty>
  <!-- Here we check a version, a type and a name
   have been set-->
  <property>project.version</property>
  <property>project.type</property>
  <property>project.name</property>
</requireProperty>
```

- Checks if no `SNAPSHOT` is included as a dependency:

```
<requireReleaseDeps/>
```

- Checks that the current project is not a snapshot.

```
<requireReleaseVersion/>
```

Let's mention other existing rules:

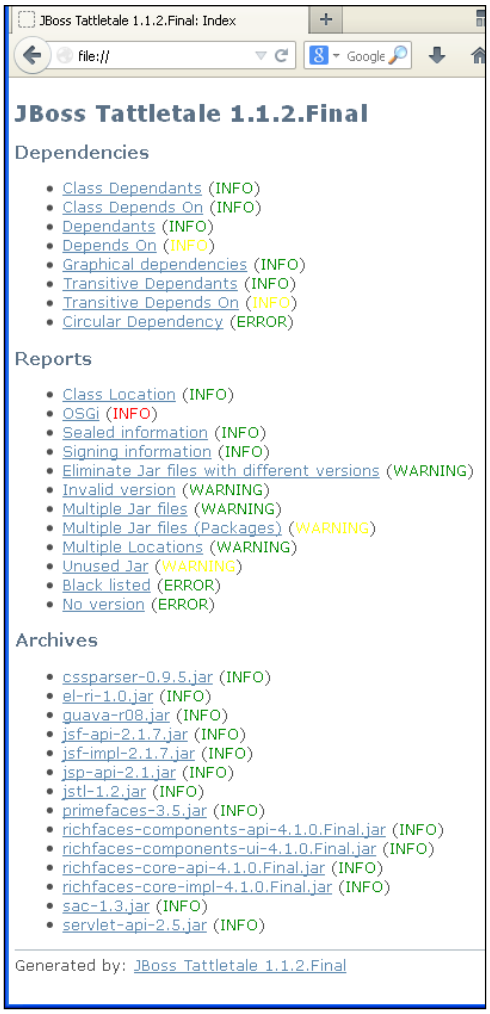
- `requireActiveProfile`: Checks if one or more profiles are active
- `requireSameVersions`: Checks that specific dependencies and/or plugins are consistent and share the same version
- `requireUpperBoundDeps`: Checks all transitive dependencies are resolved to their specified version or higher

Finally, `maven-enforcer-rule-api` provides an API allowing you to implement your own rules, extending `org.apache.maven.enforcer.rule.api.EnforcerRule`.

Tattletale

Tattletale (last Versions: 1.1.2 as stable; 1.2.0.Beta2 on 1.2.X branch) is a tool provided and supported by JBoss. It provides analyzes related to a project, its dependencies and classes, albeit more focused on JARs than on other types. Some features of Tattletale are already available through Maven `dependency` and `enforce` plugins.

Tattletale generates a tree of reports. In the general summary, the information level (`INFO`, `WARNING`, and `ERROR`) is indicated.



Dependencies

The first set deals with the following dependencies:

- `Class Dependants`: It lists the classes that depend on the considered class.
- `Class Depends On`: It lists the classes on which the considered class does depend on. Basically, this is a summary of imports.
- `Dependants`: It lists the archives on which the considered archive is depended on.
- `Depends On`: It lists the archive on which the considered archive does depend on.
- `Graphical dependencies`: It generates a GraphViz `.dot` file (nothing to see with Word model document).
- `Transitive Dependants`: It is same as `Class Dependants` and includes transitive dependencies.
- `Transitive Depends On`: It is same as `Class Depends On` and includes transitive dependencies.
- `Circular Dependency`: It lists the circular dependencies.

Reports

The second set deals with various reports:

- `Class Location`: It gives, for each archive, the list of included classes.
- `OSGi`: It tells which archive is OSGi enabled or not. In other terms, this report checks the content of the `MANIFEST` files.
- `Sealed information`: It tells which archive is or is not sealed. In other terms, this report checks whether the flag Sealed is set at `true` in the `MANIFEST` files.

Theoretically, sealed JARs guarantee all the classes of a package, (which is consequently said to be sealed) are included within the same JAR. A classic use case is when some of your classes have package-protected methods. With non-sealed JARs, nothing prevents another developer to declare a class in the same package and then access the package-protected methods and fields. Unlike, with sealed JARs, the JVM will refuse to run the code of two classes with the same package but in two different JARs. Anyway, beware the seal security is *low* and can be bypassed with very few efforts. For example, in `org.jboss.threads:jboss-threads:2.0.0.GA, MANIFEST.MF` file declares `org/jboss/threads` as sealed package. Hence, your IDE will deny you from create a class with such a package name. Yet, if you have created it from anywhere else, the project will compile.

- `Signing information`: It tells which archive is or is not signed.
- `Eliminate Jar files with different versions`: It lists the archive with the same `groupId/artifactId/type/classifier` and different versions, which are source of JAR conflicts.
- `Invalid version`: This report is related to OSGi enability. It lists the archive of which OSGi version identifier is absent or invalid.
- `Multiple Jar file`: It lists the classes that appear in different archives, and the archives where these classes appear.
- `Multiple Jar files (packages)`: It is same as `Multiple Jar file` however it includes packages.
- `Multiple Locations`: It lists the different locations of archives that appear more than once.
- `Unused Jar`: It lists the archives of which none of the content is imported anywhere. Be careful! Some classes and archive can be used in other ways out of canonical imports, such as through the Java reflection API or other dynamic instantiation.
- `Black listed`: It lists the archives that use black listed APIs.

Black-listed APIs

Some frameworks or domains impose limitations on the Java APIs that can be called, known as **white-listed APIs**. For instance, Google AppEngine or Android do not accept multithreading or Java Reflection. Such APIs, by symmetry with *white-list* APIs are designed as *black-listed* APIs.

- `No version`: It lists the archives without a version identifier.
- `JBoss AS7` (optionally): It lists the `jboss-deployment-structure.xml` file for archives

Archives

This sequence of report provides, for each JAR, WAR, and EAR, a map of values corresponding to the following keys: `Name`, `Class Version`, `Locations`, `Profiles`, `Manifest`, `Signing information`, `Requires` (list of imports), `Provides` (list of classes included within the archive, and their `SerialVersionUID` if present).

Dependency, enforce, and tattletale – conclusion

Thanks to `dependency`, `enforce`, and `tattletate` plugins, Maven features plenty of powerful tools that allow managing dependencies, and identifying and fixing the potential conflicts and cycles.

Anyway, the three plugins should be considered as complementary: while Tattletale is "report-oriented" (that means its purpose is to generate reports as support for analyze by developers), Enforce is "build-breaking-oriented"; it ensures a sequence of rules to be checked on each run, and makes the build failed as soon as a rule is transgressed, without any human monitoring or intervention.

Dynamic POMs and dependencies

Usually, POMs are written as XML files. However, never forget that XML is a representation (written, and hierarchical) of an object data structure. POMs might have been written in YaML, or even as Java POJOs. The actual POM is not the `pom.xml` file, but the intelligent object behind, of which a projection is `pom.xml`. This is why there is nothing amazing that a POM can or cannot be isomorphic to the XML file representing it.

This brings us into concepts, advanced and non-intuitive, such as effective POM and dynamic POM. The same line of reasoning can be made for the `settings.xml` file.

Effective POM and settings

The POM that is actually executed by Maven is not the one a human being can write and read in the `pom.xml` file. Maven adds blocks corresponding to references of the `settings.xml` files, such as repositories and mirrors, and default values and plugins, in more of the dependencies, either explicitly declared or resolved through transitivity, inheritance and interpolation.

Let's consider the simplest possible POM:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/
           4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>
```

```
        com.packt.dependencyManagement.chapter4
      </groupId>
      <artifactId>simplestPom</artifactId>
     <version>1.0</version>
     <!--nothing more, nothing less than a minimal declaration-->
</project>
```

By executing the `help:effective-pom` goal, you can see the actual POM, as viewed by Maven (refer the `chapter3/simplestPom/effective-pom.xml` file in the code bundle linked to this chapter).

Several blocks emerge as follows:

- The `repositories` (by default: Maven Central, `http://repo.maven.apache.org/maven2`) and `pluginRepositories` blocks (lines 9-30), from the `settings.xml` file and/or Maven's default values.
- The actual folders for `src`, `test`, `resources`, `output`, and so on (lines 33-65 and 268-272).
- The `plugins` and `pluginManagement` (lines 90-266) block. This block explains why you can simply write `mvn clean install` instead of `mvn maven-clean-plugin:clean maven-install-plugin:install` whereas you must run the complete command for less widespread plugins. Plugins available by default are:
    - `maven-antrun-plugin`
    - `maven-assembly-plugin`
    - `maven-dependency-plugin`
    - `maven-release-plugin`
    - `maven-clean-plugin`
    - `maven-install-plugin`
    - `maven-resources-plugin`
    - `maven-surefire-plugin`
    - `maven-compiler-plugin`
    - `maven-jar-plugin`
    - `maven-deploy-plugin`
    - `maven-site-plugin`
    - `maven-project-info-reports-plugin`

Similarly, you can display the effective settings used by Maven by running the `help:effective-settings` goal. For instance, for the minimal `settings.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
  <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/
            SETTINGS/1.0.0 http://maven.apache.org/xsd/
            settings-1.0.0.xsd">
</settings>
```

The actual `settings.xml` is longer (yet less verbose than the effective POM):

```
  <settings xmlns="http://maven.apache.org/POM/4.0.0"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/
            SETTINGS/1.1.0 http://maven.apache.org/xsd/
            settings-1.1.0.xsd">
      <localRepository
              xmlns="http://maven.apache.org/SETTINGS/1.1.0">
          /users/jonathan/.m2/repository
      </localRepository>
      <pluginGroups
              xmlns="http://maven.apache.org/SETTINGS/1.1.0">
         <pluginGroup>org.apache.maven.plugins</pluginGroup>
         <pluginGroup>org.codehaus.mojo</pluginGroup>
      </pluginGroups>
  </settings>
```

Two blocks emerge: the path to the local repository (corresponding to `$HOME/.m2/repository`) and the default `groupId` of plugins that do not specify it; in other terms, if a plugin of `artifactId fooPlugin` (without more information on `groupId`) is referenced, Maven will try to resolve it as `org.apache.maven.plugins:fooPlugin` and `org.codehaus.mojo:fooPlugin`.

Knowing the effective POM and settings may help you to detect conflicts.

Dynamic POM

As a disclaimer, beware the following example is used for its pedagogical interest and may fit some situations, but does not match best practices for many other projects. Among other theoretical and practical reasons, common IDEs have some difficulties to support full dynamic POMs.

Case study

Our project meets the following requirements:

- It depends on `org.codehaus.jedi:jedi-XXX:3.0.5`. Actually, the `XXX` is related to the JDK version, that is, either `jdk5` or `jdk6`.
- The project is built and run on three different environments: `PRODuction`, `UAT`, and `DEVelopment`
- The underlying database differs owing to the environment: PostGre in `PROD`, MySQL in `UAT`, and HSQLDB in `DEV`.
- Besides, the connection is set in a Spring file, which can be `spring-PROD.xml`, `spring-UAT.xml`, or `spring-DEV.xml`, all being in the same `src/main/resource` folder.

The first bullet point can be easily answered, using a `jdk-version` property.

The dependency is then declared as follows:

```
<dependency>
    <groupId>org.codehaus.jedi</groupId>
    <!--For this dependency two artifacts are available,
     one for jdk5 or and a second for jdk6-->
    <artifactId>jedi-${jdk.version}</artifactId>
    <version>${jedi.version}</version>
</dependency>
```

Still, the fourth bullet point is resolved by specifying a `resource` folder:

```
<resources>
     <resource>
         <directory>src/main/resource</directory>
         <!--include the XML files corresponding to
          the environment: PROD, UAT, DEV. Here, the
          only XML file is a Spring configuration one
          . There is one file per environment-->
         <includes>
             <include>
                 **/*-${environment}.xml
             </include>
         </includes>
    </resource>
</resources>
```

Then, we will have to run Maven adding the property values using one of the following commands:

- `mvn clean install -Denvironment=PROD -Djdk.version=jdk6`
- `mvn clean install -Denvironment=DEV -Djdk.version=jdk5`

By the way, we could have merged the three XML files as a unique one, setting dynamically the content thanks to Maven's `filter` tag and mechanism.

The next point to solve is the dependency to actual JDBC drivers.

A quick and dirty solution

A quick and dirty solution is to mention the three dependencies:

```
<!--PROD -->
<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.1-901.jdbc4</version>
    <scope>runtime</scope>
</dependency>
<!--UAT-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.25</version>
    <scope>runtime</scope>
</dependency>
<!--DEV-->
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.3.0</version>
    <scope>runtime</scope>
</dependency>
```

Anyway, this idea has drawbacks. Even though only the actual driver (`org.postgresql.Driver`, `com.mysql.jdbc.Driver`, or `org.hsqldb.jdbcDriver` as described in the Spring files) will be instantiated at runtime, the three JARs will be transitively transmitted—and possibly packaged—in a further distribution.

You may argue that we can work around this problem in most of situations, by confining the scope to `provided`, and embed the actual dependency by any other mean (such as rely on an artifact embarked in an application server); however, even then you should concede the dirtiness of the process.

A clean solution

Better solutions consist in using dynamic POM. Here, too, there will be a gradient of more or less clean solutions.

Once more, as a disclaimer, beware of dynamic POMs! Dynamic POMs are a powerful and tricky feature of Maven. Moreover, modern IDEs manage dynamic POMs better than a few years ago. Yet, their use may be dangerous for newcomers: as with generated code and AOP for instance, what you write is not what you execute, which may result in strange or unexpected behaviors, needing long hours of debug and an aspirin tablet for the headache. This is why you have to carefully weigh their interest, relatively to your project before introducing them.

With properties in command lines

As a first step, let's define the dependency as follows:

```
<!-- The dependency to effective JDBC drivers: PostGre,
 MySQL or HSQLDB-->
<dependency>
    <groupId>${effective.groupId}</groupId>
    <artifactId>
        ${effective.artifactId}
    </artifactId>
    <version>${effective.version}</version>
</dependency>
```

As you can see, the dependency is parameterized thanks to three properties: `effective.groupId`, `effective.artifactId`, and `effective.version`. Then, in the same way we added earlier the `-Djdk.version` property, we will have to add those properties in the command line, for example,:

```
mvn clean install -Denvironment=PROD -Djdk.version=jdk6 \
          -Deffective.groupId=postgresql \
          -Deffective.artifactId=postgresql \
          -Deffective.version=9.1-901.jdbc4
```

Or add the following property

```
mvn clean install -Denvironment=DEV -Djdk.version=jdk5 \
          -Deffective.groupId=org.hsqldb \
          -Deffective.artifactId=hsqldb \
          -Deffective.version=2.3.0
```

Then, the effective POM will be reconstructed by Maven, and include the right dependencies:

```
<dependencies>
     <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-core</artifactId>
         <version>3.2.3.RELEASE</version>
         <scope>compile</scope>
     </dependency>
     <dependency>
         <groupId>org.codehaus.jedi</groupId>
        <artifactId>jedi-jdk6</artifactId>
        <version>3.0.5</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>9.1-901.jdbc4</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Yet, as you can imagine, writing long command lines like the preceding one increases the risks of human error, all the more that such lines are "write-only". These pitfalls are solved by profiles.

