Quick answers to common problems

Apache Maven 3 Cookbook

Over 50 recipes towards optimal Java software engineering with Maven 3

Srirangan

[PACKT] open source* community experience distilled PUBLISHING

Apache Maven 3 Cookbook

Over 50 recipes towards optimal Java software engineering with Maven 3

Srirangan

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Apache Maven 3 Cookbook

First published: August 2011

Production Reference: 1180811

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

ISBN 978-1-849512-44-2

www.packtpub.com

Cover Image by Parag Kadam (paragvkadam@gmail.com)

Credits

Author
Srirangan

Reviewer
Carsten Ziegeler
Emmanuel Venisse

Acquisition Editor
Sarah Cullington

Development Editor
Chris Rodrigues

Technical Editor
Priyanka S

Copy Editor
Leonard D'Silva

Project Coordinator
Michelle Quadros

Proofreader
Lisa Brady

Indexer
Hemangini Bari

Graphics
Nilesh Mohite

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Srirangan is a passionate programmer with nine years of experience in freelance, open source, and Enterprise. He has executed projects in a broad range of technologies including Python, PHP, Scala, Java, Adobe Flex, HTML5, Javascript, and so on.

He is the creator of Review19 (`http://www.review19.com`); an innovative, real-time Agile team collaboration and project management tool. He is also involved with India Defence (`http://www.indiadefence.in`) which is India's largest web property dedicated to the defense sector.

Sri is a senior consultant in Inphina Technologies (`http://www.inphina.com`), a rapidly expanding, high-end technology startup in New Delhi focusing on cloud computing (Google App Engine, Hadoop) and emerging technologies.

He is an enthusiastic open source contributor and his open source projects are available on GitHub and BitBucket:

`https://github.com/Srirangan`

`https://bitbucket.org/srirangan`

To know more you can also visit the following links:

Blog - `http://srirangan.net`

Twitter - `http://twitter.com/srirangan`

LinkedIn - `http://www.linkedin.com/in/srirangan`

About the Reviewer

Carsten Ziegeler is a senior developer and software architect for JEE and portal applications at Adobe Systems. He is a member of the Apache Software Foundation and has been participating for more than twenty years in several open source projects. Carsten is a member of several Apache communities and project management committees like Felix, Sling, and Portals. In addition, Carsten is frequently writing articles, reviewing books, and presenting at various conferences.

Emmanuel Venisse has been developing, architecturing, and integrating J2EE applications for thirteen years for banks, government, holiday company projects, and so on. For the last six years, he has worked as a freelancer. For the last eight years, he's been working, in his spare time, on Apache Maven, Continuum and Archiva projects as a core developer and he's also the Continuum project leader. He has contributed to the majority of books written about Apache Maven.

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

Apache Maven is more than just build automation. When positioned at the very heart of your development strategy, Apache Maven can become a force multiplier not just for individual developers but for Agile teams and managers. This book covers implementation of Apache Maven with popular enterprise technologies/frameworks and introduces Agile collaboration techniques and software engineering best practices integrated with Apache Maven.

The *Apache 3 Maven Cookbook* is a real-world collection of step-by-step solutions for individual programmers, teams, and managers to explore and implement Apache Maven and the engineering benefits it brings into their development processes.

This book helps with the basics of Apache Maven and with using it to implement software engineering best practices and Agile team collaboration techniques. It covers a broad range of emergent and enterprise technologies in the context of Apache Maven, and concludes with recipes on extending Apache Maven with custom plugins.

We look at specific technology implementations through Apache Maven, including Java Web Applications, Enterprise Java Frameworks, cloud computing, mobile / device development, and more. We also look at Maven integration with popular IDEs, including Eclipse, NetBeans, and IntelliJ IDEA.

The book is rounded off by exploring how to extend the Apache Maven platform by building custom plugins, integrating them with existing projects, and executing them through explicit command-line calls or with Maven build phases.

What this book covers

Chapter 1, *Basics of Apache Maven*, assists you while you take your first steps with Apache Maven. It will cover setting up Apache Maven on your operating system, verifying the setup, and getting started with your first Apache Maven project.

Chapter 2, *Software Engineering Techniques*, introduces us to implementing software engineering best practices and techniques (such as Test Driven Development, build automation, dependency management, and so on) with Apache Maven.

Chapter 3, *Agile Team Collaboration*, helps you implement collaboration mechanisms with Apache Maven that aid team and distributed development.

Chapter 4, *Reporting and Documentation*, helps you generate reports and documentation, code quality, test coverage reports, and other related metrics and bundles them in a Maven site.

Chapter 5, *Java Development with Maven*, helps you take on web application and enterprise application development challenges with Java while leveraging popular frameworks including Spring and Hibernate.

Chapter 6, *Google Development with Maven*, gets you up and running with Android, Google App Engine, Google Web Toolkit development, and testing with Apache Maven.

Chapter 7, *Scala, Groovy, and Flex*, discusses these popular upcoming technologies and frameworks and gets you started on these with Apache Maven.

Chapter 8, *IDE Integration*, looks at working with Apache Maven directly from your IDE without having to switch back to the terminal.

Chapter 9, *Extending Apache Maven*, shows you how to extend Apache Maven features by development in Java, Apache Ant, and JRuby.

What you need for this book

Apache Maven is based on the Java platform, thus the main requirement is the Java SDK. Recipes in the first chapter guide you on how to set up the Java SDK on your machine. Other than this, Apache Maven automatically downloads all dependencies during execution. Make sure you have fairly good internet access available while working with Apache Maven.

For coding popular IDEs such as IntelliJ, IDEA, Eclipse, and NetBeans can be used. I prefer and recommend IntelliJ IDEA, but even a text editor such as (Scite, TextMate, or Notepad++) should be good enough.

Who this book is for

This book is for Java developers, teams, and managers who want to implement Apache Maven in their development process, leveraging the software engineering best practices and Agile team collaboration techniques it brings along. The book is also specifically for the developer who wishes to get started in Apache Maven and use it with a range of emergent and enterprise technologies including Enterprise Java, Frameworks, Google App Engine, Android, and Scala.

Conventions

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

Code words in text are shown as follows: "The `archetype: generate` command would have generated a sample Apache Maven project for us".

A block of code is set as follows:

```
<mirrors>
  <mirror>
    <id>TestRepository</id>
    <name>My test repository</name>
    <url>http://localhost:8080/nexus-webapp-
      1.8.0/content/repositories/TestRepository/</url>
    <mirrorOf>*</mirrorOf>
  </mirror>
</mirrors>
```

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

```
<groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skip>true</skip>
      </configuration>
```

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

```
$ mvn install -Dmaven.test.skip=true
```

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: "Enter the administrative **Username** and **Password**."

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 book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

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/support`, 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
Basics of Apache Maven

In this chapter, we will cover:

- Setting up Apache Maven on Windows
- Setting up Apache Maven on Linux
- Setting up Apache Maven on Mac
- Verifying the Apache Maven installation
- Creating a new project
- Compiling and testing a project
- Understanding the Project Object Model
- Understanding the build lifecycle
- Understanding build profiles

Apache Maven originated as an attempt to simplify the build process for the now defunct Apache Jakarta Alexandria project. Its formative years were then spent in the Apache Turbine project where it eventually came to replace a brittle and fragile build system based on Apache ANT.

Given Maven's tremendous potency and effectiveness in solving a majority of our day-to-day challenges, it has become hugely popular and is now widely used not only by developers but by other roles in a team including scrum masters, product owners, and project managers.

In recent years, Maven has clearly emerged as an important force-multiplier for Agile teams and organizations.

On its official website, `http://maven.apache.org`, Apache Maven's objectives are listed as:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices in development
- Allowing transparent migration to new features

Whatever the reasons that made you choose Maven (be it build automation, dependency management, standardization, testability, lifecycle management, continuous integration, or any other industry best practice), the recipes in this book will get you up and running in the shortest time possible.

In the upcoming recipes, we will set up Maven on various platforms and host environments followed by selectively exploring the core concepts of the **Project Object Model** and the **Maven build lifecycle**.

Setting up Apache Maven on Windows

We will look at installing and setting up Apache Maven on the Windows operating system. Maven is a command-line tool and needs to be integrated with the Windows environment variables. The process is quite simple and Java dependent.

There is a chance Apache Maven may have been pre-installed on your machine. Verify that Maven isn't already installed before proceeding.

See the recipe *Verifying the Maven installation* in this chapter

Getting ready

As mentioned, a prerequisite for working with Maven is the availability of the Java Development Kit. Make sure that the JDK is available before proceeding. This can be verified by running the following command line:

```
Java -version
```

It will give the following output:

```
java version "1.6.0_21"
Java(TM) SE Runtime Environment (build 1.6.0_21-b06)
Java HotSpot(TM) Client VM (build 17.0-b16, mixed mode, sharing)
```

If you do not have JDK installed, you can download it at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Got JDK? Next you need to get your hands on Maven. You can download it from:

```
http://maven.apache.org/download.html
```

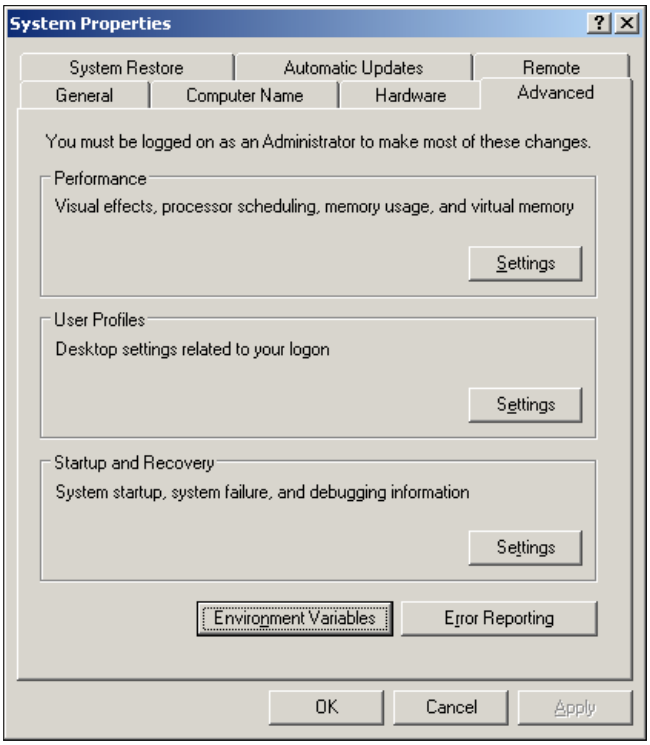
After downloading Maven, extract the archive into any folder. For Windows, it is advised that the path doesn't contain any white-space characters. I extracted Maven in the `D` drive.

```
D:\apache-maven-3.0.2\
```

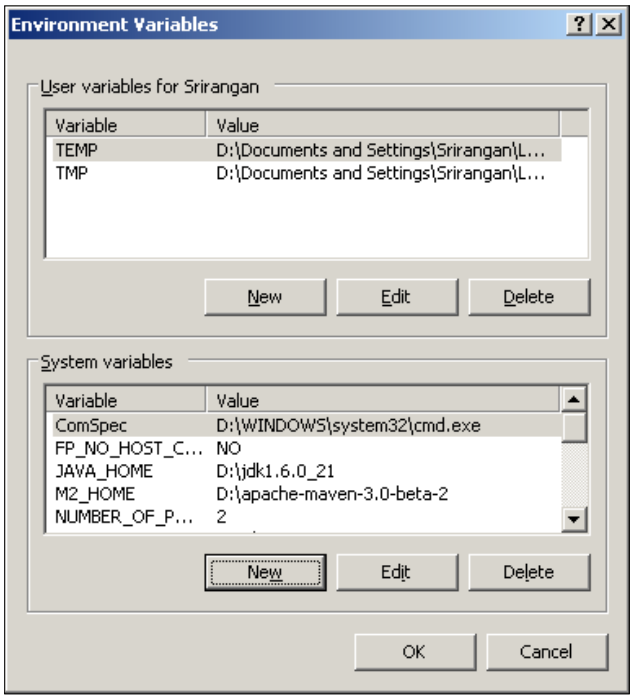
How to do it...

To start using Maven, we need to configure Windows environment variables. The `M2_HOME` variable needs to be set and the `PATH` variable needs to be modified to include the Maven binaries folder.

You can set the environment variables by accessing the **System** settings in the **Control Panel.**



Select **Environment Variables** and then the **New** button to create a new environment variable.



Create a new environment variable for `M2_HOME` pointing to the Maven base directory. For me, the value of `M2_HOME` will be `D:\apache-maven-3.0.2\`.

The `PATH` environment variable will already exist. Select it and click **Edit** to modify. It must be modified by appending the following text at the end:

```
;%M2_HOME%\bin
```

Apache Maven is now ready and available for use in the command line. It is also available for integration with IDEs and other development tools, but more on that in upcoming chapters.

See also

- *Setting up Apache Maven on Linux* in this chapter
- *Setting up Apache Maven on Mac* in this chapter
- *Verifying your Apache Maven installation* in this chapter
- *Working with Eclipse and Maven* in *Chapter 8*, *IDE Integration*
- *Working with NetBeans and Maven* in *Chapter 8*, *IDE Integration*
- *Working with IntelliJ and Maven* in *Chapter 8*, *IDE Integration*

Setting up Apache Maven on Linux

The Linux distribution used in this book is Ubuntu 10.04. If you use any other Linux distribution, the steps should nevertheless remain similar. Apache Maven is a command-line tool; it just needs to be extracted and configured with the operating system's environment.

There is a chance Apache Maven may have been pre-installed on your machine. Verify that Maven isn't already installed before proceeding.

See the recipe *Verifying the Maven installation* in this chapter

Getting ready

A prerequisite for working with Maven is the availability of the Java Development Kit. Make sure the JDK is available before proceeding. This can be verified by running the command line:

```
java -version
```

It should give the following output:

```
java version "1.6.0_18"
OpenJDK Runtime Environment (IcedTea6 1.8.1) (6b18-1.8.1-0ubuntu1)
OpenJDK Client VM (build 16.0-b13, mixed mode, sharing)
```

If you do not have JDK installed, you can download it at: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Got JDK? Next you need to get your hands on Maven. You can download it from: `http://maven.apache.org/download.html`.

After downloading Maven, extract the archive into a folder. For example, I extracted Maven into my home folder, `/home/srirangan/apache-maven-3.0.2/`. This will be an installation for a single user.

How to do it...

The next step is to add commands for exporting the `PATH` and `M2_HOME` environment variables in the `.bashrc` file. This file can be found in the user's home folder, which for me is `/home/srirangan/.bashrc`.

```
export M2_HOME=/home/srirangan/apache-maven-3.0.2
export PATH=${PATH}:${M2_HOME}/bin
```

Apache Maven is now ready and available for use in the command line. It is also available for integration with IDEs and other development tools, but more on that in upcoming chapters.

See also

- *Setting up Apache Maven on Windows* in this chapter
- *Setting up Apache Maven on Mac* in this chapter
- *Verifying your Apache Maven installation* in this chapter
- *Working with Eclipse and Maven* in *Chapter 8*, *IDE Integration*
- *Working with NetBeans and Maven* in *Chapter 8*, *IDE Integration*
- *Working with IntelliJ and Maven* in *Chapter 8*, *IDE Integration*

Setting up Apache Maven on Mac

Installing Maven on the Mac OS X isn't very different from the installation and setup on Linux. This really shouldn't be a surprise because OS X is built on top of BSD Linux in the first place. Apache Maven is a command-line tool; it just needs to be extracted and configured with the operating system's environment.

There is a chance Apache Maven may have been pre-installed on your machine. Verify that Maven isn't already installed before proceeding.

See the recipe *Verifying the Maven installation* in this chapter

Getting ready

A pre-requisite for working with Maven is the availability of the Java Development Kit. Make sure the JDK is available before proceeding. This can be verified by running the following command line:

```
java -version
```

It should give the following output:

```
java version "1.5.0_19"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_19-b02-306)
Java HotSpot(TM) Client VM (build 1.5.0_19-138, mixed mode, sharing)
```

If you do not have JDK installed, you can download it at:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

Got JDK? Next you need to get your hands on Maven. You can download it from:

```
http://maven.apache.org/download.html
```

After download, extract the archive into a folder. For example, you can extract the archive in `/usr/local/maven/`. This can, of course, be any directory of your choice and not `/usr/local/maven` in particular.

How to do it...

The next step is to add commands to export the `PATH` and `M2_HOME` environment variables in the `.bash_login` file.

```
export M2_HOME=/usr/local/maven
export PATH=${PATH}:${M2_HOME}/bin
```

Apache Maven is now ready and available for use in the command line. It is also available for integration with IDEs and other development tools, but more on that in upcoming chapters.

See also

- *Setting up Apache Maven on Windows* in this chapter
- *Setting up Apache Maven on Linux* in this chapter
- *Verifying your Apache Maven installation* in this chapter
- *Working with Eclipse and Maven* in *Chapter 8*, *IDE Integration*
- *Working with NetBeans and Maven* in *Chapter 8*, *IDE Integration*
- *Working with IntelliJ and Maven* in *Chapter 8*, *IDE Integration*

Verifying the Apache Maven installation

Did you just try and install Apache Maven? It's time to verify it. Alternatively, your operating system may have Apache Maven pre-installed.

Whether user installed or pre-installed, here's how to verify that your workstation has Apache Maven extracted and the operating system configured correctly.

How to do it...

- You have installed Maven on your system, or so you think? Before you start using it, the setup and availability of Maven needs to be verified. This is done by executing the following command line:

  ```
  mvn -version
  ```

- If Maven has been correctly installed, you will see something resembling the following output:

  ```
  Apache Maven 3.0.2
  Java version: 1.6.0_18
  Java home: /usr/lib/jvm/java-6-openjdk/jre
  Default locale: en_IN, platform encoding: UTF-8
  OS name: "linux" version: "2.6.32-24-generic" arch: "i386" Family:
  "unix"
  ```

  If the output you get is similar, then you know Maven is available and ready to be used.

- If your operating system cannot find the `mvn` command, make sure that your `PATH` environment variable and `M2_HOME` environment variable have been properly set.

See also

- *Creating a new project* in this chapter

Creating a new project

All done with downloads, setups, configurations, installations, and verifications? Great! Let's get down to business. If not, refer to the first few recipes of this chapter.

In this recipe, we will create our first Apache Maven project. To be more specific, we will use the Maven `archetype:generate` goal to generate our first Maven Java project.

Then we take a look at the Maven project structure to get an idea of what constitutes a Maven project and what goes where by convention.

How to do it...

- Start the command-line terminal and run the following command:

  ```
  $ mvn archetype:generate
  ```

- If this is the first time you are running this command, you will see that downloads are taking place in the command line.

- Then you will see a rather large list of archetypes, each having a number, a name, and a short description explaining what they are. We'll look at what archetypes are a little later. For now, select the default archetype. Here we have considered that it is archetype number `101` named `maven-archetype-quickstart`.

```
[INFO] No archetype defined. Using maven-archetype-quickstart
(org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
1: remote -> docbkx-quickstart-archetype (-)
….
100: remote -> maven-archetype-profiles (-)
101: remote -> maven-archetype-quickstart (An archetype which
contains a sample Maven project.)
…
375: remote -> javg-minima
Choose a number: 101:
```

- You will be asked to select a version of the archetype. The `default` is the latest stable version of the archetype; let's choose that.
- Next, you will have to enter the Maven "project co-ordinates" such as `groupId`, `artifactId`, `version`, and `package`.

  `GroupId` co-ordinates are used to specify the hierarchical location of a project within a Maven repository. In this case, the repository is the local Maven repository present in your filesystem. `GroupId` co-ordinates are typically the root package and thus can be shared by multiple projects within an organization.

  The `artifactId` is an identifier for your project and `version` here refers to the project version. Packages refer to the `reverse dns` root package name that is commonly used in Java and a host of other programming languages. `artifactID` identifiers will be used when artifacts are deployed in repositories and used as dependencies for other projects; more on that later, as it's not important right now.
- Upon completion you should see the following:

```
[INFO] ------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------
```

How it works...

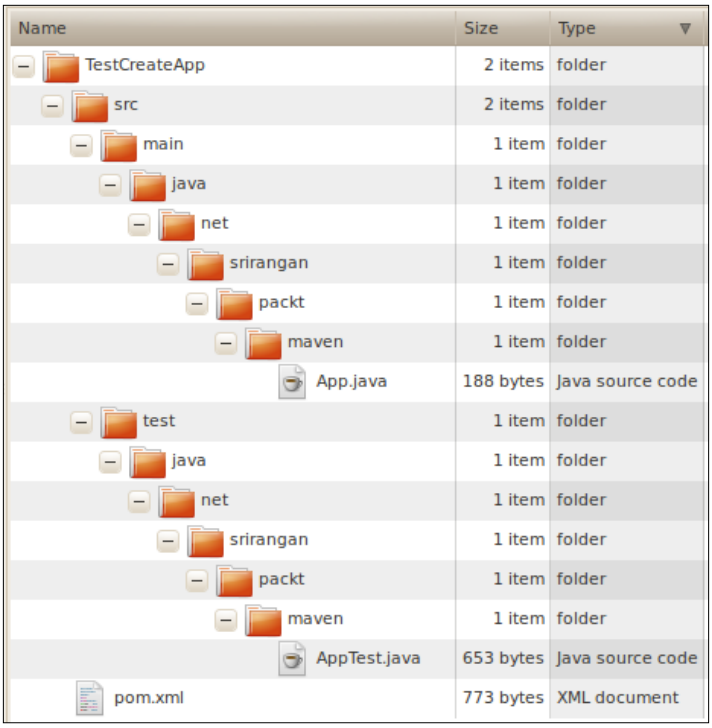
We just asked Maven, actually a Maven plugin, to create or rather generate a new project for us. A new project for Apache Maven can either be "hand-crafted" with manual `pom.xml` and folder creations or generated through Maven project archetypes.

So what exactly has Maven created for us? Simple answer—the following:

```
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- net
    |           `-- srirangan,oackt
    |               `-- maven
    |                   `-- App.java
    `-- test
        `-- java
            `-- net
                `-- srirangan,oackt
                    `-- maven
                        `-- AppTest.java

11 directories, 3 files
```

It created a folder for the project which contains the main **pom.xml** file and the source folder which contains subfolders for the application and test sources. These then contain the complete package structure and a sample application with a unit test case.



Compiling and testing a project

Let's hope that you've read the previous recipe titled *Creating a new project* or that you have a Maven project available because now we're entering the interesting part; compiling and testing the project using Maven.

If you're new to Apache Maven, which is probably why you have this book in your hands, this will be your first introduction to the Apache Maven build lifecycle.

Getting ready

- Start your command-line terminal window
- Navigate to the project folder that contains the `pom.xml` file
- Run the following command:

```
mvn compile
```

How to do it...

Now Apache Maven begins to download dependencies, if they aren't available in your local repository, and then proceeds to compile the project.

```
[INFO] ------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------
```

If the terminal shows `BUILD SUCCESS`, it means Maven has finished compilation and build of the application.

Test Driven Development (**TDD**) is a popular practice that is advocated for and followed religiously by some of the very best software craftsmen in the industry. Maven, having recognized this, makes testing part of the default build lifecycle. This makes TDD easier for teams who are trying to implement it in their development process.

When the default conventions are being used in a Maven project, they have the `src/test` directory that contains all the tests for the code. Run the following command to run the tests:

```
mvn test
```

This command triggers Maven to run the tests present in the `.../src/test` folder. When this is completed, you'll see a brief report echoed on the terminal.

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

```
Running net.srirangan.packt.maven.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.037
sec

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

How it works...

Run the following command:

```
$ mvn compile
```

It will trigger the Java compiler associated with the project. By default, it is set to JDK1.5, but the project `pom.xml` can be modified to use other versions.

The compiled code is placed in the "target" directory. The target directory will contain the compiled artifact (that is, a JAR file for a Java project by default) along with directories for compiled classes and tests. It will further contain the `pom.properties` file along with test reports and temporary files.

```
├──TestSimpleApp-1.0-SNAPSHOT.jar
├──classes
│   └──net
│       └──srirangan
│           └──packt
│               └──maven
│                   └──App.class
├──maven-archiver
│   └──pom.properties
├──surefire
│   └── ...temp files...
├──surefire-reports
│   └──net.srirangan.packt.maven.AppTest.txt
│   └──TEST-net.srirangan.packt.maven.AppTest.xml
└──test-classes
    └──net
        └──srirangan
            └──packt
                └──maven
|                   └──AppTest.class
```

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.

See also

- *Understanding the Project Object Model* in this chapter
- *Understanding the build lifecycle* in this chapter
- *Understanding build profiles* in this chapter

Understanding the Project Object Model

Every Apache Maven project contains a `pom.xml` file. The `pom.xml` file is the XML representation of the project and thus contains all metadata pertaining to the project.

This includes project configuration, defect tracking system details, project organization and licenses, project paths, dependencies, and so on.

Structure of a POM

The structure of a typical Apache Maven Project POM file is described as follows:

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

  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>

  <!-- Project Meta Data -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <organization>...</organization>
```

```
    <developers>...</developers>
    <contributors>...</contributors>

    <!-- Environment -->
    <issueManagement>...</issueManagement>
    <ciManagement>...</ciManagement>
    <mailingLists>...</mailingLists>
    <scm>...</scm>
    <prerequisites>...</prerequisites>
    <repositories>...</repositories>
    <pluginRepositories>...</pluginRepositories>
    <distributionManagement>...</distributionManagement>
    <profiles>...</profiles>
</project>
```

Project co-ordinates

Project co-ordinates are the minimal basic fields that a POM definition must contain. The three co-ordinate fields are `groupId`, `artifactId`, and `version`.

These three fields mark a specific location within the repository, hence the term *co-ordinates*.

Sections of the POM

The previously displayed sample POM contains four major sections. They are explained as follows:

- The basics: This section contains project co-ordinates, dependency management, and inheritance details. Additionally, it also contains modules and project level properties.
- Build settings: This section contains the build details.
- Project metadata: This section contains project-specific details such as name, organization, developers, URL, inception year, and so on.
- Environment: This section contains all information regarding the environment including details of the version control being, issue management, continuous integration, mailing lists, repositories, and so on.

See also

- *Understanding the build lifecycle* in this chapter
- *Understanding build profiles* in this chapter

Understanding the build lifecycle

The build lifecycle explicitly defines the process of building, testing, distributing an artifact, and is at the heart of every Maven project.

There are three inbuilt build lifecycles: default, clean, and site.

Default lifecycle

The default lifecycle handles the project compilation, test, and deployment. While it contains over 20 build phases, the following are the most important phases:

- Validate: validates that all project information is available and is correct
- Compile: compiles the source code
- Test: runs unit tests within a suitable framework
- Package: packages the compiled code in its distribution format
- Integration-test: processes the package in the integration-test environment
- Verify: runs checks to verify that the package is valid
- Install: installs the package in the local repository
- Deploy: installs the final package in a remote repository

Whenever you execute a build phase, all prior build phases are executed sequentially. Hence, executing `mvn integration-test` will execute the validate, compile, test, and package build phases before executing the integration-test build phase.

Clean lifecycle

The clean lifecycle handles the project cleaning and contains the following build phases:

- Pre-clean: executes processes required before project cleaning
- Clean: removes all files generated by previous builds
- Post-clean: executes processes required to finalize project cleaning

Site lifecycle

The site lifecycle handles the generation and deployment of the project's site documentation:

- Pre-site: executes processes required before generation of the site
- Site: generates the project's site documentation

- Post-site: executes processes required to finalize the site generation and prepares the site for deployment
- Site-deploy: deploys the site documentation to the specified web server

See also

- *Understanding the Project Object Model* in this chapter
- *Understanding build profiles* in this chapter

Understanding build profiles

Projects in Maven are generally portable. This is done by allowing configuration within the POM, avoiding all filesystem references, and depending extensively on the local repository to store the required metadata.

However, this isn't always possible as under some circumstances configuration with filesystem references become unavoidable. For these cases, Maven introduces the concept of build profiles.

Build profiles are specifications made in the POM and can be triggered as and when required. Some ways to trigger profiles are:

- Explicit command-line trigger
- Maven settings trigger
- Environment specific trigger

Explicit command-line trigger

Profiles can be directly triggered through the command line using the `-P` option. The list of profiles, separated by commas, that are to be activated should be mentioned after the `-P` flag:

```
mvn install -P profile-1,profile-2
```

In this case, only the profiles explicitly mentioned in the command will be activated and all other profiles stay dormant for this build.

The reverse is possible as well. You can explicitly specify which profiles should not be activated through the command line:

```
mvn install -P !profile-1,!profile-2
```

Maven settings trigger

Maven settings can also directly activate profiles if they are specified in the `<activeProfiles>` section of the settings file.

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>profile-1</activeProfile>
    <activeProfile>profile-2</activeProfile>
  </activeProfiles>
  ...
</settings>
```

Profiles listed in the Maven settings get activated by default every time and they don't need any explicit specification in the command line.

Environment specific trigger

Profiles can also be triggered based on the current build environment. The environment in which the profile is to be activated is directly defined within the POM profile declarations.

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>environment</name>
        <value>dev</value>
      </property>
    </activation>
  </profile>
</profiles>
```

In the code just written, the profile is only activated in the `dev` build environment. A template of a typical Apache Maven command, wherein the environment is specified explicitly in the command, is given as follows:

```
mvn groupId:artifactId:goal -Denvironment=dev
```

See also

- *Understanding the Project Object Model* in this chapter
- *Understanding the build lifecycle* in this chapter

2
Software Engineering Techniques

In this chapter, we will cover some of the most prevalent, popular, and proven software engineering practices like:

- Build automation
- Project modularization
- Dependency management
- Source code quality checks
- Test driven development
- Acceptance testing automation
- Deployment automation

These techniques have been around for more than a decade and are well-known by practitioners of software engineering. The benefits, trade-offs, and pros and cons of these practices are well-known and will only need little mentioning.

These practices are not inter-dependent, but some of them are inter-related in the larger scheme of things. One such example would be the relation between project modularization and dependency management. While nothing stops either from being implemented in isolation, they are more beneficial when implemented together.

These techniques can be further supplemented by the industry's best practices such as continuous integration, maintaining centralized repositories, source code integration, and so on, which will be studied in detail in *Chapter 3*, *Agile Team Collaboration.*

Our focus here will be on steadily understanding these software engineering techniques within the context of Maven projects and we will look at practical ways to implement and integrate them.

Build automation

Build automation is the scripting of tasks that software developers have to do on a day-to-day basis. These tasks include:

- Compilation of source code to binary code
- Packaging of binary code
- Running tests
- Deployment to remote systems
- Creation of documentation and release notes

Build automation offers a range of benefits including speeding up of builds, elimination of bad builds, standardization in teams and organizations, increased efficiency, and improvements in product quality. Today, it is considered as an absolute essential for software engineering practitioners.

Getting ready

You need to have a Maven project ready. If you don't have one, run the following in the command line to create a simple Java project:

```
$ mvn archetype:generate -DgroupId=net.srirangan.packt.maven
-DartifactId=MySampleApp
```

How to do it...

The `archetype:generate` command would have generated a sample Apache Maven project for us. If we choose the `maven-archetype-quickstart` archetype from the list, our project structure would look similar to the following:

```
└──src
    ├──main
    │   └──java
    │       └──net
    │           └──srirangan
    │               └──packt
    │                   └──maven
    └──test
        └──java
            └──net
                └──srirangan
                    └──packt
                        └──maven
```

In every Apache Maven project, including the one we just generated, the build is pre-automated following the default build lifecycle. Follow the steps given next to validate the same:

1. Start the command-line terminal and navigate to the root of the Maven project.
2. Try running the following commands in serial order:

```
$ mvn validate
...
$ mvn compile
...
$ mvn package
...
$ mvn test
...
```

You just triggered some of the phases of the build life cycle by individual commands. Maven lets you automate the running of all the phases in the correct order. Just execute the following command, `mvn install`, and it will encapsulate much of the default build lifecycle including compiling, testing, packaging, and installing the artifact in the local repository.

How it works...

For every Apache Maven project, regardless of the packaging type, the default build lifecycle is applied and the build is automated. As we just witnessed, the default build lifecycle consists of phases that can be executed from the command-line terminal.

These phases are:

- `Validate`: Validates that all project information is available and correct
- `Compile`: Compiles the source code
- `Test`: Runs unit tests within a suitable framework
- `Package`: Packages the compiled code in its distribution format
- `Integration-test`: Processes the `package` in the integration test environment
- `Verify`: Runs checks to verify if the `package` is valid
- `Install`: Installs the `package` in the local repository
- `Deploy`: Installs the final `package` in a remote repository

Each of the build lifecycle phases is a Maven plugin. We will have a detailed look at Apache Maven plugins in *Chapter 9*, *Extending Apache Maven*. When you execute them for the first time, Apache Maven will download the plugin from the default online Maven Central Repository that can be found at `http:// repo1.maven.org/maven2` and will install it in your local Apache Maven repository.

This ensures that build automation is always set up in a consistent manner for everyone in the team, while the specifics and internals of the build are abstracted out.

Maven build automation also pushes for standardization among different projects within an organization, as the commands to execute build phases remain the same.

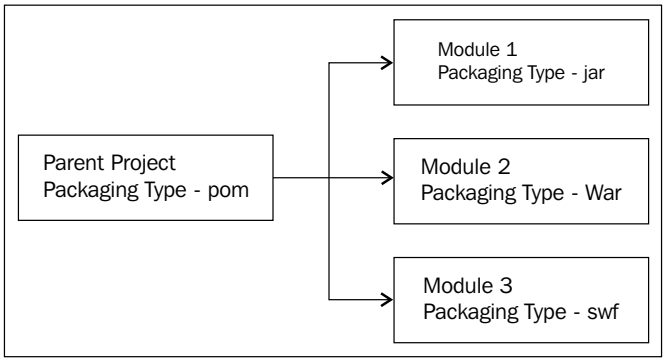
See also

- *The project object model* section in *Chapter 1*, *Basics of Maven*
- *Test driven development* section in *Chapter 2*, *Software Engineering Techniques*
- *Deployment automation* section in *Chapter 2*, *Software Engineering Techniques*

Project modularization

Considering that you're building a large enterprise application, it will need to interact with a legacy database, work with existing services, provide a modern web and device capable user interface, and expose APIs for other applications to consume. It does make sense to split this rather large project into subprojects or **modules**.

Apache Maven provides impeccable support for such a project organization through Apache Maven **Multi-modular projects**. Multi-modular projects consist of a "Parent Project" which contains "Child Projects" or "Modules". The parent project's `POM` file contains references to all these sub-modules. Each module can be of a different type, with a different packaging value.



Getting ready

We begin by creating the parent project. Remember to set the value of `packaging` to `pom`, as highlighted in the following code:

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

```
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>net.srirangan.packt.maven</groupId>
  <artifactId>TestModularApp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <name>MyLargeModularApp</name>

</project>
```

This is the base parent `POM` file for our project `MyLargeModularApp`. It doesn't contain any sub-modules for now.

How to do it...

To create your first sub-module, start the command-line terminal, navigate to the parent `POM` directory, and run the following command:

```
$ mvn archetype:generate
```

This will display a list of archetypes for you to select. You can pick archetype number 101, `maven-archetype-quickstart`, which generates a basic Java project. The `archetype:generate` command also requires you to fill in the Apache Maven project co-ordinates including the project `groupId`, `artifactId`, `package`, and `version`.

After project generation, inspect the `POM` file of the original parent project. You will find the following block added:

```
<modules>
  <module>moduleJar</module>
</modules>
```

The sub-module we created has been automatically added in the parent `POM`. It simply works—no intervention required!

We now create another sub-module, this time a Maven web application by running the following in the command line:

```
$ mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp
```

Let's have another look at the parent `POM` file; we should see both the sub-modules included:

```
<modules>
  <module>moduleJar</module>
  <module>moduleWar</module>
</modules>
```

Our overall project structure should look like this:

```
MyLargeModularApp
├──MyModuleJar
│   └──src
│       ├──main
│       │   └──java
│       │       └──net
│       │           └──srirangan
│       │               └──packt
│       │                   └──maven
│       └──test
│           └──java
│               └──net
│                   └──srirangan
│                       └──packt
│                           └──maven
└──MyModuleWar
    └──src
        └──main
            ├──resources
            └──webapp
                └──WEB-INF
```

How it works...

Compiling and installing both sub-modules (in the correct order in case sub-modules are interdependent) is essential. It can be done in the command line by navigating to the parent `POM` folder and running the following command:

```
$ mvn clean install
```

Thus, executing build phase on the parent project automatically gets executed for all its child projects in the correct order.

You should get an output similar to:

```
------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] MyLargeModularApp ..................... SUCCESS [0.439s]
[INFO] MyModuleJar ........................... SUCCESS [3.047s]
[INFO] MyModuleWar Maven Webapp .............. SUCCESS [0.947s]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
------------------------------------------------------------------------
```

Dependency management

Dependency management can be universally acknowledged as one of the best features of Apache Maven. In Multi-modular projects, where dependencies can run into tens or even hundreds, Apache Maven excels in allowing you to retain a high degree of control and stability.

Apache Maven dependencies are transient, which means Maven will automatically discover artifacts that your dependencies require. This feature has been available since Maven 2, and it especially comes in handy for many of the open source project dependencies we have in today's enterprise projects.

Getting ready

Maven dependencies have six possible scopes:

- `Compile`: This is the default scope. `Compile` dependencies are available in the classpaths.
- `Provided`: This scope assumes that the JDK or the environment provides dependencies at runtime.
- `Runtime`: Dependencies that are required at runtime and are specified in the runtime classpaths.
- `Test`: Dependencies required for test compilation and execution.
- `System`: Dependency is always available, but the JAR is provided nonetheless.
- `Import`: Imports dependencies specified in `POM` included via the `<dependencyManagement/>` element.

How to do it...

Dependencies for Apache Maven projects are described in project POM files. While we take a closer look at these in the *How it works...* section of this recipe, here we will explore the Apache Maven dependency plugin.

According to `http://maven.apache.org/plugins/maven-dependency-plugin/`:

> *"The dependency plugin provides the capability to manipulate artifacts. It can copy and/or unpack artifacts from local or remote repositories to a specified location."*

It's a decent little plugin and provides us with a number of very useful goals. They are as follows:

```
$ mvn dependency:analyze
Analyzes dependencies (used, unused, declared, undeclared)

$ mvn dependency:analyze-duplicate
Determines duplicate dependencies

$ mvn dependency:resolve
Resolves all dependencies

$ mvn dependency:resolve-plugin
 Resolves all plugins

$ mvn dependency:tree
Displays dependency trees
```

How it works...

Most Apache Maven projects have dependencies to other artifacts (that is, other projects, libraries, and tools). Management of dependencies and their seamless integration is one of Apache Maven's strongest features. These dependencies for a Maven project are specified in the project's `POM` file.

```
<dependencies>
  <dependency>
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <scope>...</scope>
  </dependency>
</dependencies>
```

In Multi-modular projects, dependencies can be defined in the parent `POM` files and can be subsequently *inherited* by child `POM` files as and when required. Having a single source for all dependency definitions makes dependency versioning simpler, thus keeping large projects' dependencies organized and manageable over time.

The following is an example to show a Multi-modular project having a MySQL dependency. The parent `POM` would contain the complete definition of the dependency:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.2</version>
    </dependency>
  <dependencies>
</dependencyManagement>
```

All child modules that require MySQL would only include a stub dependency definition:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

There will be no version conflicts between multiple child modules having the same dependencies.

The dependencies scope and type are defaulted to `compile` and JAR. However, they can be overridden as required:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <type>war</type>
</dependency>
```

There's more...

System dependencies are not looked for in the repository. For them, we need to specify the path to the JAR:

```
<dependencies>
  <dependency>
    <groupId>sun.jdk</groupId>
    <artifactId>tools</artifactId>
    <version>1.5.0</version>
    <scope>system</scope>
    <systemPath>${java.home}/../lib/tools.jar</systemPath>
  </dependency>
</dependencies>
```

However, avoiding the use of system dependencies is strongly recommended because it kills the whole purpose of Apache Maven dependency management in the first place.

Ideally, a developer should be able to clone code out of the SCM and run Apache Maven commands. After that, it should be the responsibility of Apache Maven to take care of including all dependencies.

System dependencies would force the developer to take extra steps and that dilutes the effectiveness of Apache Maven in your team environment.

Source code quality checks

In a perfect world, you are the best programmer alive and you are effortlessly able to mind control every other programmer in your entire team and force him / her to code the way you want.

In the real world, you and your team can, at best, agree upon a set of programming standards and implement automated source code quality checks into your project build to verify that a certain level of code quality is upheld.

These automated source code quality checks and verifications still cannot ensure that the application itself is designed correctly. However, it can help some of the lesser experienced programmers adhere to standards expected of them.

Getting ready

The Apache Maven PMD plugin automatically runs the PMD code analysis tool on the source code and generates a site report with results. In a typical configuration, the build fails if PMD detects quality issues in the source.

This plugin introduces four goals:

- `pmd:pmd` creates a PMD site report based on the rulesets and configuration set in the plugin
- `pmd:cpd` generates a report for PMD's Copy/Paste Detector (CPD) tool
- `pmd:check` verifies that the PMD report is empty and fails the build if it is not
- `pmd:cpd-check` verifies that the CPD report is empty and fails the build if it is not

How to do it...

The following steps need to be taken to integrate source code quality checks into your Apache Maven project's build cycle.

If you don't have an Apache Maven Java project, create one by running the following goal:

```
mvn archetype:generate
```

Launch the project's `POM` file in a text editor for editing. The PMD plugin needs to be integrated into your project. It can be added to the project `POM` file under the `reporting` element:

```
<reporting>
  <plugins>
    <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
     <version>2.5</version>
  </plugin>
  </plugins>
</reporting>
```

This can be used to run the PMD checks with default rulesets and configuration.

Here's an optional step: if you wish to use a custom set of rules and configuration for code-quality checks, it can be done by adding a configuration block to the plugin declaration. Have a look at the following code:

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

```
        <version>2.5</version>
        <configuration>
          <rulesets>
            <ruleset>/rulesets/basic.xml</ruleset>
            <ruleset>/rulesets/controversial.xml</ruleset>
            <ruleset>http://localhost/design.xml</ruleset>
          </rulesets>
          <sourceEncoding>utf-8</sourceEncoding>
          <minimumTokens>100</minimumTokens>
          <targetJdk>1.6</targetJdk>
        </configuration>
      </plugin>
```

To execute these PMD checks, start the command line, navigate to the project `POM` folder, and execute the `pmd` goal in the `pmd` plugin, as shown as follows:

```
mvn pmd:pmd
```

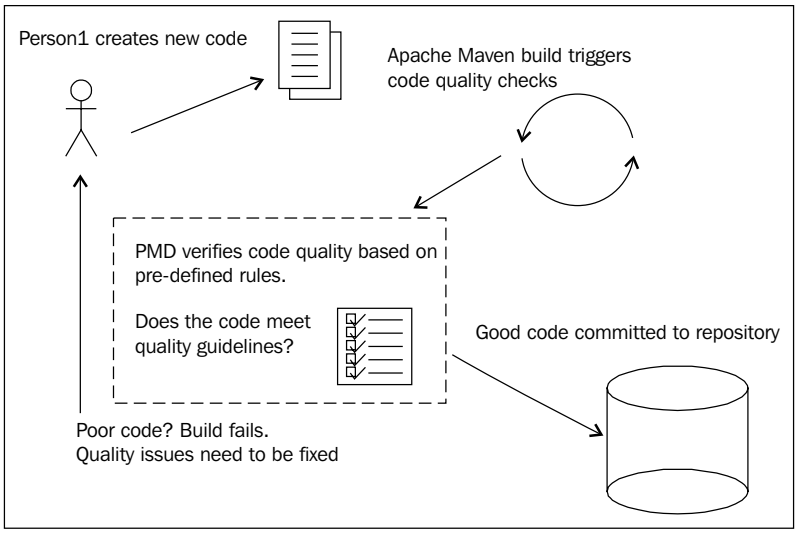
The PMD checks can be integrated with Maven's default build lifecycle, as shown in the following code:

```
<build>
  <plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
    <version>2.5</version>
    <executions>
      <execution>
      <goals>
        <goal>check</goal>
      <goal>cpd-check</goal>
      </goals>
    </execution>
    </executions>
  </plugin>
  </plugins>
</build>
```

How it works...

PMD is an open source tool that scans Java code and generates code quality reports. The reports are generated based on identification of potential bugs, dead code, non-optimized code, duplicate code, and so on.

The following diagram visualizes the build cycle with a quality check integrated with the repository:



Embedding the Apache Maven PMD plugin eliminates the need to otherwise install or configure PMD as a third-party application.

As we just saw, there are two ways to invoke PMD's code quality checks. You could either do it manually or automatically. In the first case, the individual developer would be responsible for executing code quality check each time a change is made. This approach needlessly adds another task for the already busy programmer. It makes more sense to follow the second approach for full automation.

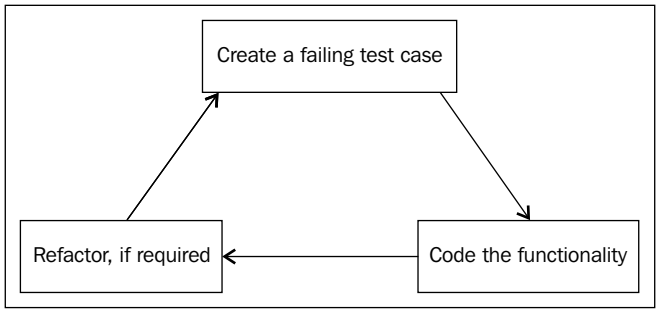
Source code quality checks are most beneficial when they are fully integrated with your build automation and continuous integration processes. Whenever a build is run on the developer's machine, Apache Maven automatically executes the PMD plugin. And when the code is committed into the SCM, the SCM triggers an Apache Maven build that would execute the PMD plugin automatically. If the new code fails to meet the code quality standards, the build should fail and the team must be automatically notified.

Test Driven Development

Test Driven Development's origins can be traced to the Test-First programming concept introduced by Extreme Programming in 1999.

Test Driven Development or TDD, as it's more commonly known, introduces very short, iterative development cycles wherein the programmer first writes a failing test case, then builds the functionality followed by code refactoring, if required.

Apache Maven makes unit testing and integration testing an integral part of the build lifecycle, thus enabling individual programmers and teams to easily implement the practice of TDD.



Getting ready

We will need a simple Apache Maven Java project to get started with TDD. If you don't have one, generate a Java project using the `archetype:generate` command:

```
$ mvn archetype:generate
```

We'll use the popular JUnit framework to create a unit test. Mention the dependency in your `POM`:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>
```

If you generated your project using the `maven-archetype-quickstart` archetype, it would already contain the JUnit dependency and sample tests. However, this archetype may include an older version of JUnit 3.8.1. You need to edit your `POM` file and revise the dependency version to 4.8.2 or some of the newer features of JUnit will remain unavailable.

How to do it...

Test suites and test cases reside in the `<project_base_dir>/src/test/java` folder. Create your first test case for an existing class using your IDE. Popular IDEs such as Eclipse, NetBeans, IntelliJ, and others, which support JUnit and Maven make this process a breeze.

```
package net.srirangan.packt.maven;

import org.junit.Test;
import static org.junit.Assert.assertEquals;
```

```
public class MyClassTest {
  @Test
  public void testMultiply() {
    MyClass tester = new MyClass();
    assertEquals("Result", 50, tester.multiply(10, 5));
  }
}
```

Here we have created a test case for a class named `MyClass` that implements one method called `multiply`. Do note that `MyClass` and `MyClassTest` must reside in the same package.

If you need to merge several test cases, they can be combined into a test suite:

```
package net.srirangan.packt.maven;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses( { MyClassTest.class })
public class AllTests {
}
```

How it works...

Executing the test build phase from the command line executes the tests and all other phases required for the test phase:

```
$ mvn test
```

Alternatively, the test phase is automatically executed in the default build lifecycle. Therefore, executing `install`, for example, will run all the phases including the test phase:

```
$ mvn install
```

Apache Maven outputs the test results on the console. You should see something similar to:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running net.srirangan.packt.maven.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.023 sec

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

The Maven Surefire plugin has a test goal bound to the test phase of the build lifecycle. This executes all tests under the `<project_base_dir>/src/test/java` folder with filenames matching the following patterns:

- `**/Test*.java`
- `**/*Test.java`
- `**/*TestCase.java`

If one or more of the tests fail, the build fails. The output is shown on the console while an XML version can be found at `<project_base_dir>/target/surefire-reports`.

See also

Acceptance testing automation section in *Chapter 2*, *Software Engineering Techniques*

Acceptance testing automation

Selenium is a popular automation testing framework which works with a variety of technologies including Java, C#, Ruby, Groovy, Python, PHP, and Perl.

In order to write automation tests, Selenium provides the Selenium IDE, which is a plugin for Mozilla Firefox that primarily allows you to record and playback tests and export them into various languages including Java.

Selenium Maven Plugin allows you to specify automation tests created for Selenium in your Maven project and integrate it with the Maven build lifecycle.

Getting ready

First we need a web application project to get started. This command ought to do it:

```
$ mvn archetype:generate -DgroupId=net.srirangan.packt.maven
-DartifactId=MySampleWebApp -DarchetypeArtifactId=maven-archetype-webapp
```

Include the relevant dependencies in the project `POM`:

```
<dependency>

  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>
```

How to do it...

Starting the Selenium server requires it to be synced with the pre-integration test phase of your build lifecycle. This can be done by adding the following to the project `POM`:

```
<plugins>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>selenium-maven-plugin</artifactId>
    <executions>
      <execution>
        <phase>pre-integration-test</phase>
        <goals>
          <goal>start-server</goal>
        </goals>
        <configuration>
          <background>true</background>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
```

This will start the Selenium server before running the integration tests.

However, to run the Selenium tests, we will need to start the web application server as well. The Maven Jetty plugin allows us to accomplish the same:

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.10</version>
  <executions>
    <execution>
      <id>start-jetty</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <scanIntervalSeconds>0</scanIntervalSeconds>
        <daemon>true</daemon>
      </configuration>
    </execution>
    <execution>
```

```
        <id>stop-jetty</id>
        <phase>post-integration-test</phase>
        <goals>
          <goal>stop</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

We have the Selenium and Jetty servers, that is, the test and web application servers configured to automatically initiate with the integrated test phase of the Maven build lifecycle.

What's next? Of course, creating our first test! Introduce the JSP file `src/main/webapp/index.jsp` in your project with the following markup:

```
<html>
  <body>
    <h1>Hello, World</h1>
  </body>
</html>
```

With the Jetty application server running, try accessing `http://localhost:8080/mywebapp`. You should be seeing the header text as `Hello, World`. If not, fix your web application project to render the same.

Our Selenium test case will load the contents of `http://localhost:8080/mywebapp` and will try to assert if the result contains a header with the text `Hello, World`

```
import junit.framework.TestCase;
import org.junit.Test;
import com.thoughtworks.selenium.DefaultSelenium;
import com.thoughtworks.selenium.SeleniumException;

public class SeleniumHelloWorldExample extends TestCase {
  private DefaultSelenium selenium;

  @Override
  public void setUp() throws Exception {
    super.setUp();
    selenium = createSeleniumClient("http://localhost:8080/");
    selenium.start();
  }

  @Override
  public void tearDown() throws Exception {
    selenium.stop();
```

```
    super.tearDown();
  }

  protected DefaultSelenium createSeleniumClient(String url) throws
Exception {
    return new DefaultSelenium("localhost", 4444, "*firefox", url);
  }

  @Test
  public void testHelloWorld() throws Exception {
    try {
      selenium.open("http://localhost:8080/mywebapp/index.jsp");
      assertEquals("Hello, World", selenium.getText("//h1"));
    } catch (SeleniumException ex) {
      fail(ex.getMessage());
      throw ex;
    }
  }
}
```

It's all set! Stir up that command line, navigate to the project base directory, and execute the following command:

```
$ mvn integration-test
```

How it works...

If you look at the preceding configuration, the Selenium and Jetty plugins are configured to start in the pre-integration test phase. Thus, when the integration-test build phase is reached, Jetty and Selenium are ready and available.

Now the execution of our test case begins. In the `setup()`, we create a Selenium client. The client is the used in the test case `testHelloWorld()` to load a web page and assert a simple condition, based on the response of that web page, thus simulating a user loading a web page on his / her browser and asserting a condition.

See also

Test driven development section in *Chapter 2*, *Software Engineering Techniques*

Deployment automation

The Maven Deploy Plugin is used to add artifact(s) to a remote repository during the deploy phase of the build lifecycle.

The deploy plugin introduces two goals:

- `deploy:deploy`: To deploy a project and all its artifacts
- `deploy:deploy-file`: To deploy a single artifact file

Getting ready

Deployment to a repository means not only to copy the artifacts to a folder but to update metadata regarding the artifacts as well. It requires:

- **Target repository**: Target repository is where the artifacts will be deployed. Its location, the protocol for access (FTP, SCP, SFTP), and user-specific account information are required.
- **Target artifacts**: Target artifacts are the artifacts which are to be deployed. `Artifact groupId`, `artifactId`, `version`, `packaging`, and `classifier information` are required.
- **Deployer method** to actually perform the deployment: This can be implemented either in a cross platform (wagon transport) or system-specific manner.

How to do it...

The project `POM` must include a valid `<distributionManagement/>` element, which provides a `<repository/>` element defining the remote repository location for the artifact.

```
<distributionManagement>
  <repository>
    <id>srirangan.repository</id>
    <name>MyPrivateRepository</name>
    <url>...</url>
  </repository>
</distributionManagement>
```

For this, you need to specify a server definition in your `settings.xml (<USER_HOME>/.m2/settings.xml or <M2_HOME>/conf/settings.xml)` to provide authentication information for the repositories.

```
<server>
  <id>srirangan.repository</id>
  <username>srirangan</username>
```

```
    <password>myTopSecretPassword</password>
  </server>
```

The deployment can be executed from the command line by navigating to the project folder and executing the following command:

```
$ mvn deploy
```

How it works...

The Apache Maven Deploy plugin is invoked usually during the `deploy` phase of the build life cycle. As we just saw, a `<distributionManagement/>` element with a `<repository/>` element in it are required to enable this plugin.

Snapshots and releases can be separated by defining a `<snapshotRepository/>` element, whereas site deployments require a `<site/>` element.

The `<distributionManagement/>` element is inherited allowing for registry of this information in the parent `POM` file to make them available to all sub-modules.

The actual deployment takes place based on the protocol defined in the repository with the commonly used protocols being FTP and SSH. `wagon-ftp` and `wagon-ssh-external` are providers for these two protocols.

There's more...

If the remote repository is accessible through FTP, then the project POM `build` elements need to include the specification of the `wagon-ftp` extension.

```
<distributionManagement>
  <repository>
    <id>sri-ftp-repository</id>
    <url>ftp://...</url>
  </repository>
</distributionManagement>

<build>
  <extensions>
    <extension>
      <groupId>org.apache.maven.wagon</groupId>
      <artifactId>wagon-ftp</artifactId>
      <version>1.0-beta-6</version>
    </extension>
  </extensions>
</build>
```

For deployment using SSH, note the changes in the extension `artifactId` and the URL.

```
<distributionManagement>
  <repository>
    <id>sri-ssh-repository</id>
    <url>scpexe://....</url>
  </repository>
</distributionManagement>

<build>
  <extensions>
    <extension>
      <groupId>org.apache.maven.wagon</groupId>
      <artifactId>wagon-ssh-external</artifactId>
      <version>1.0-beta-6</version>
    </extension>
  </extensions>
</build>
```

Authentication information can be provided in the `settings.xml`, but encrypted passwords aren't supported. Hence SSH-based deployments allow secure deployments when required.

3
Agile Team Collaboration

In this chapter, we will cover:

- Creating centralized remote repositories
- Performing continuous integration with Hudson
- Integrating source code management
- Team integration with Apache Maven
- Implementing environment integration
- Distributed development
- Working in offline mode

This chapter covers techniques such as distributed development, continuous integration, environment integration, centralized remote repositories implementation, and so on. One thing common to the recipes covered here is that they are context sensitive and cater to situations that arise in medium to large software development teams.

This is a chapter for Agile teams, but can be useful for any team, regardless of the methodologies followed. However, it is recommended that one is familiar with practices that are followed by Agile teams because for many situations these practices are ideal.

A good example of this is the practice of pair programming. While it is an extremely effective practice for any team, in the context of a distributed team, pair programming becomes critical and its success can have a direct impact on the health of the team and the project.

Some of these techniques are well documented and have been popularized by software engineering methodologies such as Extreme Programming. The recipes here offer glimpses at how these can be implemented using Maven in situations, where one is dealing with Agile, distributed and cross functional teams.

Creating centralized remote repositories

In *Chapter 2*, *Software Engineering Techniques*, we explored Dependency Management capabilities of Apache Maven in the recipe *Dependency management*. Apache Maven projects can be dependent on other "artifacts". These artifacts can be other projects or external libraries. Apache Maven stores all packaged artifacts in a local repository. The local repository exists on your filesystem and its location is configured in `${M2_HOME}/conf/settings.xml` or `${USER_HOME}/.m2/settings.xml`.

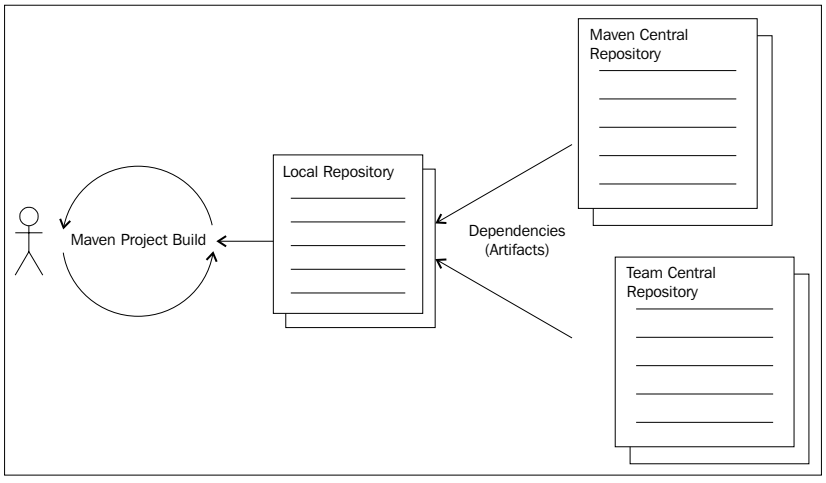
In the instances where we have dependencies that aren't available in the local repository, we see that Apache Maven automatically downloads them from the Maven Central Repository.

It is possible that our projects have unique dependencies that aren't available in the **Apache Maven Central Repository**. Apache Maven does let you manually add libraries into the local repository through the command line. However, to expect every project team member to manually download and install the library in their local repository isn't a pragmatic approach as it adds a lot of manual steps for each programmer in the team and undermines the very purpose of Apache Maven's dependency management features.

Developers want to spend most of their time doing interesting work and being productive rather than getting drawn into project configuration tasks such as ensuring all the prerequisite dependency artifacts are available in their local repository.

A good solution here is to configure Apache Maven to work with a centralized remote repository. All dependencies need to be made available and maintained on this remote repository and everybody's Maven instance automatically installs missing dependencies on their local repositories from this remote source.

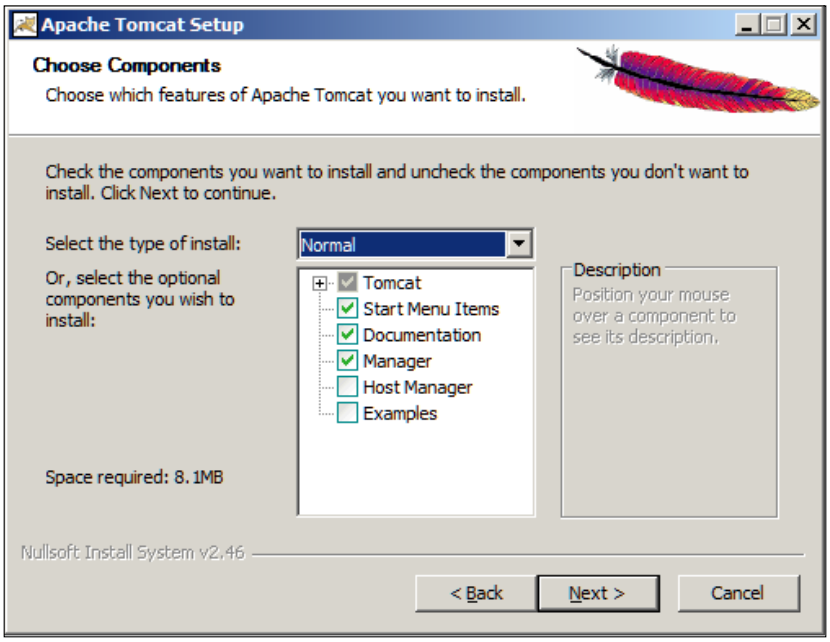
The following diagram illustrates how this works:



Getting ready

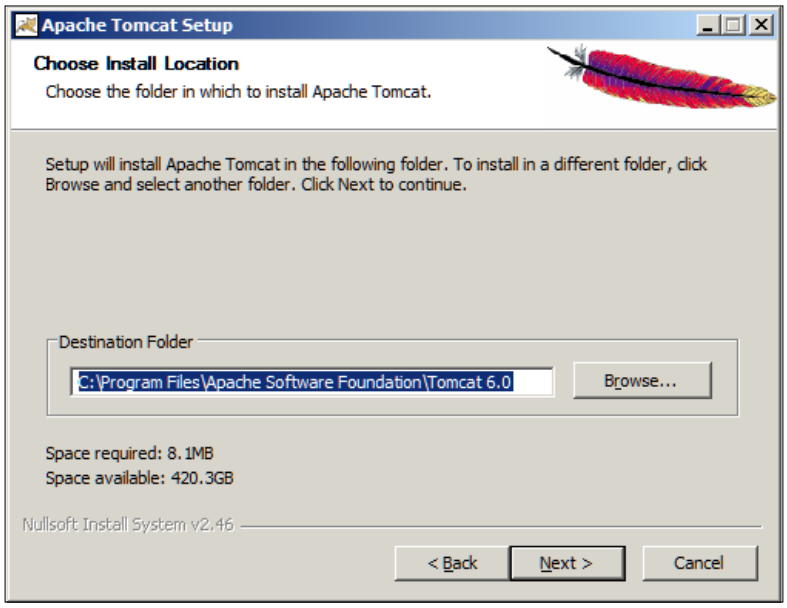
In this recipe, we will be exploring the Sonatype Nexus Repository Manager. While other alternatives such as Apache Archiva, Artifactory, and so on exist, Nexus powered by Sonatype's support and community proves to be an excellent choice for programmers, teams, and enterprises. Nexus, being a web application, needs a web application server to be deployed upon. Here Apache Tomcat 6 is the chosen application server:

- Install it on the Windows 7 operating system by downloading the latest stable version of the service installer at `http://tomcat.apache.org`.
- On Windows, you can install **Tomcat** using the **Installer | Setup** file available for download. Run the installer (`.exe`) and you will get a wizard, as shown in the following screenshots:

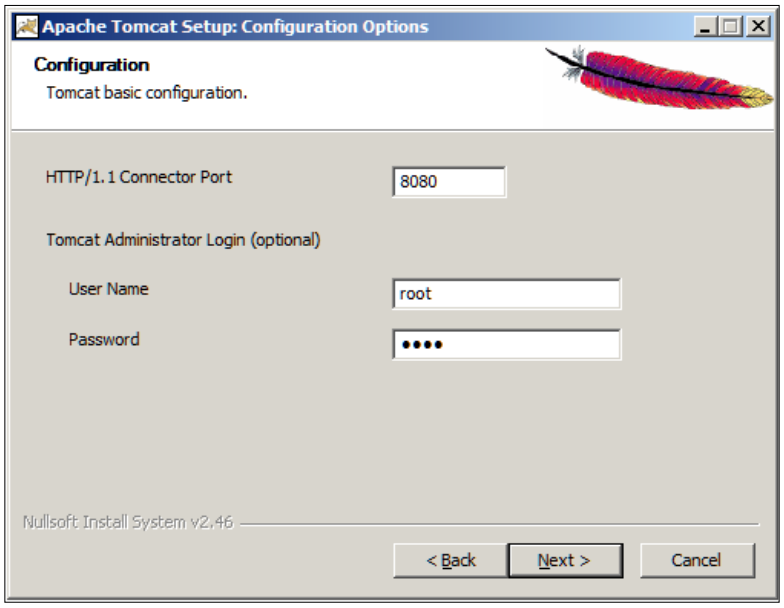


Ensure that you install the **Manager** as displayed in the preceding screenshot. On Ubuntu, you'll need to install an additional package, `tomcat6-manager`. We will be using the manager to deploy the Sonatype Nexus WAR package.

- The next step for the **Tomcat** installation on Windows is the selection of the root directory for **Tomcat**.

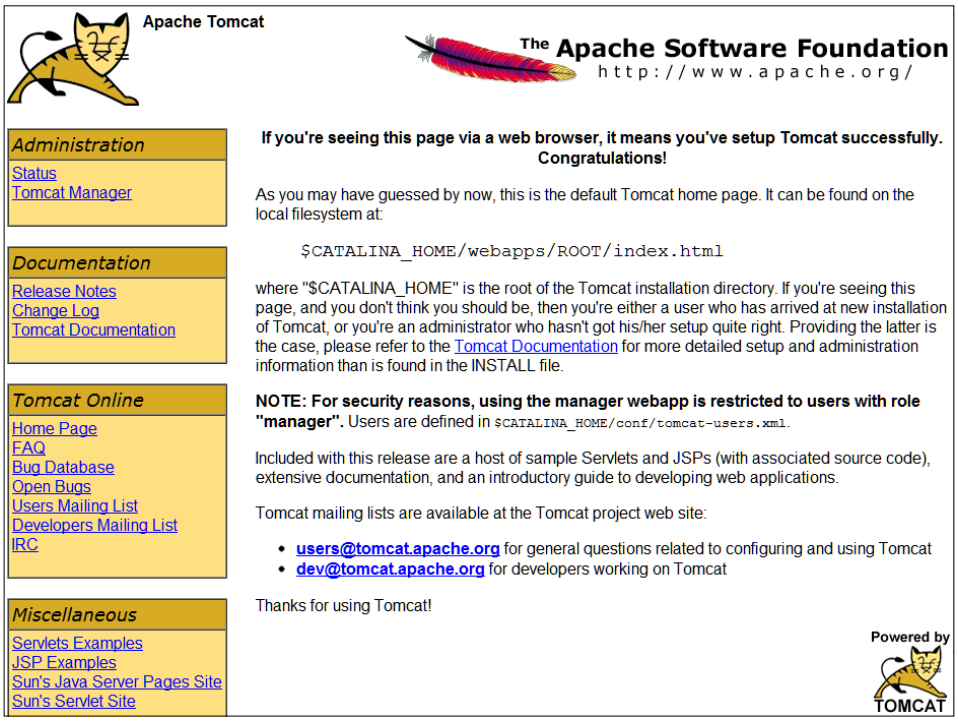


- Enter the administrative **Username** and **Password**. Make sure you remember it, as it will be needed for accessing the Tomcat manager.



- Open your web browser and visit `http://localhost:8080`.

  You should see a confirmation page similar to the one shown in the next screenshot:

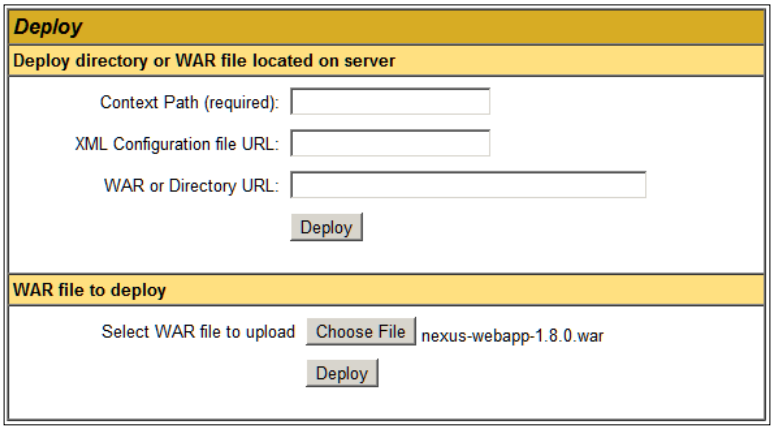


How to do it...

Now that we've got the Tomcat application server ready, the next thing for us is to install Nexus Open Source. This can be done as follows:

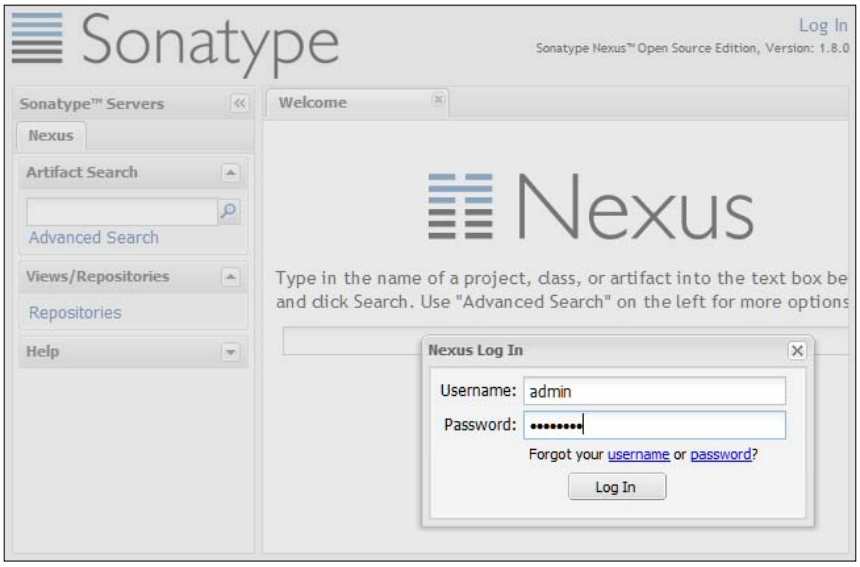
1. Download the latest stable WAR package of Nexus Open Source from `http://nexus.sonatype.org`. Once this is completed, start the Tomcat Manager. On my Windows installation, the URL for Tomcat Manager is `http://localhost:8080/manager/html`.

2. The Tomcat Manager will present a web-based interface to deploy the Nexus Open Source WAR. This deployment needs to be done, as shown in the following screenshot:



3. You will see two forms. We'll select the second form and just upload the Nexus WAR file. Alternatively, you can extract the WAR file on your machine and use the first form by giving specific details for the context-path configuration file and directory where Nexus was extracted.
4. After a successful deployment, Nexus will now be available at

   `http://localhost:8080/nexus-webapp-1.8.0`.

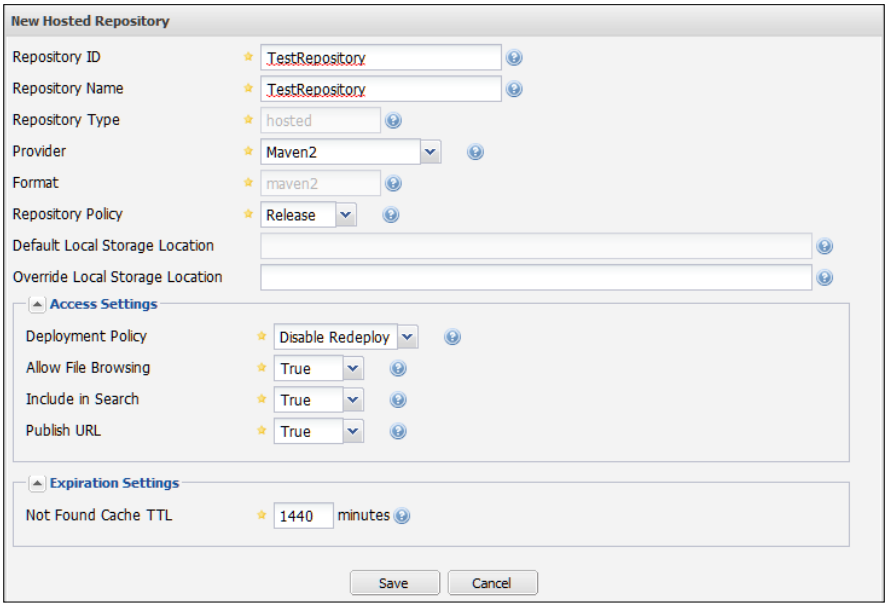


5. The default administrative login for Nexus is:

   Username: **admin**

   Password: **admin123**

6. Once you're logged in, you can create a repository for your project:



7. You will find a repository URL available after creation. Make a note of this URL as it needs to be used in your Apache Maven configuration or POM file.

8. The Maven `settings.xml` file now needs to be modified and a new mirror is to be added:

```
<mirrors>
  <mirror>
    <id>TestRepository</id>
    <name>My test repository</name>
    <url>http://localhost:8080/nexus-webapp-
      1.8.0/content/repositories/TestRepository/</url>
    <mirrorOf>*</mirrorOf>
  </mirror>
</mirrors>
```

How it works...

While setting up the Nexus repository and configuring Apache Maven is complete, one critical aspect still remains. Nexus needs to be installed on the LAN so that everyone in the team can access it.

Ideally, it should not reside on a development machine. A dedicated box, accessible on the LAN will be perfect. Setting up a LAN is obviously beyond the scope of this recipe. However, chances are that you are already on a LAN. In such a case, a preferred repository URL would be something like:

```
http://<permamnet-hostname-on-network>/nexus-webapp-1.8.0/content/
repositories/TestRepository/
```

Out of all the repository managers available, Nexus is recommended because it is developed by Sonatype; which includes many core developers from Apache Maven. Hence, interoperability with various versions of Apache Maven is better supported.

Repository managers are generally recommended, as they prevent the trouble of each developer uniquely downloading the same dependencies over and over again. Only a single download is required, and once downloaded, they are available to everyone in the team or organization. In large organizations with multiple projects, this ends up saving considerable bandwidth, and more importantly, developer time.

Another advantage is that internal projects / artifacts can be deployed onto the team / organizational repository and each developer can access internal dependencies from the repository instead of having to clone and locally build the project.

Performing continuous integration with Hudson

In *Chapter 2*, *Software Engineering Techniques*, we saw how our project builds could be automated in the *Build automation* section. Now that we can build and test our project with a single command, the question of how often this should be done arises.

"Ship It!" by *Jared Richardson and William Gwaltney Jr*. says:

> *"Ideally, you will rebuild every time the code changes. That way you'll know immediately if any change broke your build."*

Sure, programmers always try and do it, but how often have you pulled an update from the SCM and forgotten to build before committing your code? To avoid this occurrence, builds can be triggered each time code is committed into the SCM. Such a setup is known as **continuous integration**. It has become extremely popular and is extremely effective for small and large teams working in an Agile environment.

The key idea here is to launch a build every time a developer commits code. This catches a bad commit in which the developer may have committed erroneous code or forgotten to add any requisite files.

Getting ready

To implement continuous integration in your team, you will need to use a continuous integration server. There are a number of options here such as Cruise Control, Apache Continuum, Hudson, and so on. In this recipe, we will take a look at Hudson (`http://hudson-ci.org/`), which is a continuous integration tool that integrates well with Maven projects.

Hudson, being a web application, needs a web application server to be deployed upon. Here Apache Tomcat 6 is the chosen application server.

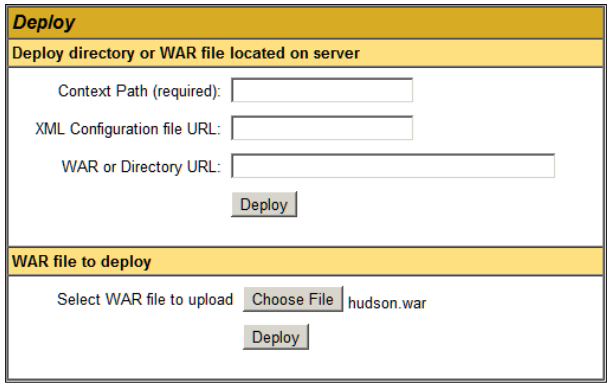
Downloading and installing Apache Tomcat 6 was covered in the *Getting ready* section of the preceding recipe. You would have to follow the same steps for this recipe before proceeding any further.

How to do it...

1. Now that we've got the Tomcat application server ready, the next thing for us is to download Hudson from `http://hudson-ci.org`.

   Download the latest stable WAR package of Hudson. Once this is completed, start the Tomcat Manager. On Windows, the URL for Tomcat Manager is `http://localhost:8080/manager/html`.

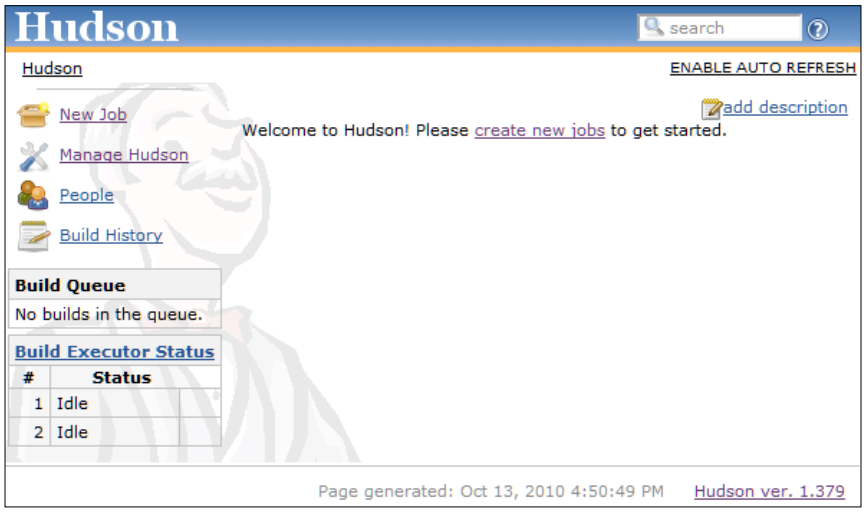
2. The Tomcat Manager will present a web-based interface to deploy the Hudson WAR. This deployment needs to be done, as shown in the following screenshot:



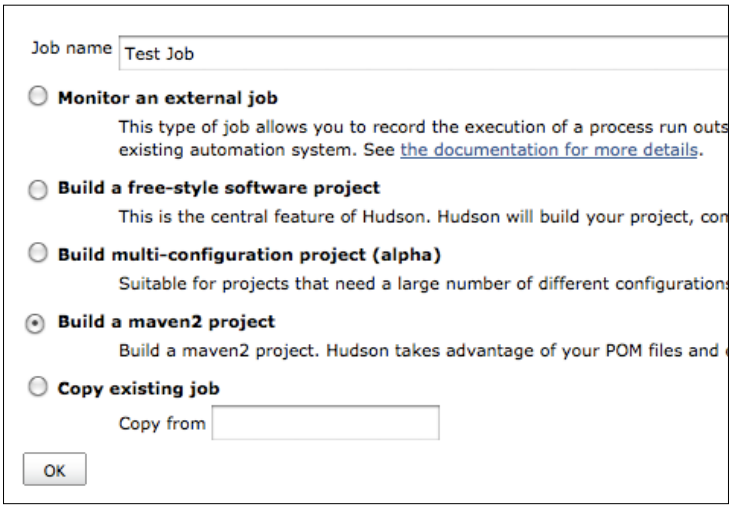
   Note that there are two forms available on the deployment page. We will directly upload the WAR file using the second form.

3. After a successful deployment, Hudson will now be available at:

   `http://localhost:8080/hudson`.



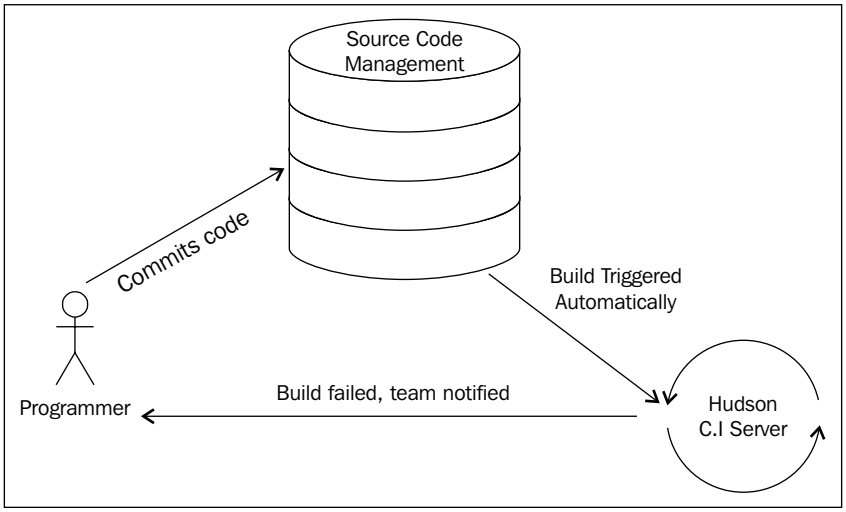
- **Hudson** needs to be configured. This can be done by selecting **Manage Hudson** | **Configure System**. Make sure you configure JDK, Maven, and SVN/CVS properties.
- Next we create a Hudson job for our project. Hudson provides a job type dedicated for Maven; the job type "**maven2**" can be used for Maven 2 and Maven 3 projects.



- Create the job by entering the project and Source Code Management properties. Build triggers can be specified and so one can build settings including e-mail notifications, if required.

How it works...

Once set up correctly, every time code is committed to SVN / CVS, the Hudson job gets triggered. The job then in-turn proceeds to trigger a Maven build. Hudson also continues to monitor the console output of the Maven build. Hudson looks for patterns in the Maven output and confirms if the build and tests are completed successfully or not.



Depending on Hudson and job configuration, Hudson may e-mail the project team (or mailing list) if any build or test failures occur.

Integrating source code management

If your team is not using **Revision Control System** (also known as **Version Control**), here is some advice: stop everything and implement it right now. This advice also applies to individual programmers or pairs who have used SCMs in the past but don't find it relevant for their current setup.

Working with an SCM lets you keep track of changes, associate important versions of the source with milestones, create branches for parallel development, implement scheduled backups, and configure best-practice techniques such as continuous integration and so on.

Getting ready

There are many revision control systems for you to choose from. Even enterprise quality version control systems today are free and open source. Therefore, while choosing a version control system, the only problem there is a problem of plenty. Today, version control systems come in two flavors–centralized and distributed.

Centralized version control systems have a repository based on centralized remote servers and developers download one specific revision / version of the source code (usually the latest version, that is, HEAD revision) and work with it. Changes are made on the local workstation and can be "committed" onto the centralized version control system. Popular centralized SCMs include CVS, Subversion (SVN), MS Visual Source Safe, and so on.

Distributed version control systems are, however, fundamentally different in the sense that at any given moment, the developer workstation has all the revisions of the source code and is independently self-sufficient for all development work. Developers choose to either make their source repository servers available to other users directly, or share it via a centralized source repository. Popular distributed SCMs are Mercurial and GIT.

How to do it...

The first step is, of course, to choose a version control system. The primary open source candidates include Subversion (SVN), Mercurial, and GIT. Your choice would depend on various parameters including the standards of your organization, your team's comfort level with new adoptions, and so on.

No matter which SCM suite you choose, be it centralized or distributed, some concepts remain similar. For instance, the Subversion `svn checkout` command will download the HEAD revision of the source repository on the developer machine while Mercurial and GIT commands, which are `hg clone` and `git clone`, will create an instance of the entire repository on the developer box.

The following table introduces basic commands for the three popular open source revision / version control systems, that is, Subversion (SVN), Mercurial (HG), and Git.

These commands are not identical to each other and each revision control system has its own individual learning curve.

Subversion commands	*Mercurial commands*	*Git commands*
svn checkout	*Hg init*	*git init*
svn add	*hg clone*	*git clone*
svn delete	*hg add*	*git add*
svn commit	*hg commit*	*git commit*
svn update	*hg status*	*git status*
svn diff	*hg pull*	*git diff*
svn log	*hg push*	*git reset*
svn merge	*hg serve*	*git merge*
and so on.	*and so on.*	*and so on.*

The source code configuration (SCM) information can be added to the POM file of an Apache Maven project.

```
  <scm>
    <connection>...</connection>
    <developerConnection>...</developerConnection>
    <tag>...</tag>
    <url>...</url>
  </scm>
...
```

The preceding code snippet shows details of a revision control repository added in a project POM file. The `connection` and `developerConnection` elements represent the read-only and read/write paths of the repository respectively. The `tag` element dictates which tag is being worked upon, while the `url` field is an optional field for a publicly browseable view of the source code.

How it works...

The workflow for the three revision control systems–SVN, Mercurial, and Git–can be different depending on the configuration. However, a likely configuration for smaller teams (2-12 people) would involve a centralized repository even while you are leveraging benefits of Distributed version control systems such as Git or Mercurial.

In such a case, the first step for a new developer would be to extract the source code on his / her workstation. With SVN, this would be the `svn Checkout` command, while for Git or Mercurial you would use the `clone` command. In SVN, it is likely that you would be getting only a copy of the HEAD revision while the `clone` command for Git / Mercurial extracts the entire repository.

The next step that would come in after the programmer has made changes is to commit the code back into the repository. This is where the advantages of Git and Mercurial really shine through. If you are using either one of those, you can commit your code into your local repository and continue to work. However, if you're stuck with SVN, at this point, before committing your code to the central repository, you need to update your local revision and merge changes and resolve conflicts right away.

It is a best practice for programmers to commit small bunches of changes as often as possible. This is where decentralized revision control systems really help because you can commit small changes into your local repository and continue working. But if you're using SVN, you need to update your revision, merge changes, and resolve conflicts. Doing this very often can lead to significant loss of productivity; hence a lot of developers using SVN or similar centralized version control systems abandon the practice of committing code very often in small bunches and commit code mostly once a day or so.

However, in a DVCS, you would commit code often and push your changes to the central repository, perhaps once a day if you are working in a central repository configuration.

There's more...

Latest versions of popular IDEs such as Eclipse, NetBeans, IntelliJ IDEA, and so on have plugins for these various revision control systems and embed their user with the development process itself. In fact, Eclipse's Team suite helps abstract different revision control commands and standardizes the experience, regardless of which underlying system being used.

A popular option these days is also to make use of online cloud-based source code repositories and revision control providers. They generally provide one or more SCM systems and have various pricing options including free options. Some of these include:

- Google Code: Mercurial, SVN–`http://code.google.com`
- BitBucket: Mercurial–`http://www.bitbucket.org`
- GitHub: Git–`https://github.com`
- SourceForge: CVS, SVN, Bazaar, Git, Mercurial–`http://www.sourceforge.net`

Team integration with Apache Maven

Most enterprise projects are generally executed by a team of programmers in an organization. Teams may consist of developers and contributors. Teams can be onsite, offshore, or distributed, which is a mix of onsite and offshore team members. Alternatively, there might be multiple teams working on different modules of the same project.

Team information is integral toward achieving completeness in an Apache Maven project, and this information can be defined inside an Apache Maven project POM file.

Getting ready

To set up team information in a POM, make sure you have a Maven project and have write permissions to this file. The team information setup consists of:

- Licenses
- Organization
- Developers
- Contributors

Make sure you have all this team meta information available before we proceed.

How to do it...

License information is entered in the POM file in the following structure:

```
<licenses>
  <license>
    <name>...</name>
    <url>...</url>
    <distribution>...</distribution>
    <inceptionYear>...</inceptionYear>
    <comments>...</comments>
  </license>
</licenses>
```

The licensing information can include the name, description, URL, and inception year along with additional comments.

Adding license information inside the POM is important because it is likely that the license purchased by a business would be a team license and individuals in the team would need this information to get started. While teams can share this information on wikis and mailing lists, it makes sense that the project POM file contains this information. After all, the underlying idea of the POM file is that it contains all requisite information for that particular project and all you should ever need is the source code, POM file, and an Apache Maven instance to get started.

Organization information can be set up in the following format:

```
<organization>
  <name>...</name>
  <url>...</url>
</organization>
```

This includes a very basic set of information consisting of just the name and URL of the organization. This can be a private company or an open source group. For teams working with (**ISV) Independent Software Vendors** and implementing client projects, generally the organization information would be the client information because typically, it is the clients who would own the Intellectual Property rights. However, this may vary from case-to-case and will need to be confirmed by a business representative.

Projects generally have multiple developers and information for all the developers can be included in the POM file with the following format:

```
<developers>
  ...
  <developer>
    <id>...</id>
    <name>...</name>
    <email>...</email>
    <url>...</url>
    <organization>...</organization>
    <organizationUrl>...</organizationUrl>
    <roles>
      <role>...</role>
      <role>...</role>
    </roles>
    <timezone>...</timezone>
    <properties>
      <picUrl>...</picUrl>
    </properties>
  </developer>
  ...
</developers>
```

This developer information includes:

- `id`, `name`, and `email`: These correspond to the developer's ID, the developer's name, and e-mail address.
- `organization` and `url`: These are the developer's organization name and it's URL.
- `roles`: They specify the standard actions that the person is responsible for. For example, developer, architect, quality assurance, and so on.
- `timezone`: A numerical offset in hours from GMT to where the developer lives.
- `properties`: This element is where any other properties about the person are mentioned. For example, links to a personal image or an instant messenger handle.

Additionally, for developers, projects also have contributors who generally play supportive roles such as making a bug-fix, working on documentation, and so on. Their information can be set up in the POM file as well:

```
<contributors>
  ...
  <contributor>
    <name>...</name>
    <email>...</email>
    <url>...</url>
    <organization>...</organization>
    <organizationUrl>...</organizationUrl>
    <roles>
      <role>...</role>
    </roles>
    <timezone>...</timezone>
    <properties>
      <gtalk>...</gtalk>
    </properties>
  </contributor>
  ...
</contributors>
```

For small teams or teams located in the same geographical area, adding this team information to the project POM file may seem redundant. However, for instances such as multiple distributed teams working on the same project, or even large open source projects with developers and contributors scattered around the world, having this information stored in the project POM file really helps.

Of course, this information could also be present in wikis, and teams would leverage mailing lists and instant messaging to communicate. Having team information stored in the project POM doesn't seek to replace either of these, but rather just focuses on POM completeness.

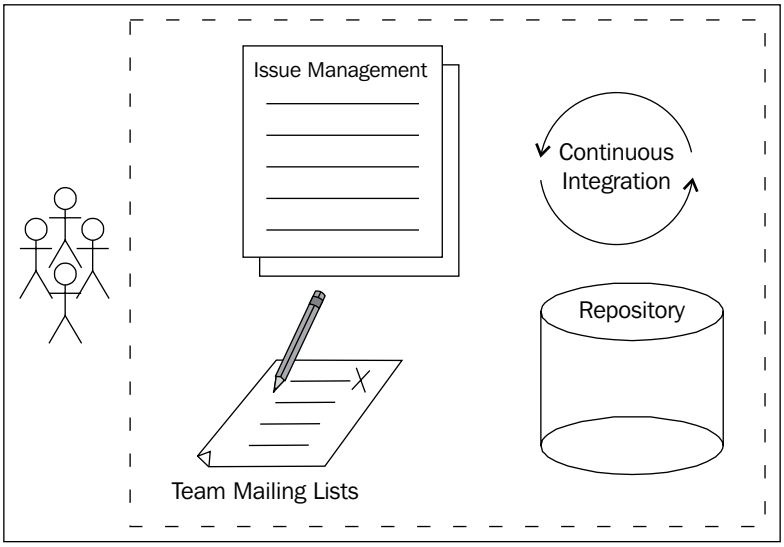
How it works...

This team information, while not critical to the project, is often used by various plugins. It serves to ensure the completeness of the POM file and ensures the POM includes all information regarding the project and not just technical information.

Several IDE (NetBeans, Eclipse, and IntelliJ) plugins or other software used in the development process could also make use of this information. For example, the licenses stored in the project POM may be required to run proprietary development software such as IntelliJIDEA, Adobe Flash Builder (for Flex projects), and so on.

Implementing environment integration

As you would have discovered by now, your project's source code does not live in isolation. Often you'll find your team (and yourself!) working with an issue tracker system, continuous integration system, team mailing lists, source code revision tools, and so on.



This is the environment in which the project team resides and having this information in the POM file is critical to achieve completeness of the project POM file.

Getting ready

To set up environment information in a POM, make sure you have a Maven project and have "write" permission to this file. The environment information includes:

- Issue management
- Continuous integration
- Mailing lists
- SCM

Having these details is a prerequisite before we can proceed.

How to do it...

Information for issue management tools (including JIRA, Bugzilla, and so on) can be set up in the POM file in the following format:

```
<issueManagement>
  <system>...</system>
  <url>...</url>
</issueManagement>
```

This information includes the name of the system and the URL by which it can be accessed.

We looked at the practice of continuous integration earlier in this chapter in the recipe *Performing continuous integration with Hudson*. The CI server's metadata can be entered in the POM file in this format:

```
<ciManagement>
  <system>hudson</system>
  <url>http://localhost:8080/hudson</url>
  <notifiers>
    <notifier>

      <sendOnError>true</sendOnError>
      <sendOnFailure>true </sendOnFailure>
      <sendOnSuccess>true </sendOnSuccess>
      <sendOnWarning>true </sendOnWarning>

    </notifier>
  </notifiers>
</ciManagement>
```

While much of the CI configuration is specific to the CI tool itself, depending on which one you use, (Hudson, Continuum, Cruise Control, and so on), Maven allows the mention of some recurring settings that are common across various CI tools. These include Boolean values of the e-mailing triggers–`sendOnError`, `sendOnFailure`, `sendOnSuccess`, and `sendOnWarning`.

Does your team have more than two members? You need a mailing list to provide an easy way of communicating with everyone involved in the project, including the product owner and other stakeholders if necessary.

Mailing lists have been around literally for decades now and there are many applications to work with here. Google Groups (`http://groups.google.com`) provides an easy way to quickly set up a mailing list. Google Groups is especially recommended if your project is hosted on Google Code or your organization uses the Google Apps infrastructure.

Here's the format for entering this mailing list information:

```
<mailingLists>
  <mailingList>
    <name>...</name>
    <subscribe>...</subscribe>
    <unsubscribe>...</unsubscribe>
    <post>...</post>
    <archive>...</archive>
    <otherArchives>
      <otherArchive>...</otherArchive>
    </otherArchives>
  </mailingList>
</mailingLists>
```

Finally, to ensure the completeness of the POM, the only remaining project infrastructure entity information that is missing is the SCM. This can be entered as follows:

```
<scm>
  <connection>...</connection>
  <developerConnection>...</developerConnection>
  <tag>...</tag>
  <url>...</url>
</scm>
```

The `connection` element should point to read-only access while `developerConnection` points to read/write access to the repository. The connection format that needs to be used is:

```
scm:<scm_provider><delimiter><provider_specific_part>
```

How it works...

Once this information is available in the Project POM file (`pom.xml`), it can be used in various ways such as being used by new team members getting up to speed with the development ecosystem, by advanced IDEs such as IntelliJ IDEA to provide smoother integration with the bug tracking tools, source code repositories tools, and so on.

This helps in achieving "completeness" of the Apache Maven Project POM in which the underlying philosophy calls for all information concerning the project to be contained in the Project Object Model (`pom.xml` file).

See also

- *Chapter 3*, *Performing continuous integration with Hudson* section.
- *Chapter 3*, *Integrating source code management* section.

Distributed development

Like it or not, it is a globalized world today and has been so for the software industry for at least two or three decades. Software creative professionals along with professionals of many other industries find themselves working with distributed cross functional teams. Rather than going into our shells, this situation can be a great opportunity to work with people from differing cultural, geographical, and professional backgrounds.

And while diversity is a gift and must be celebrated, it does bring along new challenges that cannot be swept under the carpet and need to be addressed. One does need to enforce certain practices that transcend cultural differences and make distributed software development practical and productive.

You are supposed to work with people half way across the world. They don't speak your language, they don't share your work culture, and neither do they share your time zone. To put it mildly, this can be quite a challenge.

Yet if done correctly, it has proven itself to be of great benefit to the business. The key here is, if done "correctly".

So how does one do it correctly? I suggest keep it simple.

First, speak a common language. Without this, one can't proceed.

Second, have honest communication with positive intent. Human beings, especially those who speak differing languages, have this great ability to judge the moods of each other. It is not a secret that over 90 percent of human communication in non-verbal.

If you are negative, that will be the first thing your distributed peers will pick up. This needs to be stressed, as it is extremely important. Off shoring / outsourcing can be a controversial subject, but if you are going to work with a distributed team, your intent should be to do your best, regardless of your political stance on this subject.

How to do it...

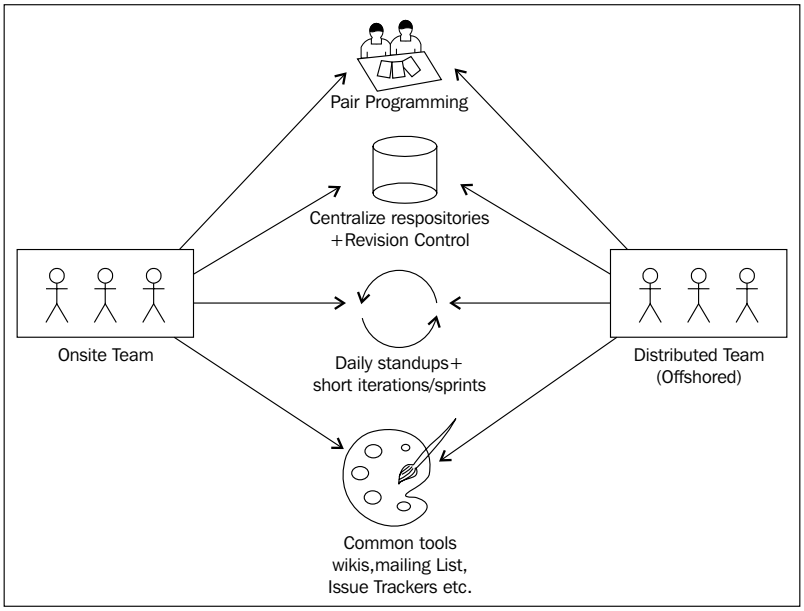
While the practices described in other recipes of this chapter are suitable for any team, they especially resonate and make most sense if implemented for distributed teams.

1. Start with setting up:
    - ❑ Remote source code repository / revision control system
    - ❑ Continuous integration server
    - ❑ Issue tracking systems
    - ❑ Mailing list systems

2. Configure them correctly and make sure everything has been tried and tested by you or someone suitable in your local team. Note that all these systems need to work off the Internet or on private company networks access, whichever is available to your distributed teammates.
3. Once this is ready, create the requisite accounts and access controls for your distributed partners. Share this information with them either on the wiki, on the mailing list, or even through the Apache Maven Project POM file / Maven Project Site.

The next steps depend on the experience levels of the local and distributed team. Do they understand how everything should work? If not, ask them to buy my book. :-)

4. A good way to get started with less experienced team members, whether distributed or local, is through pair programming. In a team of developers with mixed experience levels, it is strongly recommended that developers start pairing up with more experienced team members. The quickest learning happens by example and experienced team members get an opportunity to motivate and mentor a beginner by showing them how it's done.
5. A developer pair can set up the project on the developer box, make their first code change, their first build, and first commit. This will be a tremendous learning experience for the young developer. Rotate pairs often and soon you should have achieved cohesiveness in your distributed team. Screen and voice sharing tools such as Skype can be effectively employed for pair programming. Pair programming has clear benefits and is more effective than day long trainings and sessions.



How it works...

Eventually, everyone in your distributed team should be comfortable executing the following:

- Updating the local code base from the revision control system
- Picking up a task from the issue / task management system
- Writing the tests for the tasks that they are implementing
- Writing the code to implement the tasks
- Running a local build, and checking that code compiles, and all existing and new tests execute successfully
- Updating and merging changes in case multiple commits have been made while the programmer was coding
- Committing the code

Once your team is here, you know you have done a good job developing not only software, but also shaping individuals and building teams. This, for the senior developers, can be an extremely rewarding experience.

See also

- *Chapter 3*, *Creating centralized remote repositories* section
- *Chapter 3*, *Performing continuous integration with Hudson* section
- *Chapter 3*, *Integrating Source Code Management* section

Working in offline mode

Stuck on a tropical island with no Internet access? Will you give in to that intrinsic urge to write more code?

Maven, always a good friend of the developer, lets you work away from centralized SCMs, central repositories, and issue tracking systems. It's called the **Maven offline mode**.

Getting ready

You will need an existing or new project to work on. If your project has dependencies that are not available on the local repository, make sure you have the packages JAR/WAR/EAR available.

How to do it...

It's simple. Just use an `-o` switch for any Maven command. For example:

```
mvn install -o
mvn test -o
```

Does the build fail because of the unavailability of a dependency?

If you have the packaged file (JAR/WAR/EAR), dependencies can be manually installed in your local repository using the following command:

```
mvn install:install-file -DgroupId=%GROUP_ID% -DartifactId=%ARTIFACT_
ID% -Dversion=%VERSION% -Dfile=%COMPONENT%.jar -Dpackaging=jar
-DgeneratePom=true
```

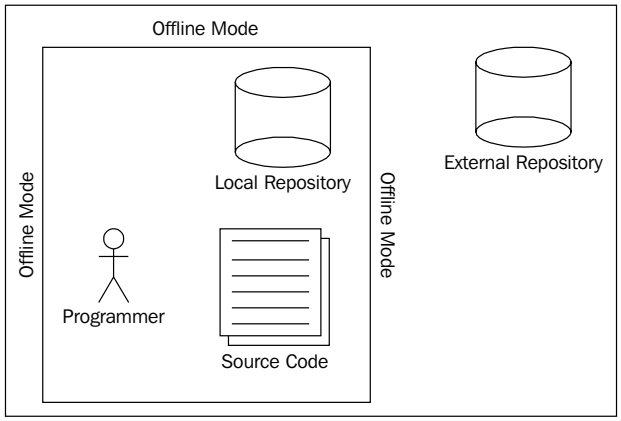
If you are getting tired of manually mentioning offline mode, "`-o`" in every command, you could set it permanently in the Maven settings file `settings.xml`. The settings file can be found in the `<M2_HOME>/conf/` folder or the `<USER_HOME>/m2/` directories.

You need to set the offline configuration element as follows:

```
<settings …>
..
<offline>true</offline>
..
</settings>
```

How it works...

When an Apache Maven command is executed in offline mode, then Apache Maven restricts all access to external / remote repositories: Apache Maven Offline Mode expects all relevant dependency artifacts to be available in the local repository.



Apache Maven Offline Mode can be leveraged to speed up or execute commands while there isn't any network access available, as it allows all commands and features possible as long as there isn't any need to retrieve a dependency / artifact from a remote repository.

When used in conjunction with a distributed version control system (such as Git or Mercurial), the programmer can achieve a higher level of productivity without any network access.

4

Reporting and Documentation

In this chapter, we will cover:

- Documenting with a Maven site
- Generating Javadocs with Maven
- Generating unit test reports
- Generating code coverage reports
- Generating code quality reports
- Setting up the Maven dashboard

As seen in *Chapter 3*, *Agile Collaboration Techniques*, large projects are often integrated with a support environment that includes issue trackers, and continuous integration systems which complement the project's online documentation with real-time data reflecting the current status.

For open source and enterprise distributed projects, the site and documentation are arguably as important as the strategy and code. Such projects revolve around the team/community and proper reporting and documentation is a catalyst for the nurturing and growth of the community and maintaining high productivity of the development team. For enterprise projects, thorough documentation not only brings tangible value to the team, but adds to the overall worth of the project.

In this chapter, we will see how Apache Maven and its plugins can be used to automate most of our reporting, documenting, and analyzing requirements. The reports generated by Maven plugins can broadly be categorized as follows:

- Technical documentation: Javadocs
- Test coverage reports: Cobertura, Clover, and so on

- Code quality reports: Checkstyle, PMD, CPD, FindBugs, and so on
- Unit testing reports: Surefire

We will begin with a study of the Maven site build phase and then take it from there to examine specific plugins for each of the reporting categories just mentioned.

Documenting with a Maven site

A Maven project site is the foundation for the end users and the developers alike. End users look to the site for user guides, API docs, and mailing list archives; and developers look to the site for design documents, reports, issue tracking, and release road-maps.

A Maven project site can consist of everything from unit test failures to code quality reports. It makes them available in a simple HTML or "website" format. HTML pages are rendered using a consistent project template.

Additionally it can be presented as a PDF as well.

Maven project sites often contain the project Javadocs and binary releases. They can be published to a remote server for distributed access.

Getting ready

A Maven site exists for a given Maven project. Our first step is, therefore, to create a new Maven project if we don't have one for the site. Executing the following command in the terminal will create a sample Maven project.

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

How to do it...

Now that we have the Maven project ready, let's start working with the Maven site:

1. Working with Maven sites, true to Maven's commitment to simplicity, has been reduced to just a few terminal commands:

   ```
   $ cd sample-project
   $ mvn site:run
   ```

   We have generated a basic Maven site. Now, our next logical stop is to customize and configure it to our specific needs. This includes the specification of project meta information, creating a requisite menu, setting up remote deployment, configuring authentication information, and so on.

2. The next step is to configure the site descriptor to customize the Maven project site. Here's an example of the site descriptor:

```
<project name="Sample Project">
  <bannerLeft>
    <name>...</name>
    <src>...</src>
    <href>...</href>
  </bannerLeft>
  <body>
    <menu name="...">
      <item name="..." href="..."/>
    </menu>
    <menu ref="reports"/>
  </body>
</project>
```

   A quick look and it's easy to figure out that the site descriptor defines the upper-left banner and navigation menus for the site.

   The site descriptor goes in the folder `<project-root>/src/site` and is conventionally named `site.xml`.

   Your project's site structure should look like this:

```
+- src/
   +- site/
      +- apt/
      |  +- index.apt
      |
      +- xdoc/
      |  +- other.xml
      |
      +- fml/
      |  +- general.fml
      |  +- faq.fml
      |
      +- site.xml
```

3. Once you have a project site that is ready to be served to the intended audience (developers and user community), the site plugin's `deploy` goal can be used to deploy a number of protocols including FTP, SFTP, DAV, and SCP.

   Here's an example of configuring the remote server:

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

```
      <id>sample-project.website</id>
      <url>dav:https://dav.sample.com/sites/sample-project</url>
    </site>
  </distributionManagement>
  ...
</project>
```

4. Remote server authentication can be defined in the Maven `settings.xml` file:

```
<settings>
  ...
  <servers>
    <server>
      <id>sample-project.website</id>
      <username>username</username>
      <password>password</password>
    </server>
    ...
  </servers>
  ...
</settings>
```

5. If the remote server is a Unix/Linux service, the file and directory permissions and modes need to be set correctly. This can also be done by entries in the Maven `settings.xml` file:

```
<settings>
  ...
  <servers>
    ...
    <server>
      <id>hello-world.website</id>
      ...
      <directoryPermissions>0775</directoryPermissions>
      <filePermissions>0664</filePermissions>
    </server>
  </servers>
  ...
</settings>
```

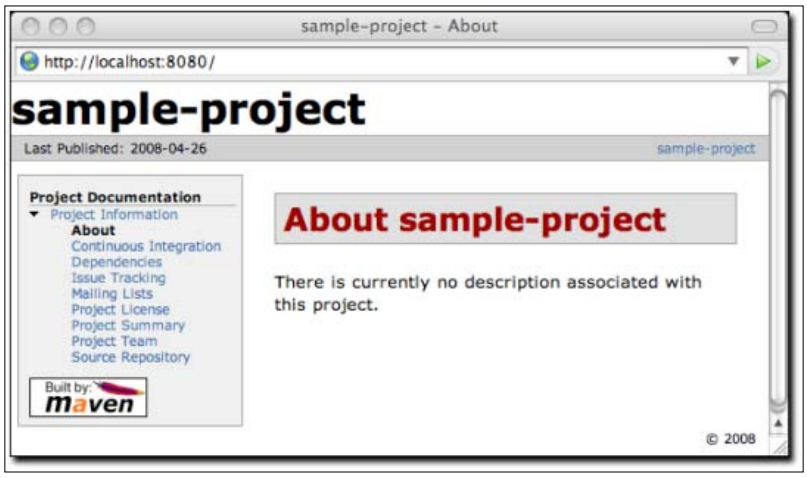
How it works...

To build the site for your project and start an embedded instance of the Jetty web server, use the following command:

```
$ mvn site:run
```

The site can be viewed by pointing your web browser to the URL `http://localhost:8080`.

The following screenshot shows the generated site:



The `$ mvn site:deploy` invokes the plugin goals that generates the JARs and then uses the SCP or FTP to remotely deploy the site. The configuration used for remote deployment was covered in steps 3, 4, and 5 in the *How to do it* section of this recipe.

Depending on the configuration, the method of deployment is automatically chosen and the appropriate command is executed behind the scenes.

For more information about the Maven site, check out `http://maven.apache.org/guides/mini/guide-site.html`

Generating Javadocs with Maven

Javadoc is a documentation generator introduced by Sun Microsystems (now part of Oracle) for generating API documentation in HTML format directly from the Java source code.

Over the years, Javadoc's format has become the de facto industry standard for documenting Java classes, methods, interfaces, and so on. It has proven to be very popular, and has transcended Java, and been readily adopted in other technologies including Microsoft .NET, PHP, ActionScript, Python, and a number of JVM-based programming languages.

The Maven Javadoc plugin uses the Javadoc tool and generates API documentation directly from the source code in HTML format.

Structure of a Javadoc comment:

```
/**
 * @author      Firstname Lastname <address @ example.com>
 * @version     2010.0331
 * @since       1.6
 */
public class Test {
  // class body
}
```

Getting ready

To get started, let us have a quick look at the various goals made available to us by the Maven Javadoc plugin. These goals are important because we later make use of them in order to generate, archive, or fix the project documentation.

This plugin comes with ten major goals. They are as follows:

- `javadoc:javadoc`: Generates documents for the project
- `javadoc:test-javadoc`: Generates documents for the test classes
- `javadoc:aggregate`: Generates documents for an aggregator project
- `javadoc:test-aggregate`: Generates documents for tests of an aggregator project
- `javadoc:jar`: Creates an archive of the documents
- `javadoc:test-jar`: Creates an archive of the tests' docs
- `javadoc:aggregate-jar`: Archives documents of an aggregated project
- `javadoc:test-aggregate-jar`: Archives test documents of an aggregated project
- `javadoc:fix`: Fixes documents and tags for Java files
- `javadoc:test-fix`: Fixes documents and tags for Java test files

How to do it...

Javadoc can be generated as part of the "site" phase of the build lifecycle. This is perhaps the most popular configuration for Javadoc generation. To implement this, the Javadoc plugin needs to be part of the `reporting` element in the project POM file.

```
<project>
  ...
  <reporting>
    <plugins>
```

```
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-javadoc-plugin</artifactId>
          <version>2.7</version>
          <configuration>
            ...
          </configuration>
        </plugin>
      </plugins>
      ...
    </reporting>
    ...
  </project>
```

The command would be the same as the Maven site build phase command and the Javadocs would be generated as part of the Maven site.

```
$ mvn site
```

While integrating the Javadoc generation with the Maven build lifecycle is the most beneficial, in case a situation arises wherein you need to use the Javadoc plugin in a standalone mode, the following configuration can be used.

Add the plugin to the build element of the project POM file:

```
  <project>
    ...
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-javadoc-plugin</artifactId>
          <version>2.7</version>
          <configuration>
            ...
          </configuration>
        </plugin>
      </plugins>
      ...
    </build>
    ...
  </project>
```

The following commands can be directly executed in the console/command-line terminal and the goal specific action will take place.

```
$ mvn javadoc:javadoc
$ mvn javadoc:jar
$ mvn javadoc:aggregate
$ mvn javadoc:aggregate-jar
$ mvn javadoc:test-javadoc
$ mvn javadoc:test-jar
$ mvn javadoc:test-aggregate
$ mvn javadoc:test-aggregate-jar
```

The aggregate goal in the Javadoc plugin can be used in a Multi-modular project environment where there is a need to aggregate all the Javadocs of all the modules into a single Javadoc report.

The following project structure is a good candidate for using `javadoc:aggregate`:

```
Project
  |-- pom.xml
  |-- Module1
  |   `-- pom.xml
  |-- Module2
  |   `-- pom.xml
  `-- Module3
    `-- pom.xml
```

How it works...

When the Maven Javadoc plugin is configured to be part of the `reporting` element in the project `pom.xml` file, then it is executed as part of the site phase of the Maven project build lifecycle.

Hence the command to generate it is:

```
$ mvn site.
```

If the Maven Javadoc plugin has been installed in the standalone mode, which means it has been explicitly defined in the `build` element of the project `pom.xml` file under the plugins sub-element, then it is not automatically executed as part of the project build lifecycle and its goals need to be explicitly executed, as shown in the *How to do it* section of this recipe.

Under this configuration, it behaves like any other Maven plugin that is not automatically associated with the project build. This is a rare configuration, but maybe suitable for testing/evaluation purposes.

Generating unit test reports

The Maven Surefire plugin is executed in the "test" phase of the build lifecycle to run the unit tests, generate plain text, and XML reports of the tests.

The Surefire plugin comes with one goal: `surefire:test`.

By default, Surefire reports can be found in `{basedir}/target/surefire-reports`.

Getting ready

To get started, you need to include the dependency of the Maven Surefile plugin in your project `POM.xml` file.

```
<project>
  ...
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.6</version>
        </plugin>
        ...
      </plugins>
    </pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>

        </plugin>
      ...
      </plugins>
  </build>
  ...
</project>
```

You may also need to change the default configuration to suit your preferences. Configuration elements can also be included as depicted:

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

```
      <configuration>
          <reportFormat>brief</reportFormat>
          <useFile>false</useFile>
      </configuration>
  </plugin>
```

Surefire will work with unit tests created for:

- TestNG
- JUnit (3.8 or 4.x)
- Plain Java (POJO) tests

It will also work when a combination of the test frameworks just mentioned are being used in the same project/module.

How to do it...

The Surefire plugin is invoked in the "test" phase of the Maven build lifecycle. Thus the following command invokes Surefire:

```
$ mvn test
```

To see the Surefire output on a console, try:

```
$ mvn test -Dsurefire.useFile=false
```

Sometimes there may be a need to skip all tests, skip individual tests, execute individual tests, or modify the test reports' output destination or format. For these aspects, Maven is remarkably easy to customize and can achieve the desired results with minimal fuss.

To permanently skip tests in a project, edit the POM with:

```
<project>
  ...
  <build>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skip>true</skip>
      </configuration>
    </plugin>
    ...
  </build>
  ...
</project>
```

Tests can also be skipped on a one-off basis through an additional parameter in the command line.

The following command skips the compilation and execution of tests:

```
$ mvn install -Dmaven.test.skip=true
```

This command only skips the execution of tests:

```
$ mvn install -DskipTests=true
```

By default, Surefire includes all tests of the following patterns:

- `**/Test*.java`
- `**/*Test.java`
- `**/*TestCase.java`

If your test classes don't match these patterns, you can manually force Surefire to include a test:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
      <include>_____.java</include>
    </includes>
  </configuration>
</plugin>
```

In the preceding configuration, you can include a pattern for the test classes and Surefire plugin will execute the same.

In some situations, especially when refactoring or dealing with legacy code that isn't maintained, you may need to exclude certain tests from the unit test phase of the build lifecycle. This can be done by configuring the `exclude` property:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <excludes>
      <exclude>**/TestCircle.java</exclude>
      <exclude>**/TestSquare.java</exclude>
    </excludes>
  </configuration>
</plugin>
```

Sample reports of the Surefire plugin's test executions in two formats (text and XML) are shown as follows:

```
-------------------------------------------------------------------
Test set: net.srirangan.packt.maven.AppTest
-------------------------------------------------------------------
    Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.039 sec

<?xml version="1.0" encoding="UTF-8" ?>
<testsuite failures="0" time="0.033" errors="0" skipped="0" tests="1"
name="net.srirangan.packt.maven.AppTest">
  <properties>
    <property name="java.runtime.name" value="Java(TM) SE Runtime
Environment"/>
    <property name="sun.boot.library.path" value="C:\Program Files\
Java\jdk1.6.0_21\jre\bin"/>
    <property name="java.vm.version" value="17.0-b17"/>
    //... other environment properties ...
  </properties>
  <testcase time="0.009" classname="net.srirangan.packt.maven.AppTest"
name="testApp"/>
</testsuite>
```

How it works..

As the Surefire plugin gets executed, every unit test is executed sequentially.

As each test is executed, a report is generated in the `{basedir}/target/surefire-reports` directory. Reports are organized as per the project package structure as the tests are organized in the same package structure.

These reports can be rendered to display on the console based on a command-line argument, as seen in the *How to do it* section of this recipe.

As witnessed before, the Surefire plugin configuration can be modified in the project POM file. We had earlier set up the `skip` configuration parameter to be `true`. With this, all tests would be skipped by this plugin.

The Surefire reports contain a list for all the test cases that were executed along with the time elapsed, class name/references, successes, failures, and errors encountered during execution.

Additionally, the report will also contain environment and property information including the relevant paths, JDK versions, and so on.

Generating code coverage reports

In the *Generating unit test report* recipe, we leveraged the Surefire plugin to generate reports of our unit tests. These reports contained information about the success or failure of each individual test. However, what it lacked was an assessment of the scope of the unit tests itself in relation to the source code base.

Are these unit tests sufficient? Do they provide test coverage to all important portions of the functional code?

Cobertura is a tool that helps you answer some of these questions. It analyzes your source code and test code, calculates the percentage of code accessed by the tests and thus identifies portions of the code that aren't covered by tests.

Cobertura instruments Java bytecode after compilation, and generates HTML and XML based reports showing the percentage of code coverage for each class, each package, and the project overall.

Getting ready

Cobertura Maven plugin `http://mojo.codehaus.org/cobertura-maven-plugin/` embeds the code coverage features of Cobertura within your Maven build lifecycle. It is used to analyze code coverage in your project and helps highlight portions of your source code which lack enough unit test coverage.

The plugin provides us five goals to work with. They are:

- `cobertura:check`: Checks the last instrumentation results
- `cobertura:clean`: Cleans up rogue files being tracked
- `cobertura:dump-datafile`: Provides a data file dump
- `cobertura:instrument`: Instruments the compiled classes
- `cobertura:cobertura`: Instruments, tests, and generates the report

Installing the Cobertura Maven plugin is a two step process. It is explained as follows:

1. The first step involves the initial configuration in the reporting element of the project POM file. Implement the plugin within the project POM file, as shown as follows:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.4</version>
    </plugin>
```

```
    ...
  </plugins>
</reporting>
```

2. The second step is the instrumentation configuration, as shown in the following code. The `configuration/instrumentation` element can be used to override the default Cobertura instrumentation settings and meet our project needs.

   The following settings show how to ignore a package and exclude certain packages and patterns:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <configuration>
        <instrumentation>
         <ignores>
           <ignore>com.test.somepackage.*</ignore>
         </ignores>
         <excludes>
           <exclude>com/test/somepackage/**/*.class</exclude>
           <exclude>com/test/**/*Test.class</exclude>
         </excludes>
       </instrumentation>
      </configuration>
    </plugin>
  </plugins>
</build>
```

   The following settings bind the execution of the plugin with the clean phase of the build lifecycle:

```
<executions>
  <execution>
    <goals>
      <goal>clean</goal>
    </goals>
  </execution>
</executions>
```

How to do it...

The Cobertura report can be generated by executing the following command in the console:

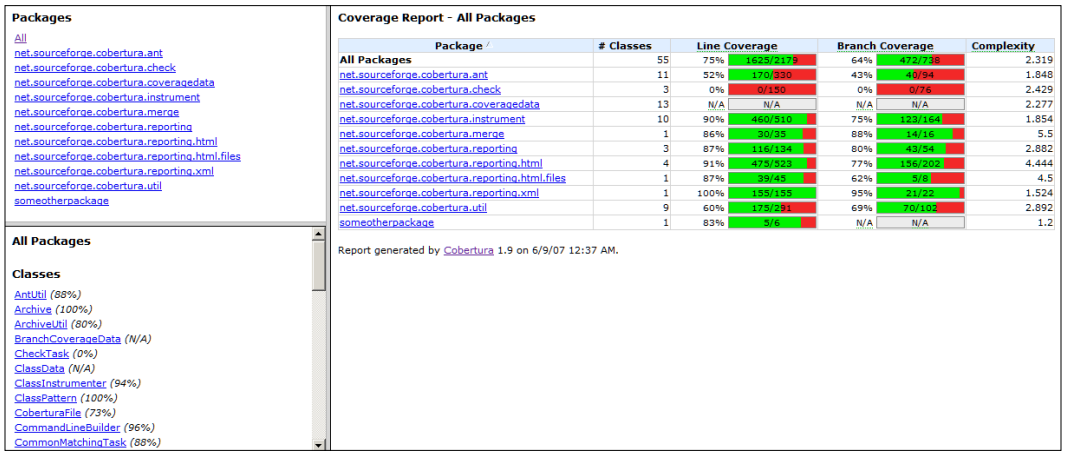
```
$ mvn cobertura:cobertura
```

This will generate the report in a standalone mode.

If the Cobertura plugin is bound to the reports phase of the project's build cycle, the Maven site command will also lead to the generation of the Cobertura report:

```
$ mvn site
```

The following screenshot shows the **Coverage Report** grouped as **Packages** in a web browser:



How it works...

As seen in the preceding screenshot, Cobertura can be configured to be triggered in the site phase of the Maven project build lifecycle. Alternatively, it can be triggered explicitly from the command line.

When executed, Cobertura scans the source folder of the Maven project and begins analyzing the source code package-wise and class-wise. At this point, it also takes into account the configuration settings for inclusions and exclusions. After analysis, Cobertura produces "pretty" reports in the form of HTML files viewable through the web browser. This can be made part of the Maven project site as well as the reports produced by other tools, for example, Sonar.

In an ideal scenario, Cobertura is part of the build lifecycle and automatically checks the code coverage on every build.

Builds can be configured to fail if the code coverage doesn't meet acceptable levels.

Generating code quality reports

There is good code, and then there is "good code". Presumably, there is a high level of understanding in your team and everyone agrees on common conventions and standards for code quality.

Once we've reached this agreement as a team, we must make sure all team members adhere to it as the code base of your projects evolves over time.

Checkstyle is a Java development tool to help developers adhere to coding standards. The Maven Checkstyle plugin `http://maven.apache.org/plugins/maven-checkstyle-plugin/` allows developers to generate reports regarding the "code style" implemented by the team in the project source code.

The Maven Checkstyle plugin introduces two goals:

- `checkstyle:checkstyle`: Performs analysis and generates a report
- `checkstyle:check`: Checks for violations against the last Checkstyle run

The minimum requirement for this plugin includes JDK 1.5 or higher and Maven 2.0.6 or higher.

Getting ready

To get started, you need to update the reporting element of the project POM file to include the Checkstyle plugin:

```
...
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>2.6</version>
    </plugin>
  </plugins>
</reporting>
...
```

How to do it...

The plugin can then be executed and reports generated by invoking the Maven site command:

```
$ mvn site
```

Depending on your project configuration, the invocation of `$ mvn site` can lead to the execution of many plugins and report generators. For situations where Checkstyle reports need to be generated in isolation from any other process or from the build lifecycle, the following command can be used:

```
$ mvn checkstyle:checkstyle
```

Based on your teams' understanding and code conventions, Checkstyle can be configured to match the same. To configure the Checkstyle plugin, it needs to be added to the build section of the project POM file. An example is shown as follows:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <enableRulesSummary>false</enableRulesSummary>
           ...
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

The following screenshot shows a sample Checkstyle report with a list of files, number of **Infos**, **Warnings**, and **Errors**. You can also see a package level summary of the same:

Summary

Files	Infos	Warnings	Errors
16	0	123	17

Files

Files	I	W	E
org/apache/maven/plugin/checkstyle/CheckstyleExecutor.java	0	4	0
org/apache/maven/plugin/checkstyle/CheckstyleExecutorException.java	0	4	0
org/apache/maven/plugin/checkstyle/CheckstyleExecutorRequest.java	0	60	2
org/apache/maven/plugin/checkstyle/CheckstyleReport.java	0	0	3
org/apache/maven/plugin/checkstyle/CheckstyleReportGenerator.java	0	18	0
org/apache/maven/plugin/checkstyle/CheckstyleReportListener.java	0	7	0
org/apache/maven/plugin/checkstyle/CheckstyleResults.java	0	0	1
org/apache/maven/plugin/checkstyle/CheckstyleViolationCheckMojo.java	0	0	1
org/apache/maven/plugin/checkstyle/DefaultCheckstyleExecutor.java	0	2	6
org/apache/maven/plugin/checkstyle/ReportResource.java	0	6	0
org/apache/maven/plugin/checkstyle/VelocityTemplate.java	0	9	1
org/apache/maven/plugin/checkstyle/rss/CheckstyleRssGenerator.java	0	1	0
org/apache/maven/plugin/checkstyle/rss/CheckstyleRssGeneratorRequest.java	0	9	0
org/apache/maven/plugin/checkstyle/rss/DefaultCheckstyleRssGenerator.java	0	3	1
org/codehaus/plexus/util/interpolation/RegexBasedInterpolator.java	0	0	1
org/codehaus/plexus/util/interpolation/ValueSource.java	0	0	1

How it works...

Checkstyle is typically executed as a part of the Maven site phase in the build lifecycle. However, as seen in the preceding *How to do it...* section, it can also be executed explicitly from the command line.

Regardless of how it is executed, during execution, Checkstyle covers a series of standard checks which include:

- Annotations
- Block checks
- Class design
- Duplicate code
- Imports

The checks can be configured as part of the plugin rules and Checkstyle in itself supports a far greater number of checks. Complete details of these can be found at `http://checkstyle.sourceforge.net/`.

Setting up the Maven dashboard

The Maven dashboard plugin commits to centralizing and enabling effective sharing of information, created by a number of other Maven reporting plugins. The list of plugins supported by the dashboard is as follows:

- Test coverage reports
- Cobertura: Calculates the percentage of code accessed by tests
- Clover: Calculates the test coverage metrics
- Code quality reports
- Checkstyle: Performs static code style analysis
- PMD/CPD: Performs Java source code analysis and copy-paste detection
- FindBugs: Performs Java source code analysis to detect bug patterns
- JDepend: Calculates design quality metrics by package
- Taglist: Performs static code analysis to find tags in the code, like `@todo` or `//TODO` tags
- Unit testing reports
- Surefire: Executes the unit tests of an application

Getting ready

The goals made available by the dashboard plugin are:

- `dashboard:dashboard`: Generates the dashboard to aggregate all reports
- `dashboard:persist`: Preserves the dashboard in a backend database

How to do it...

The dashboard plugin can be configured in the project's POM file by including the plugin in the POM's `build` and `reporting` elements.

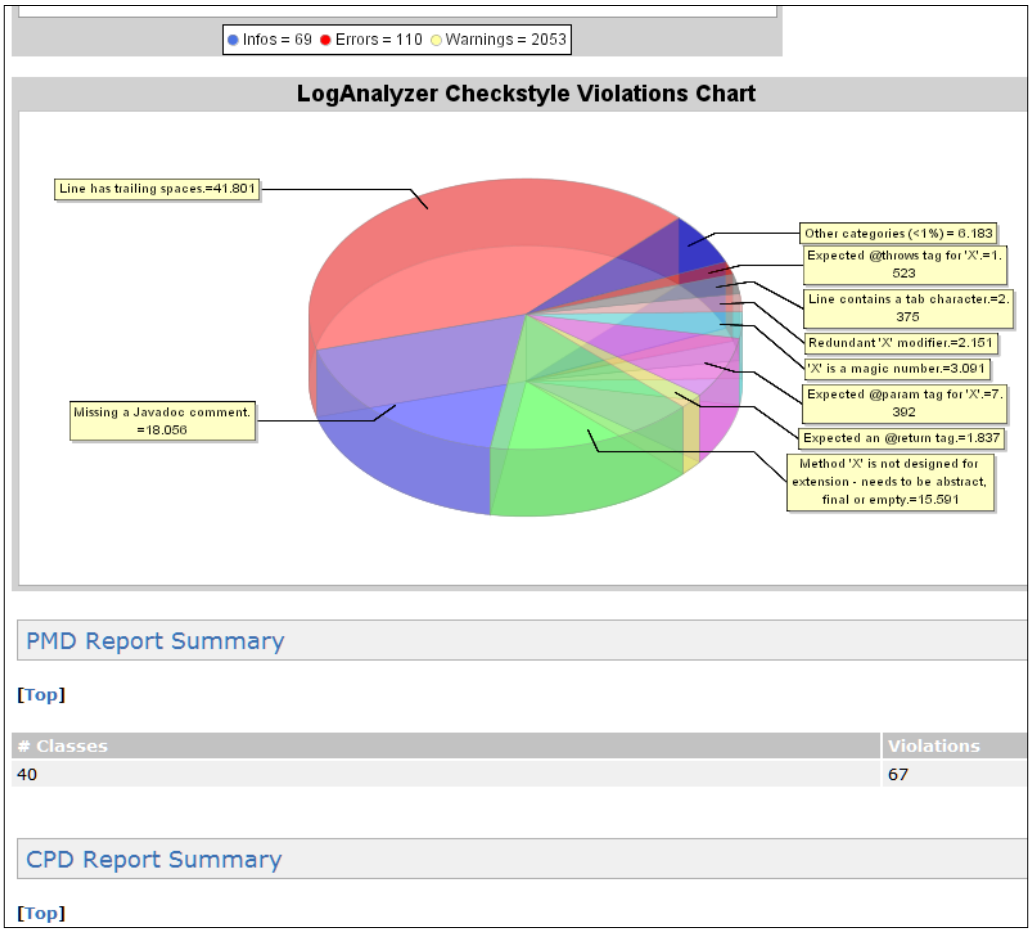
```
<project>
  ...
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>dashboard-maven-plugin</artifactId>
          <version>1.0.0-beta-1</version>
        </plugin>
        ...
      </plugins>
    </pluginManagement>
  ...
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>dashboard-maven-plugin</artifactId>
        <version>1.0.0-beta-1</version>
      </plugin>
      ...
    </plugins>
  </build>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>dashboard-maven-plugin</artifactId>
        <version>1.0.0-beta-1</version>
```

```
        </plugin>
        ...
      </plugins>
    </reporting>
    ...
  </project>
```

Usage from the command line includes the generation of all reports and then the generation of the dashboard to include all reports generated before:

```
$ mvn site
$ mvn dashboard:dashboard
```

The following screenshot shows the Maven dashboard report for a single Maven project:



How it works...

In this recipe's introduction, we saw the list of plugins that the Maven dashboard supports. The Maven dashboard is in fact dependent on these plugins, as a majority of its operations are performed on the results of these plugins, as compiled by Maven site.

The Maven dashboard works on the reports generated by supported plugins. The **Metrics Collector** uses these XML reports for data analysis, while the **Graph Generator** renders a visual representation of these finds.

Finally, the DB service is used to keep these in a database so that analysis reports and visualizations are available on the dashboard.

5
Java Development with Maven

In this chapter, we will cover:

- Building a web application
- Running a web application
- Enterprise Java development with Maven
- Using Spring Framework with Maven
- Using Hibernate persistence with Maven
- Using Seam Framework with Maven

One of the primary reasons stated for the adoption of Maven is its tremendous potential in enterprise Java development. Not only does it make practical sense for companies and teams to implement Maven in their projects for methodology-related and best practice implementation related reasons (see *Chapter 2*, *Software Engineering Techniques* and *Chapter 3, Agile Team Collaboration*), but Maven does provide inherent capabilities for Java Web and Java EE development which makes it an ideal choice with the best of both worlds, that is, technical capability as well as management and methodology support.

This chapter will begin with a recipe on creating and building web applications with Maven while moving on to a recipe on running this web application on an embedded **Web Application Server** (**Jetty**).

The next recipe will look at Java EE development with the Maven EAR plugin to package our EJBs in EAR packages for use with our web application projects.

Then we will look at a host of popular enterprise frameworks including Spring Framework, Hibernate ORM Framework, and JBoss Seam Framework. These frameworks are highly popular, and with their popularity growing by the day among Agile teams, they are quickly replacing the cumbersome J2EE development paradigm of the late 90s and early 2000s. They represent the future of Web and Enterprise Java Development. The recipes in this chapter will focus on specific plugins for these frameworks and the implementation of these frameworks in your Maven projects.

Building a web application

This recipe will see us creating a simple web application with a Maven Archetype plugin. The web application will:

- Run in an embedded web application server (Jetty)
- Have some dependencies added
- Contain a simple servlet
- Have a WAR file generated

Our goal here is for you to be able to start using Maven to accelerate web application development.

Getting ready

We start off with the concept of a (**POWA**) **Plain Old Web Application**, that is, a web application project consisting of one servlet and one **Java Server Page** (**JSP**). The archetype template that will be used to generate a POWA is `org.sonatype.mavenbook.simpleweb`.

How to do it...

Start the command-line terminal and execute the following `generate` command:

```
$ mvn archetype:generate -DarchetypeArtifactId=maven-archetype-
webapp -DartifactId=testWebApp -DgroupId=net.srirangan.packt.maven
-Dversion=1.0-SNAPSHOT -Dpackage=net.srirangan.packt.maven
```

You should get the following output. Notice that you will get a prompt asking for the package.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1 ----------------------
-------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.0-alpha-5:generate (default-cli) @
standalone-pom >>>
```

```
[INFO]
[INFO] <<< maven-archetype-plugin:2.0-alpha-5:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.0-alpha-5:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Using property: groupId = net.srirangan.packt.maven
[INFO] Using property: artifactId = testWebApp
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = net.srirangan.packt.maven
Confirm properties configuration:
groupId: net.srirangan.packt.maven
artifactId: testWebApp
version: 1.0-SNAPSHOT
package: net.srirangan.packt.maven
 Y: : y
[INFO] ----------------------------------------------------------
[INFO] Using following parameters for creating OldArchetype: maven-
archetype-webapp:1.0 ----------------------------------------------
--------
[INFO] Parameter: groupId, Value: net.srirangan.packt.maven
[INFO] Parameter: packageName, Value: net.srirangan.packt.maven
[INFO] Parameter: package, Value: net.srirangan.packt.maven
[INFO] Parameter: artifactId, Value: testWebApp
[INFO] Parameter: basedir, Value: C:\Projects
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] ********************* End of debug info from resources from
generated POM **********************
[INFO] OldArchetype created in dir: C:\Projects\testWebApp
[INFO] ----------------------------------------------------------
[INFO] BUILD SUCCESS ----------------------------------------------
[INFO] Total time: 6.591s
[INFO] Finished at: Mon Nov 15 21:25:38 IST 2010
[INFO] Final Memory: 9M/114M --------------------------------------
-----------------
```

The `archetype:generate` command executed before has created a new project folder, `testWebApp`, which contains the following POM file:

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

```
    <artifactId>testWebApp</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>testWebApp Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
    <build>
      <finalName>testWebApp</finalName>
    </build>
  </project>
```

Now, we add a build plugin to compile the project with JDK 1.6. The `build` element of the POM file should look similar to the following:

```
  ..
    <build>
    <finalName>testWebApp</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
      </plugin>
    </plugins>
    </build>
  ..
```

How it works...

The Apache Maven web application project will work like any other Maven project. The project structure looks like this:

```
├──src
|   └──main
|       ├──resources
|       └──webapp
```

```
|           └──WEB-INF
└──target
    ├──classes
    ├──maven-archiver
    ├──surefire
    └──testWebApp
        ├──META-INF
        └──WEB-INF
            └──classes
```

You can run the commands to compile and test the project.

```
$ mvn compile
$ mvn test
$ mvn install
```

On installation, it will be packaged and deployed in the local repository like any other Maven project. It can also be deployed into a remote repository.

There's more...

If you have a look, you will find the project `source/main/webapp` folder consisting of `index.jsp`. Make suitable changes to that file such as adding some HTML:

```
<html>
<body>
<h2>Hello World!</h2>
</body>
</html>
```

Now that we have the build phase plugin set up, we can proceed to built the project:

```
$ mvn install
```

After a successful build, the target folder has been generated. In the target folder, you will find a WAR file named `testWebApp.war`. Take a closer look at the project POM file and you will find that the packaging in there will be set to `WAR` and the final artifact reflects this packaging property:

```
...
<artifactId>testWebApp</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
...
```

See also

- *Chapter 5*, *Running a web application* section
- *Chapter 2*, *Deployment automation* section

Running a web application

Maven can automate the process of deploying, cleaning, and redeploying the WAR on a developer machine web application server, making the entire process less cumbersome for the developer. This is done with the Jetty plugin.

According to the Wikipedia:

> *"Jetty is a pure Java-based HTTP server and servlet container (application server). Jetty is a free and open source project under the Apache 2.0 License. Jetty deployment focuses on creating a simple, efficient, embeddable and pluggable web server. Jetty's small size makes it suitable for providing web services in an embedded Java application. It also offers support for Web Sockets, OSGi, JMX, JNDI, JASPI, AJP, and other Java technologies".*

In the preceding recipe, we built the web application project and generated a WAR file. Now deploying this WAR would generally include setting up Apache Tomcat, unpacking a distribution, copying your application's WAR file to a `webapps/` directory, and then starting your Tomcat container.

Although you could still do this, the Maven Jetty plugin makes this entire process much easier.

Getting ready

Let us add the `plugin` element shown in the following code to your web application project's build configuration:

```
<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <version>6.1.17</version>
    </plugin>
    ...
  </plugins>
</build>
```

How to do it...

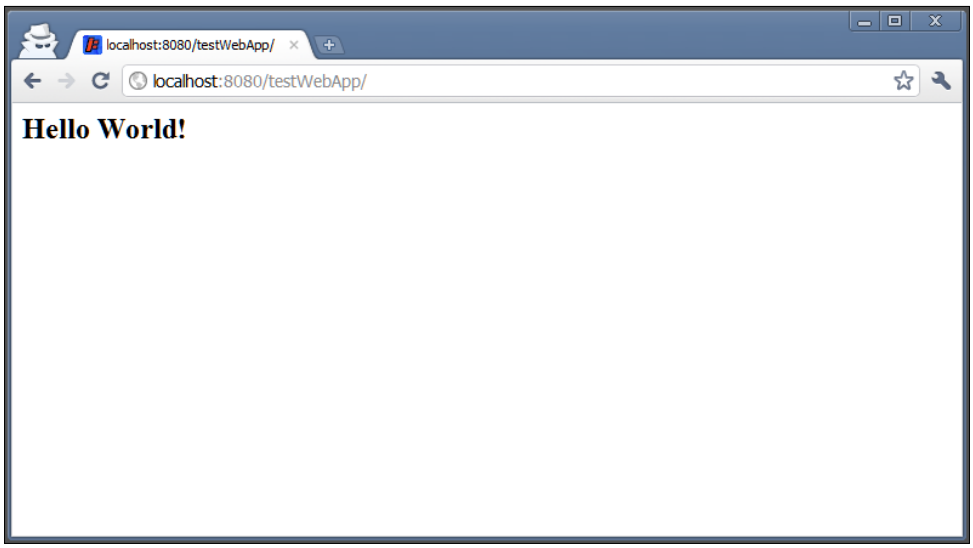
Once the Jetty plugin has been set up in the project POM file, the `jetty:run` goal is available:

```
$ mvn jetty:run
```

When running the preceding command for the first time, Maven will automatically download all dependencies and start the Jetty plugin.

```
[INFO] Starting jetty 6.1.26 ...
2010-11-16 07:37:00.135:INFO::jetty-6.1.26
2010-11-16 07:37:00.247:INFO::No Transaction manager found - if your
webapp requires one, please configure one.
2010-11-16 07:37:00.484:INFO::Started
SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Now **testWebApp** is running on **localhost:8080/testWebApp**:



A few notes specifically for Windows are:

Make sure the inbuilt or external firewall does not interfere and allows the Jetty server to execute on port **8080**.

Make sure the path to the Maven repository does not contain spaces. This may be verified in the `Maven settings.xml` file.

How it works...

The underlying process varies depending on the goal that has been executed. The Jetty plugin makes the following goals available:

```
$ mvn jetty:run
```

This goal doesn't require the project to be assembled as a WAR file and will deploy the web application directly from the project. This saves time and is used often during development.