Profiles and settings

As an easy improvement, you can define **profiles** within the POM itself. The profiles gather the information you previously wrote in the command line, for example:

```
<profile>
     <!-- The profile PROD gathers the properties related
       to the environment PROD-->
     <id>PROD</id>
```

```
        <properties>
            <environment>PROD</environment>
            <effective.groupId>
                postgresql
            </effective.groupId>
            <effective.artifactId>
               postgresql
            </effective.artifactId>
            <effective.version>
                9.1-901.jdbc4
            </effective.version>
            <jdk.version>jdk6</jdk.version>
        </properties>
        <activation>
            <!-- This profile is activated by default:
              in other terms, if no other profile in
              activated, then PROD will be-->
            <activeByDefault>true</activeByDefault>
        </activation>
    </profile>
```

Or:

```
    <profile>
        <!-- The profile DEV gathers the properties
         related to the environment DEV-->
        <id>DEV</id>
        <properties>
            <environment>DEV</environment>
            <effective.groupId>
                org.hsqldb
            </effective.groupId>
            <effective.artifactId>
               hsqldb
            </effective.artifactId>
            <effective.version>
                2.3.0
            </effective.version>
            <jdk.version>jdk5</jdk.version>
        </properties>
        <activation>
            <!-- The profile DEV will be activated if,
              and only if, it is explicitly called-->
            <activeByDefault>false</activeByDefault>
        </activation>
    </profile>
```

The corresponding command lines will be shorter:

```
mvn clean install
```

(Equivalent to `mvn clean install -PPROD`)

Or:

```
mvn clean install -PDEV
```

You can list several profiles in the same POM, and one, many or all of them may be enabled or disabled.

Nonetheless, multiplying profiles and properties hurts the readability. Moreover, if your team has 20 developers, then each developer will have to deal with 20 blocks of profiles, out of which 19 are completely irrelevant for him/her. So, in order to make the thing smoother, a best practice is to extract the profiles and inset them in the personal `settings.xml` files, with the same information:

```
<?xml version="1.0" encoding="UTF-8"?>
  <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/
                 SETTINGS/1.0.0 http://maven.apache.org/xsd/
                 settings-1.0.0.xsd">
      <profiles>
          <profile>
              <id>PROD</id>
              <properties>
                  <environment>PROD</environment>
                 <effective.groupId>
                     postgresql
                 </effective.groupId>
                 <effective.artifactId>
                     postgresql
                 </effective.artifactId>
                 <effective.version>
                     9.1-901.jdbc4
                 </effective.version>
                 <jdk.version>jdk6</jdk.version>
              </properties>
              <activation>
                  <activeByDefault>true</activeByDefault>
              </activation>
          </profile>
      </profiles>
</settings>
```

Dynamic POMs – conclusion

As a conclusion, the best practice concerning dynamic POMs is to parameterize the needed fields within the POM. Then, by order of priority:

- Set an enabled profile and corresponding properties within the `settings.xml`.

```
mvn <goals>  \
           [-f <pom_Without_Profiles.xml> \]
           [-s <settings_With_Enabled_Profile.xml>]
```

- Otherwise, include profiles and properties within the POM

```
mvn <goals> \
           [-f <pom_With_Profiles.xml> \]
           [-P<actual_Profile> \]
           [-s <settings_Without_Profile.xml>]
```

- Otherwise, launch Maven with the properties in command lines

```
mvn <goals> \
            [-f <pom_Without_Profiles.xml> \]
            [-s <settings_Without_Profile.xml>]
            -D<property_1>=<value_1> \
            -D<property_2>=<value_2> \
(...)
            -D<property_n>=<value_n>
```

Summary

Thus, at the end of this chapter, we are able to:

- Use the `type/packaging` and `classifier` tags, to make the difference between two artifacts with the same `groupId/artifactId/version`
- Create a new type
- Use various plugins to analyze our dependencies and detect early potential problems, and then fix them
- Set up POMs, dynamically owing to environment, properties, profiles, and so on

4
Migration of Dependencies to Apache Maven

Starting a new project from scratch is, from many viewpoints, the best situation: you can introduce Maven with best practices from the beginning. Anyway, often you receive a project which is built on another system: Apache Ant or even an Eclipse configuration (the `.classpath`, `.project`, and `.launch` files). In such a situation, migrating to Maven is perfectly mastered, provided you follow a rigorous process.

Case study

Let's consider a project, built thanks to an Ant `build.xml` file and targets. Let's have a look and comment what we can read:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<project name="ProjectFromAnt" default="generateJar">
  <description>
    Example of project to migrate from Ant to Maven 3.
    The main target is generateJar, that compiles the
    sources and compress them as a Java ARchive.
  </description>
```

Let's set some properties which are listed as follows:

- Folder for Java sources:

```
<property name="src"
          location="src"/>
```

- Folder for unit tests:

```
<property name="test"
          location="test"/>
```

- Folder for dependencies:

```
<property name="libdir"
          location="lib"/>
<property name="classesdir"
          value="target/classes"/>
<property name="final.name"
          value="projectFromAnt"/>
<property name="defaulttargetdir"
          value="target"/>
<property name="testclassesdir"
          value="target/test-classes"/>
<property name="testclassesdir"
          value="target/test-classes"/>
```

- Gives the dependencies needed to compile; there is no easy equivalent to different scopes, such as `provided` and `runtime`:

```
<path id="build.classpath">
  <fileset dir="${libdir}">
```

- This JAR contains a **POM**:

```
    <include name="commons-lang-2.6.jar"/>
```

- This JAR does not contain any POM. But `Manifest.MF` is included. Moreover, you can find it in most of public Maven repos:

```
    <include name="commons-cli-1.0.jar"/>
```

- This JAR includes neither a POM nor a relevant `Manifest.MF`. Besides, you have to download it manually:

```
    <include name="ftp4j-1.7.2.jar"/>
    <!-- May have summed up as:
    <include name="**/*.jar"/>
    -->
  </fileset>
</path>
```

- Target to clean build folders:

```
<target name="clean" description="Cleans folders">
  <delete dir="${classesdir}"/>
```

```
  <delete dir="${defaulttargetdir}"/>
</target>
```

- Target to compile the sources:

```
<target name="compile"
        description="Compiles the code"
        depends="clean">
  <mkdir dir="${classesdir}"/>
  <javac destdir="${classesdir}"
         srcdir="${src}"
         classpathref="build.classpath"/>
</target>
```

- Target to compress the `*.class` files as a JAR archive:

```
<target name="generateJar"
        description="Create the jar"
        depends="compile">
  <jar jarfile="${defaulttargetdir}/${final.name}.jar"
       basedir="${classesdir}"/>
</target>
```

- Target to compile the test sources:

```
<target name="compile-tests" depends="compile">
  <delete dir="${testclassesdir}"/>
  <mkdir dir="${testclassesdir}"/>
  <javac destdir="${testclassesdir}">
```

- Folder for unit tests sources:

```
    <src>
      <pathelement location="${test}"/>
    </src>
    <classpath>
      <path refid="build.classpath"/>
      <pathelement path="${classesdir}"/>
    </classpath>
  </javac>
</target>
```

- Runs the unit tests:

```
<target name="run-unit-test"
        depends="compile-tests"
        description="runs the unit tests">
  <junit dir="./"
```

```
            failureproperty="test.failure">
        <classpath>
          <path refid="build.classpath"/>
          <pathelement path="${testclassesdir}"/>
          <pathelement path="${classesdir}"/>
```

- JUnit archive is added a dependency for tests; you can consider it as an equivalent to `<scope>test</scope>`:

```
          <fileset dir="lib">
            <include name="junit-4.11.jar"/>
          </fileset>
        </classpath>
      </junit>
    </target>
  </project>
```

In this example, we used JARs with their version numbers in suffix; but in real life, most of the time, you find JARs named `commons-cli.jar`, `junit.jar`, or `ftp4j.jar`, rather than `commons-cli-1.0.jar`, `junit-4.11.jar`, or `ftp4j-1.7.2.jar`.

The Eclipse `.classpath` equivalent file is similar to the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src"
                  path="src"/>
  <classpathentry kind="src"
                  path="test"/>
  <classpathentry kind="con"
                  path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry exported="true"
                  kind="lib"
                  path="lib/commons-cli-1.0.jar"/>
  <classpathentry exported="true"
                  kind="lib"
                  path="lib/commons-lang-2.6.jar"/>
  <classpathentry exported="true" kind="lib"
                  path="lib/ftp4j-1.7.2.jar"/>
  <classpathentry kind="lib"
```

```
                    path="lib/junit-4.11.jar"/>
  <classpathentry kind="output"
                    path="bin"/>
</classpath>
```

Our task is to migrate this project, and its underlying dependencies, to Maven. The scope of this work does not include the migration of any script to Maven. So, at the end of this chapter, we expect to hold a state-of-the-art `pom.xml` file.

The project relies on three JARs to compile, one more for JUnit.

Setting the folders

First of all, set the folders: `src` for project sources and `test` for unit tests:

```
<build>
  <sourceDirectory>src</sourceDirectory>
  <testSourceDirectory>test</testSourceDirectory>
</build>
```

Introducing Maven with standard libraries

The second step is to set the dependencies. External artifacts can be divided into two categories:

- Those which "respect/follow" Maven conventions. In 2013, the majority of de facto standard frameworks and libraries, most of which are open source, are built and published through Maven: Spring, Hibernate, and so on.
- Those which do not "respect/follow" Maven conventions. Although this case is less common, many projects remain reluctant to share Maven rigidity. Among them, let's mention former Sun JARs, Apache Tomcat, several libraries from Google, and so on.

So, you will have to review all the dependencies of `build.xml` and check whether they respect `n-uplet groupId/artifactId/version` (in more of `type`, `classifier`, and so on).

Available POM

Let's consider the dependency to `commons-lang-2.6.jar`. If you open the JAR, you will find a POM in the folder `commons-lang-2.6.jar!/META-INF/maven/commons-lang/commons-lang`. This POM will acquaint you with the complete project, among which is the data you are investigating for the block:

```
<dependency>
  <groupId>commons-lang</groupId>
  <artifactId>commons-lang</artifactId>
  <version>2.6</version>
</dependency>
```

A simple copy and paste to your POM, and the game is over for this dependency.

Unavailable POM

The previous situation is the best you can happen to encounter. Let's see how to manage with unfavorable cases.

Disclosing information from Manifest.MF

Identifying the right version may be a useful tactic. Open `Manifest.MF` within `commons-cli-1.0.jar!/META-INF/`:

```
Manifest-Version: 1.0
Created-By: Apache Jakarta Maven
Built-By: hen
Package: org.apache.commons.*
Name: org.apache.commons.*
Specification-Title: commons-cli
Specification-Version: 1.0
Specification-Vendor: Apache Software Foundation
Implementation-Title: org.apache.commons.*
Implementation-Version: 1.0
Implementation-Vendor: Apache Software Foundation
```

You can conclude the corresponding `Version` is `1.0`.

Online tools

Let's consider the dependency to `commons-cli-1.0.jar`. You have the name of the archive, but neither the `groupId` nor the `artifactId` attributes (which may differ from the archive name). You should use tools such as **MvnRepository** (`http://mavenrepository.com/`), **TheCentralRepository** (`http://search.maven.org/#browse`), **findJAR** (`http://www.findjar.com/`), and so on.

For `commons-cli`, with version 1.0, MvnRepository gives the following `groupId` and `artifactId`:

```
<dependency>
  <groupId>commons-cli</groupId>
  <artifactId>commons-cli</artifactId>
  <version>1.0</version>
</dependency>
```

As an alternative, you can decompress the JAR and read the full qualified name of the classes, for example, `org.apache.commons.cli.GnuParser`. Then, a quick search on findJAR retrieves the corresponding archive:

You can add MvnRepository and findJAR search plugins within Mozilla Firefox, using the URLs `https://addons.mozilla.org/fr/firefox/addon/mvnrepository-search/` and `https://addons.mozilla.org/fr/firefox/addon/search-with-findjarcom/` respectively.

Checksums

The third dependency you have to identify is that of JUnit. Thanks to MvnRepository, you can guess the `groupId` and `artifactId` attributes. Yet, no piece of information related to the version is available within the JAR itself (no POM and nothing relevant in `Manifest.MF`). So, how to distinguish between JUnit 4.11 from 4.0 or even 3.8.1?

You will need a bit of savvy and insight. You can rely on the archive size, or, better, compare the checksum of the archive you hold against those provided in a website such as TheCentralRepository.

Besides, JUnit is used only in unit tests, so obviously its scope will be `test`.

At last, the dependency is:

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

Next steps

If needed, the natural next step is to upload the three JARs onto a remote repository, like Artifactory or Nexus.

If this is not possible, because the organization you belong to does not have any central repository or any other reason, then think of installing the JARs on your local repository. The command is as follows:

```
$ mvn install:install-file \
  -DgroupId= commons-lang -DartifactId= commons-lang -Dversion=2.6 \
  -Dpackaging=jar -Dfile=/complete/path/to/commons-lang-2.6.jar
```

Non-Maven standard libraries

Some of the non-Maven standard libraries are given in the following section.

State

At this point, you can delete the three JARs from the `lib` folder, and replace them with three dependencies in your POM. The last point to deal with is `ftp4j-1.7.2.jar`. To build this study case, the JAR was extracted from a ZIP file downloaded from the Sauron Software website: `http://www.sauronsoftware.it/projects/ftp4j/`. These ZIP and JAR files include neither POM nor other relevant information. Moreover, `ftp4j-1.7.2.jar` does not appear among archives known by popular websites.

You can decide on an arbitrary designation, such as `sauron:ftp4j:jar:1.7.2`, and then install on your local repository thanks to the `mvn install-install-file` command. But a main drawback of purely local installs of JARs that are not equivalent to a remote one is that any new developer integrating with the team will have to perform such operations, as well as any team member, each and every time he or she deletes his or her local repository. Besides, this will prevent from abiding by the best practice of building the release versions in a continuous integration environment, purging its own local repository before each build.