```
$ mvn jetty:run-exploded
```

This goal is used to deploy the exploded WAR file assembled from the project.

```
$ mvn jetty:run-war
```

This goal is used to deploy the WAR file assembled from the project.

Jetty then starts as an application server and the web application is available at the URL `http://localhost:8080/<web-app-context>`.

Enterprise Java development with Maven

The Maven EAR plugin primarily is used to generate the J2EE Enterprise Archive file. The **EAR** file (or **Enterprise Archive**) is a Java EE file format used for packaging one or more modules for deployment onto application servers. It also contains deployment descriptors which are XML files (like `application.xml`) that describe deployment information for the package.

According to the Maven EAR plugin FAQ:

> *"An EAR archive is used to deploy standalone EJBs, usually separated from the web application. Thus there is no need for a web application to access these EJBs. The EJBs are still accessible, though, using EJB clients."*

The EAR plugin supports the following artifacts:

- `ejb`
- `war`
- `jar`
- `ejb-client`
- `rar`
- `ejb3`
- `par`

- `sar`
- `wsr`
- `har`

The minimum requirements for this plugin are Maven 2.0.6 or higherer and JDK 1.4 or higher.

Getting ready

If you don't have a Maven Java project, create one by executing the following command:

```
$mvn archetype:generate -DarchetypeArtifactId=maven-archetype-j2ee-simple
```

This command creates a simple J2EE Enterprise application with the following structure:

```
├──ear
├──ejbs
│  └──src
│      └──main
│          └──resources
│              └──META-INF
├──primary-source
├──projects
│  └──logging
├──servlets
│  └──servlet
│      └──src
│          └──main
│              └──webapp
│                  └──WEB-INF
└──src
    └──main
        └──resources
```

As of today, this archetype contains a bug. The `generate pom.xml` must be modified and the `site` sub module must be removed.

How to do it...

The configuration of the EAR plugin is similar to the configuration of the other plugins. The following needs to be specified in your Maven Java project:

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

```
      <plugins>
        [...]
        <plugin>
          <artifactId>maven-ear-plugin</artifactId>
          <version>2.4.2</version>
        </plugin>
    [...]
  </project>
```

The Maven EAR plugin provides the following goals:

- `ear:ear`

  Builds the Enterprise Archive files

- `ear:generate-application-xml`

  Generates EAR deployment description files

- `ear:help`

  Displays help information

How it works...

This plugin can be executed on an individual goal basis or be called as part of the project's "package" phase of the build lifecycle.

An example of individual goal executions is:

```
$ mvn ear:ear
$ mvn ear:generate-application-xml
```

A sample output is shown as follows:

```
[INFO] ------------------------------------------------------------
------
[INFO] Building project 1.0
[INFO] ------------------------------------------------------------
------
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ project -
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
[INFO] Building sub projects 1.0
[INFO] ------------------------------------------------------------
------
```

```
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ projects
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
[INFO] Building logging 1.0
[INFO] ------------------------------------------------------------
------
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ logging -
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
[INFO] Building core project classes 1.0
[INFO] ------------------------------------------------------------
------
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ primary-s
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
[INFO] Building servlets 1.0
[INFO] ------------------------------------------------------------
------
[INFO]
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ servlets
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
[INFO] Building servlet 1.0
[INFO] ------------------------------------------------------------
------
[INFO]
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ servlet -
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
```

```
[INFO] Building enterprise java beans 1.0
[INFO] ------------------------------------------------------------
------
[INFO]
[INFO] --- maven-ear-plugin:2.5:generate-application-xml (default-
  cli) @ ejbs ---
[INFO] Generating application.xml
[INFO]
[INFO] ------------------------------------------------------------
------
[INFO] Building ear assembly 1.0
[INFO] ------------------------------------------------------------
------
[INFO]
[INFO] --- maven-ear-plugin:2.4.2:generate-application-xml (default-
  cli) @ ear ---
[INFO] Generating application.xml
```

An example of the `package` build phase call is:

```
$ mvn package
```

Using Spring Framework with Maven

Spring is a popular open source application framework for the Java platform. While Spring Framework's core features can be used by any Java application, specialized extensions of the Spring Framework for enterprise development are available. Spring has become a popular alternative to the Enterprise JavaBean model.

One of Spring Framework's strongest aspects is its support of Inversion of Control. In a broad sense, this can be described as allowing beans to be defined (through XML files or annotations) and these can be "injected" as and when required.

This makes for easier design and architecture of enterprise applications and thus easier development. It allows better leveraging of interfaces, allowing proper decoupling of API definitions and implementations.

And since beans can be easily "injected" into tests (Spring Framework provides extensive unit testing support), high testability of enterprise applications remains possible.

This recipe assumes that you have basic knowledge of the Spring Framework itself. The rest of the recipe helps you integrate Spring Framework's components with your Maven Java project.

Getting ready

You need a Maven Java project to implement the Spring Framework; if you don't have one, running the following command in the terminal will create a sample Java project:

```
$ mvn archetype:generate -DarchetypeArtifactId=maven-archetype-
quickstart -DgroupId=net.srirangan.packt.maven -DartifactId=TestAppCreate
-Dversion=1.0.0
```

Alternatively, you can also create a Maven Java web application as shown in the first recipe of *Chapter 5*, *Building a web application*.

How to do it...

Spring Framework artifacts need to be defined as dependencies in the project POM file. The following list contains all Spring Framework dependencies which allow you to acquire Spring 3 artifacts for your Maven project. (Source: Keith Donald on `http://blog.springsource.com`)

```
<!-- Shared version number properties -->
<properties>
    <org.springframework.version>3.0.5.RELEASE</org.springframework.
version>
</properties>

<dependencies>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-expression</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-oxm</artifactId>
  <version>${org.springframework.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-web</artifactId>
    <version>${org.springframework.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${org.springframework.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc-portlet</artifactId>
    <version>${org.springframework.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${org.springframework.version}</version>
    <scope>test</scope>
  </dependency>

  </dependencies>
```

How it works...

Specifying the Spring Framework artifacts as dependencies works similarly to any other dependency being defined in the project POM. They are made available to your application.

Spring also requires the availability of the context configuration XML file. This file traditionally contained bean declarations, but since Spring 3 supports annotation-based bean declarations, the context file could contain additional configuration information such as the package paths which Spring should scan for annotation-based beans.

Here is a Spring 3 context file `<project-root>/src/main/resources/context.xml`, which defines the package path that must be scanned for beans:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
      xmlns:tx="http://www.springframework.org/schema/tx"
```

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-
            context.xsd
          http://www.springframework.org/schema/tx
          http://www.springframework.org/schema/tx/spring-tx.xsd">

        <context:component-scan base-package="net.srirangan.packt.maven"
          />

    </beans>
```

There's more...

You'll now find brief explanations of two of the core development and design concepts that Spring Framework brings along, namely, Inversion of Control (IoC) and Aspect Oriented Programming (AOP).

Inversion of Control

The **Inversion of Control** (**IoC**) container is a central concept of the Spring Framework. It provides a consistent mechanism to configure and manage JavaBeans. Typically, the IoC container is configured using XML files, which contain the bean definitions.

Aspect Oriented Programming

Aspect Oriented Programming focuses on core business logic programming by isolating supporting functions and logic. The Spring Framework supports **Aspect Oriented Programming** (**AOP**) based on interceptions and runtime configuration. Compared to AspectJ, Spring AOP framework is much simpler while supporting all major features.

Unit testing with Spring Framework

If you are using Spring in your project, you will definitely need to write tests that require the injection of Spring Beans for broad-ranged test coverage. We will look at how that can be achieved by creating a `service` class, exposing it as a Spring 3 bean, writing tests which inject the Service class, and testing it.

Let's start by creating `MyServiceOne.java` in `src/main/java/net/srirangan/packt/maven/services`

```
package net.srirangan.packt.maven.services;

import org.springframework.stereotype.Service;

@Service("myServiceOne")
public class MyServiceOneImpl {
```

```
  public String getA() {
    return "A";
  }
}
```

Notice the `@Service` annotation just preceding the class definition and note that the `@Service` annotation contains the bean name `@Service("...bean..name..here...")`

Our next step is to ensure that our `context.xml` file's component scan element contains the services package.

```
<context:component-scan base-
  package="net.srirangan.packt.maven.services" />

Finally we define a test called MyServiceOneTest.java under
  "src/test/java/net/srirangan/packt/maven/services":
package net.srirangan.packt.maven.services;

import org.junit.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test
  .context.junit4.AbstractJUnit4SpringContextTests;

@ContextConfiguration("classpath:context.xml")
public class MyServiceOneTest extends
  AbstractJUnit4SpringContextTests {

  @Autowired
  private MyServiceOne myServiceOne;

  @Test
  public void shouldGetA() {
    // .. test body here
  }

}
```

Few quick pointers to note in the preceding test:

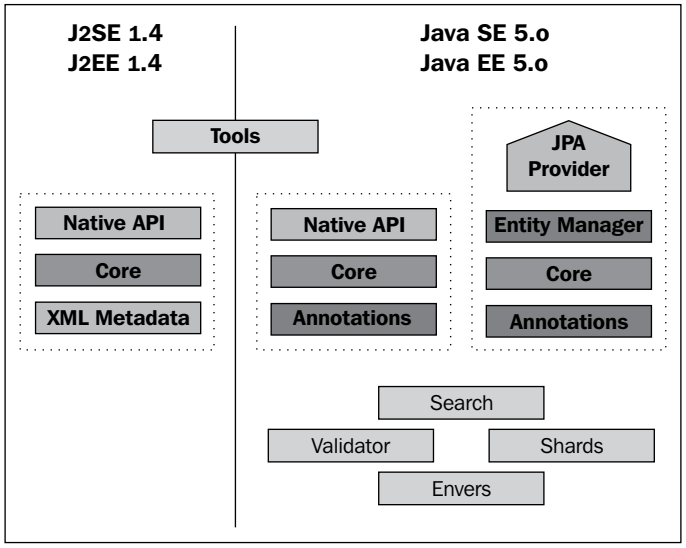
- Test class extends `AbstractJUnit4SpringContextTests`, which initializes Spring
- The `@ContextConfiguration` annotation defines the location of the context file for the Test
- Use of the `@Autowired` annotation to inject the bean

Using Hibernate persistence with Maven

Persistence is definitely a common feature for a majority of web and enterprise applications. Hibernate replaces interaction with JDBC by providing an object-relational mapping layer. In plain English, Hibernate lets you read and write Java objects into a database in a more seamless and business-oriented manner.

Hibernate's inherent simplicity has lead to its enormous growth and adoption. This simplicity is demonstrated by the fact that neither a deep understanding of the Hibernate API nor any real SQL skills are required to implement and efficiently leverage the framework into your application.

The following image shows the various tools provided by Hibernate and its evolution:



Experienced software engineers have been consistent in advocating Hibernate not only for the features mentioned and described, but for arguably the biggest advantage that Hibernate adoption generates, to make developers more efficient with their time while standardizing data access code style across projects and teams.

In pre-Hibernate days, a majority of the developer's time was drained by monotonous, repetitive yet bulky code interacting with the underlying data store. The effective implementation of Hibernate successfully reduces this and lets the developer focus on more challenging aspects of the project such as design, business/domain logic, and testing.

Not surprisingly, we find Hibernate's stated goal being:

> *"..to relieve the developer from 95 percent of common data persistence related programming tasks"*

Getting ready

To use Hibernate in our Maven project, we will implement the Maven Hibernate3 plugin found at `http://mojo.codehaus.org/maven-hibernate3`.

But first, we will create a simple Maven Java project and implement Hibernate-based persistence. Use the following command to create a Maven Java project based on `maven-archetype-quickstart`:

```
$ mvn archetype:generate -DarchetypeArtifactId=maven-archetype-
quickstart -DgroupId=net.srirangan.packt.maven -DartifactId=TestAppCreate
-Dversion=1.0.0
```

This will create a simple Maven Java project with a valid POM file. If you have some basic knowledge of Hibernate 3, it will be helpful in going along. This recipe will help merge this knowhow of Hibernate 3 with your Maven project. Hibernate, in itself, is a vast topic and has entire books dedicated to it. However, we will take a quick look at implementing Hibernate in this project and complementing it with the use of the Hibernate Maven plugin.

How to do it...

The first step is to identify and include all of Hibernate's artifacts and its dependencies into your Maven project. Open the `pom.xml` file of your Maven project and include the following dependencies:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
    <version>3.4.0.GA</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.3.2.GA</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
```

```
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.6.1</version>
   <type>jar</type>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.6.1</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.12.1.GA</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.15</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
```

The included Hibernate artifacts are:

- `hibernate-annotations 3.4.0.GA`
- `hibernate-core 3.3.2.GA`

Hibernate is notorious for requirements of further dependencies, some of them at runtime. We have pre-included them in the preceding code. They are:

- `slf4j-api 1.6.1`
- `slf4j-simple 1.6.1`
- `javassist 3.12.1.GA`
- `mysql-connector-java 5.1.15`

Our example will be showcasing Hibernate's interaction with an underlying MySQL data store.

We will now create two Java files and one Hibernate properties file. I am using the following package structure `net.srirangan.packt.maven.TestHibernateApp`. However, of course you are free to use your own. The project structure we create is:

```
├──java
│   └──net
│       └──srirangan
│           └──packt
│               └──maven
│                   └──TestHibernateApp
│                       ├──app
│                       ├──├──App.java
│                       └──domain
│                       ├──├──User.java
└──resources
│   └──hibernate.properties
```

User.java is a Hibernate 3 entity. We use annotations to limit the need for XML configuration files for Hibernate.

```
package net.srirangan.packt.maven.TestHibernateApp.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Table
@Entity
public class User {

    private Long id;

    private String username;

    private String password;

    public void setUsername(String username) {
        this.username = username;
    }

    @Column
    public String getUsername() {
        return username;
    }
```

```
        public void setPassword(String password) {
            this.password = password;
        }

        @Column
        public String getPassword() {
            return password;
        }

        public void setId(Long id) {
            this.id = id;
        }

        @Id
        @GeneratedValue
        public Long getId() {
            return id;
        }

    }
```

Notice the use of the `@Table`, `@Entity`, `@Id`, `@Column` annotations. These are configurable and customizable, of course, but the defaults work great as well. In this case, the table name is the same as the class/entity name and column names and types are exactly as defined in the entity.

`hibernate.properties` is the next file we define. It contains external data source configuration and some Hibernate behavior-specific configuration as well.

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/testdatabase
hibernate.connection.username=testUser
hibernate.connection.password=testPass
hibernate.connection.pool_size=1
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.hbm2ddl.auto=create
```

The first six properties are specific to the underlying MySQL database and the connection pool size available. `hibernate.show_sql` echoes all queries onto the console, while `hibernate.hbm2ddl.auto=create` generates the tables in the database each time the application is executed. Other possible values for this field are `validate`, `update`, and `create-drop`. While `validate` and `update` are self-explanatory, `create-drop` creates the schema on application startup and drops it after execution is completed.

Finally, we create the main application to use our entity in `App.java`:

```
package net.srirangan.packt.maven.TestHibernateApp.app;

import net.srirangan.packt.maven.TestHibernateApp.domain.User;

import org.hibernate.Session;
import org.hibernate.cfg.AnnotationConfiguration;

public class App  {

    public static void main( String[] args ) {

        AnnotationConfiguration configuration = new
AnnotationConfiguration()
         .addPackage("net.srirangan.packt.maven.TestHibernateApp.
domain")
         .addAnnotatedClass(User.class);

        Session session = configuration.buildSessionFactory().
openSession();

        User user1 = new User();
        user1.setUsername("hello");
        user1.setPassword("world");
        session.save(user1);

        session.close();

    }
}
```

It's a very simple example that starts a session, creates a user, and saves it in the database.

When you execute this application, the console will show a log of Hibernate starting up and an insert query to save the `User` that was created.

If you check your database, a new table named `user` was created with one record inserted with the `username` as `hello` and `password` as `world` while `id` would be `1`.

```
mysql >> show tables;
|| *Tables_in_testdatabase* ||
|| user ||

mysql >> select * from user;
|| *id* || *password* || *username* ||
|| 1 || world || hello ||
```

We will now have a look at the Maven Hibernate plugin which provides a set of goals to simplify Hibernate development and automate certain aspects of it. To integrate the Maven Hibernate3 plugin with your Maven Java project, you need to define the plugin in the `build | plugins` element of the project POM file.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>hibernate3-maven-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <components>
          <component>
            <name>hbm2ddl</name>
            <implementation>jdbcconfiguration</implementation>
          </component>
          <component>
            <name>hbm2hbmxml</name>
            <outputDirectory>src/main/resources</outputDirectory>
          </component>
        </components>
        <componentProperties>
          <drop>true</drop>
          <configurationfile>/src/main/resources/hibernate.cfg.xml
            </configurationfile>
        </componentProperties>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>jdbc.artifact.groupid</groupId>
          <artifactId>jdbc-driver</artifactId>
          <version>1.0</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
```

Interestingly, to run the Maven Hibernate3 plugin, you don't need to specify the Hibernate artifacts (JARs) as dependencies in the POM file.

The requirements of the Hibernate plugin include Maven 2.0.6 or higher and JDK 1.4 or higher.

It is also recommended that you set up the `build | pluginManagement`:

```
<pluginManagement>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>hibernate3-maven-plugin</artifactId>
      <version>2.2</version>
    </plugin>
    ...
  </plugins>
</pluginManagement>
```

How it works...

The Maven Hibernate3 plugin provides us with seven goals. Each of these goals can be executed by the command line, provided you are in the Maven project root folder.

- `hibernate3:hbm2cfgxml` to generate the `hibernate.cfg.xml` file:

  ```
  $ mvn hibernate3:hbm2cfgxml
  ```

- `hibernate3:hbm2ddl` to generate database schema:

  ```
  $ mvn hibernate3:hbm2ddl
  ```

- `hibernate3:hbm2doc` to generate database schema docs:

  ```
  $ mvn hibernate3:hbm2doc
  ```

- `hibernate3:hbm2hbmxml` to generate `hbm.xml` files:

  ```
  $ mvn hibernate3:hbm2hbmxml
  ```

- `hibernate3:hbm2java` to generate Java classes from `*.hbm.xml` files:

  ```
  $ mvn hibernate3:hbm2java
  ```

- `hibernate3:hbmtemplate` to renders templates against Hibernate Mapping information:

  ```
  $ mvn hibernate3:hibernate3
  ```

Using Seam Framework with Maven

Seam Framework is a next generation Java web application framework developed by JBoss, Red Hat. It combines two frameworks, namely, the Enterprise JavaBeans (EJB3) and Java Server Faces (JSF). It also introduces the concept of "bijection" (similar to Spring Framework's injection) which allows for injection and removal of beans using simple `@In` and `@Out` annotations.

This recipe assumes you have basic knowledge of the Seam Framework and helps integrate Seam within your Maven web application project.

Getting ready

Seam is a web application framework. Hence you need a Maven web application project to implement it in. If you don't have one ready, create it by running the following command in the console:

```
$ mvn archetype:generate -DarchetypeArtifactId=maven-archetype-
webapp -DartifactId=testWebApp -DgroupId=net.srirangan.packt.maven
-Dversion=1.0-SNAPSHOT -Dpackage=net.srirangan.packt.maven
```

How to do it...

Seam3 artifacts and examples are published on the JBoss Community Maven Repository. These include source and JavaDoc artifacts as well. To access them, you need to modify your Maven `settings.xml` to recognize and access the JBoss public repository.

```
<profiles>
  <profile>
    <id>jboss-public-repository</id>
    <activation>
      <property>
        <name>jboss-public-repository</name>
        <value>!false</value>
      </property>
    </activation>
    <repositories>
      <repository>
      <id>jboss-public-repository-group</id>
      <name>JBoss Public Maven Repository Group</name>
        <url>http://repository.jboss.org
          /nexus/content/groups/public</url>
        <releases>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
          <enabled>false</enabled>
          <updatePolicy>never</updatePolicy>
        </snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>jboss-public-repository-group</id>
        <name>JBoss Public Maven Repository Group</name>
```

```
        <url>http://repository.jboss.org
          /nexus/content/groups/public</url>
        <releases>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
          <enabled>false</enabled>
          <updatePolicy>never</updatePolicy>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
 </profiles>
```

The `pluginRepositories` can be defined at the project POM level, but it may be more convenient to do it at the Maven `settings.xml` level if the dependency spans across multiple projects.

Seam3 artifacts need to be individually included as dependencies in your project/module POM file as and when required.

The Seam3 artifacts available are:

- Faces

```
<dependency>
   <groupId>org.jboss.seam.faces</groupId>
   <artifactId>seam-faces</artifactId>
   <version>3.0.0.Alpha3</version>
</dependency>
```

- International

```
<dependency>
   <groupId>org.jboss.seam.international</groupId>
   <artifactId>seam-international</artifactId>
   <version>3.0.0.Alpha1</version>
</dependency>
```

- JMS

```
<dependency>

   <groupId>org.jboss.seam.jms</groupId>
   <artifactId>seam-jms</artifactId>
   <version>3.0.0.Alpha1</version>
</dependency>
```

- Remoting

```
<dependency>
   <groupId>org.jboss.seam.remoting</groupId>
   <artifactId>seam-remoting-core</artifactId>
   <version>3.0.0-Beta1</version>
</dependency>
```

- REST

```
<dependency>
   <groupId>org.jboss.seam.rest</groupId>
   <artifactId>seam-rest</artifactId>
   <version>3.0.0.Alpha1</version>
</dependency>
```

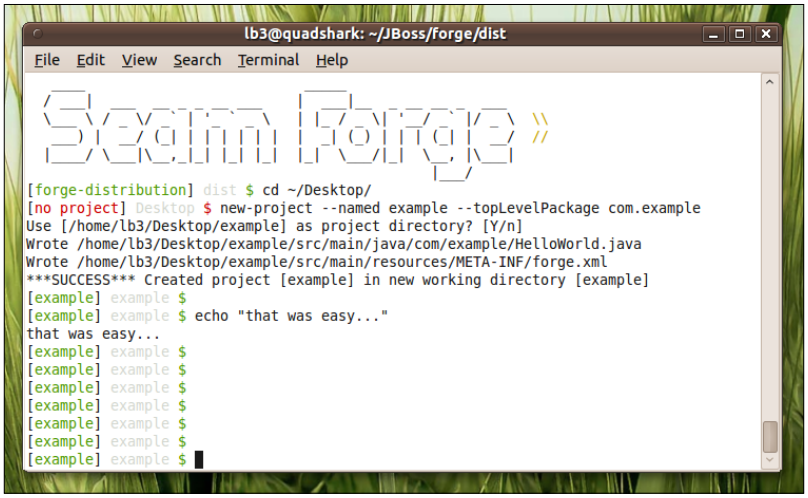
- XML

```
<dependency>
   <groupId>org.jboss.seam.xml</groupId>
   <artifactId>seam-xml-config</artifactId>
   <version>3.0.0.Alpha2</version>
</dependency>
```

The Seam Framework community has also developed **Seam Forge**, which is defined according to `http://seamframework.org` as:

> *"A core framework for rapid-application development in a standards-based environment. Plugins / incremental project enhancement for Java EE and more"*

Think of it as a command-line based application to achieve rapid code generation for Java/Seam. The following screenshot shows the Seam Forge command line in an active state:



Seam Forge is a Maven project which is publicly hosted on `GitHub.com`. You will need a Git client installed to "clone" (similar to `svn checkout`) the project on to your workstation.

You will need Maven 3 and Java 6

There are five steps for downloading, installing, and executing Seam Forge. They are as follows:

1. The first step is to clone the project using a Git or Git client:

   ```
   $ git clone git://github.com/seam/forge.git seam-forge
   ```

   This will clone, checkout, download the Seam Forge sources on to a new folder created on your current working directory called `seam-forge`.

2. Navigate into this folder:

   ```
   $ cd seam-forge
   ```

3. Build/install Seam Forge:

   ```
   $ mvn install
   ```

4. Once you see `build success`, navigate into the `dist` folder:

   ```
   $ cd dist
   ```

5. Execute!:

   ```
   $ mvn exec:java
   ```

How it works...

A Seam Framework Maven web application works exactly like any typical Maven web application; it is compiled and tested, then artifacts are generated and made ready for deployment on an application server.

Seam Framework artifacts are of course dependencies that are required during the compile and test phases of the project lifecycle.

6
Google Development with Maven

In this chapter, we will cover:

- Setting up the Android development environment
- Developing an Android application
- Testing and debugging an Android application
- Developing a Google Web Toolkit application
- Testing and debugging a Google Web Toolkit application
- Developing a Google App Engine application

Google has been a technology company like no other, and in less than a decade it has established itself as a major frontrunner in the Java technology space as well with commercially successful, technically path breaking and innovatory platforms, concepts, and framework.

In this chapter, we look at three of the most popular Google offerings in the Java world today:

- The Android platform
- Google Web Toolkit
- Google App Engine

The Android platform is already a dominant player in a highly competitive mobile market where Google, for a change, did not enjoy the first mover advantage. However, sound engineering practices and honest, principle driven policies have quietly yet quickly placed Google's Android platform at an enviable position; wherein a vast number of device makers and service providers have adopted and pushed the platform to an audience which was craving for openness and access on their mobiles.

The **Google Web Toolkit** (**GWT**) solves a very different type of problem. GWT provides a set of APIs and tools which let you create rich web applications, without having to deal with the cumbersome and often unsuccessful process of cross-browser compatibility in a very fragmented browser market.

The **Google App Engine** (**GAE**) offering, unlike Android, did enjoy the first mover advantage. Bundled with the Google Apps offering, it was the first introduction of cloud computing to small and large enterprises alike. Google App Engine lets you use the cloud APIs on offer, develop hosted applications, and deploy them on the **Google cloud**. The Google cloud in turn offers unmatchable performance, risk free hosting, and unlimited scalability.

This chapter gets you up and running, leveraging Maven to help your development team make the most of these offerings from Google.

Setting up the Android development environment

The Android platform is one of the fastest, if not the fastest, growing mobile and device platforms today. It has been strongly promoted by Google since 2008, along with a host of industry partners and heavy weights, and is quickly replacing alternatives as the dominant mobile platform today.

Android's primary strengths lie in its open platform, architecture, and underlying technologies. For developers, it is a charm to work with as it's based on the very mature Java platform. Hence tools, IDEs, and development methodologies are already available and the relative learning curve is tiny compared to other proprietary platforms.

The prerequisites for Android development on Maven primarily include downloading and setting up the **Android Software Development Kit** (**Android SDK**) and Android artifacts in the Maven repository (local or remote).

Getting ready

At this moment, we assume that you are comfortable developing with Maven and understand the basics of the Maven development paradigm. If not, a good place to start is with *Chapter 1, Basics of Apache Maven* of this book.

Our first step is to get a hold of the Android SDK. Google maintains an impressive Android developer portal at `http://developer.android.com` which contains all the relevant downloads.

The specific URL for the Android SDK download is: `http://developer.android.com/sdk/`.

How to do it...

Follow the subsequent steps to install and set up the Android SDK:

1. Extract the downloaded Android SDK into any folder.
2. Define the `ANDROID_HOME` environment variable. The value of `ANDROID_HOME` should be the root directory of the Android SDK. The following commands can be used to define `ANDROID_HOME`:

   ```
   ### Linux ### export ANDROID_HOME=/opt/android-sdk
   ### Windows ### set ANDROID_HOME=C:\android-sdk
   ```

3. An optional step here is to install the `ANDROID_HOME/tools` to the `PATH` environment variable. (Check the *Maven Installation* section in *Chapter 1*, *Basics of Apache Maven* for instructions on modifying the `PATH` variable)

   The following steps are now optional as Android SDK artifacts have been made available in the Maven Central Repository.

4. The installation and setting up of the Android SDK is followed by the installation of the **Android API JARS** (or **Android artifacts**) in the Maven repository.
5. The Maven Android SDK deployer tool allows you to deploy in a local repository as well as in a remote repository server. The Maven Android SDK deployer tool is available at: `http://github.com/mosabua/maven-android-sdk-deployer`.

   The source needs to be downloaded from the GitHub repository of the Maven Android SDK deployer and needs to replace the Android SDK. The Android API JARS (artifacts) can now be installed in local and remote repositories. We will be running Maven build phase commands to do the same.
6. Installation into a Maven local repository is pretty simple. Open the command line and navigate to the Maven Android SDK deployer root folder and execute the following command:

   ```
   $ mvn clean install
   ```

7. To install these on a remote repository, make sure the remote repository is defined in the Maven `settings.xml` file. Additionally, the server credentials need to be added as well:

   ```
   <settings>
     <servers>
           <server>
               <id>android.repo</id>
               <username>your username</username>
               <password>your password</password>
   ```

```
        </server>
    </servers>
</settings>
```

Once everything is set up, the following command will install the artifacts in the remote repository:

```
$ mvn deploy
```

8. To use the Maven Android plugin goals on the command line with the short plugin name "android", `pluginGroups` need to be defined in the `settings.xml` file:

```
<pluginGroups>
  <pluginGroup>
    com.jayway.maven.plugins.android.generation2
  </pluginGroup>
</pluginGroups>
```

How it works...

In the preceding steps, we set up the Android SDK on your operating system. This means the SDK is available for use directly or through external applications. It is also available for use for Apache Maven projects and plugins.

Then we followed it up by deploying the Android SDK and API artifacts (JAR/library files) in our repository. By deploying it to the repository, any new Maven module that has a dependency on Android will be able to reference the artifacts from the repository.

This step is now optional as Android artifacts are directly available from the Maven Central Repository. Deployment information has still been preserved in this recipe for reference.

See also...

- *Chapter 1*, *Basics of Maven*

Developing an Android application

In the preceding recipe, we covered the basics of setting up the Maven Android development environment. Now that we're ready, we'll proceed to the actual development of an Android application with Maven.

Getting ready

If you haven't set up the Maven Android development environment, you need to do it before you can continue with this recipe. Refer to the previous recipe for guidance.

How to do it...

You can set up your Maven Android application project by either directly configuring the project `pom.xml` file or by running the Maven `archetype:generate` command. Steps for both the methods are as follows:

The first step is to create a Maven project with the POM file, as shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>0.1-SNAPSHOT</version>
  <packaging>apk</packaging>
  <name>...</name>
  <dependencies>
    <dependency>
      <groupId>com.google.android</groupId>
      <artifactId>android</artifactId>
      <version>2.1_r1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <groupId>
          com.jayway.maven.plugins.android.generation2
        </groupId>
        <artifactId>maven-android-plugin</artifactId>
          <version>2.2.3-SNAPSHOT</version>
          <configuration>
          <sdk>
```

```
            <platform>2.1</platform>
          </sdk>
            <deleteConflictingFiles>
              true
            </deleteConflictingFiles>
            </configuration>
            <extensions>true</extensions>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <source>1.5</source>
              <target>1.5</target>
            </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

The notable highlights from the POM are:

- Packaging has been set to `apk`
- Dependencies for Android artifacts have been defined with the scope `provided`
- Build plugins have been configured for integration with the Maven build cycle

1. To create the project from the Maven archetype, execute the following command in the terminal:

```
$ mvn archetype:generate -DarchetypeCatalog=http://kallisti.eoti.
org:8081/content/repositories/snapshots/archetype-catalog.xml


[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
-------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------
-------------
[INFO]
[INFO] --- maven-archetype-plugin:2.0-alpha-5-SNAPSHOT:generate
(default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
```

```
...
Define value for property 'groupId': : org.eoti.android.test
Define value for property 'artifactId': : TestAndroid
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.eoti.android.test: :
Confirm properties configuration:
groupId: org.eoti.android.test
artifactId: TestAndroid
version: 1.0-SNAPSHOT (hit enter)
package: org.eoti.android.test (hit enter)
 Y: : (hit enter)
[WARNING] Don't override file F:\work\TestAndroid\src\main\
android\res\values\strings.xml
[WARNING] Don't override file F:\work\TestAndroid\src\main\
android\res\layout\main.xml
[INFO] ------------------------------------------------------------
-------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
-------------
[INFO] Total time: 1:02.860s
[INFO] Finished at: Sun Apr 18 19:58:56 PDT 2010
[INFO] Final Memory: 6M/11M
[INFO] ------------------------------------------------------------
-------------
```

Once the archetype creation command is successfully executed, you should see the following project structure generated:

```
C:.
|  pom.xml
|
└──src
    |  AndroidManifest.xml
    |
    └──main
        ├──android
        |  └──res
        |      ├──drawable
        |      |      icon.png
        |      |
```

```
|       ├──layout
|       |      main.xml
|       |
|       └──values
|              strings.xml
|
└──java
    └──org
        └──eoti
            └──android
                └──test
                        MyActivity.java
```

This can be worked with as any other Maven project.

2. In order to run it on the emulator, the following command can be executed after initializing the emulator or USB test device:

```
# Package & Deploy
mvn install
```

3. To uninstall the application from the emulator, run the Maven clean command.

```
# Uninstall from Emulator
mvn clean
```

How it works...

As we just saw, a Maven Android project can simply be generated by an `archetype:generate` command. This not only greatly simplifies things, but also makes one more productive.

This archetype is provided by the **Maven Android Plugin Project**: `http://code.google.com/p/maven-android-plugin/`.

Once set up, a Maven Android project is quite similar to a generic Maven Java project, and all the build phases and plugins are available and can be configured in a way no different to a Maven Java project.

Testing and debugging an Android application

Now that you've got your Android development environment fully set up, and your first Android application up and running, we now proceed to the details of setting up, testing, and debugging practices for a development team.

The first two recipes of this chapter are a prerequisite for this recipe and it is recommend that you go through them if you haven't already.

Luckily for us, integration of **Android App Development** with Maven works like a charm as an already mature Android development platform leverages the best of what the Maven environment has to offer in terms of Test Driven Development and other best practices.

In this recipe, we will not only explore unit testing, but we will delve into integration and instrumentation tests for your Android application as well.

Getting ready

Unit testing and Test Driven Development has been covered in elaborate detail in previous chapters. See *Chapter 2*, *Software Engineering Techniques* for more on Test Driven Development (TDD). Continuing any further requires understanding of the core practice of TDD.

How to do it...