So, how to deal with this "uncommon" and "exotic" JAR?

Quick and (very) dirty

The quickest way is to add the JAR as a dependency, declared with scope `system`:

```
<!--Quick and dirty:-->
<dependency>
  <groupId>unknownGroupId</groupId>
  <artifactId>ftp4j</artifactId>
  <version>1.7.2</version>
  <scope>system</scope>
  <systemPath>
    ${project.basedir}/lib/ftp4j-1.7.2.jar
  </systemPath>
</dependency>
```

Please notice that we decided to set the `version` attribute to `1.7.2`, but rigorously, we should have set it at `UNKNOWN` or `unknownVersion`.

The disadvantage of this dirty solution is that when you generate your archive, the JARs dependent on the scope `system` will not be exported. Quite a limitation, isn't it?

So, this solution should be opted for **only** in situations when the JAR **is needed to** compile, for instance, when it contains *annotation processors* or a *code generator*.

Notwithstanding, we may have used the plugin Maven Install, in order to install automatically the JARs on the local repository, with such a code similar to the following code:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-install-plugin</artifactId>
  <executions>
    <execution>
      <id>install-ftp-jar</id>
      <phase>validate</phase>
      <configuration>
        <file>${basedir}/lib/ftp-1.7.2.jar</file>
        <repositoryLayout>default</repositoryLayout>
        <groupId>UNKNOWN</groupId>
        <artifactId>ftp</artifactId>
        <version>1.7.2</version>
        <packaging>jar</packaging>
        <generatePom>true</generatePom>
      </configuration>
      <goals>
        <goal>install-file</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

In any case, this solution is quick and dirty, too, for many reasons: first, the module dependent on JAR will be installed on each build, increasing the build time. Second, as many `<execution>` blocks as dependencies must be added: the solution cannot be considered as long-term sustainable. Moreover, this solution may work with Maven 2.2 and yet raise strange issues with Maven 3, thereby preventing from upgrading the Maven version.

(A bit) slower and (far) cleaner

You have to declare a `<remote repository>` of which URL is local, such as:

```
<repositories>
  <repository>
    <id>pseudoRemoteRepo</id>
    <releases>
      <enabled>true</enabled>
      <checksumPolicy>ignore</checksumPolicy>
    </releases>
    <url>file://${project.basedir}/lib</url>
  </repository>
</repositories>
```

Then, declare the dependencies like:

```
<dependency>
  <groupId>UNKNOWN</groupId>
  <artifactId>ftp4j</artifactId>
  <version>1.7.2</version>
</dependency>
```

Move and/or rename the JARs, for instance, from the `lib/ftp-1.7.2.jar` folder to the actual one, such as `lib/UNKNOWN/ftp4j/1.7.2/ftp4j-1.7.2.jar`.

Summary

Thus, at the end of this chapter, you are able to migrate a wide range of projects to Maven:

- Getting information from the POM when available
- Searching and finding clues from the archives
- As a last resort, setting a pseudo-remote repository

However, this step of migration is not sufficient: despite you having performed migration, other tasks, such as detecting conflicts and fixing them, are still to be completed.

The complete POM, built on the former Ant `build.xml` file, looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>
      com.packt.dependencyManagement.chapter4
    </groupId>
    <artifactId>ProjectFromAnt</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>Project from Ant to Maven</name>
    <repositories>
      <repository>
        <!--Fake remote repository, to retrieve the dependency to
        ftp-1.7.2-->
        <id>pseudoRemoteRepo</id>
        <releases>
          <enabled>true</enabled>
          <checksumPolicy>ignore</checksumPolicy>
        </releases>
        <url>file://${project.basedir}/lib</url>
      </repository>
    </repositories>
    <build>
      <sourceDirectory>src</sourceDirectory>
      <testSourceDirectory>test</testSourceDirectory>

      <!-- Quick and dirty solution ; we left it as a comment-->
  <!--
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-install-plugin</artifactId>
          <executions>
            <execution>
              <id>install-ftp-jar</id>
              <phase>validate</phase>
              <configuration>
                <file>${project.basedir}/lib/ftp-1.7.2.jar</file>

                <repositoryLayout>default</repositoryLayout>
                <groupId>UNKNOWN</groupId>
                <artifactId>ftp</artifactId>
                <version>1.7.2</version>
```

```
              <packaging>jar</packaging>
              <generatePom>true</generatePom>
            </configuration>
            <goals>
              <goal>install-file</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
-->
  </build>
  <dependencies>
    <dependency>
      <!--Retrieved from its POM-->
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.6</version>
    </dependency>
    <dependency>
      <!--groupId and artifactId were retrieved thanks to
      http://findjar.com-->
      <groupId>commons-cli</groupId>
      <artifactId>commons-cli</artifactId>
      <!--Version was retrieved from the Manifest.MF-->
      <version>1.0</version>
    </dependency>

    <!-- Quick and dirty solution ; we left it as a comment-->
<!--
    <dependency>
      <groupId>unknownGroupId</groupId>
      <artifactId>ftp4j</artifactId>
      <version>1.7.2</version>
      <scope>system</scope>
      <systemPath>
        ${project.basedir}/lib/ftp4j-1.7.2.jar
      </systemPath>
    </dependency>
-->

    <dependency>
      <groupId>UNKNOWN</groupId>
```

```
      <artifactId>ftp4j</artifactId>
      <version>1.7.2</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <!-- The version was retrieved by comparison of checksums
      and sizes-->
      <version>4.11</version>
      <!-- The scope is test, because in the original Ant file
      JUnit was designed as needed for tests-->
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

With the recipe exposed in this chapter, you should be able to migrate almost any project. Nonetheless, if truly special needs are encountered, do not forget that, as a last resort, Maven can execute any Ant script, thanks to the `org.apache.maven.plugins:maven-antrun-plugin:1.7` plugin and goal run. Anyway, the actual goal of migration to Maven would be lost, as well as main features such as repositories. So, you should consider your case very carefully.

5
Tools within Your IDE

Developers in the 2010s are lucky enough to dispose of various Integrated Development Environments (IDEs). The three most widely used are Eclipse, NetBeans, and IntelliJ IDEA. All are of good quality. They feature many tools that, once learned, allow increasing your productivity: auto-complete, dynamic management and visualization of dependencies, and so on.

Case study

Let's consider a basic three-tier project, organized on three layers: HSQLDB/Hibernate (Hibernate playing a simple role of JDBC layer), Spring, and JSF + Primefaces.

Functionally, the presentation layer consists of a date and hour widget, and a color picker. When a color is selected, a JSF bean is called, and then a Spring service, then a DAO runs a query in database.

Expected display in browser

This study case will illustrate the IDEs features related to the Maven dependency management.

To execute this project, you have to launch the database (`mvn exec:java -Dexec.mainClass="org.hsqldb.Server"`), then build, and deploy the WAR on a servlet container (`mvn clean install jetty:run`).

IntelliJ IDEA

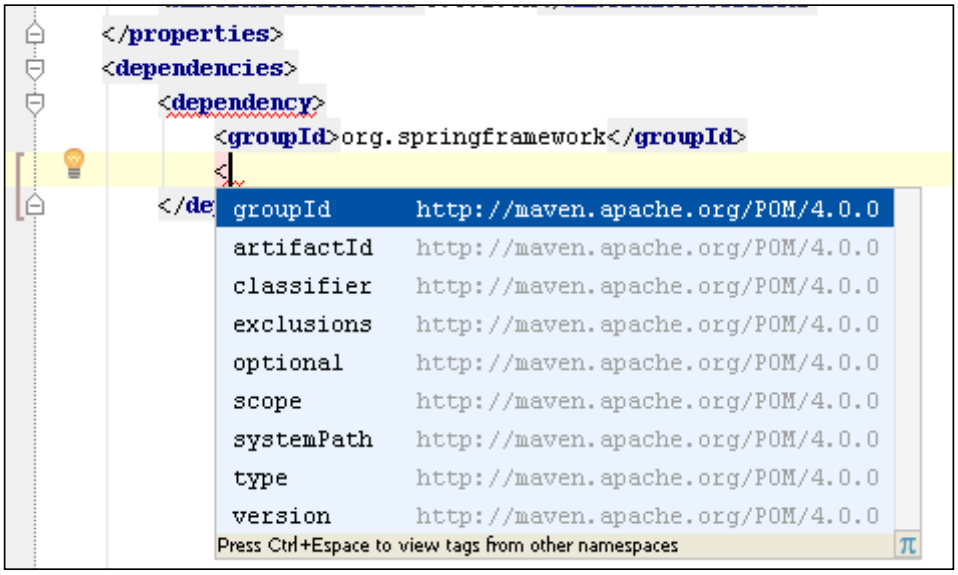
JetBrains IntelliJ IDEA is considered as a premium IDE by numerous developers and architects. Two versions are available: **Community Edition** (free and limited) and **Ultimate Edition** (complete with all features). The latest version is 12.1, which is also known as Leda. Next version, 13, known as Cardea, is expected to be released in December 2013.

XML with XSD completion

For any XML file, when an XML schema (XSD) or a DTD is indicated, IntelliJ IDEA (Community and/or Ultimate) can provide automatic completion. Therefore this is the case for POM files (and this explains why we have taken care to write complete XML headers since the beginning of this book).

In IntelliJ IDEA, two modes of auto-completion exist: the classic one, through pressing a combination of *Ctrl* + *Space bar* keys, and a smart mode ("guessing" what you intend to do), through pressing the *Ctrl* + *Shift* + *Space bar* keys.

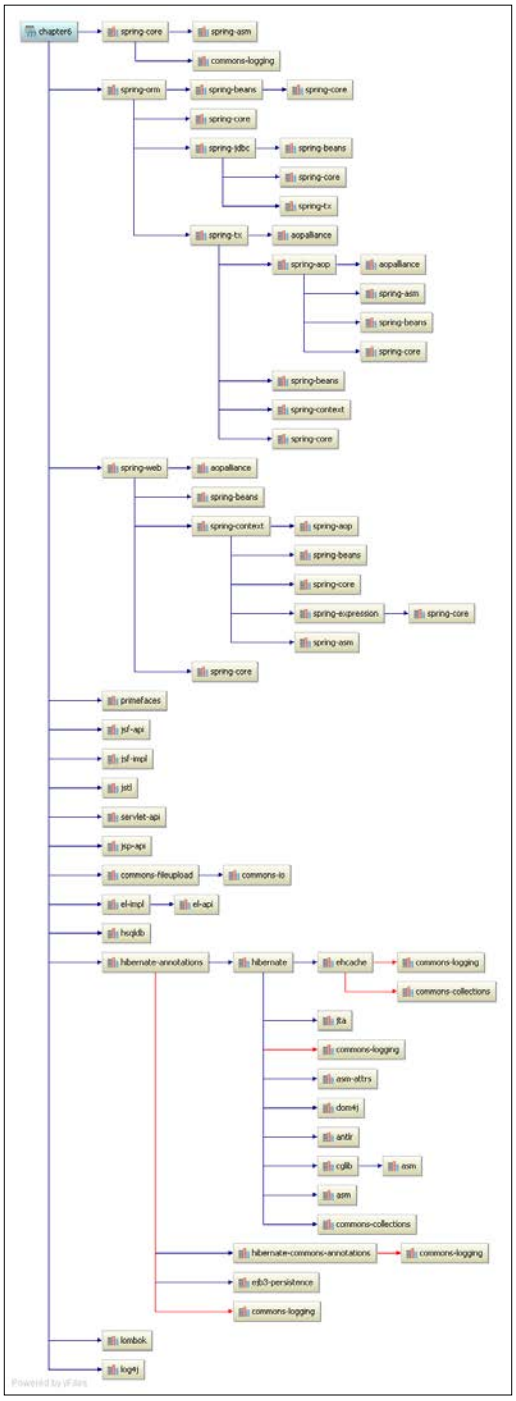
Auto-completion increases your productivity and your typing lane, saving the time of searching for the exact command or writing a complete instruction, and avoiding spelling errors as shown in the following screenshot:



Module Dependency Graph

IntelliJ IDEA Ultimate Edition has a plugin called **Module Dependency Graph**. On editing a POM, just press *Ctrl* + *Shift* + *Alt* + *U* (or right-click on the POM interface and then navigate to **Maven** | **Show Dependencies...**), a figure will be generated, on which red arrows show a potential conflict.

This features allows you to see the dependencies of your project in a user-friendly manner, all the more if you are allergic to text outputs produced by a `mvn dependency:tree`.



Dependency graph generated by IntelliJ IDEA 12.1 Ultimate

The graph can be filtered by scope, exported to a `.PNG` file, or printed.

DSM (or Dependency Structure Matrix) is a plugin available with IntelliJ IDEA Ultimate. While not linked specifically to Maven (the same tool is included in Sonar among others), it provides very useful information on your projects and modules: cycles, dependencies, relationships, and so on. For more details, consult `http://blogs.jetbrains.com/idea/2008/01/intellij-idea-dependency-analysis-with-dsm/` and `http://www.dsmweb.org/en/dsm.html`.

The figure can also be displayed on editing the POM by pressing a combination of *Ctrl* + *Alt* + *U* (or right-click on the POM interface and then navigate to **Maven** | **Show Dependencies...**).

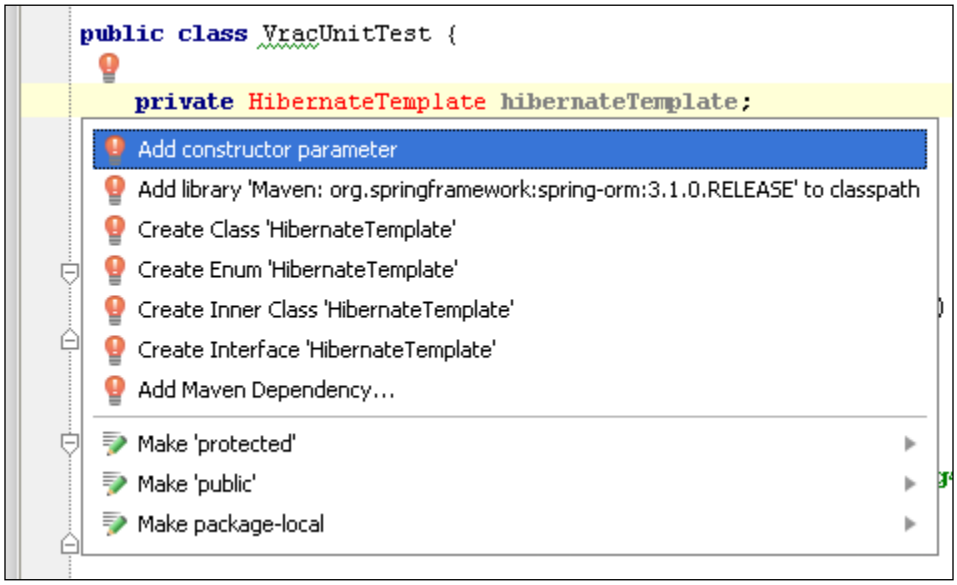
Dependency addition

Another cool feature of IntelliJ IDEA is the suggestion of imports and/or of Maven dependency addition.

Dependency addition from Java code

Dependencies can be added on-the-fly, while writing code, that is, without leaving the Java editor: a Maven dependency is added in the background, selected among those proposed by IntelliJ IDEA after scanning the repository. Clearly, the ability to add dependencies while writing Java code, therefore without switching of context between Java and Maven's XML POM, makes you save time and productivity.

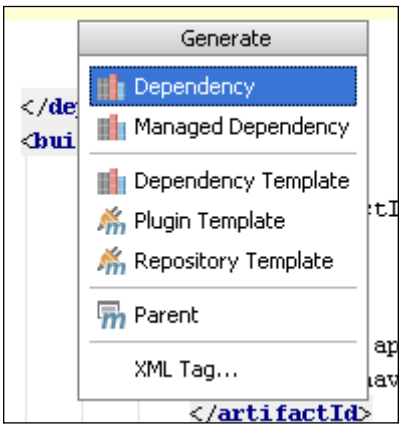
When you call a class that is not yet referenced to in your imports, IntelliJ IDEA will consider it as an error (because it makes your code unable to compile). Then, move the caret to this piece of code, and press *Alt* + *Enter* (or wait for a red lamp to be displayed, then click on it): IntelliJ IDEA will propose several choices, among which one is **Add library 'Maven: ...' to classpath**.



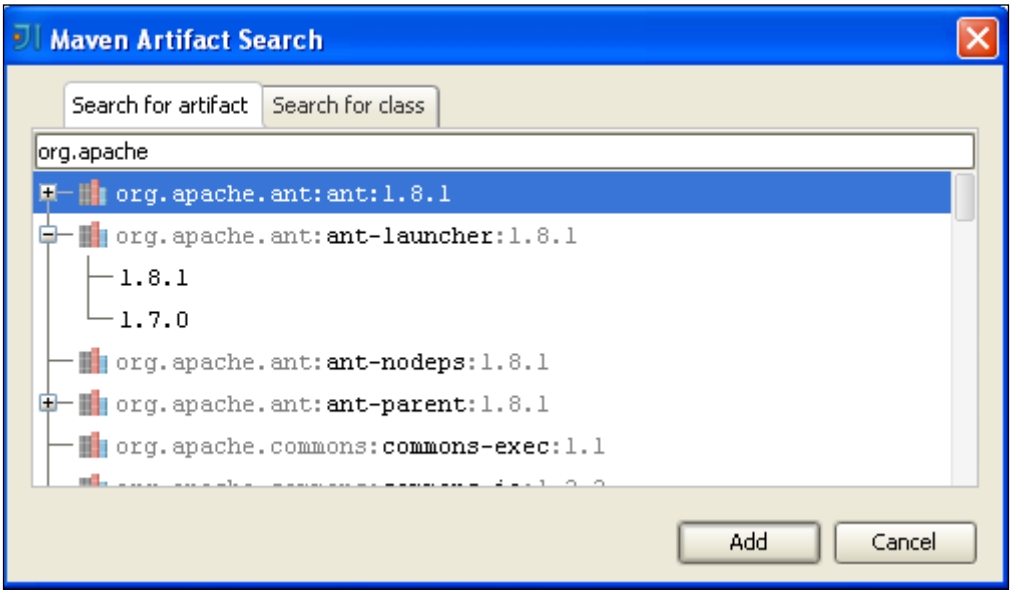
Import and dependency suggestions by IntelliJ IDEA 12.1 Community

Dependency search and generation within a POM

While editing a POM, you can search and add a dependency. Just press the *Alt* + *Insert* key combination. A pop-up appears as shown in the following screenshot:



Click on **Dependency** or **Managed Dependency**. Then you can search and select a dependency by designation or by an embedded class as shown in the following screenshot:



The same operations can be performed for plugins, repositories, and so on.

Conclusion on IntelliJ IDEA

IntelliJ IDEA is very complete in its support of Maven and provides powerful tools dedicated to dependency management.

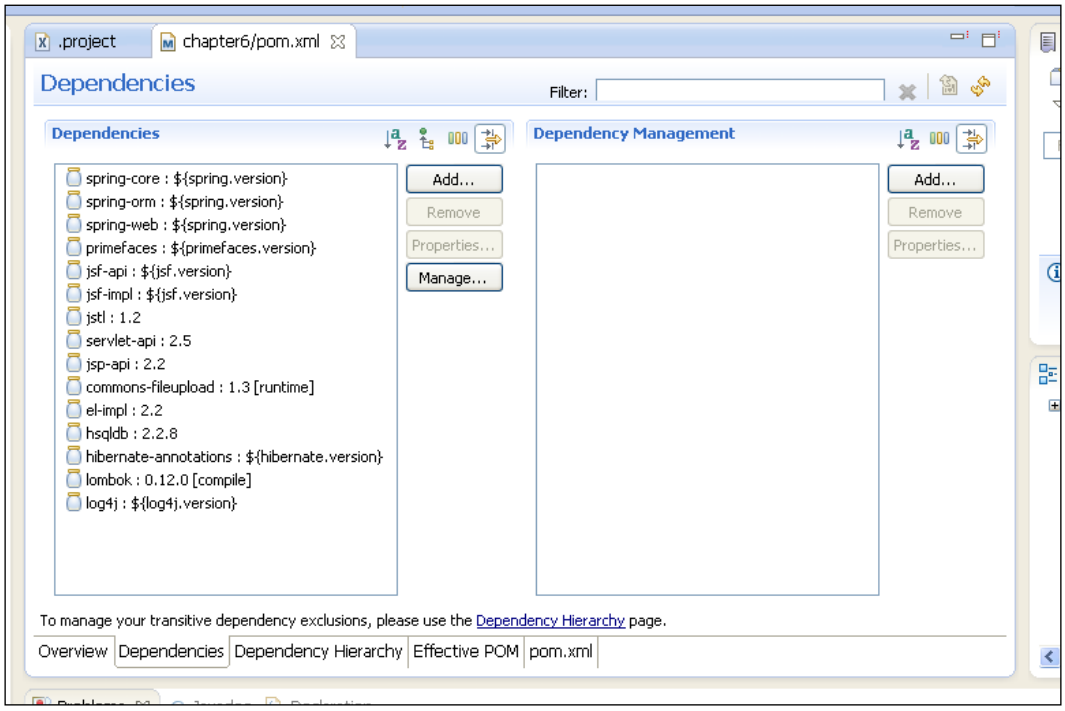
Eclipse

Eclipse is released by the Eclipse Foundation. It is the most widespread IDE in Java projects. The latest version available is 4.3, also known as **Kepler**.

On opening a POM, Eclipse switches to a special view, dedicated to POMs. By default, five tabs are displayed in the POM editor. **Overview** summarizes macro-information related to the POM, such as details on the project, SCM, organization, and so on. You can fill in the form to update the POM, instead of writing XML blocks. The `pom.xml` file displays the text content of the file. The three other tabs are more interesting from a dependency management viewpoint.

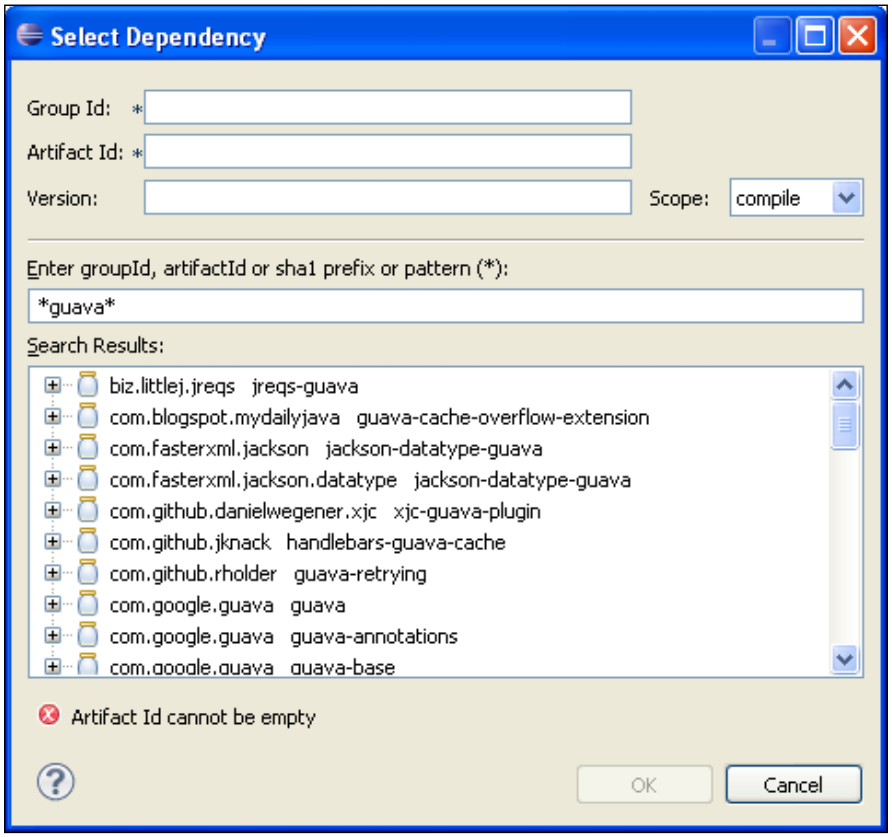
Dependency view

Dependencies lists the dependencies referenced in the POM. On the upper right corner, a text field allows to filter them as shown in the following screenshot:



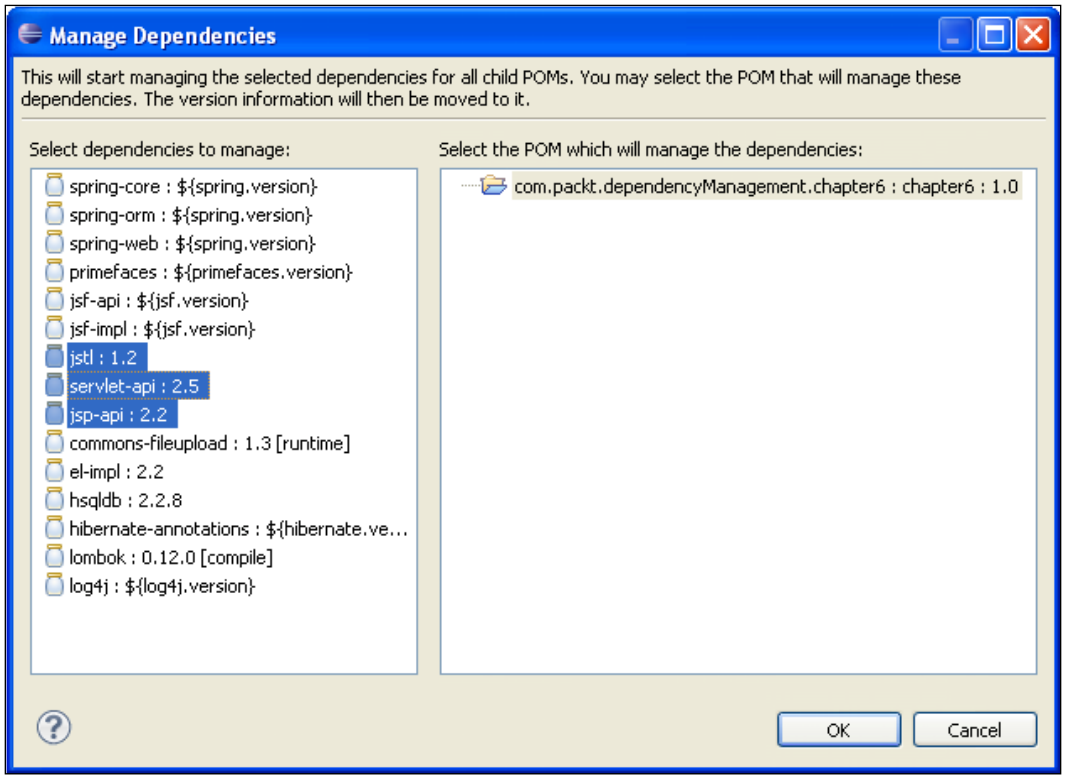
Dependencies view in Eclipse 4.3 (Kepler)

On clicking on **Add** button, Eclipse opens a modal window. Then you can add an artifact, entering its `groupId/artifactId/version` if you know them, or performing a search. For instance, in the following screenshot, we search all the artifacts containing `guava` in their designation. If we select one of the results, then it will be added in the list of dependencies as shown in the following screenshot:



Select dependency view in Eclipse 4.3 (Kepler)

In the main **Dependency** view, the **Manage** button allows to pull up some of the dependencies to a parent POM as shown in the following screenshot:

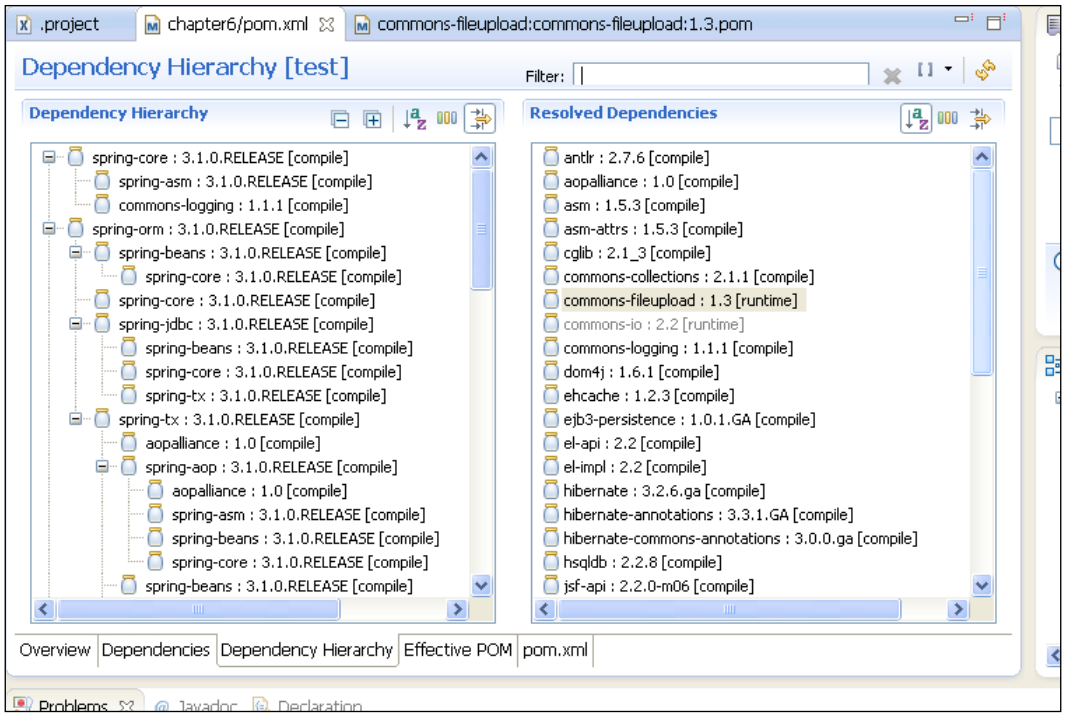


Manage dependencies view in Eclipse 4.3 (Kepler)

Dependency Hierarchy view

Dependency Hierarchy lists, displays the hierarchy in a left pane, the tree-sorted dependencies, direct ones as well as transitive. On the right, Eclipse displays the **Resolved Dependencies**. On clicking on one dependency, Eclipse opens this artifact POM. Therefore, you can examine a dependency POM from your current project one.