For unit testing with the Surefire plugin, please follow the ensuing steps:

1. The `testSourceDirectory` element needs to be configured in the build of the project's POM file.

```
<build>
  <sourceDirectory>src/main/java</sourceDirectory>
  <testSourceDirectory>src/test/java</testSourceDirectory>
  ...
</build>
```

2. Surefire executes these tests as part of the "test" phase of the Maven build cycle. The following command can be executed to run these tests:

```
$ mvn test
```

   Instrumentation tests can be set up in a modular format with one module for the application while another for the tests and the two modules tied together with a parent POM.

```
<dependency>
  <groupId>net.srirangan.packt.maven.testandroidapp</groupId>
  <artifactId>intents</artifactId>
  <version>0.1</version>
  <type>apk</type>
</dependency>
```

How it works...

As shown in the code of the previous section, the `sourceDirectory` has been modified to the Maven standard `src/main/java` and the `testSourceDirectory` has been added with the value `src/test/java`.

This is in alignment with Maven standards for any Java project. Here, the main application source code lies in the `sourceDirectory`, that is, `src/main/java` while the supporting test suites lie in the `testSourceDirectory`, which is `<project root>/src/test/java`.

Instrumentation tests are integration tests that run on the emulator or device and can interact with another deployed application. These are bundled into the application.

The setup of the instrumentation tests with the Maven Android plugin is similar to setting up any normal application. The instrumentation tests are a separate application with an added dependency to the application, which needs to be tested. Adding the type of `apk` to the dependency allows the Maven Android plugin to find the Android package of the application.

See also

- *Project modularization* section in *Chapter 2*, *Software Engineering Techniques*

Developing a Google Web Toolkit application

According to `http://code.google.com/webtoolkit/`:

> *"Google Web Toolkit (GWT) is a development toolkit for building and optimizing complex browser-based applications. Its goal is to enable productive development of high-performance web applications without the developer having to be an expert in browser quirks, XMLHttpRequest, and JavaScript. GWT is used by many products at Google, including Google Wave and the new version of AdWords. It's open source, completely free, and used by thousands of developers around the world."*

This is how Google describes Google Web Toolkit (GWT), and it would be a futile effort to try to define it in a different way. GWT lets you create high-end, advanced, cross-browser applications while abstracting the underlying client-side browser technologies, giving you a set of Java APIs and widgets to interact with.

Getting ready

The Google Web Toolkit framework is itself beyond the scope of this recipe. A good place to get started with GWT is `http://code.google.com/webtoolkit/overview.html`.

This recipe focuses on GWT developers who wish to leverage Maven in their development process. It also assumes basic knowledge of Maven itself. Please make sure you are familiar with the concepts in *Chapter 1*, *Chapter 2*, and *Chapter 3* of this book.

How to do it...

1. Let us start by generating a Google Web Toolkit (GWT) project by running the following archetype command in the terminal:

   ```
   mvn archetype:generate -DarchetypeRepository=repo1.maven.org
   -DarchetypeGroupId=org.codehaus.mojo -DarchetypeArtifactId=gwt-
   maven-plugin -DarchetypeVersion=2.1.0-1
   ```

   This creates a new GWT project with the following tree structure:

   ```
   C:.
   └───TestGwtApp
       ├───.settings
       └───src
           ├───main
           │   ├───java
           │   │   └───net
           │   │       └───srirangan
           │   │           └───packt
           │   │               └───maven
           │   │                   └───gwt
           │   │                       ├───client
           │   │                       ├───server
           │   │                       └───shared
           │   ├───resources
           │   │   └───net
           │   │       └───srirangan
           │   │           └───packt
           │   │               └───maven
           │   │                   └───gwt
           │   │                       └───client
           │   └───webapp
           │       └───WEB-INF
           └───test
               ├───java
               │   └───net
               │       └───srirangan
               │           └───packt
               │               └───maven
               │                   └───gwt
               │                       └───client
   ```

```
└──resources
        └──net
            └──srirangan
                └──packt
                    └──maven
                        └──gwt
```

2. The project can be compiled with the following command:

```
$ mvn compile
```

Additional Compilation Options:

$ mvn compile -Dgwt.logLevel=[LOGLEVEL]
LOGLEVEL can be ERROR, WARN, INFO, TRACE, DEBUG, SPAM, or ALL

$ mvn compile -Dgwt.style=[PRETTY|DETAILED]

How it works...

An initial look at the structures reveals that it is similar to any typical Maven Java project. We have the `src/main/java` and `src/test/java` folders for source and test packages respectively, and a sub folder for resources in both test and source directories.

In addition, the `sourceDirectory` contains the `webapp` folder similar to a Maven web application.

See the *Building a Web Application* recipe in *Chapter 5*, *Java Development with Maven*.

Open the GWT project POM file that was generated in the preceding step.

```
<modelVersion>4.0.0</modelVersion>
<groupId>net.srirangan.packt.maven.gwt</groupId>
<artifactId>TestGwtApp</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
<name>GWT Maven Archetype</name>
```

Note that the packaging has been set to `war`. This is done for any Maven web application as well as any GWT application.

However, as displayed in the following code, the GWT application's POM file contains additional dependencies to GWT artifacts:

```
<dependencies>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-servlet</artifactId>
    <version>2.1.0-1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <version>2.1.0-1</version>
    <scope>provided</scope>
  </dependency>
<dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <version>2.1.0-1</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>generateAsync</goal>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The build plugin configuration shown in the preceding code now synchronizes the Maven GWT project with the Maven build lifecycle.

There's more...

The Maven-GWT plugin provides the following goals:

Goal	Report?	Description
gwt:clean	No	Clean up the webapp directory for GWT module compilation output.
gwt:compile	No	It invokes the GWT compiler for the project source. See compiler options.
gwt:css	No	It creates a CSS interfaces for `.css` files. It uses the utility tool provided in `gwt sdk`, which creates a corresponding Java interface for accessing the class names used in the file.
gwt:debug	No	It extends the `gwt` goal and runs the project in the GWT hosted mode with a debugger port hook (optionally suspended).
gwt:eclipse	No	A goal which creates Eclipse launch configurations for GWT modules.
gwt:generateAsync	No	It generates an Asyn interface.
gwt:i18n	No	It creates I18N interfaces for constants and messages files.
gwt:mergewebxml	No	It merges GWT servlet elements into deployment descriptor (and non GWT servlets into shell).
gwt:resources	No	It copies GWT Java source code and module descriptor as resources in the build output directory. It is an alternative to declaring a `<resource>` in the POM with finer filtering as the module descriptor is read to detect sources to be copied.
gwt:run	No	It is a goal which runs a GWT module in the GWT hosted mode.
gwt:sdkInstall	No	Install a GWT (home built) SDK in local repository.
gwt:source-jar	No	Add GWT Java source code and module descriptor as resources to the project `jar`. It is an alternative to `gwt:resources` for better Eclipse project synchronization.
gwt:test	No	Mimic Surefire to run `GWTTestCases` during integration test phase, until `SUREFIRE-508` is fixed.

Additional details, updates, and documentation for this plugin are available at `http://mojo.codehaus.org/gwt-maven-plugin/`.

Testing and debugging a Google Web Toolkit application

If you haven't set up and created your Maven-Google Web Toolkit (GWT) application, stop following this recipe and move back to the preceding recipe. Here's where we deal with setting up a Test Driven Development based workspace for our Maven-GWT projects.

Getting ready

As mentioned before, the preceding recipe is a prerequisite for everything we're going to explore here. Also, you need to be familiar with Test Driven Development (TDD) covered in the *Test Driven Development* recipe in *Chapter 2*, *Software Engineering Techniques*.

Unit testing of the GWT application would consist of unit testing of the custom libraries and custom APIs that are being used. This is the same as unit testing for any Java project using the Surefire plugin. This part has already been covered in *Chapter 2*, *Software Engineering Techniques.*

Integration testing for GWT projects is where things get interesting. The GWT-Maven plugin testing support is bound to the Maven integration test phase of the build lifecycle. It is not meant for standalone execution.

How to do it...

1. We begin with a configuration to enable integration goal execution. To get `gwt:test` to run, you should include the test goal in your plugin configuration executions and you should invoke mvn integration test (or mvn install).

   Add the GWT-Maven-Plugin to the build plugins, as shown in the following code:

```
<project>
[...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>2.1.0-1</version>
        <executions>
          <execution>
            <goals>
              <goal>test</goal>
```

```
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

This integrates the plugin test goal with the integration test phase of Maven.

2. Selenium mode can be used for running your test suite on real browsers using Selenium RC. The test mode parameter must be set to `selenium`, that is, `Dgwt.test.mode=selenium` on the command line. The Selenium parameter needs to be configured with the host running the Selenium RC browser, as shown in the following code:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven-plugin</artifactId>
  <version>2.1.0-1</version>
  <configuration>
    <selenium>myhost:4444/*firefox"</selenium>
  </configuration>
</plugin>
```

3. Google Web Toolkit is an alternative to Selenium. It comes with a custom remote browser server. This can be launched using the `gwt:browser` goal, which will trigger the launch of the default operating system browser:

```
$ mvn gwt:browser
```

Testsuite can be run on a remote browser by setting the mode parameter to `remoteweb`. The `remoteweb` parameter has to be configured to declare your remote browser server, as shown:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven-plugin</artifactId>
  <version>2.1.0-1</version>
  <configuration>
    <remoteweb>rmi://myhost/ie</remoteweb>
  </configuration>
</plugin>
```

How it works...

As discussed earlier, your project's `testSourceDirectory` will contain typical unit tests meant to be run by Surefire, and in addition, will contain GWT-specific integration tests. It is a recommended best practice to follow a differing naming convention to keep the two tests separate.

Here our integration test class files are post-fixed with `GwtTest.java`, which allows us to set up a rule to exclude them from the Surefire unit test runner:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <excludes>
      <exclude>**/*GwtTest.java</exclude>
    </excludes>
  </configuration>
</plugin>
```

By default, the `gwt-maven` plugin uses `GwtTest*.java` as an inclusion pattern so that such tests will not match the standard Surefire pattern. While using this convention, you don't have to change your configuration.

GWT integration tests run on an inbuilt browser, which unfortunately does not purge the need for testing on a "real" browser.

This is where we leverage the Selenium testing framework for real world acceptance testing.

See also

- *Test Driven Development* section in *Chapter 2*, *Software Engineering Techniques*
- *Acceptance testing automation* section in *Chapter 2*, *Software Engineering Techniques*

Developing a Google App Engine application

According to `http://code.google.com/appengine/`:

> *"Google App Engine enables you to build web applications on the same scalable systems that power Google applications. App Engine applications are easy to build, easy to maintain, and easy to scale as your traffic and data storage needs grow. With App Engine, there are no servers to maintain: You just upload your application, and it's ready to serve to your users."*

Ever since Google offered the **Google App Engine** (**GAE**) platform, it has revolutionized the industry giving birth to a game changing concept–Cloud Computing.

Google App Engine (as we know it today) lets you develop server hosted web applications in Python and Java based on the Google App Engine APIs on offer. These applications can be deployed onto Google data centers giving you extremely efficient, well-run, and scalable web applications.

Getting ready

Google App Engine as a development and cloud computing platform is beyond the scope of this recipe (and book!). Here, we assume that you are knowledgeable about the GAE platform and have a fair idea of application development. This recipe helps you leverage Maven for your GAE application development.

Visit `http://code.google.com/appengine/` for more information on Google App Engine.

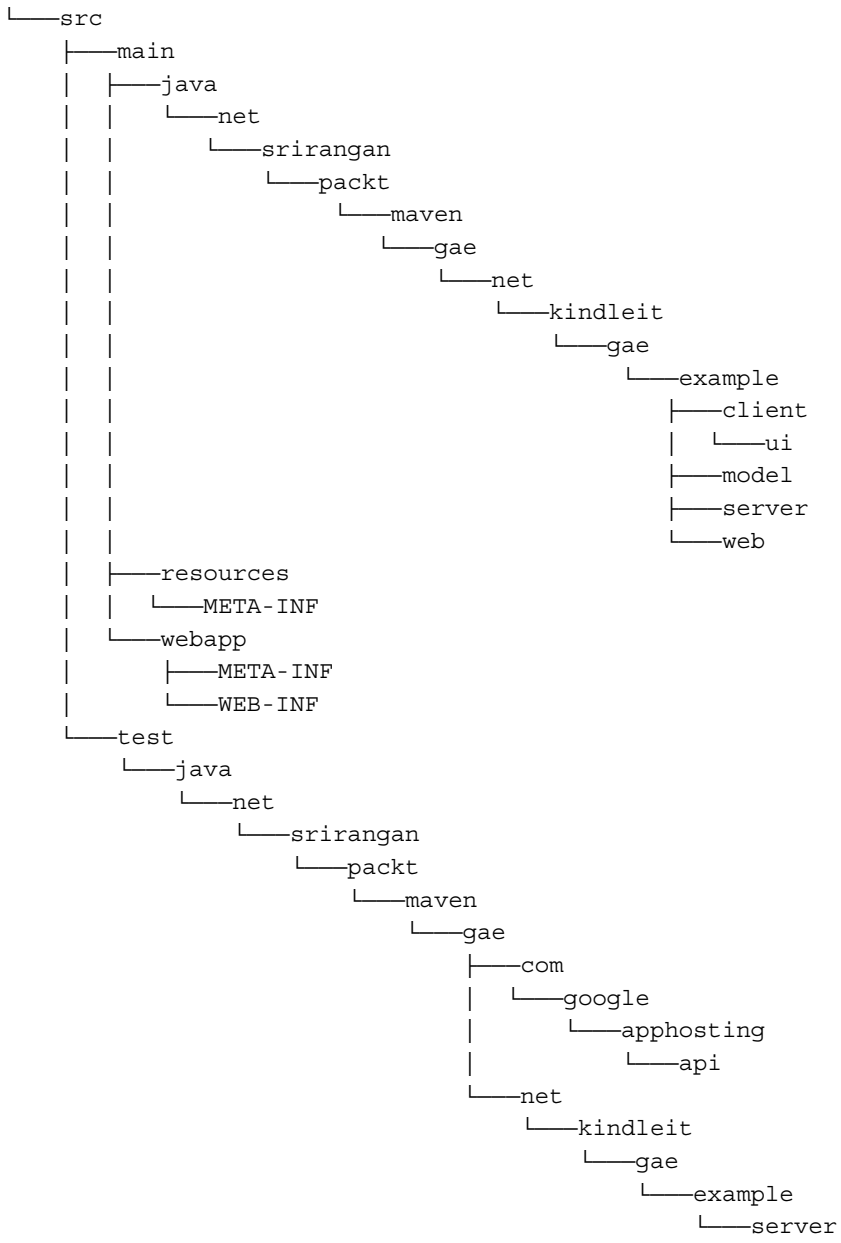
This recipe also assumes basic knowledge of Maven itself. Please make sure you are familiar with the concepts in *Chapter 1*, *Chapter 2*, and *Chapter 3* of this book.

How to do it...

Let's get started by creating a Maven-Google App Engine (GAE) project using an archetype command. Execute the following in the command line terminal:

```
mvn archetype:create -DarchetypeGroupId=net.kindleit
-DarchetypeArtifactId=gae-archetype-gwt -DgroupId=net.srirangan.packt.
maven.gae -DartifactId=TestGaeApp
```

This command creates a new Google App Engine project for you in a folder named after the `artifactId` provided in the command, `TestGaeApp`.



You can see a typical Maven web application structure with `sourceDirectory` (`src/main/java`) and `testSourceDirectory` (`src/test/java`) being generated along with the package structure.

An existing Maven web application project can also be converted to a Google App Engine project by making the following modifications to the existing project POM file:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>net.kindleit</groupId>
        <artifactId>maven-gae-plugin</artifactId>
        <version>[plugin version]</version>
          <dependencies>
            <dependency>
              <groupId>net.kindleit</groupId>
              <artifactId>gae-runtime</artifactId>
              <version>${gae.version}</version>
              <type>pom</type>
            </dependency>
          </dependencies>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Use the following command to run the GAE application on the local server:

```
$ mvn gae:run
```

Use the following command to deploy the GAE application on the Google appspot cloud:

```
$ mvn gae:deploy
```

How it works...

Inspecting the POM file of the generated project reveals that:

- Packaging type remains WAR, similar to Maven web and GWT apps
- Dependencies, that is, Google App Engine artifacts have been included

- Build plugin integration has been set up for synchronization with the Maven build lifecycle
- Profiles have been generated for integration-build, release-build, and release

Apart from the two main goals described in the preceding commands, the Maven Google App Engine plugin also provides the following:

Goal	Description
gae:cron-info	Displays times for the next several runs of each `cron` job from Google's servers.
gae:debug	*Extends the* `run` goal for running the project with a debugger port hook (optionally suspended). It is a simple utility method, as the `run` goal supports the passing of `jvm` options in the command line.
gae:deploy	Uploads a WAR project onto Google's servers.
gae:enhance	Enhances classes.
gae:logs	Retrieves logs from Google's servers.
gae:rollback	Rolls back an update on Google's servers.
gae:run	Runs the WAR project locally on the Google App Engine development server. You can specify `jvm` flags via the `jvmFlags` in the configuration section.
gae:start	Runs the WAR project locally on the Google App Engine development server, without executing the package phase first. This is intended to be included in your project's POM and runs in the pre-integration test phase.
gae:stop	Stops a running instance of the Google App Engine development server. This is intended to be included in your project's POM and runs in the post-integration test phase.
gae:unpack	Downloads and unzips the SDK to your Maven repository. Use this goal, if you don't wish to specify a `gae.home` or `Dappengine.sdk.home` property. The plugin will now search for the SDK in that default location.
gae:update-indexes	Updates data store indexes.

Check out `http://www.kindleit.net/maven_gae_plugin/` for more information, updates, and documentation of this plugin.

There's more...

There are additional archetypes provided to generate your GAE project in an integrated state with popular Java Web Frameworks and technologies. The following commands can be used for the same:

```
### JSP
mvn archetype:generate -DarchetypeGroupId=net.kindleit
-DarchetypeArtifactId=gae-archetype-jsp -DgroupId=... -DartifactId=...
-DarchetypeRepository=http://maven-gae-plugin.googlecode.com/svn/
repository


## Objectify JSP
mvn archetype:generate -DarchetypeGroupId=net.kindleit
-DarchetypeArtifactId=gae-archetype-objectify-jsp -DgroupId=...
-DartifactId=... -DarchetypeRepository=http://maven-gae-plugin.
googlecode.com/svn/repository


### Wicket
mvn archetype:generate -DarchetypeGroupId=net.kindleit
-DarchetypeArtifactId=gae-archetype-wicket -DgroupId=... -DartifactId=...
-DarchetypeRepository=http://maven-gae-plugin.googlecode.com/svn/
repository


### JSF based example
mvn archetype:generate -DarchetypeGroupId=net.kindleit
-DarchetypeArtifactId=gae-archetype-jsf  -DgroupId=... -DartifactId=...
-DarchetypeRepository=http://maven-gae-plugin.googlecode.com/svn/
repository


### GWT based example
mvn archetype:generate -DarchetypeGroupId=net.kindleit
-DarchetypeArtifactId=gae-archetype-gwt  -DgroupId=... -DartifactId=...
-DarchetypeRepository=http://maven-gae-plugin.googlecode.com/svn/
repository
```

7
Scala, Groovy, and Flex

In this chapter, we will cover:

- Integrating Scala development with Maven
- Integrating Groovy development with Maven
- Integrating Flex development with Maven

Java has been the dominant enterprise programming platform for over a decade and a half. In many ways, it has shaped the way the industry does business, especially with the advent of the Internet and followed by cloud computing. It is very evolved and mature, and has good infrastructure support.

For all its successes, however, it has some serious problems competing with modern dynamic programming languages in the context of ease of programming, especially while getting started in the initial stages of a project.

An advantage that Java traditionally enjoyed over the likes of Python, Ruby, and so on is that a lot of enterprises were committed to **Java Virtual Machine** (**JVM**) as their platform due to its inherent advantages. This always worked in favor of the Java programming language.

However, all this has changed with modern languages such as Scala, Groovy, and so on supporting JVM bytecode, which made them compatible with existing enterprise infrastructure and assets. Furthermore, **RIA** (**Rich Internet Applications**) technology, such as Flex, never faced the JVM challenge in the first place due to the widespread adoption of the Flash Player.

Apache Maven's flexible plugin-based architecture allows the tools to evolve with time and lends its benefits to developers and development teams that are keen to leverage this new breed of modern programming languages.

Integrating Scala development with Maven

Scala stands for "Scalable Language". It is defined as a multi-paradigm programming language and has integrated support for object-oriented programming and functional programming.

It runs on the JVM and on Android. Furthermore, it can read existing Java libraries which give it a huge advantage in cases where there is a lot of existing code infrastructure in Java.

The Scala bytecode is in many ways, identical to the Java bytecode. In many cases, Scala generates more optimal byte-code than Java.

Twitter, Foursquare, and a whole bunch of exciting software startups have adopted Scala. Scala helped Twitter get over its scalability issues when the micro blogging pioneer first hit the mainstream and needed to scale-up as it served millions of new users each day.

This is, of course, not to say that scalable applications can't be built in pure Java. However, the Scala programming language features itself (such as pattern matching, concurrency, and Actors) and frameworks built on top of Scala (for example, Akka, GridGain, and so on.) allow application scalability with ease.

Below is an example of a popular sorting algorithm implemented in Scala:

```
def qsort: List[Int] => List[Int] = {
  case Nil => Nil
  case pivot :: tail =>
  val (smaller, rest) = tail.partition(_ < pivot)
  qsort(smaller) ::: pivot :: qsort(rest)
}
```

The creator of Scala, Martin Odersky, has given a number of popular talks and presentations examining Scala's comparability. When Scala code is up against comparable code in other programming languages including Java, C#, Ruby, and so on, the conciseness of Scala really stands out.

The Scala implementation of quicksort in the preceding code demonstrates this very conciseness of Scala code.

While the Scala community recommends **SBT** (**Simple Built Tool**) for pure Scala projects, often you'll be implementing modules in Scala while other modules of the project would have been built with Java. This is one of the strongest cases for using Maven in a multi-modular project with Java and Scala-based modules. It is probably the reason why SBT is more or less compatible with Maven's conventions and even provides a feature for "Mavenizing" an existing SBT project.

Getting ready

You'll need Apache Maven installed. The Maven version preferably should be Maven 3 but Maven 2.0.7 or higher is supported. JDK 1.5 or higher is recommended. Do note, installation of Scala is not required. Apache Maven will take care of the Scala dependency.

How to do it...

Generate the Scala project archetype:

```
$ mvn archetype:generate
```

This will show a list of available archetypes in which you need to select the archetype for a simple Scala project, which is `scala-archetype-simple`.

Maven interactive mode will also ask you for archetype `version`, `groupId`, `artifactId`, and other Maven project co-ordinates. This is the same step we've encountered in previous recipes. You can choose reasonable values for `groupId`, `artifactId`, and `package`. For me, the `groupId` was `net.srirangan.packt.maven`, `artifactId` was `scalaexample`, `version` was the `default value`, and `package` was the same as `groupId`.

On completion, your terminal will look similar to the following lines:

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 1:38.142s
[INFO] Final Memory: 10M/114M
[INFO] ------------------------------------------------------------
```

The completion of this operation has created a project folder with the same name as the project artifact ID. The project directory tree structure would look similar to this:

```
Folder PATH listing for volume OS
Volume serial number is 7C69-DF5D
C:.
└───src
    ├───main
    │   └───scala
    │       └───net
    │           └───srirangan
    │               └───packt
    │                   └───maven
    │                       └───App.scala
    └───test
```

```
            └──scala
                └──samples
                    └──junit.scala
                    └──scalatest.scala
                    └──specs.scala
```

Also generated is a simple `Hello World!` Scala application–`App.scala`:

```
package net.srirangan.packt.maven
object App {
  def foo(x : Array[String]) = x.foldLeft("")((a,b) => a + b)
  def main(args : Array[String]) {
    println( "Hello World!" )
    println("concat arguments = " + foo(args))
  }
}
```

The `pom.xml` file is configured in a way that integrated the build lifecycle with the Maven Scala plugin:

```
<build>
    <plugins>
        <plugin>
          <groupId>org.scala-tools</groupId>
          <artifactId>maven-scala-plugin</artifactId>
            <executions>
              <execution>
                <goals>
                  <goal>compile</goal>
                  <goal>testCompile</goal>
                </goals>
              </execution>
            </executions>
            ...
        </plugin>
    </plugins>
</build>
```

To compile the project, run:

```
$ mvn compile
```

To run the tests, use the default project lifecycle command:

```
$ mvn test
```

How it works...

If you inspect the Scala project `pom.xml` file that was generated, you would see dependencies defined for Scala: `library`, `testing`, `specs`, and `scalatest`.

```
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.9.0-1
    </version>
  </dependency>
  <dependency>
    <groupId>org.scala-tools.testing</groupId>
    <artifactId>specs_2.9.0-1</artifactId>
    <version>1.6.5</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.scalatest</groupId>
    <artifactId>scalatest</artifactId>
    <version>1.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

The first dependency listed here is the Scala programming language itself, that is, `org.scala-lang.scala-library`. The other two dependencies are for running Scala tests. There may also be a fourth dependency which is not listed in the preceding code for `junit`. It is unrelated to Scala, but has been explained and looked at in detail earlier in *Chapter 2*, *Software Engineering Techniques*.

Looking at the build settings and plugin configuration hereafter, we see that the source and test directories are being set to `src/main/scala` and `src/test/scala` respectively. The plugin execution is being bound to the `compile` and `testCompile` build phases.

```
<build>
    <sourceDirectory>src/main/scala</sourceDirectory>
    <testSourceDirectory>src/test/scala</testSourceDirectory>
    <plugins>
      <plugin>
        <groupId>org.scala-tools</groupId>
        <artifactId>maven-scala-plugin</artifactId>
        <version>2.15.0</version>
        <executions>
```

```
            <execution>
              <goals>
                <goal>compile</goal>
                <goal>testCompile</goal>
              </goals>
              <configuration>
                <args>
                  <arg>-make:transitive</arg>
                  <arg>-dependencyfile</arg>
                  <arg>${project.build.directory}/.scala_dependencies</
arg>
                </args>
              </configuration>
            </execution>
          </executions>
        </plugin>
        ...
      </plugins>
    </build>
```

There's more...

In addition to what was described up to now, the Maven Scala plugin has more features which meet the needs of most developers and development situations.

The following table lists all goals made available by the Maven Scala plugin, their respective Apache Maven console commands, and brief explanations:

Goal	Maven command	Details
scala:compile	`$ mvn scala:compile`	Compiles the Scala source directory
scala:console	`$ mvn scala:console`	Launches the Scala REPL with all project dependencies made available
scala:doc	`$ mvn scala:doc`	Generates the documentation for all Scala sources
scala:help	`$ mvn scala:help`	Displays the Scala compiler help
scala:run	`$ mvn scala:run`	Executes a Scala class
scala:script	`$ mvn scala:script`	Executes a Scala script
scala:testCompile	`$mvn scala:testCompile`	Compiles the Scala test sources

Integrating Groovy development with Maven

Groovy, which is an object-oriented programming language for the Java Virtual Machine (JVM), provides dynamic programming features similar to Python, Ruby, Smalltalk, and so on. It is often used as a scripting language and interacts freely with existing Java code and libraries.

Groovy was created by James Strachan and Bob McWhirter and is currently being lead by Guillaume Laforge. It is released under the Apache License v2.0.

Groovy is defined as an agile and dynamic programming/scripting language built for the Java Virtual Machine. While it is inspired by Python, Ruby, and Smalltalk; for a Java developer it promises an "almost-zero" learning curve.

In the past few years, Groovy has become popular for scripting and **DSLs** (**Domain Specific Languages**) in addition to its use in application development. In an application development environment, Groovy enhances developer productivity by reducing scaffolding code while providing built-in capability for unit and mock testing.

Groovy is based on the JVM and thus interoperates with existing Java-based infrastructure, libraries, and frameworks.

You can find more information on the official Groovy website `http://groovy.codehaus.org/`.

Getting ready

Apache Maven needs to be installed and set up along with JDK. Apache Maven 3+ and JDK 1.6+ are recommended. The Groovy setup is not required as the dependency is taken care of by Maven.

How to do it...

Generate the Groovy project archetype:

```
$ mvn archetype:generate
```

This will show a list of available archetypes in which you need to select the archetype for a basic Groovy project, which is `gmaven-archetype-basic`. Maven interactive mode will also ask you for the archetype `version`, `groupId`, `artifactId`, and other Maven project co-ordinates.

Alternatively, you can try executing:

```
$ mvn archetype:generate -DarchetypeGroupId=org.codehaus.groovy.
maven.archetypes -DarchetypeArtifactId=gmaven-archetype-basic
-DarchetypeVersion=<VERSION>
```

On completion, your terminal will look similar:

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 1:38.142s
[INFO] Final Memory: 10M/114M
[INFO] ------------------------------------------------------------
```

The completion of this operation has created a project folder with the same name as the project artifact ID. The project directory tree structure would look similar to this:

```
C:.
└───pom.xml
└───src
    ├───main
    │   └───groovy
    │       └───net
    │           └───srirangan
    │               └───packt
    │                   └───maven
    │                       └───Example.groovy
    │                       └───Helper.java
    └───test
        └───groovy
            └───net
                └───srirangan
                    └───packt
                        └───maven
                            └───ExampleTest.groovy
                            └───HelperTest.groovy
```

What has been generated is a simple "Hello world!" Groovy application–`Example.groovy`:

```
package net.srirangan.packt.maven
/**
 * Example Groovy class.
 */
class Example
{
```

```
  def show() {
    println 'Hello World'
  }
}
```

The `pom.xml` file is configured in a way that integrated the build lifecycle with the Maven Groovy plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.groovy.maven</groupId>
      <artifactId>gmaven-plugin</artifactId>
      <version>1.0</version>
        <executions>
          <execution>
            <goals>
              <goal>generateStubs</goal>
              <goal>compile</goal>
              <goal>generateTestStubs</goal>
              <goal>testCompile</goal>
            </goals>
                    </execution>
        </executions>
    </plugin>
  </plugins>
</build>
```

To compile the project, run:

```
$ mvn compile
```

To run the tests, use the default project lifecycle command:

```
$ mvn test
```

How it works...

If you inspect the POM file (`pom.xml`) of the Groovy project that was generated, you will find declared dependencies, as shown in the following code:

```
<dependency>
  <groupId>org.codehaus.groovy.maven.runtime</groupId>
  <artifactId>gmaven-runtime-1.6</artifactId>
  <version>1.0</version>
</dependency>
```

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

The first dependency listed here is the GMaven runtime, that is, `gmaven-runtime-1.6`. The other dependency listed in the preceding code is `junit` which is unrelated to Groovy, but has been explained and looked at in detail earlier in *Chapter 2, Software Engineering Techniques*. It is used for running unit tests.

With the help of the GMaven plugin, these dependencies are integrated with the Maven build lifecycle and the build workflow for the developer is achieved through Apache Maven.

One of the advantages of Apache Maven is on display here, as despite the various differences in underlying technologies, libraries, and toolkits, the Maven build lifecycle, the interface for the developer, and external tools remains consistent.

There's more...

Groovy Shell provides the developer with a very convenient command-line shell to execute Groovy commands. The GMaven plugin provides for command-line access to the Groovy shell with the following command:

```
$ mvn groovy:shell
```

The Groovy Shell can be accessed. If you prefer a GUI access to the shell, run the following command:

```
$ mvn groovy:console
```

Integrating Flex development with Maven

Adobe Flex is one of the pioneers and leaders in the **Rich Internet Applications** (**RIA**) space, and with the advent of Adobe AIR, it has proved itself as a viable option for desktop and mobile applications as well.

Here's how Adobe describes Flex on `http://www.adobe.com/products/flex/`:

> *"Build engaging, cross-platform rich Internet applications*
>
> *Flex is a highly productive, free, open source framework for building expressive web applications that deploy consistently on all major browsers, desktops, and operating systems by leveraging the Adobe® Flash® Player and Adobe AIR® runtimes."*

Flex is a rich framework that lets you build rich applications for deployment on the Internet as well as desktops and devices. Flex application code consists of MXML (`.mxml`) and ActionScript (`.as`) files, classes, and packages. It compiles to produce a SWF (`.swf`) artifact that can be rendered in the Flash Player and Adobe AIR runtime.

While Adobe ships a good IDE, that is, Adobe Flash Builder and it brings with it an Eclipse project structure, it is generally advisable to "Mavenize" your Adobe Flex project especially if it's a large enterprise project with multiple developers involved.

As an Apache Maven project, not only can the Flex module be integrated with a unified build, but also a number of other benefits described in *Chapter 2* and *Chapter 3* can be reaped such as continuous integration, Test Driven Development, and so on.

Getting ready

You need Apache Maven installed. Preferred versions are Apache Maven 3 along with JDK 1.6 or higher. The Adobe Flex SDK is an Apache Maven project dependency and will be automatically included when required.

How to do it...

Generate a Maven Flex project using the FlexMojos application archetype:

```
$ mvn archetype:generate
```

This will show a list of available archetypes in which you need to select the `flexmojos-archetypes-application` archetype (which is number `353` for me). Maven interactive mode will also ask you for `archetype version`, `groupId`, `artifactId`, and other Maven project co-ordinates.

On completion, your terminal will look similar:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2:55.017s
[INFO] Final Memory: 10M/114M
[INFO] ------------------------------------------------------------------------
```

The completion of this operation has created a project folder with the same name as the project artifact ID. The project directory tree structure will look similar to this:

```
C:.
└───src
    ├───main
```

```
|   ├──flex
|   |   └──Main.mxml
|   └──resources
└──test
    └──flex
        └──net
            └──srirangan
                └──packt
                    └──maven
                        └──TestApp.as
```

As seen above, a Flex application, `Main.mxml`, was generated. The file template was created by Marvin Herman Froeder and is distributed with the Apache License through the `flexmojos` archetype. The contents for the file are:

```
...
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
  <mx:Text text="Hello World!"/>
</mx:Application>
```

Another file generated is `TestApp.as` in the `src/test` folder, which contains a simple test case:

```
package net.srirangan.packt.maven
{
  import flexunit.framework.TestCase;
  import Main;
  public class TestApp extends TestCase
  {
    public function testNothing():void
    {
      //TODO un implemented
      trace("Hello test");
    }
  }
}
```

Let's inspect the project dependencies defined in the POM file (`pom.xml` in the project root folder):

```
<dependencies>
  <dependency>
    <groupId>com.adobe.flex.framework</groupId>
    <artifactId>flex-framework</artifactId>
    <version>4.0.0.13875</version>
```

```
      <type>pom</type>
    </dependency>
    <dependency>
      <groupId>com.adobe.flexunit</groupId>
      <artifactId>flexunit</artifactId>
      <version>0.85</version>
      <type>swc</type>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

It's as simple and beautiful as it can get. Just the two dependencies—one for the Flex framework itself followed by the `flexunit` dependency for unit testing.

Let's now compile our Apache Maven–Flex project:

```
$ mvn compile
```

And let's run the unit tests:

```
$ mvn test
```

You can combine these two steps by running:

```
$ mvn install
```

How it works...

In the *How to do it...* section above, we created a new Adobe Flex project using an Apache Maven archetype. We have also seen how to execute Maven build lifecycle phases for the project (`compile`, `test`, `install`, and so on).

While inspecting the project POM file (`pom.xml`), we see that the `flexmojos` plugin is integrated with the default build cycle:

```
<build>
  <sourceDirectory>src/main/flex</sourceDirectory>
  <testSourceDirectory>src/test/flex</testSourceDirectory>
  <plugins>
    <plugin>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-maven-plugin</artifactId>
      <version>4.0-pre-alpha-1</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

The POM also reveals that the packaging of the project is `swf` as Flex projects generate output in the Flash Player format, namely, SWF:

```
<groupId>net.srirangan.packt.maven</groupId>
<artifactId>TestFlexApp</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>swf</packaging>
```

There's more...

In addition to the compilation and test features we witnessed above, the `flexmojos` plugin provides many more goals that cater to most of our needs as Flex developers.

The table below lists the `flexmojos` goal, its corresponding Maven command, and an explanation of its function:

FlexMojos Goal	Maven Command	Details
flexmojos:asdoc	`$ mvn flexmojos:asdoc`	Generates documentation for ActionScript source files
flexmojos:asdoc-report	`$ mvn flexmojos:asdoc-report`	Generates documentation for ActionScript source files as a report to be included in the Maven site
flexmojos:compile-swc	`$ mvn flexmojos:compile-swc`	Compiles MXML and ActionScript sources into a SWC library
flexmojos:compile-swf	`$ mvn flexmojos:compile-swf`	Compiles MXML and ActionScript sources into a SWF executable
flexmojos:copy-flex-resources	`$ mvn flexmojos:coyp-flex-resources`	Copies Flex resources into a web application project–used in a multi-modular project setup
flexmojos:flexbuilder	`$ mvn flexmojos:flexbuilder`	Generates Adobe Flash Builder (previous known as Flex Builder) configuration files
flexmojos:generate	`$ mvn flexmojos:generate`	Generates ActionScript classes based on Java classes using GraniteDS
flexmojos:optimize	`$ mvn flexmojos:optimize`	Runs post-line SWF optimization on SWC library files
flexmojos:sources	`$ mvn flexmojos:sources`	Creates a JAR containing all the sources

FlexMojos Goal	Maven Command	Details
flexmojos:test-compile	`$ mvn flexmojos:test-compile`	Compiles all the test classes
flexmojos:test-run	`$ mvn flexmojos:test-run`	Runs the tests in the Adobe Flash Player
flexmojos:test-swc	`$ mvn flexmojos:test-swc`	Builds an SWC file containing the test sources
flexmojos:wrapper	`$ mvn flexmojos:wrapper`	Generates an HTML wrapper for an SWF application

8
IDE Integration

In this chapter, we will cover:

- Creating a Maven project with Eclipse 3.7
- Importing a Maven project with Eclipse 3.7
- Creating a Maven project with NetBeans 7
- Importing a Maven project with NetBeans 7
- Creating a Maven project with IntelliJ IDEA 10.5
- Importing a Maven project with IntelliJ IDEA 10.5

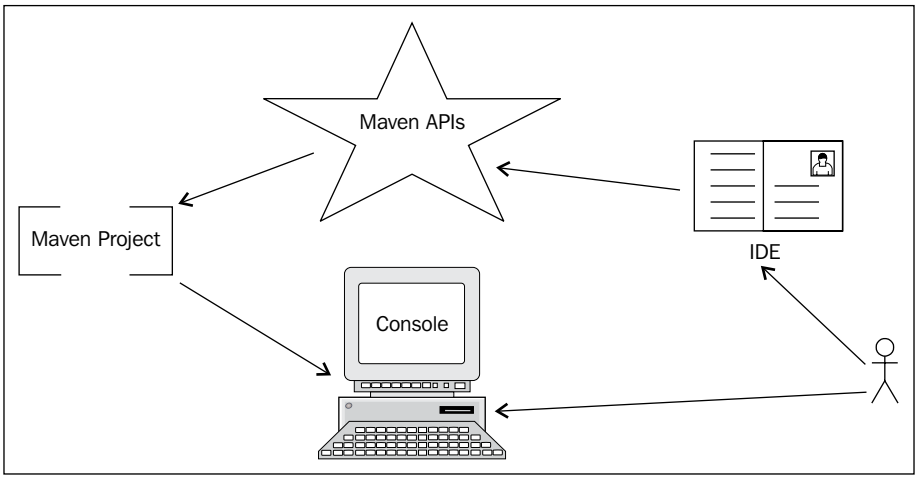
Advanced, modern **Integrated Development Environments** (**IDEs**) are being used by an overwhelming majority of developers and professional programmers in the industry. This holds true for most technologies, regardless of the underlying programming language/framework being used.

In previous chapters, most of what we covered eventually involved the running of and execution of Apache Maven commands on the terminal/console. This is the way Apache Maven works; it is, after all, a command-line tool.

In this chapter, many of the same commands will be executed by the IDE in the background while the developer works and interacts with the IDE's graphical UI.

It must be said that it isn't too hard to switch between the command line and your IDE in most modern operating systems, and it is this *"is a command-line tool"* property of Maven and its well-defined standards and conventions that has enabled its easy integration with other tools in the development infrastructure, including the developer's IDE.

The three most popular development IDEs for developers today would surely be Eclipse, NetBeans, and JetBrains IntelliJ IDEA. Each of these has excellent integration with Maven, Maven projects, and the extended Maven ecosystem.



This integration is provided through IDE plugins that strictly adhere to the Maven standards.

A project that has been integrated with the IDE almost always remains uncorrupted and independent of the IDE, fully compliant with Maven standards, and thus ready for use from the console/terminal, if the developer (or any third-party application) chooses to do so.

Creating a Maven project with Eclipse 3.7

For many years now, Eclipse remained a popular IDE for enterprise and open source development in Java and other technologies. Eclipse is free and open source and shares its roots and origin with VisualAge. According to Wikipedia:

> *"Eclipse began as an IBM Canada project. It was developed by Object Technology International (OTI) as a Java-based replacement for the Smalltalk-based VisualAge family of IDE products, which itself had been developed by OTI. In November 2001, a consortium was formed to further the development of Eclipse as an open source product and platform. In January 2004, the Eclipse Foundation was created."*

Eclipse is more than just an IDE; it also is a platform for extension and plugin/add-on development. Here then comes **M2Eclipse**, which is an Eclipse plugin developed by Sonatype providing extensive Eclipse-Maven integration.

M2Eclipse isn't the only Eclipse-Maven integration plugin, but it is definitely one of the most complete yet easy-to-use plugins. As of Eclipse 3.7, M2Eclipse is pre-installed in the Java-specific bundle.

Getting ready

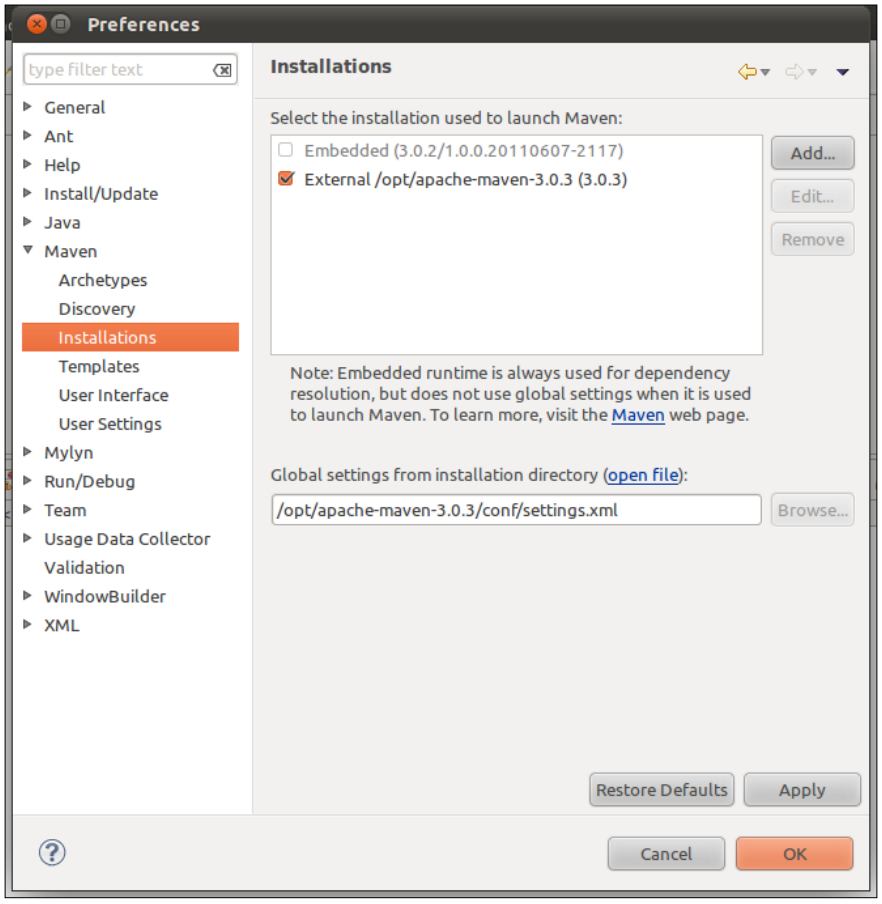
To get started, you need to download and install Eclipse 3.7 from the Eclipse website:

`http://www.eclipse.org/downloads/`.

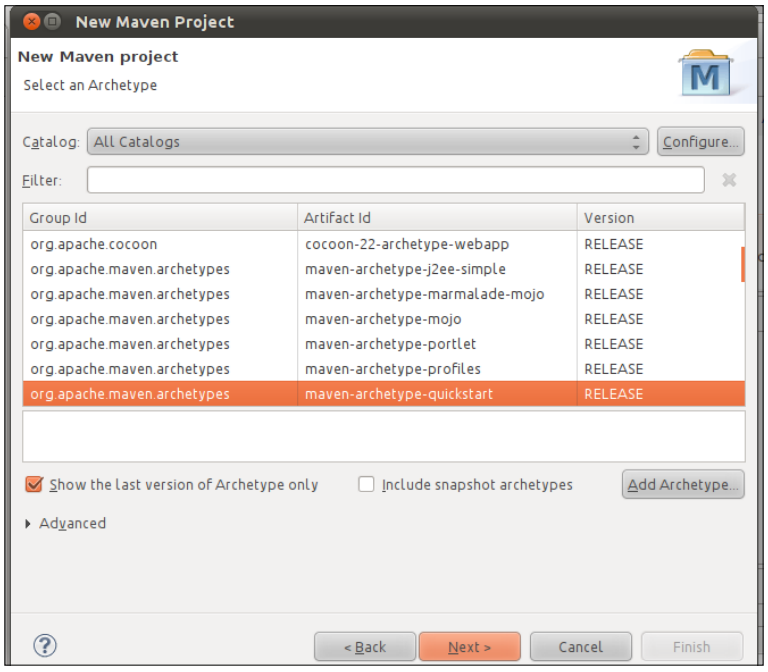
Make sure you download a Java-specific package of Eclipse so that the Maven M2Eclipse plugin is bundled. If your Eclipse installation does not have M2Eclipse pre-installed, you can manually install the plugin from the Eclipse Marketplace.

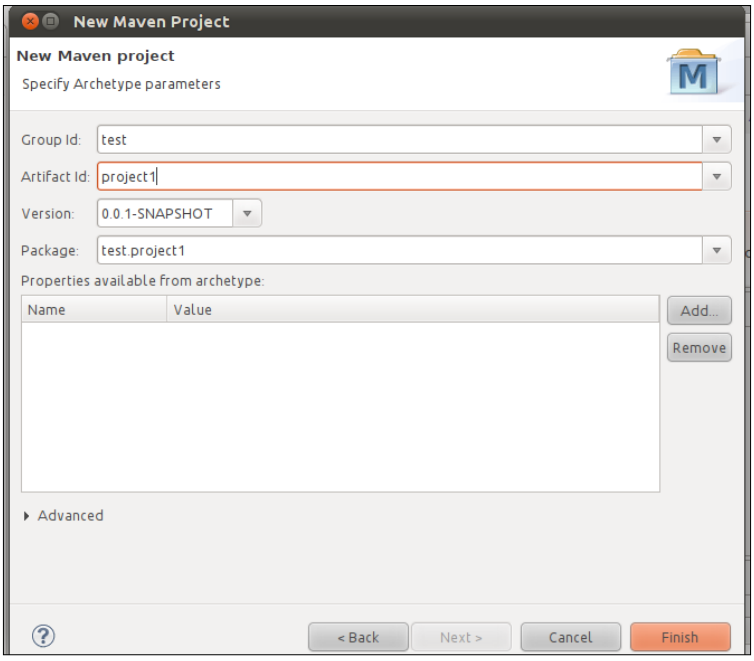
How to do it...

1. Make sure the correct installation of Maven is configured in Eclipse. You can find this setting in **Window | Preferences | Maven** | **Installations**. You can continue to use the Eclipse embedded Maven or choose an external installation, as shown in the following screenshot:



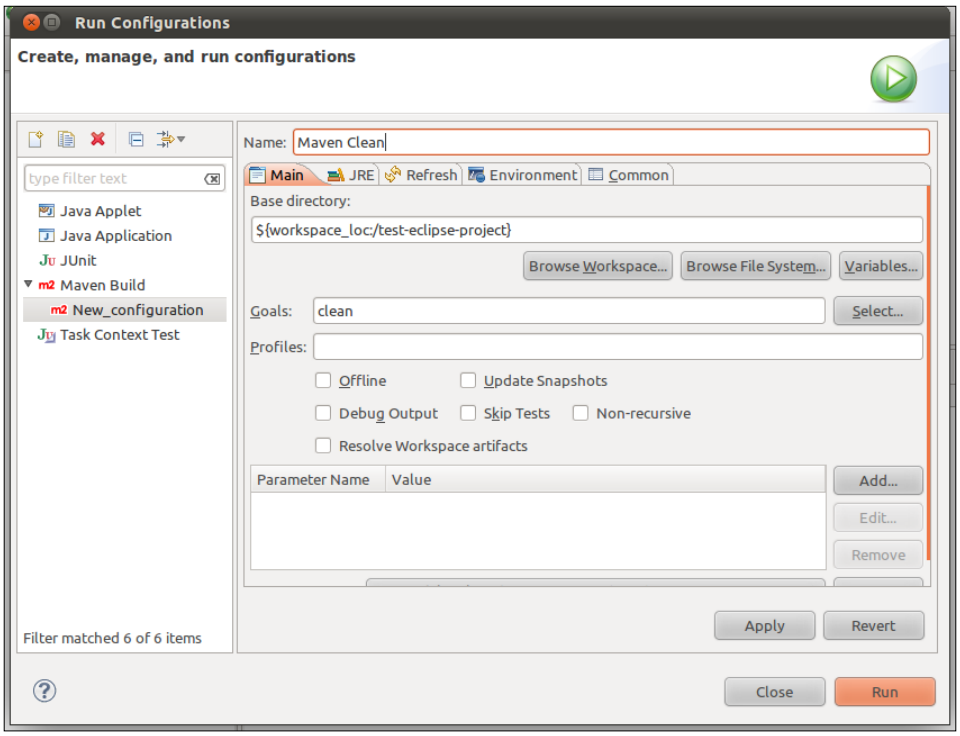
2. Create a new project (*Ctrl+N*), select **Maven Project**, and continue on the wizard to select an archetype and set up the project co-ordinates.





A new project has been created, which is both a valid Eclipse project and a valid Apache Maven project.

3. Now we would like to execute Maven commands through Eclipse. This is done by creating Eclipse **Run Configurations** for Maven goals. We start by selecting the **Run** | **Run** configurations. Select **Maven Build** and create a new configuration.

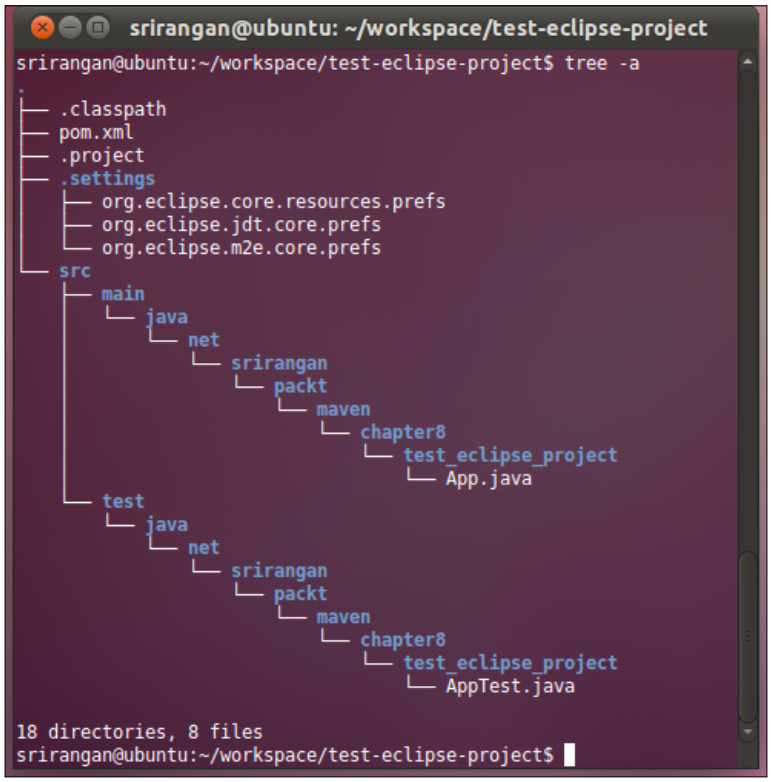


The **Base directory**, **Goals**, **Profiles**, and other settings can be configured while creating or modifying the **Run configuration**.

How it works...

We just created a project that is an Eclipse and Maven project. If you inspect the filesystem, you will find that a valid `pom.xml` file exists in the project root, which makes it a valid Maven project.

In addition, you will also find `.classpath` and `.project` files and the `.settings` folder which are Eclipse-specific resources, making it a valid Eclipse project as well.



Once you create and execute a Maven build run configuration, Eclipse starts a read-only console at the path defined in the **Run configuration**. Then the goals defined in the **Run configuration** are executed using Maven APIs and the output is shared on the read-only console.

Importing a Maven project with Eclipse 3.7

In the previous recipe, *Creating a Maven project with Eclipse 3.7*, we created a project compatible with Eclipse and Apache Maven. We also created custom Maven build run configurations to execute lifecycle or other Maven plugin goals from within the IDE itself.

However, most of the time your Apache Maven project would exist on a shared filesystem or version control/code repository. This would be shared by your entire team and it is a best practice not to share IDE files on the code repository.

In such situations, you need to import your Apache Maven project into the Eclipse IDE. M2Eclipse allows you to do the same through an easy and intuitive wizard.

Getting ready

Firstly, you need to download and install Eclipse 3.7 from the Eclipse website:

`http://www.eclipse.org/downloads/`.

Make sure you download a Java-specific package of Eclipse so that the Maven M2Eclipse plugin is bundled in, otherwise you will have to manually install the plugin from the Eclipse Marketplace.

You will also need a pre-existing Apache Maven project to import into Eclipse IDE.

How to do it...

Launch the project import wizard by selecting File | Import and then selecting Maven | Existing Maven Projects.

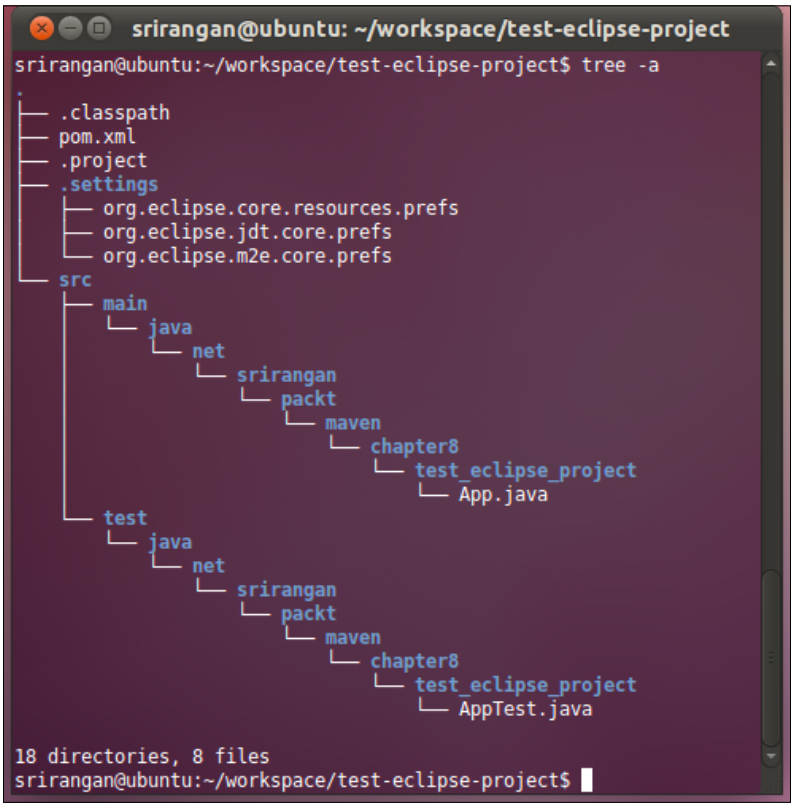
When you enter the root directory of your project, you will find the listing of the Maven projects (and their modules, if they have any) in the following screenshot.

Select one or more projects that you want to import and hit Finish.

How it works...

Eclipse (well in fact, the M2Eclipse plugin) has now created the requisite Eclipse-specific file and converted the Maven project into a "Maven + Eclipse" project.

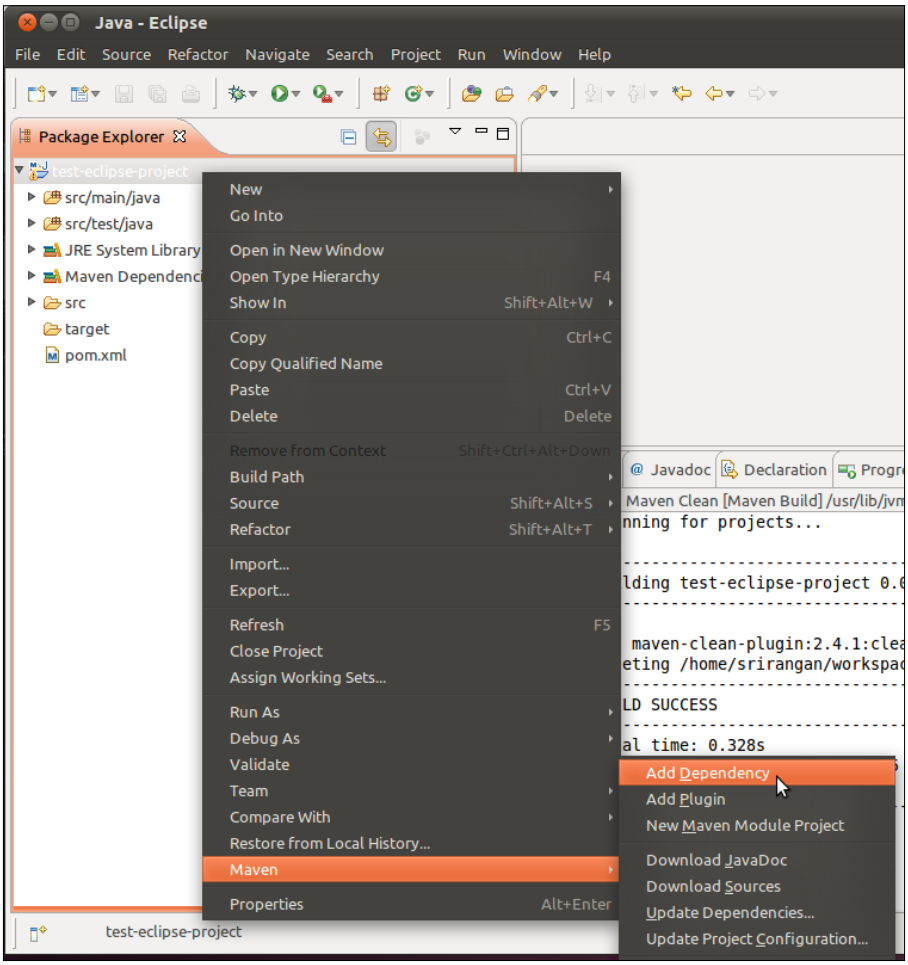
You will find **.project**, **.classpath** files, and the **.settings** folder in your filesystem:



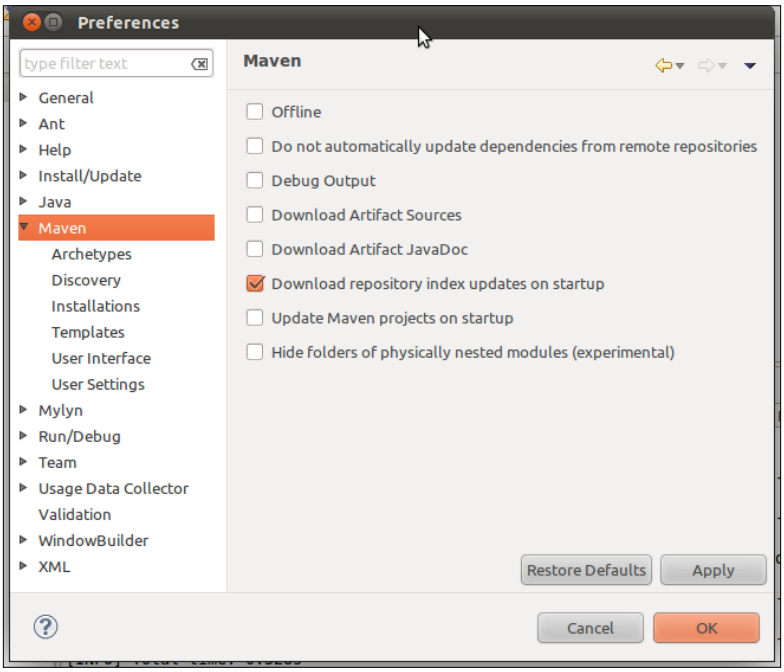
There's more...

Project-specific Maven options are available by right-clicking on the project and selecting the Maven menu item.

As shown in the following screenshot, these options include adding of dependencies, plugins, downloading JavaDocs and sources, creating new modules, and so on:



Global Maven settings for Eclipse can be modified by selecting **Windows | Preferences | Maven**:



Here you can configure the M2Eclipse behavior, defaults for Maven integration, **Archetypes**, **Discovery**, **Installations**, **User Settings**, and so on.

Creating a Maven project with NetBeans 7

Much like Eclipse, NetBeans is an open source platform-framework and an IDE (Integrated Development Environment) for developing with Java and other technologies.

It is a free and open source software maintained by the NetBeans community and backed by Sun/Oracle Corporation. According to `http://netbeans-org`:

> *"The NetBeans IDE is written in Java and runs everywhere where a JVM is installed, including Windows, Mac OS, Linux, and Solaris... The NetBeans Platform allows applications to be developed from a set of modular software components called modules. Applications based on the NetBeans platform (including the NetBeans IDE) can be extended by third party developers."*

The Maven module will either be pre-included in your NetBeans installation or can be installed on top of the NetBeans IDE.

The Maven module comes with an embedded instance of Maven or alternatively can be configured to work with one or more standalone instances of Maven.

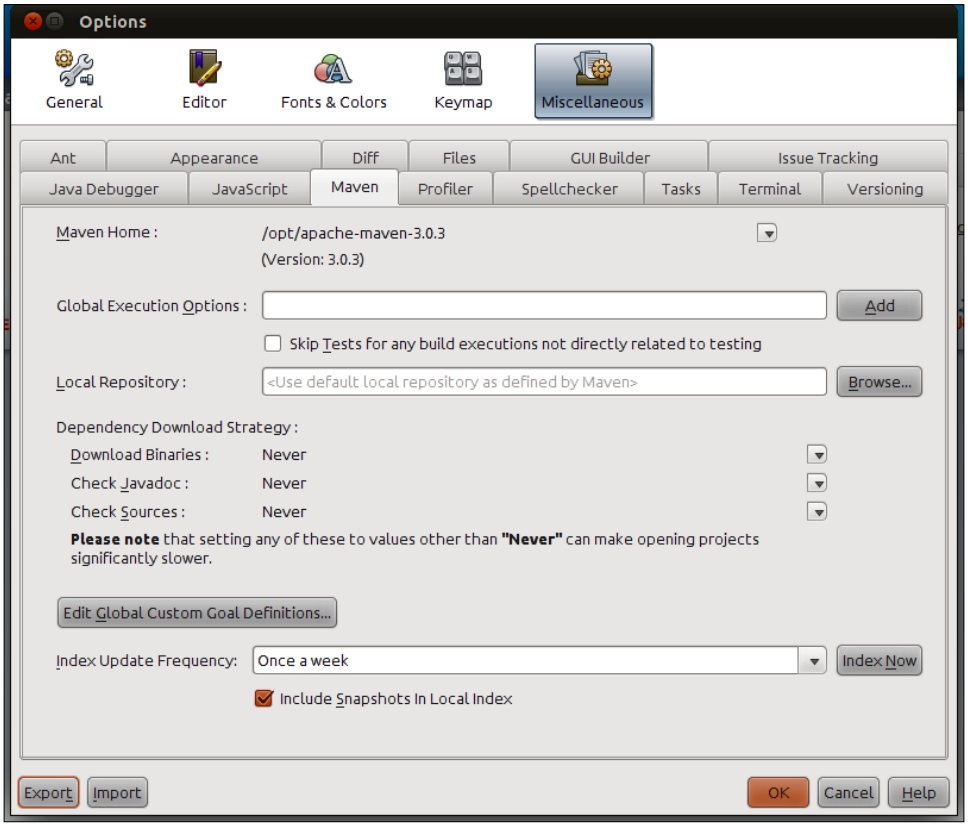
It provides comprehensive integration between NetBeans and Maven, and you the developer could almost do without executing Maven commands from the terminal/console.

Getting ready

Download the NetBeans 7 bundle for Java EE from the NetBeans website.

`http://netbeans.org/downloads/index.html`

Install NetBeans 7 and configure your Maven home directory in **Tools | Options | Miscellaneous | Maven**.

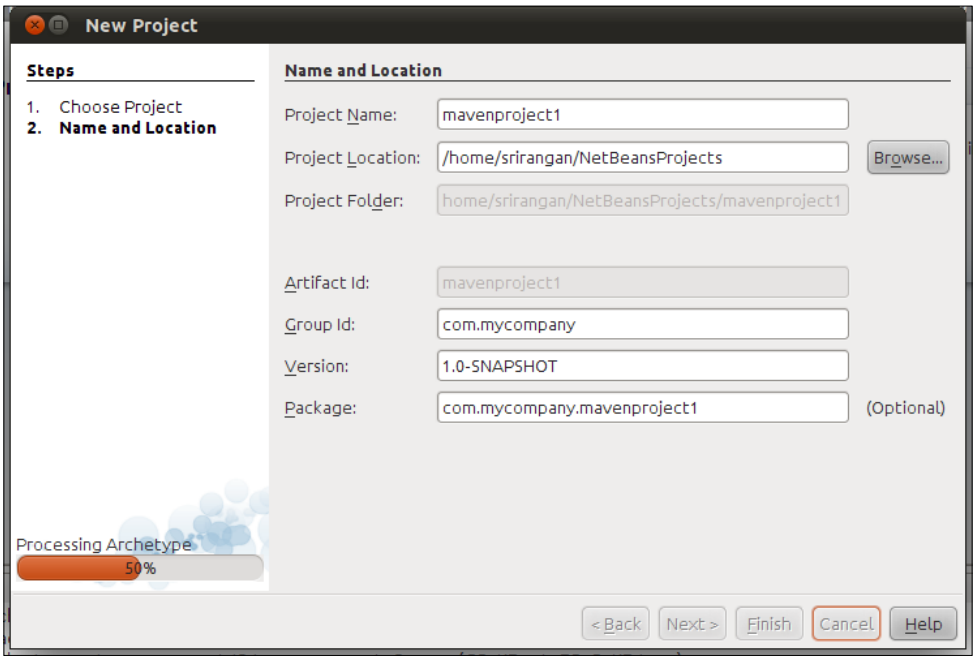


If you have an existing Apache Maven installation, you can point to that or continue to use the bundled version of Apache Maven.

How to do it...

Launch the **New Project** wizard by selecting **File | New Project** and then select **Maven | Java Project**.

Next, specify the Maven project co-ordinates and complete the wizard:

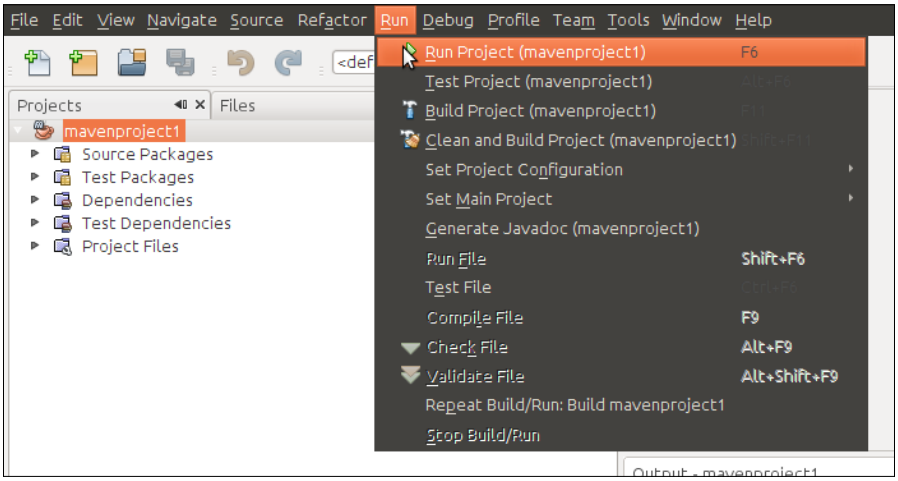


Here you can enter the **Project Name** and **Project Location** on your filesystem and the Apache Maven project co-ordinates.

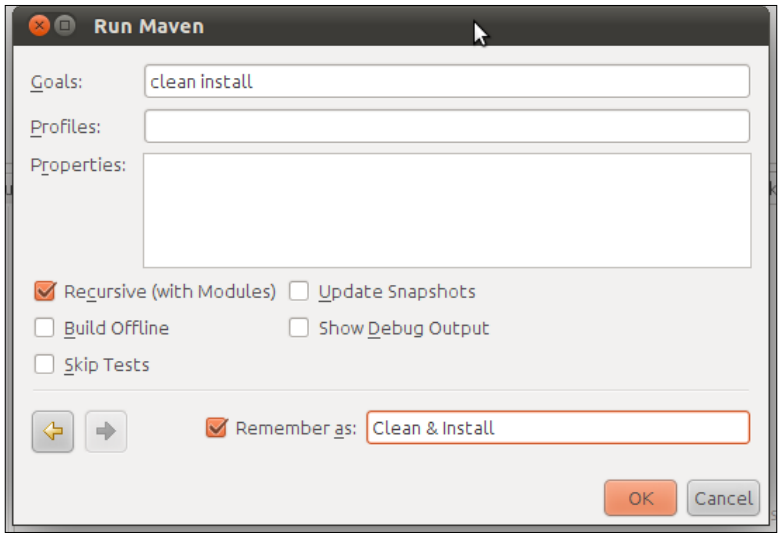
The **Artifact ID** is picked up from the project name, while the text fields for **Group Id**, **Version**, and **Package** can be edited.