This feature allows you to see your project declared dependencies, the transitive dependencies as they are resolved, and consequently, to know which version of a given artifact is actually loaded.



Dependency Hierarchy view in Eclipse 4.3 (Kepler)

Effective POM view

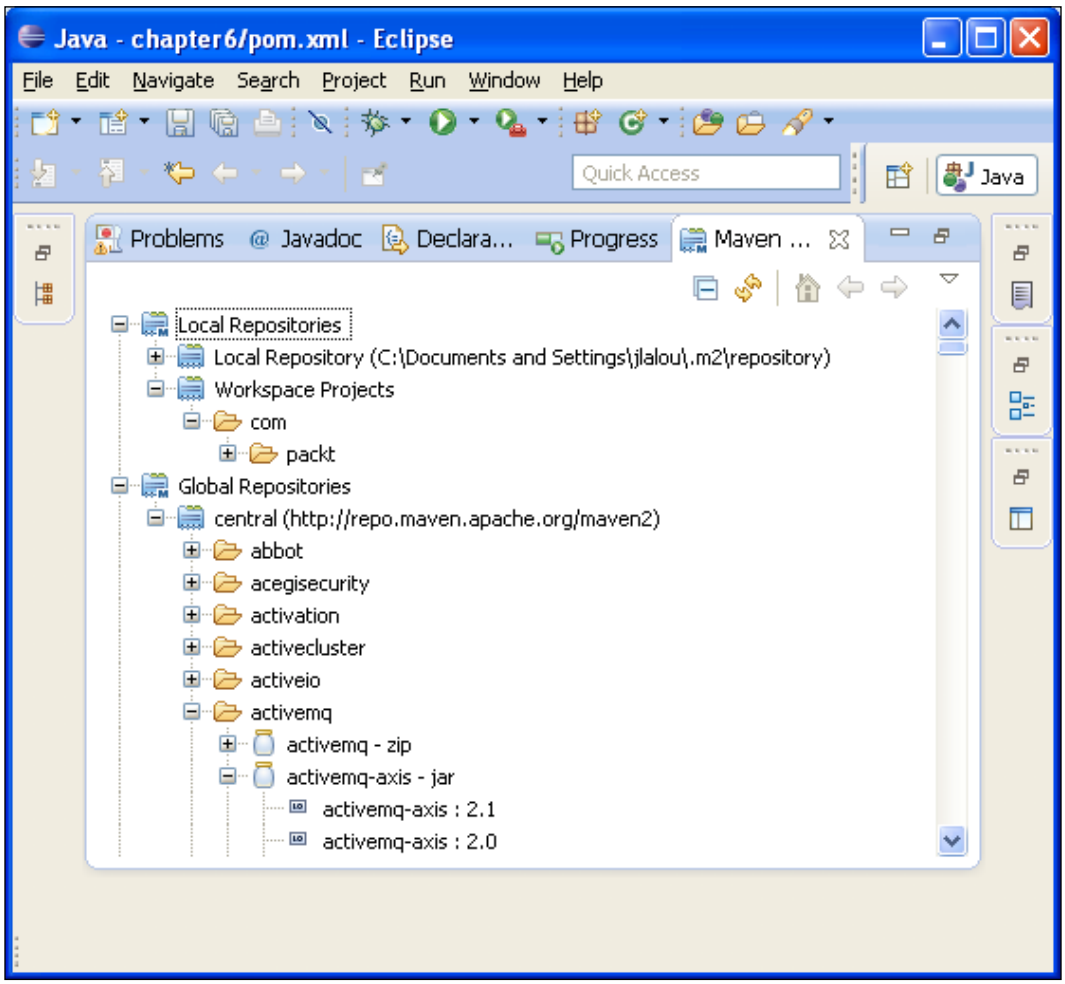
Effective POM view displays the actual POM as it is resolved and completed by Maven. For details see the section, *Dynamic POMs and dependencies* in *Chapter 3*, *Dependency Designation (advanced)*.

Dependency Graph

Until Eclipse 4.X branch, a **Dependency Graph** was featured. More precisely, the Dependency Graph was bundled with **m2e**, the main plugin for Maven integration within Eclipse, until m2e 1.0 release. Anyway owing to its authors, this plugin was not efficient with big projects; its development was ended and the plugin was removed from future versions of Eclipse.

Maven Repository view

Eclipse provides a view, **Maven Repository**, which provides a synthetic view of your repositories (both local and remote) and their content as shown in the following screenshot:



Maven Repository view in Eclipse 4.3 (Kepler)

This view synthesizes all remote and local repositories used by the project and their contents.

You can collapse, expand, and explore the different folders. You can also force a reindex of a repository.

Conclusion on Eclipse

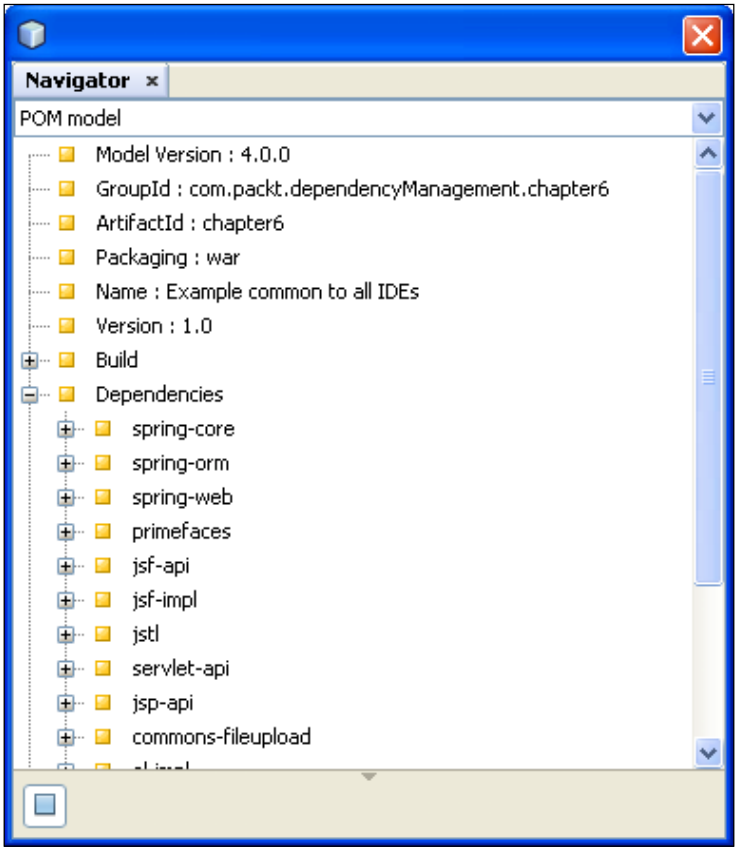
Eclipse supports Maven well, yet this support is less broad than IntelliJ IDEA.

NetBeans

NetBeans is the official IDE promoted by Sun and now Oracle, for Java projects. Latest available version is NetBeans 7.3.

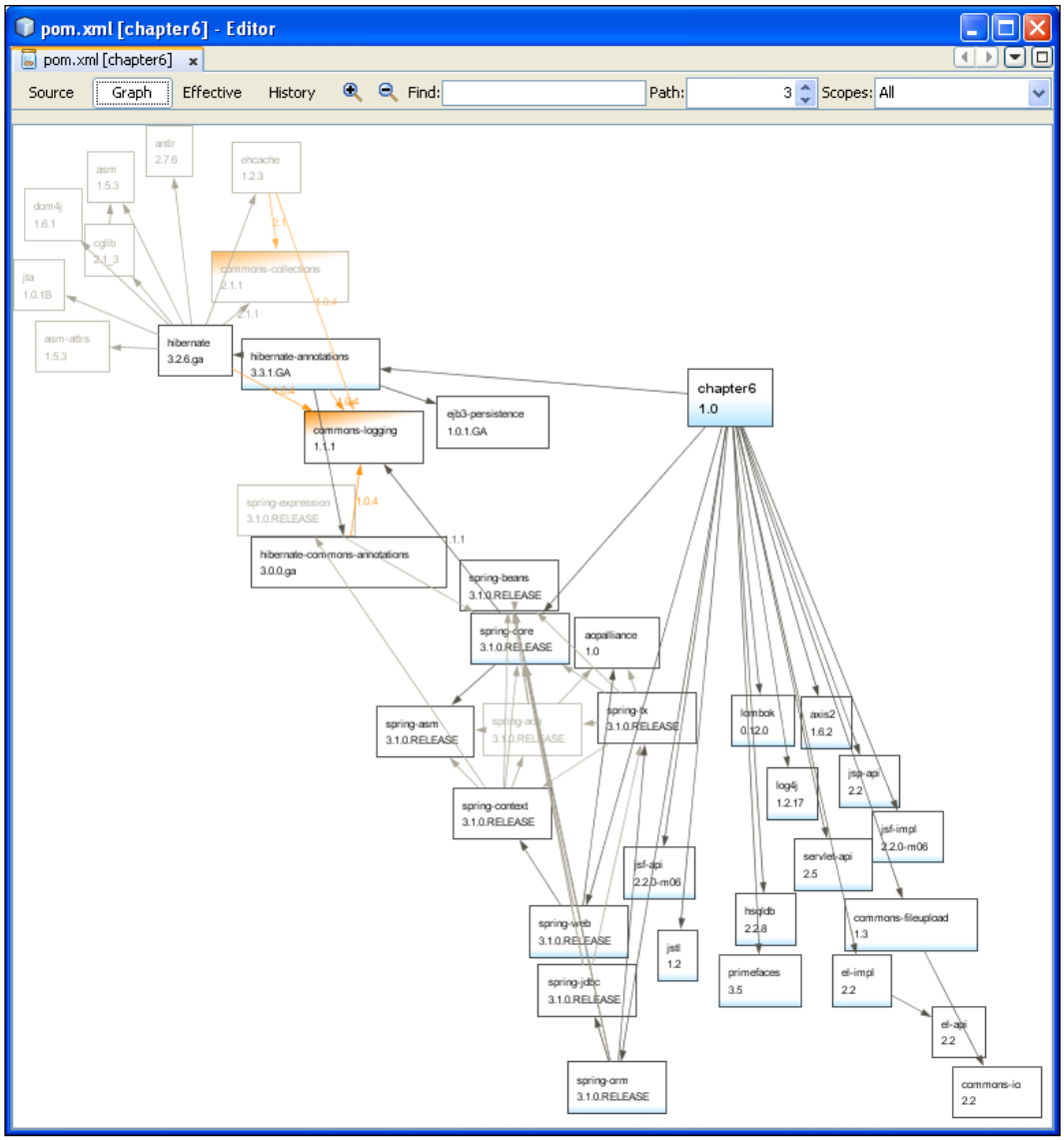
Overview

In similarity to Eclipse, on opening a POM, NetBeans switches to a POM-specific view. By default, four tabs are displayed. **Source** displays the text content of the POM. By the way, the **Navigator** frame displays the POM as a tree as shown in the following screenshot:



Navigator frame in NetBeans 7.3

Another view, **Effective**, displays the effective POM. A virtual left column is added, to know the source of each block: this is useful for projects with complex or multi-level inheritance. The **Graph** view displays a graph of dependencies as shown in the following screenshot:



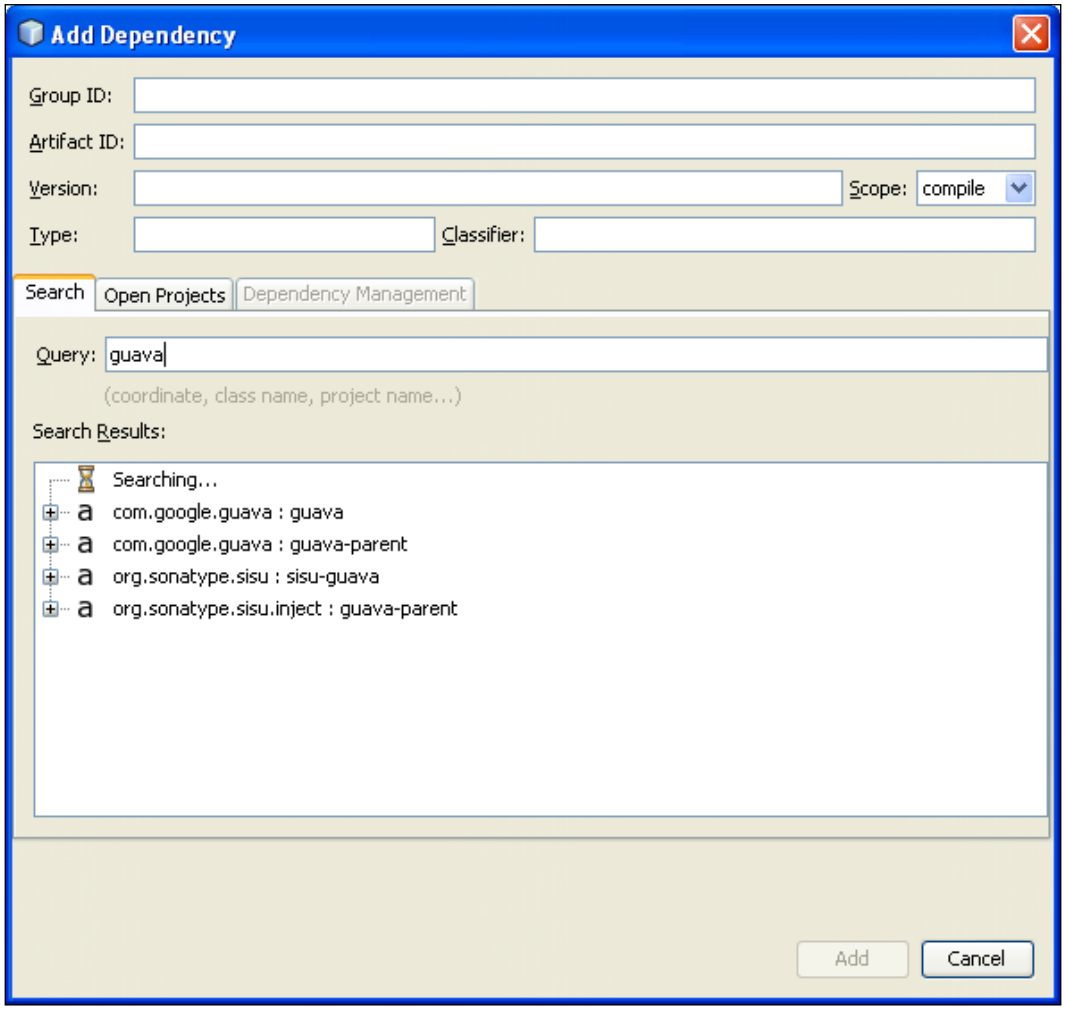
Graph view in NetBeans 7.3

The dependencies can be filtered. Moreover, the maximum path depth to be highlighted can be set.

At last, the **History** view is common to all text files, and allows easy comparison of the currently edited file.

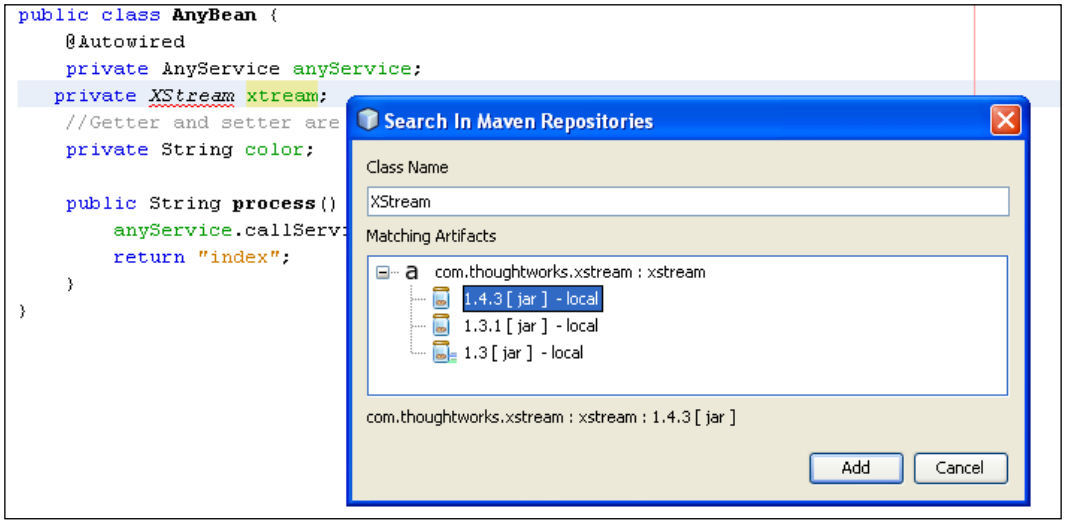
Dependency addition

In similarity to IntelliJ IDEA and Eclipse, you can add a dependency from the POM (by pressing *Alt* + *Insert*) and generate the corresponding code, thanks to a search and addition popup as shown in the following screenshot:



Add Dependency popup in NetBeans 7.3

At last, with NetBeans you can add dependencies into the POM from Java code. The process is similar to the one in IntelliJ: move the caret to the piece of code needing an import, and type *Alt* + *Enter* (or click on the red lamp replacing the line number on the left). NetBeans will propose several choices, among which one is **Search dependency at Maven repositories for…**. Then a modal window is opened, in which you can filter the right dependency to add as shown in the following screenshot:



Search in Maven Repositories popup in NetBeans 7.3

NetBeans can help developers and architects in managing the dependencies of their projects.

Summary

As a conclusion, we can say that the three main IDEs have many features that can alleviate the burden of managing dependencies, and therefore increase your productivity with:

- Auto-complete
- Dependency visualization
- Dynamic dependency addition

As usual, each and every developer will choose his or her favorite IDE owing to their usage, even though switching timely of an IDE still remains possible, when some particular features lack in the permanent one.

The following table compares the pros and cons of IDEs:

IDE	Pros	Cons
IntelliJ IDEA 12.1 (Leda) `http://www.jetbrains.com/idea/download/`	• Complete set of features • Code generation from POM and Java code • Dependency graph • XML completion	• Some plugins are not available in Community Edition • No "effective POM" view
Eclipse 4.3 (Kepler) `http://www.eclipse.org/downloads/`	• POM-specific view • Maven Repository viewer • Effective POM view • Navigation to artifacts POMs	• No dependency graph • Poor code generation tools
NetBeans 7.3 `https://netbeans.org/downloads/`	• POM-specific view • Effective POM view • Code generation from POM and Java code • Dependency graph	• Dependency graph hard to read • No navigation to artifacts POMs

6
Release and Distribute

Until now, we have dealt with the artifacts on which your project *does* depend on. In this chapter, we will broaden the horizon and see your project as a released and *depended on* one.

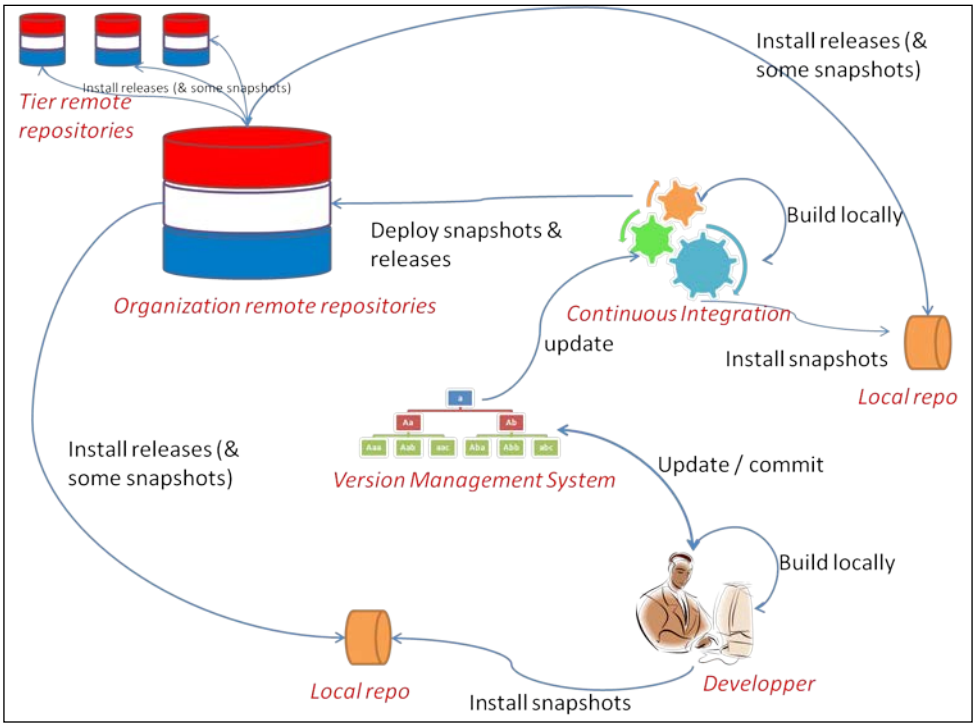
We will review miscellaneous tips, tools, how-tos, and best practices. These might not always be 100 percent-Maven focused, however, that will improve your productivity and efficiency.

Best practices before release

Here are some pieces of advice, which are quite obvious, however, people tend to ignore them:

- Firstly, clean the POM, regularly. Frequency depends on your context, but cleanup should be performed at least once per release. The cleanup should deal with conflicts of dependencies, removal of unused JARs, version upgrade, and so on.
- Next, *release* versions should be built **only** on a continuous integration platform, such as Atlassian Bamboo, Apache Hudson/Jenkins, Continuum, and so on. Maven does not make a difference between `foo:bar:version` built locally and the "official" `foo:bar:version` built on continuous integration (even though the same source code should be compiled to the same bytecode, the rule "compile once, run everywhere" may fail). While discrepancies are common and inescapable with snapshots, such divergences should not be admitted with releases.

- Besides, the local repository of the machine on which release versions are built should be **purged** before building the release (goal `mvn dependency:purge-local-repository`). Yet, if multiple builds are run at the same time, issues may happen. As a workaround, consider having separate local repositories (accessed through—`Dmaven.repo.local=/path/to/local/repo/for/releases` in the command line) or even `settings.xml` files for different build profiles.
- Actually, this last point is part of a more general work organization. Schematically, a modern project can be considered as having several steps, which include development, building, and deployment. The following diagram shows a good organization:



Efficient work organization with Maven: version control management, continuous integration, and repositories

Developers update the source code from **version control systems** (**VCS**), such as SubVersion, Git, and so on. They build locally snapshots, which are installed on their local repositories. Developers commit the code to the VCS. The source code is checked out by the continuous integration platform to be built and tested. The artifacts generated by a successful build are deployed on a remote repository, which is common to the organization. The developers' local repositories are fed with release versions downloaded from the organization repository (anyway, in some cases, developers may have to download and install manually some artifacts). If possible, the organization repository is the single entry point to feed external repositories, and get tier artifacts. Such a process guarantees a consistency of artifacts used within an organization.

Fixing conflicts with tier-parties

Your project dependencies are not the only artifacts that can raise conflicts—most of the time, a Java/JEE application is not purely standalone—and it lays on a tier application or container, which is either an application server or anything else. However, these tier applications embed their own dependencies—the amount of which can be in dozens—conflicts between resident application and host dependencies often happen. You have full interest to anticipate these conflicts and treat the problems ahead. The question that arises: "How to deal with them?"

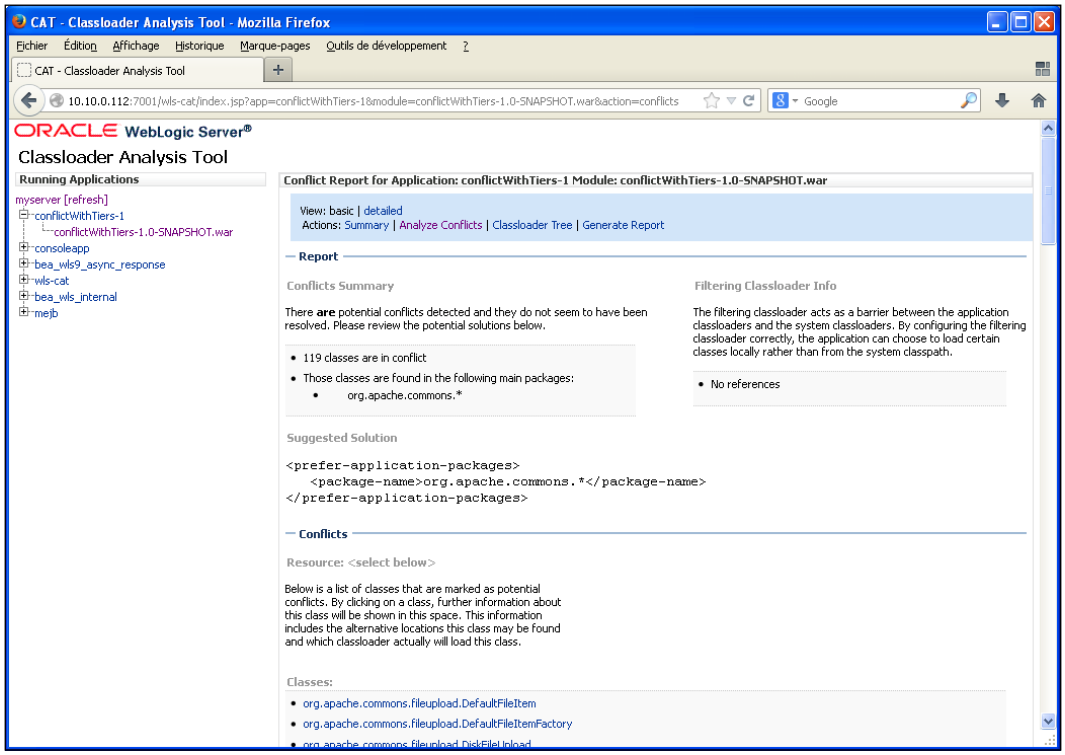
Let's consider the artifact `com.packt.dependencyManagement.chapter6:conflictWithTiers`. It was generated through an archetype:

```
mvn archetype:generate -B \
      -DgroupId=com.packt.dependencyManagement.chapter6 \
      -DartifactId=conflictWithTiers \
      -DarchetypeGroupId=org.apache.struts \
      -DarchetypeArtifactId=struts2-archetype-blank \
      -DarchetypeVersion=2.3.8 \
```

Maven Archetype plugin

Maven Archetype allows us to create projects owing to a template. Such template structures are available for hundreds of frameworks and situations.

As is, the brand project has conflicts with Oracle WebLogic 11*g* on which it was targeted to be deployed. WebLogic has powerful tools to detect and fix such problems (among many others). **Classloader Analysis Tool** (**CAT**) deserves to have a look at it. For instance, CAT reports the conflict with Apache Commons JARs and suggests the fix, as shown in the following screenshot:



Classloader Analysis Tool from WebLogic

In this situation, the solution consists in adding two configuration files, `weblogic.xml` and `weblogic-application.xml`, which are specific to WebLogic. The files are added within the folder `src/main/webapp/WEB-INF/folder`:

- Add the following code into the `weblogic.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
  xmlns="http://www.bea.com/ns/weblogic/weblogic-web-app"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/
   weblogic-web-app
  http://www.bea.com/ns/weblogic/weblogic-web-app/
   1.0/weblogic-web-app.xsd">
```

```
    <container-descriptor>
      <!-- Tells WebLogic to prefer libraries from WEB-
       INF folder rather than those from WebLogic self-
       classloader -->
     <prefer-web-inf-classes>true
     </prefer-web-inf-classes>
    </container-descriptor>
    <context-root>/conflictWithTiers</context-root>
</weblogic-web-app>
```

- Add the following code into the `weblogic-application.xml` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE weblogic-application PUBLIC
  "-//BEA Systems, Inc.//DTD WebLogic Application
   7.0.0//EN"
  "http://www.oracle.com/technology/weblogic/weblogic-
   application/1.1/weblogic-application.xsd">
<weblogic-application>
  <!--Apache Commons provoke conflicts with WebLogic:
    with the block below, we tell WebLogic the packages
    for which to prefer the dependencies of the WAR over
    the dependencies of the WebLogic itself-->
    <prefer-application-packages>
      <package-name>org.apache.commons.*</package-name>
      <!--For this block to be taken in account, the tag
       <prefer-web-inf-classes> from weblogic.xml must
       be set at *true* -->
    </prefer-application-packages>
</weblogic-application>
```

Similar mechanisms exist for most of application servers. For a JBoss AS 7 of which you would like to exclude `commons-lang`, you would have to fill a `src/main/webapp/WEB-INF/folder/jboss-deployment-structure.xml` file with content similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure
    xmlns="urn:jboss:deployment-structure:1.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:jboss:deployment-structure:1.2
     http://www.jboss.org/schema/jbossas/jboss-deployment-
     structure-1_2.xsd">
  <deployment>
    <exclusions>
      <!-- Reference to $JBOSS_HOME/modules/org/jboss/as/jaxr/
       main -->
      <module name="org.jboss.as.jaxr"/>
```

```
        <!--You have to add as many blocks as modules with
          conflicts-->
        <module name="org.apache.velocity"/>
        <module name="org.apache.cxf"/>
        <module name="org.apache.juddi.scout"/>
        <!-- module etc.-->
      </exclusions>
      <dependencies>
        <!-- This is the name of dependency exported by the module
          of name javaee.api, in the module.xml file -->
        <module name="org.apache.commons.lang"/>
      </dependencies>
    </deployment>
</jboss-deployment-structure>
```

This is similar for other application servers such as `%JONAS_BASE%/conf/cloassloader-default-filtering.xml` for JOnAS, `WEB-INF/glassfish-web.xml` for GlassFish, and so on.

As a conclusion, try to identify the potential conflicts at an earlier stage, that is, when a dependency is added or updated.

Releasing the source code

Once all code is compiled and tested, and the dependencies have been checked, it is time to release the software.

For a final Version `x.y.0`, best practice consists in:

- Tagging the Version `x.y.0` and removing its `x.y.0-SNAPSHOT` status
- Creating a new branch `x.y` that will generate snapshots (`x.y.1-SNAPSHOT`, `x.y.2-SNAPSHOT`, and so on) and releases (`x.y.1`, `x.y.2`, and so on)
- Upgrading the version of the trunk to `x.(y+1).0-SNAPSHOT`

The Maven Release plugin

Maven Release plugin can help you in performing these tasks with the minimum human intervention possible. Here is an example of a minimal POM using Maven Release:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packt.dependencyManagement.chapter6</groupId>
    <artifactId>exampleOfRelease</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <properties>
        <!-- Manually set BRANCH_NAME before running the plugin-->
        <BRANCH_NAME>1.0</BRANCH_NAME>
    </properties>
    <scm>
        <connection>
          scm:svn:http://svn.mycompany.extension:8066/myProject/
          branches/${BRANCH_NAME}
        </connection>
        <developerConnection>
            scm:svn:http://svn.mycompany.extension:8066/myProject/
            branches/${BRANCH_NAME}
        </developerConnection>
        <url>
            http://svn.mycompany.extension:8066/myProject/branches
        </url>
    </scm>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>
                    maven-release-plugin
                </artifactId>
                <version>2.4.1</version>
                <configuration>
                    <!-- Tell Maven to tag the version-->
                    <remoteTagging>true</remoteTagging>
                    <!-- Should be parameterized in settings.xml
                     file -->
                    <username>myUserName</username>
                    <!-- Should be parameterized in settings.xml
                      file -->
                    <password>myPassWord</password>
                    <tagBase>
                      http://svn.mycompany.extension:8066/
                      myProject/tags/
                    </tagBase>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Delivering artifacts and distributions

In a project life, you are led to release your artifacts, either independently of any other binary, or included in a wider archive.

Artifacts

A good practice is to implement a simple rule; as stated earlier, team members can build any `SNAPSHOT` versions, but the task of delivering a release version should be reserved exclusively to the continuous integration system. Actually, theoretically, the same code on any platform should produce the same bytecode and hence be archived. Anyway, on loading them from the local repository, Maven does not discriminate artifacts that have been locally built from those that have been downloaded from a remote repository, provided they have the same designation. Therefore, to be sure the same release artifact is common to all developers; the release versions should never be locally built. This avoids human errors, inconsistency of JARs, and the nightmare of hunting and fixing them.

Release distribution

In the following sections, we will see how to distribute the artifact that is generated by our project.

A simple case

Let's consider the `com.packt.dependencyManagement.chapter6:exampleOfSimpleAssembly` project. This is a simple project with direct dependencies to Groovy, Spring, and transitively to other stuff. For such a project, you are not interested in exporting one-on-one dependencies, and having a long, ugly, unreadable classpath, hence command line as follows:

```
java ./exampleOfSimpleAssembly-1.0.jar:$M2_REPO/aopalliance/
aopalliance/1.0/aopalliance-1.0.jar:$M2_REPO/commons-logging/commons-
logging/1.1.1/commons-logging-1.1.1.jar:$M2_REPO/org/codehaus/groovy/
groovy-all/2.1.6/groovy-all-2.1.6.jar:$M2_REPO/org/springframework/
spring-aop/3.2.3.RELEASE/spring-aop-3.2.3.RELEASE.jar:$M2_REPO/org/
springframework/spring-beans/3.2.3.RELEASE/spring-beans-3.2.3.RELEASE.
jar:$M2_REPO/org/springframework/spring-context/3.2.3.RELEASE/
spring-context-3.2.3.RELEASE.jar:$M2_REPO/org/springframework/
spring-core/3.2.3.RELEASE/spring-core-3.2.3.RELEASE.jar:$M2_REPO/org/
springframework/spring-expression/3.2.3.RELEASE/spring-expression-
3.2.3.RELEASE.jar  com.packt.dependencyManagement.chapter6.MainClass
```

Rather, you would be glad to satisfy of a mere:

```
java -jar exampleOfSimpleAssembly-1.0.jar
```

Then you can use the **Maven Assembly** plugin. Assembly allows you to package your application with other material, such as sources, documentation, external dependencies, and so on. Four default aggregators called **descriptors** exist: `bin`, `src`, `project`, and `jar-with-dependencies`. The latter interests us (you can consult the Maven Assembly website for more information on the other three).

In the POM, let's add the following block:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.4</version>
    <executions>
        <execution>
            <id>generateSingle</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
            <configuration>
                <!-- This Assembly will generate the regular JAR,
                  plus a second one, suffixed with '-full'. -->
                <finalName>${project.artifactId}-
                  ${project.version}-full</finalName>
                <descriptorRefs>
                    <!-- We call the predefined descriptor
                     'jar-with-dependencies'-->
                    <descriptorRef>jar-with-dependencies
                    </descriptorRef>
                </descriptorRefs>
                <appendAssemblyId>false</appendAssemblyId>
                <archive>
                    <manifest>
                        <!-- Let's create a Manifest file, hinting
                          at the executable class to launch-->
                        <mainClass>com.packt.dependencyManagement.
                          chapter6.MainClass</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </execution>
    </executions>
</plugin>
```

On running an `mvn clean` install, two artifacts are generated:

- `exampleOfSimpleAssembly-1.0.jar` the usual one
- `exampleOfSimpleAssembly full-1.0.jar` that artifact contains all the dependencies, exploded as .class files, then re-bundled within a JAR

Then you can run the application and get the expected output (last line) as follows:

```
$ java -jar target\exampleOfSimpleAssembly-1.0-full.jar
12 August 2013 16:18:28 org.springframework.context.support.
AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.
ClassPathXmlApplicationContext@ab50cd: startup date [Mon Aug 12 16:18:28
CEST 2013]; root of context hierarchy
12 August 2013 16:18:28 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[applicationContext.xml]
12 August 2013 16:18:30 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.
factory.support.DefaultListableBeanFactory@104faf8: defining beans
[anyService,anyDao,org.springframework.context.annotation.internalConfig
urationAnnotationProcessor,org.springframework.context.annotation.intern
alAutowiredAnnotationProcessor,org.springframework.context.annotation.in
ternalRequiredAnnotationProcessor,org.springframework.context.annotation.
internalCommonAnnotationProcessor,org.springframework.context.annotation.
ConfigurationClassPostProcessor.importAwareProcessor]; root of factory
hierarchy
*** 'any data' retrieved by Groovy / Spring annotation ***
```

Compiling both Java and Groovy sources

In a project such as the one in the preceding section, you have to compile both Java and Groovy sources. To perform that, the easiest way is to set maven-compiler-plugin a little more complex than usual:

```
<plugin>
    <!-- Use this particular configuration of maven-compiler-
     plugin, in order to compile both Java and Groovy sources,
     thanks to groovy-eclipse-compiler-->
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.0</version>
```

```
        <configuration>
            <compilerId>groovy-eclipse-compiler</compilerId>
            <source>1.6</source>
            <target>1.6</target>
        </configuration>
        <dependencies>
            <dependency>
                <groupId>org.codehaus.groovy</groupId>
                <artifactId>groovy-eclipse-compiler</artifactId>
                <version>2.8.0-01</version>
            </dependency>
            <dependency>
                <groupId>org.codehaus.groovy</groupId>
                <artifactId>groovy-eclipse-batch</artifactId>
                <version>2.1.5-03</version>
            </dependency>
        </dependencies>
    </plugin>
```

Maven Shade Plugin

We may have got a similar result with using another plugin: **Maven Shade**.

A complex case

Sometimes, you face situations more complex than the previous case. Then you need to call more advanced features of Maven Assembly:

Study case

Let's consider the `com.packt.dependencyManagement.chapter6:exampleOfDistributions` artifact. It has been built merely through a Maven archetype (`mvn archetype:generate -B -DarchetypeGroupId=org.codehaus.mojo -DarchetypeArtifactId=gwt-maven-plugin -DarchetypeVersion=2.5.1`). In itself it does not have more interest than any other "Hello World" level application. This artifact can build, generate a WAR, and deploy on any servlet container or application server. Let's assume that, in the future, this WAR becomes a wide spread application that is downloaded and installed by many tier parties. As a developer and maintainer of this project, you will be interested in releasing not only the WAR, but a complete **distribution**, which consists of the considered WAR with a ready-to-use (and possibly customized) server, such as Jetty servlet container, or TomEE (a JEE6-certified application server based on Tomcat and OpenEJB). This kind of custom bundle is more frequently used than people think: Sonatype Nexus, as well as various Apache projects, Atlassian products or Liferay actually are (or can be reconfigured as) Jetty servers with an embedded web app.

Maven Assembly plugin answers the frequent case of distribution building, with some exotic folder architecture. We have already seen, in the previous section, how to use a predefined descriptor; now we will dive deeper in custom descriptors.

The target of the current exercise is to build the following with one command:

1. A WAR containing or compiled code with the needed dependencies.
2. A ZIP with a ready-to-use **Jetty 8.1.12** (last branch, 9.X, is not compatible with Java 6).
3. Another ZIP with a ready-to-use **TomEE 1.5.2**.

In order to reach this goal, you will have to call the plugin Maven Assembly. Most of the time, Assembly basic features are sufficient. But in a case like our current one, we have to follow the steps given in the following sections.

Following the process

- Download and install the binary of both servers.
    - Install Jetty archive as a regular artifact:

```
mvn install:install-file \
      -DgroupId=org.eclipse.jetty \
      -DartifactId=jetty-distribution \
      -Dversion=8.1.12.v20130726 \
      -Dpackaging=zip
      -Dfile=/path/to/file/jetty-distribution-
        8.1.12.v20130726.zip
```

    - Install TomEE as a regular archive. Hierarchically, Apache Foundation has classed TomEE as a subproject of OpenEJB (hence, we get the `groupId`).

```
mvn install:install-file \
      -DgroupId=org.apache.openejb \
      -DartifactId=tomee-webprofile \
      -Dversion=1.5.2 \
      -Dpackaging=zip \
      -Dfile=/path/to/file/apache-tomee-1.5.2-
        webprofile.zip
```

      (You can notice we declare the packaging as zip).

We have manually installed the archives on local repositories. This works, but is is not the best way to deal with. Indeed, if your organization owns a private repository (such as Nexus, Artifactory, and Archiva), then the archives must be installed on it; then all team members will retrieve the same archive, downloaded from the common repository.

- In the POM, add regular dependencies to the artifacts you have just installed: `org.eclipse.jetty:jetty-distribution:zip:8.1.12.v20130726` and `org.apache.openejb:tomee-webprofile:zip:1.5.2`.
- Call the Assembly plugin from the POM as follows:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <descriptors>
            <!--Call the descriptor to generate Jetty
             bundle-->
            <descriptor>src/main/assembly/assembly-
              jetty.xml</descriptor>
            <!--Call the descriptor to generate TomEE
             bundle-->
            <descriptor>src/main/assembly/assembly-
             tomee.xml</descriptor>
        </descriptors>
    </configuration>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

- Edit the `assembly-jetty.xml` file. The code is self documented as follows:

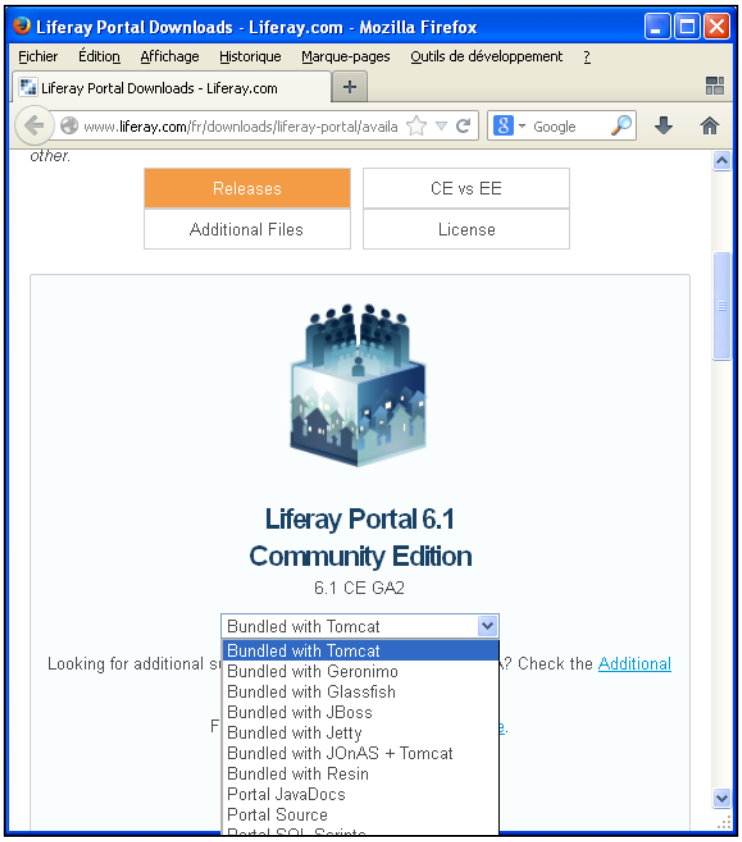
```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance"
      xsi:schemaLocation="http://maven.apache.org/
       plugins/maven-assembly-plugin/assembly/1.1.2
       http://maven.apache.org/xsd/assembly-1.1.2.xsd">
```

```
    <id>distribution-jetty</id>

    <baseDirectory>${project.artifactId}-
      ${project.version}</baseDirectory>
    <formats>
        <!-- The generated distribution will be a ZIP.
         Other available formats include most common ones
         (tar, tar.gz, etc.)-->
        <format>zip</format>
    </formats>
    <includeBaseDirectory>false</includeBaseDirectory>

    <dependencySets>
        <dependencySet>
            <outputDirectory>/</outputDirectory>
            <includes>
                <!-- Get the dependency-->
                <include>org.eclipse.jetty:jetty-
                 distribution:zip:8.1.12.v20130726
                </include>
            </includes>
            <directoryMode>744</directoryMode>
            <!-- Uncompress the archive-->
            <unpack>true</unpack>
            <unpackOptions>
                <excludes>
                   <!-- exclude webapps folder content-->
                    <exclude>**/webapps/**</exclude>
                </excludes>
            </unpackOptions>
        </dependencySet>
    </dependencySets>

    <fileSets>
        <fileSet>
          <directory>${project.build.directory}
          </directory>
            <outputDirectory>/jetty-distribution-
             8.1.12.v20130726/webapps/</outputDirectory>
            <includes>
                <!--include the jar file generated by our
                  project, then copy it into the folder
                  hinted at in the 'outputDirectory' tag,
                  just above-->
                <include>**/${project.artifactId}-
                 ${project.version}.${project.packaging}
                </include>
            </includes>
        </fileSet>
    </fileSets>
</assembly>
```

- Edit `assembly-tomee.xml` and perform the same operations.
- Now you can build the project. As output you will get the following three archives:
    - `exampleOfDistributions-1.0.war`: The web app, as usual
    - `exampleOfDistributions-1.0-distribution-jetty.zip`: A Jetty distribution, ready to unzip, run, and deploy the embedded web app
    - `exampleOfDistributions-1.0-distribution-tomee.zip`: The same for TomEE

Conclusion

- Maven Assemblies allow building and distributing complex and highly customized archives. The preceding example can be adapted to most of the market application servers. This is what Liferay offers: on downloading Liferay, you are offered a choice between several bundles, as shown in the following screenshot:



Distribution management

Any organization should have a central repository, in order to guarantee the consistency of artifacts for all teams. Many solutions exist on the market, either open source or commercial. In this section, we will deal with **Apache Archiva**. however, most of the concepts are common to other systems, such as Sonatype Nexus and JFrog Artifactory:

1. Download (`http://archiva.apache.org/download.cgi`), install, and execute Archiva, let's say on `http://localhost:8080`.
2. Create an admin account and password, let's say `admin/admin1`.
3. In your `settings.xml`, add the following block of code:

```
<servers>
    <server>
        <!-- Repository for releases -->
        <id>archiva.releases</id>
        <username>admin</username>
        <password>admin1</password>
    </server>
    <server>
        <!-- Repository for snapshots -->
        <id>archiva.snapshots</id>
        <username>admin</username>
        <password>admin1</password>
    </server>
</servers>
```

4. In the POM, add the following block of code:

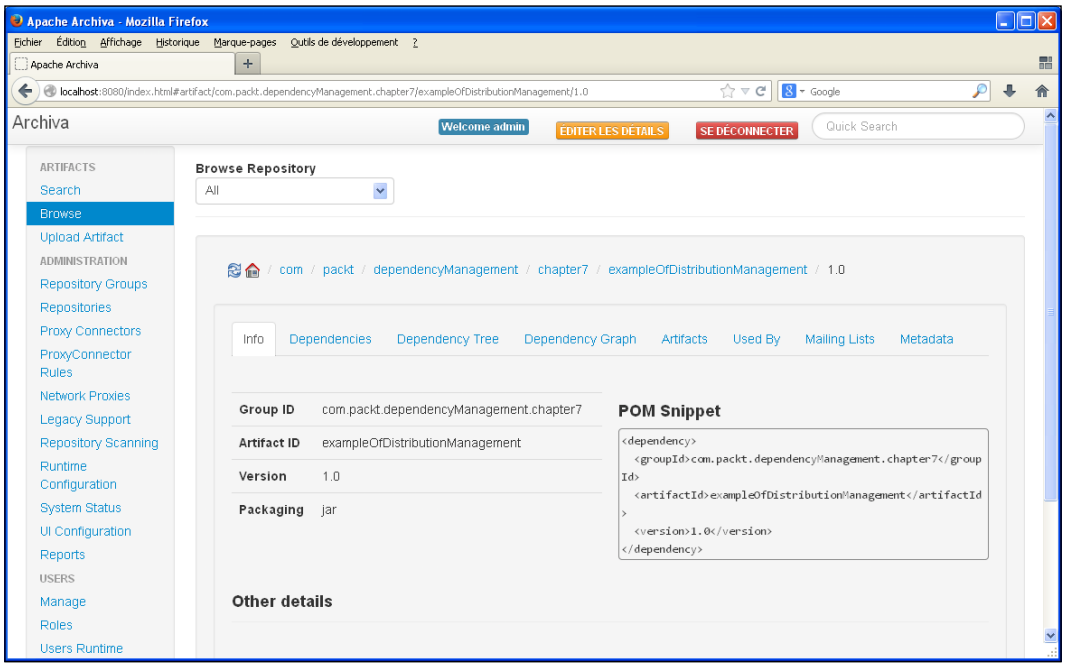
```
<distributionManagement>
    <repository>
        <!-- The distributionManagement.repository.id
         *MUST* match the value of servers.server.id
         in settings.xml-->
        <id>archiva.releases</id>
        <name>Internal Release Repository</name>
        <url>
          http://localhost:8080/archiva/
          repository/internal/
        </url>
    </repository>
    <snapshotRepository>
        <!-- The distributionManagement.repository.
          id *MUST* match the value of
          servers.server.id in settings.xml-->
```

```
            <id>archiva.snapshots</id>
            <name>Internal Snapshot Repository</name>
            <url>
              http://reposerver.mycompany.com:8080/
              archiva/repository/snapshots/
            </url>
        </snapshotRepository>
    </distributionManagement>
```

5. Run the following command:

```
mvn deploy:deploy-file \
    -Dfile=target\exampleOfDistributionManagement-1.0.jar
    -DpomFile=pom.xml
    -DDrepositoryId=archiva.releases
    -Durl=http://localhost:8080/repository/internal/
```

6. The artifact is then deployed and available in Archiva:



Here are a few remarks:

- A good practice is to *discriminate* artifacts according to the version (release or snapshot), and deploy them on different repositories. Snapshots are to be frequently uploaded and downloaded, whereas release versions should seldom be. So, on installing a repository, think of creating two subrepositories (or even two separated repositories running on two different instances and even host and ports).
- As stated earlier, artifacts, at least release ones, should not be uploaded by common developers and committers, but rather only by a "golden source", that is, in most of situations by the automatic and continuous integration system.

Summary

Thus, at the end of this chapter, we are able to:

- Prepare the release
- Detect and fix conflicts between our project dependencies and any tier party
- Deliver simple and complex distributions, thanks to the Maven Assembly plugin
- Perform the actual distribution of our artifacts

Useful Public Repositories

Here is a list of public repositories. They are known to host widely used artifacts. Some repositories discriminate releases and snapshots, others offer channels of staging, nightly, branches, and so on. Most of the time, the release channel is to be favored, but owing to your project's needs, you would select one or more channels.

At last, remind repositories should be set in `settings.xml` rather than in the `pom.xml`.

Maven Central

Maven Central is the main public repository. It contains most of the popular and open sources' artifacts.

```
<repository>
  <id>MavenCentral</id>
  <name>Maven Central, aka iBiblio</name>
  <url>
  http://repo1.maven.org/maven2/
  </url>
</repository>
```

iBiblio

iBiblio offers a public mirror for Maven Central.

```
<repository>
  <id>iBiblio</id>
  <name>iBiblio mirror of MavenCentral</name>
  <url>
    http://mirrors.ibiblio.org/pub/mirrors/maven2
  </url>
</repository>
```

"Local" mirrors exist, according to the region of the world: whereas Central is located in Missouri, iBiblio is hosted in North Carolina. The interest of "local" repositories is to lower latencies and lighten the load on Maven Central, for example, `http://uk.maven.org/maven2` in UK and Antelik's `http://maven.antelink.com/content/repositories/central/` in France (the latter is neither official nor supported by Apache Maven).

JavaNet

JavaNet offers a public repository.

```
<repository>
  <id>java.net.public</id>
  <name>Java.net PUBLIC: contains both snapshots and releases
  </name>
  <url>
    https://maven.java.net/content/repositories/public/
  </url>
  <snapshots>
     <enabled>true</enabled>
  </snapshots>
  <releases>
     <enabled>true</enabled>
  </releases>
</repository>
```

JBoss

Many artifacts, among which those released by Red Hat JBoss, are available on the JBoss repository.

```
<repository>
  <id>JBoss.public</id>
  <name>JBoss</name>
  <url>
    http://repository.jboss.org/nexus/content/groups/public/
   </url>
</repository>
```

CodeHaus

CodeHaus hosts a repository.

```
<repository>
  <id>CodeHaus.repository</id>
  <name>CodeHaus repository</name>
  <url>
    http://repository.codehaus.org/
  </url>
</repository>
```

Apache

Here is the Apache Foundation public repository.

```
<repository>
  <id>Apache.public</id>
  <name>Repository for Apache Foundation projects: releases and
  snapshots</name>
  <url>
    https://repository.apache.org/content/groups/public/
  </url>
  <releases>
     <enabled>true</enabled>
  </releases>
  <snapshots>
     <enabled>true</enabled>
  </snapshots>
</repository>
```

OSS Sonatype

Sonatype, main contributor to Apache Maven, deploys a public Nexus that points to several "small" repositories related to one organization: `http://oss.sonatype.org/`.

Index

Symbols

A

B

C

D

E

F

G

I

J

K

S

T

U

V

W

X

Thank you for buying

Apache Maven Dependency Management

About Packt Publishing

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

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

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

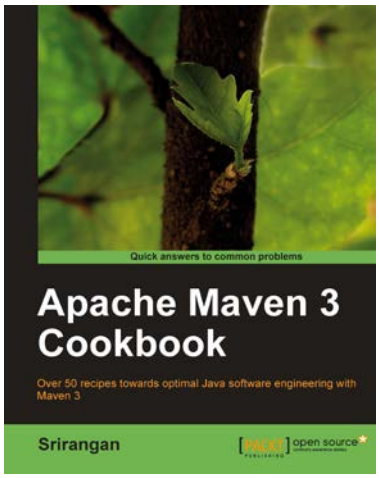
About Packt Open Source

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

Writing for Packt

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

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



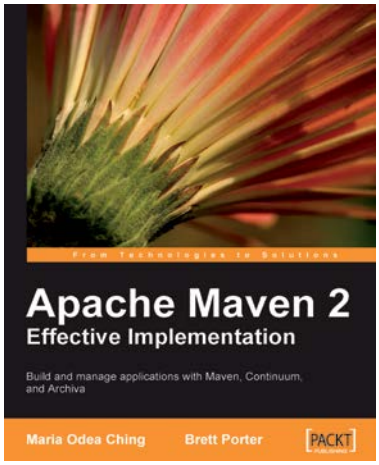
Apache Maven 3 Cookbook

ISBN: 978-1-84951-244-2 Paperback: 224 pages

Over 50 recipes towards optimal Java software engineering with Maven 3

1. Grasp the fundamentals and extend Apache Maven 3 to meet your needs
2. Implement engineering practices in your application development process with Apache Maven
3. Collaboration techniques for Agile teams with Apache Maven
4. Use Apache Maven with Java, Enterprise Frameworks, and various other cutting-edge technologies

---



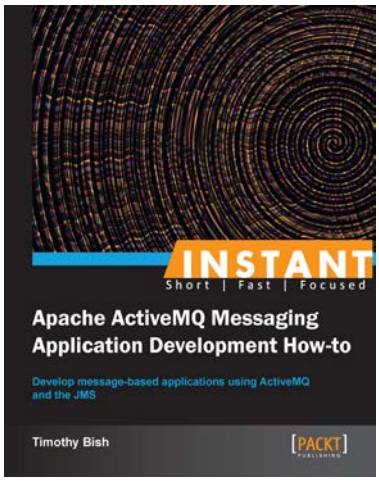
Apache Maven 2 Effective Implementation

ISBN: 978-1-84719-454-1 Paperback: 456 pages

Build and manage applications with Maven, Continuum, and Archiva

1. Follow a sample application which will help you to get started quickly with Apache Maven
2. Learn how to use Apache Archiva - an extensible repository manager - with Maven to take care of your build artifact repository
3. Leverage the power of Continuum - Apache's continuous integration and build server - to improve the quality and maintain the consistency of your build

[PACKT] open source
community experience distilled
PUBLISHING



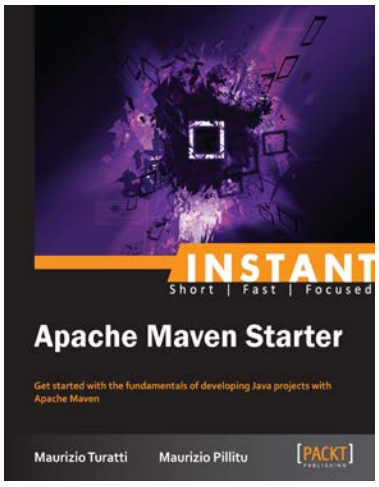
Instant Apache ActiveMQ Messaging Application Development How-to [Instant]

ISBN: 978-1-78216-941-3 Paperback: 78 pages

Develop message-based applications using ActiveMQ and the JMS

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn how to use the JMS API
3. Explore advanced messaging features in ActiveMQ
4. Useful information on common pitfalls new developers often encounter

---



Instant Apache Maven Starter [Instant]

ISBN: 978-1-78216-760-0 Paperback: 62 pages

Get started with the fundamentals of developing Java projects with Apache Maven

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create Java projects and project templates with Maven archetypes
3. Manage project dependencies, project coordinates, and multi-modules
4. Download, install, and configure Maven on different operating systems