On completion, a new project will be created, which is both a NetBeans project and an Apache Maven project.

NetBeans directly integrates with the Maven build lifecycle and provides you with a range of options to quickly clean, build, and execute your project under the **Run** and **Debug** menu options:



To create a custom Maven goal execution configuration, right-click the project, select **Custom | Goals**, and define your custom goal:

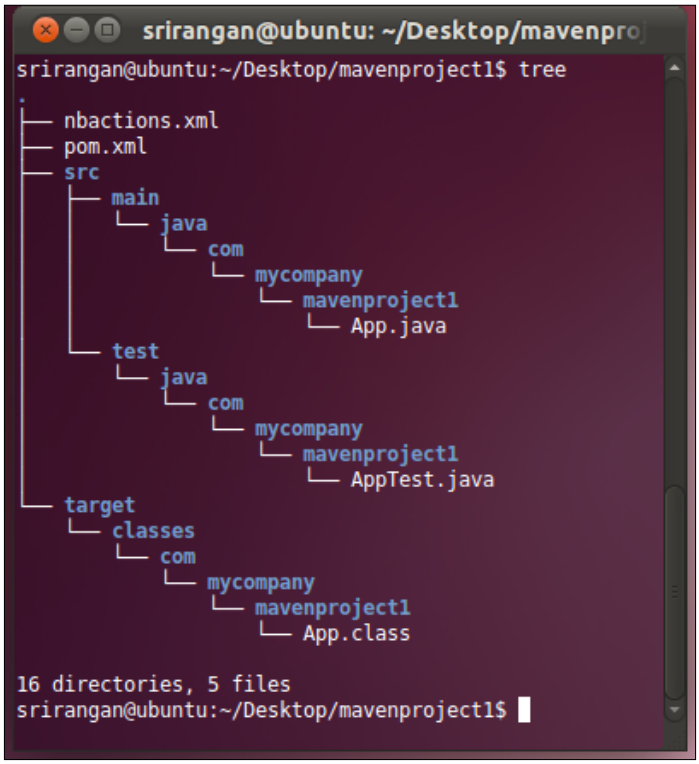


Apart from specifying the custom goal, you can also define **Profiles** and custom **Properties** for the custom run configuration.

Select the checkbox at the bottom to save this configuration for future use.

How it works...

We just created a project that is a NetBeans and Maven-compatible project. If you inspect the filesystem, you will find that a valid **pom.xml** file exists in the project root, which makes it a valid Maven project:



The presence of a project POM file makes it a valid NetBeans project, as NetBeans is fully compatible with the configurations in the POM.

In addition to the `pom.xml` file, you will also find the `nbactions.xml` file, which is a NetBeans-specific file, to store the custom goal/execution configuration.

Once you execute the project clean, build, run, or custom Maven goal, NetBeans starts a read-only console at the project root. The appropriate Maven goals are executed using Maven APIs and the output is shared on the read-only console.

Importing a Maven project with NetBeans 7

In the previous recipe, *Creating a Maven project with NetBeans 7*, we created a project compatible with NetBeans and Apache Maven. We also created custom run configurations to execute lifecycle or other Maven plugin goals from within the IDE itself.

However, most of the time your Apache Maven project would exist on a shared filesystem or version control/code repository. This would be shared by your entire team and it is a best practice not to share IDE files on the code repository.

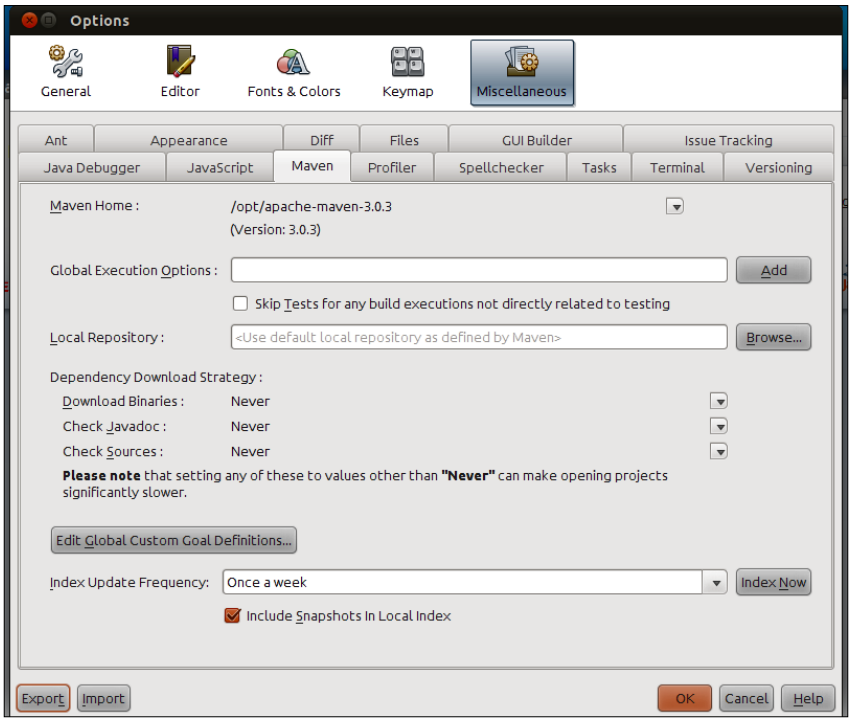
In such situations, you need to import your Apache Maven project into the NetBeans IDE. NetBeans is fully compatible with the Apache Maven Project Object Model and thus allows you to natively open any Apache Maven project.

Getting ready

Download the NetBeans 7 bundle for Java EE from the NetBeans website:

`http://netbeans.org/downloads/index.html`.

Install NetBeans 7 and configure your Maven home directory in **Tools | Options | Miscellaneous | Maven**.

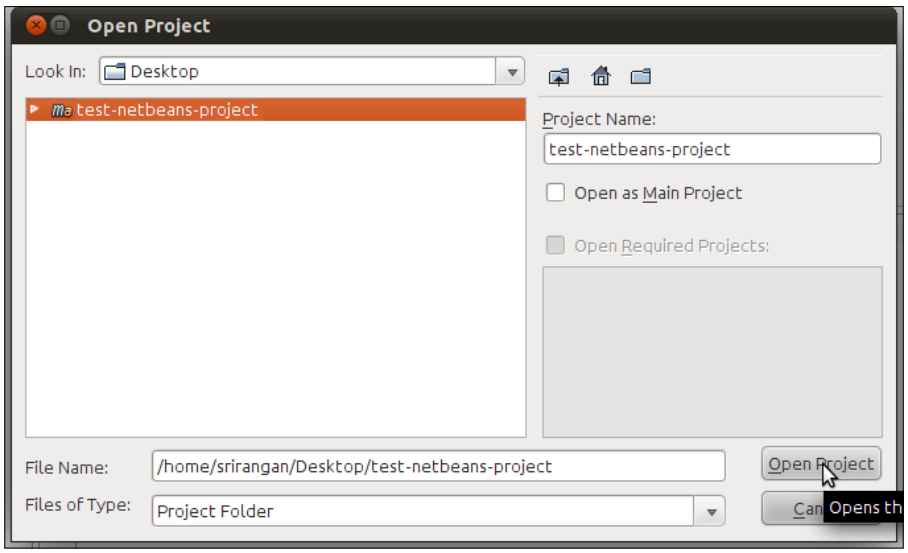


If you have an existing Apache Maven installation, you can point to that or continue to use a bundled version of Apache Maven.

In addition, you would also need an existing Apache Maven project to import as a NetBeans project.

How to do it...

To work on your Apache Maven project with NetBeans, select the **File | Open Project** dialog and navigate to your project root folder.



You will find the folder with the `pom.xml` file highlighted with a special **Maven** icon, and this folder can be selected and opened as a NetBeans 7 project.

How it works...

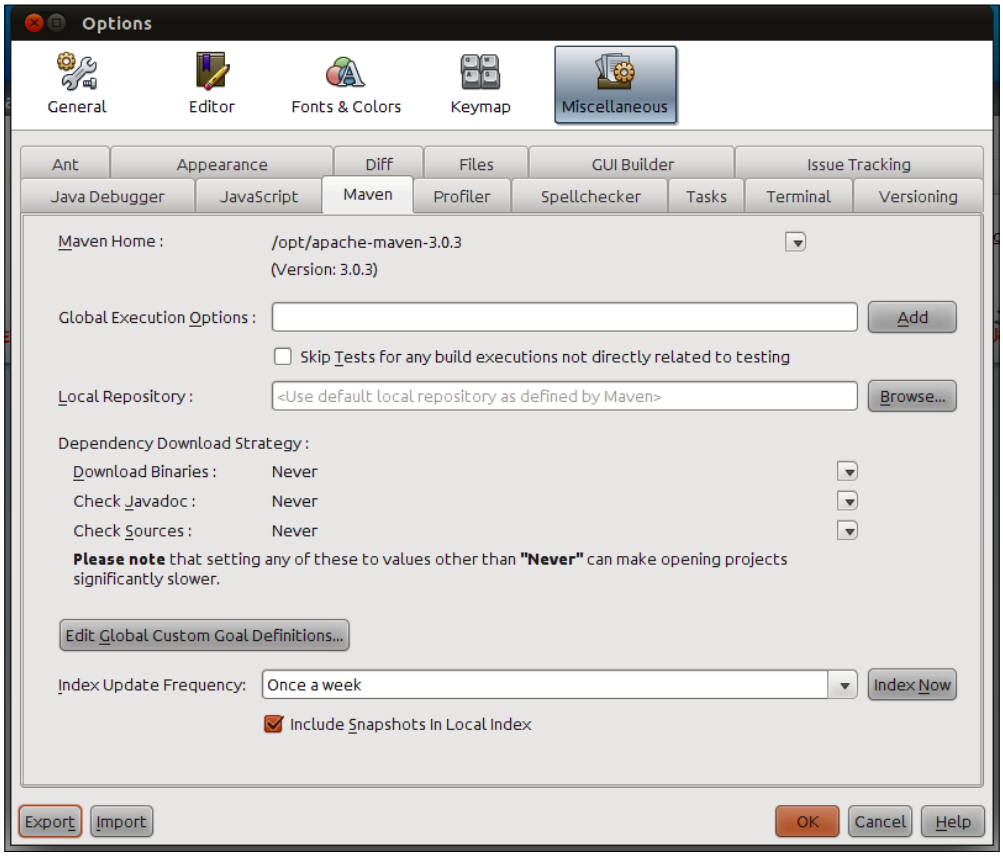
As we saw in the previous recipe, *Creating a Maven project with NetBeans 7*, an Apache Maven project is fully compatible with NetBeans 7. This means that there isn't any actual import process that converts a Maven project into a NetBeans project.

An Apache Maven project in itself is a valid NetBeans 7 project. The NetBeans IDE relies on the information and project configuration specified in the `pom.xml` file and lets the developer work seamlessly on the Maven project with the NetBeans IDE.

A Maven project on NetBeans IDE behaves no different from a native NetBeans project and the developer may work seamlessly on either without ever having to know the internals.

There's more...

Global settings for Apache Maven on NetBeans can be modified by selecting **Tools | Options | Miscellaneous** | **Maven**.



The settings that can be modified include Maven installation location, repository location, dependency download strategies (binaries, sources, docs), global custom goal definitions, and so on.

Creating a Maven project with IntelliJ IDEA 10.5

An extremely popular alternative to Eclipse and NetBeans, IntelliJ IDEA is according to many programmers, the best IDE available. While Eclipse and NetBeans are quite similar to each other, JetBrains IntelliJ IDEA offers a unique experience to the developer.

In my personal experience, after years of using Eclipse and NetBeans, when I migrated to IntelliJ IDEA, at first it was a little challenging. However, within a matter of days, I could experience productivity enhancements in my own work. Now it has been a while since I migrated to IntelliJ IDEA and I'm as strong an advocate as any for JetBrains IntelliJ IDEA.

Originating back in 2001, it was the pioneer of popular features now taken for granted, including code navigation and re-factoring. It is now available as commercial software and also as a free, open source community software. According to Wikipedia:

> *"JetBrains is a Czech software development company with offices in Prague, Czech Republic; Saint Petersburg, Russia and Boston, USA. It is best known for its Java IDE, IntelliJ IDEA and for its Visual Studio plugin ReSharper ... JetBrains was founded in 2000 as a private company, by Sergey Dmitriev, Eugene Belyaev, and Valentin Kipiatkov."*

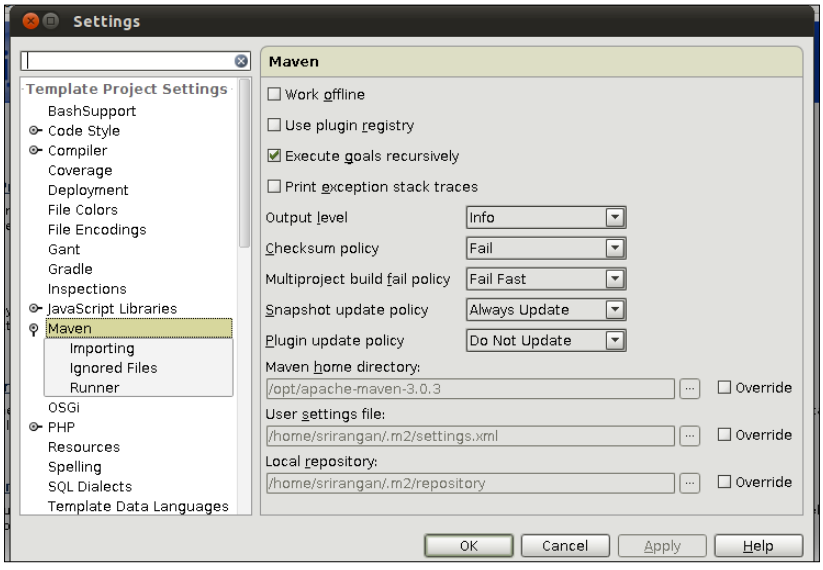
Getting ready

For this recipe, you need to download and install IntelliJ IDEA. You can download it from the JetBrains website:

`http://www.jetbrains.com/idea/download/index.html`.

IntelliJ IDEA comes pre-installed with an Apache Maven plugin and no additional installations are required. Your external installation of Apache Maven is compatible with IntelliJ IDEA.

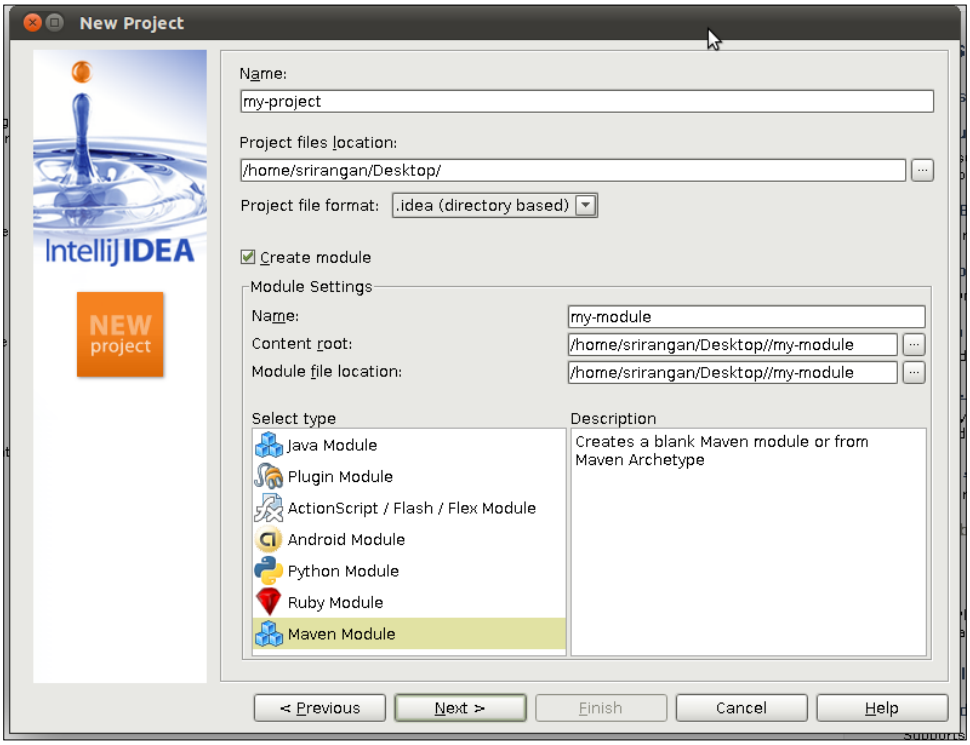
The IntelliJ IDEA Maven plugin automatically picks up your Apache Maven installation if the `M2_HOME` environment variable is defined. You can manually change the Maven installation by overriding the path through **File | Settings | Maven**.



How to do it...

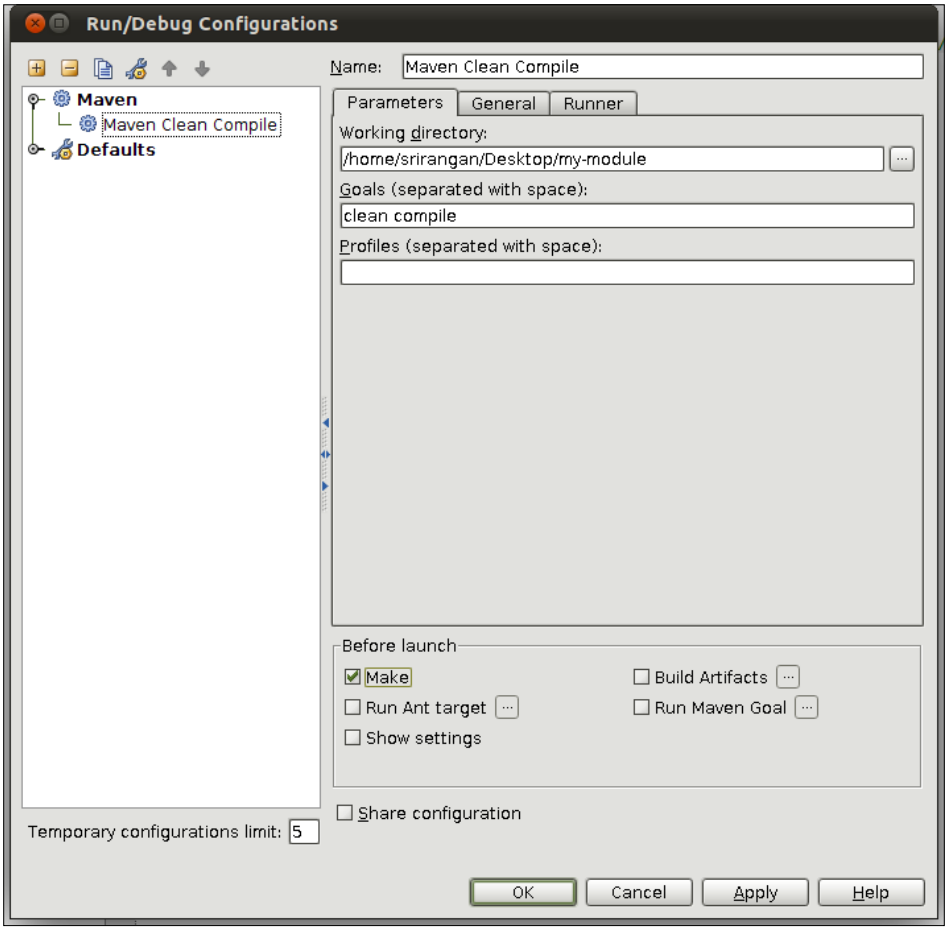
To create a new Apache Maven project through IntelliJ IDEA, you need to select the menu option **File | New Project** and create a project from scratch.

Name your project and select its location on the filesystem. Then proceed to create your first module, enter the module **Name**, and select the type **Maven Module**, as shown in the following screenshot:



In the next step, you can enter your project co-ordinates and make an Apache Maven project archetype selection.

Once your project is created, you can proceed to create a run configuration to execute one or more Apache Maven goals. Select **Run | Edit Configurations** and add a new Maven run configuration by selecting the **+**icon.



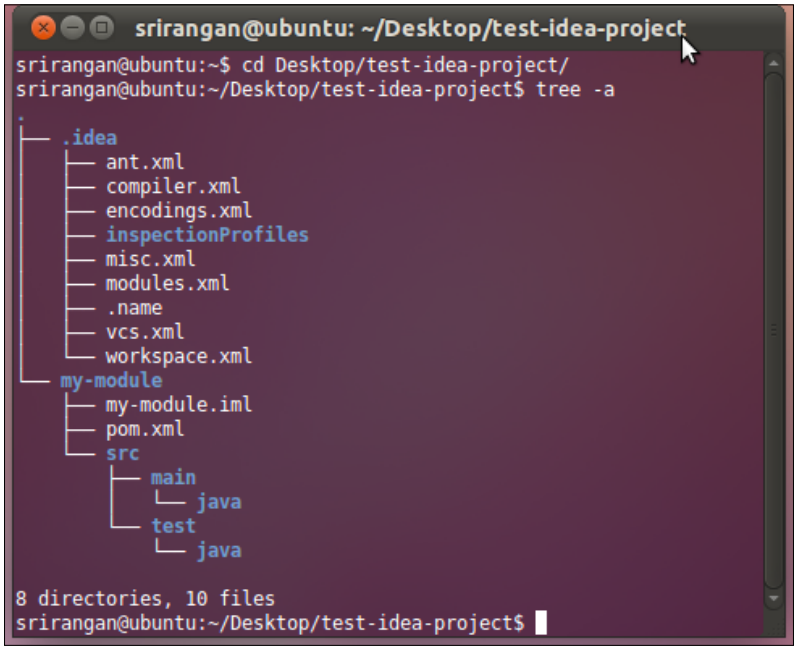
You can now select a working directory, which should be the project root containing the `pom.xml` file, specify the Maven goals to be executed, and define other IDE / Maven options.

Once the run configuration is created, it can be executed through the **Run** menu option.

How it works...

We have created an Apache Maven project that is at the same time an IntelliJ IDEA project as well.

If you inspect the filesystem, you will find the `.idea` directory and an `.iml` file which are specific to IntelliJ IDEA. The project also contains a valid `pom.xml` file, which makes it an Apache Maven project.



We also created a custom Apache Maven run configuration in IntelliJ IDEA and when that is executed, IntelliJ IDEA launches a console at the working directory defined in the Run Configuration.

Then, the specified goals are executed and the output is shared on the console.

Importing a Maven project with IntelliJ IDEA 10.5

In the previous recipe, *Creating a Maven project with IntelliJ IDEA 10.5*, we created a project compatible with IntelliJ IDEA and Apache Maven. We also created custom run configurations to execute lifecycle or other Maven plugin goals from within the IDE itself.

However, most of the time your Apache Maven project would exist on a shared filesystem or version control/code repository. This would be shared by your entire team and it is a best practice not to share IDE files on the code repository.

In such situations, you need to import your Apache Maven project into the IntelliJ IDEA IDE. IntelliJ IDEA is fully compatible with the Apache Maven Project Object Model and thus allows you to natively open any Apache Maven project.

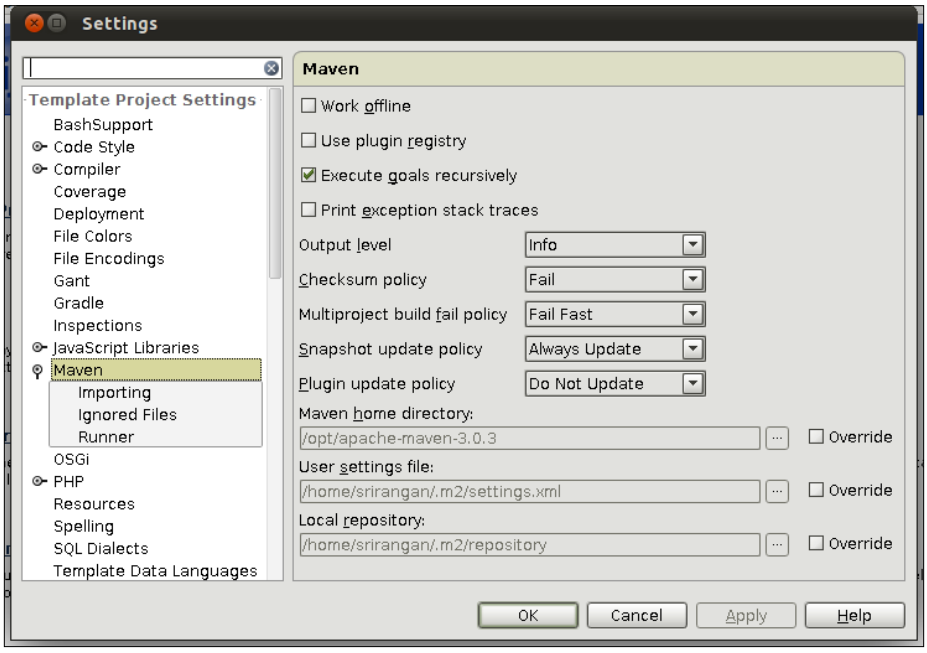
Getting ready

For this recipe, you need to download and install IntelliJ IDEA. You can download it from the JetBrains website:

`http://www.jetbrains.com/idea/download/index.html`.

IntelliJ IDEA comes pre-installed with an Apache Maven plugin and no additional installations are required. Your external installation of Apache Maven is compatible with IntelliJ IDEA.

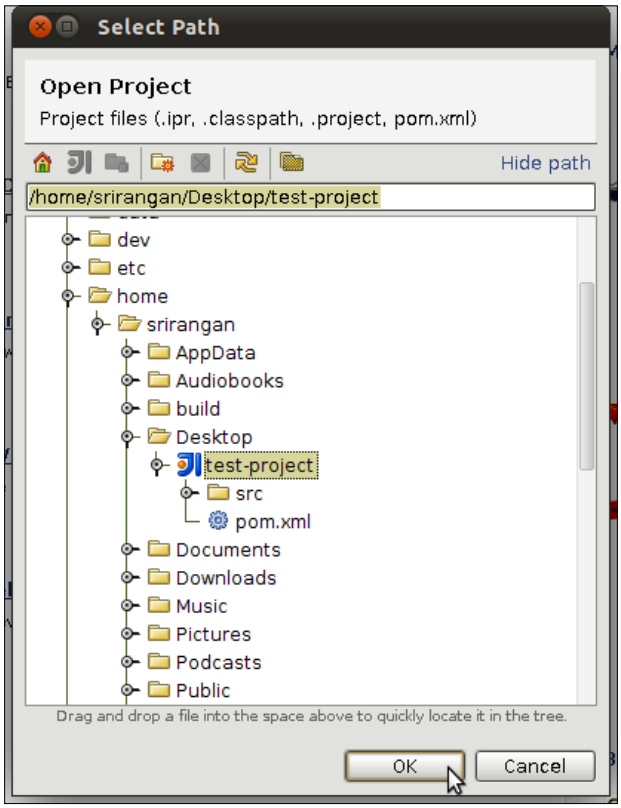
The IntelliJ IDEA Maven plugin automatically picks up your Apache Maven installation if the `M2_HOME` environment variable is defined. You can manually change the Maven installation by overriding the path through **File** | **Settings** | **Maven**.



In addition, you will also need an existing Apache Maven project to import into IntelliJ IDEA.

How to do it...

Let's begin by launching the open project dialog by selecting **File | Open Project** and then navigating to the root of your Apache Maven project:

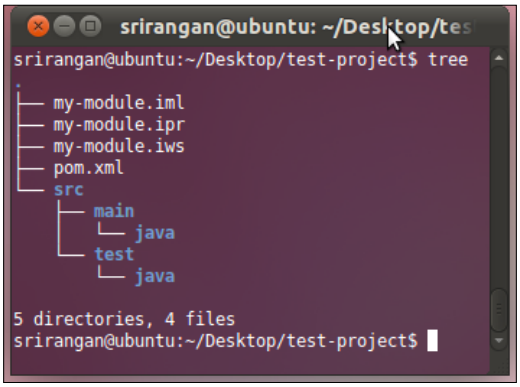


Select that folder or the **pom.xml** file it contains and click **OK**. The Apache Maven project will now automatically open up inside IntelliJ IDEA.

How it works...

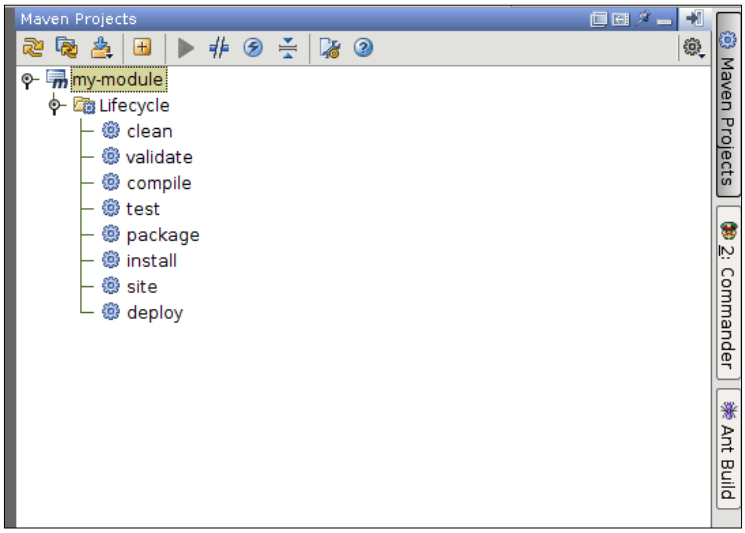
IntelliJ IDEA comes pre-installed with the Apache Maven plugin. This plugin makes Apache Maven projects directly compatible with IntelliJ IDEA and discards the need for any Apache Maven to IntelliJ IDEA "import" process.

If you inspect the filesystem, you will not only find the **pom.xml** but additional **.iml**, **.iws**, and **.ipr** files in the directory. These are project files created by IntelliJ IDEA for its internal use.



There's more...

For all Apache Maven projects, IntelliJ IDEA provides the **Maven Projects** tool window. The Maven Projects tool window lists out all Apache Maven modules along with their lifecycle goals in the tree-menu format.



A button/menu option is provided for the execution of these goals.

How to execute custom Apache Maven goals has been explained in the preceding recipe, *Creating a Maven project with IntelliJ IDEA 10.5*.

9
Extending Apache Maven

In this chapter, we will cover:

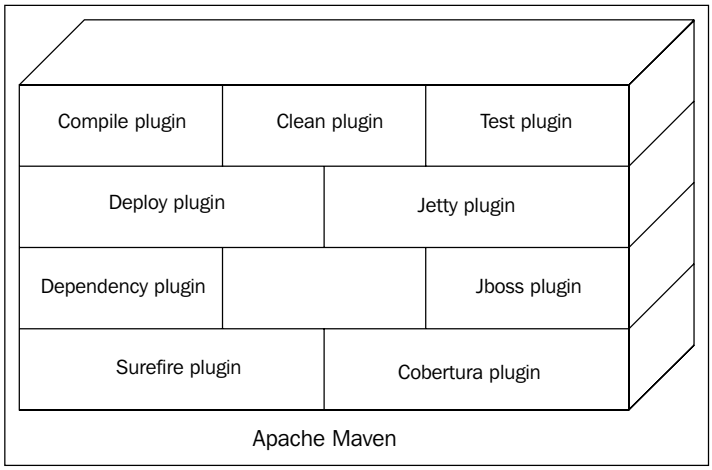
- Creating a Maven plugin using Java
- Making your Java Maven plugin useful
- Documenting your Maven plugin
- Creating a Maven plugin using Ant
- Creating a Maven plugin using JRuby

This chapter looks at ways you can extend the current functionality of Apache Maven and its plugins by writing plugins of your own. We are not going into specific details of end-to-end implementation of one particular plugin, but will broadly look at the various ways we can extend Apache Maven and present examples for each of these methods.

By design, Apache Maven is nothing more than a set of plugins wrapped together within a common framework. Every Maven functionality we have explored in this book from *Chapter 1* to *Chapter 8* has made use of one or more Apache Maven plugins. This module architecture holds true for many Java applications, including the popular JetBrains IntelliJ IDEA IDE as well.

Like Apache Maven, an Apache Maven plugin needs to be executed on the JVM (Java Virtual Machine) and hence needs to be either written in Java itself or in any other programming/scripting language that leads to compilation of artifacts compatible with the JVM. These artifacts are usually JAR files.

The following image illustrates this very structure where Apache Maven is a bundled collection of plugins:



In this chapter, we have recipes for creating these plugins in Java, Apache Ant, and JRuby. We will also look at some advanced techniques of Java plugins to make them more intuitive and usable.

Creating a Maven plugin using Java

Given its internal architecture, Apache Maven can be defined as no more than a framework with collections of plugins. Plugins are where the real action happens. Be it compilation of code, running of tests, creation of artifacts, and so on. A plugin can have one or more goals that can be called upon explicitly from the command line or sometimes be integrated with one of the project build phases.

In this recipe, we will create a **MOJO** (**Maven plain Old Java Object**), build it, include it in another project, and execute it from the command line.

Getting ready

For this recipe, you need Apache Maven 3 installed and set up correctly and you need to be familiar with Maven concepts such as archetypes, project co-ordinates, dependencies, and so on.

How to do it...

We start by generating an Apache Maven plugin project using an archetype. Archetypes, as discussed earlier, are project templates, and the Maven community has made an archetype available for generating a basic Maven plugin project.

1. Start your console and execute the following command:

   ```
   $ mvn archetype:generate -DgroupId=net.srirangan.packt.maven
   -DartifactId=maven-plug101-plugin -DarchetypeGroupId=org.apache.
   maven.archetypes -DarchetypeArtifactId=maven-archetype-mojo
   ```

2. Now build and install this project in your local repository with the following command:

   ```
   $ mvn clean install
   ```

3. Open the `MyMojo.java` file in any text editor. If you are using an IDE (such as IntelliJ IDEA or Eclipse), you can start by importing the project and then opening `MyMojo.java` for editing.

   You will see that the source code for the MOJO is already populated. We will replace that code with an even simpler example, as shown in the following code:

   ```
   import org.apache.maven.plugin.AbstractMojo;
   import org.apache.maven.plugin.MojoExecutionException;

   /**
     * @goal helloworld
   */
   public class MyMojo extends AbstractMojo {
     public void execute() throws MojoExecutionException {
       getLog().info("Hello, world.");
     }
   }
   ```

4. Let's now modify the POM file (`pom.xml`) to include this plugin:

   ```
   ...
   <build>
     <plugins>
       <plugin>
         <groupId>net.srirangan.packt.maven</groupId>
         <artifactId>maven-plug101-plugin</artifactId>
         <version>1.0-SNAPSHOT</version>
       </plugin>
     </plugins>
   </build>
   ...
   ```

5. We can now execute the plugin using the fully qualified goal command:

```
$ mvn net.srirangan.packt.maven:maven-plug101-plugin:1.0-
SNAPSHOT:helloworld
```

The result should be:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Building maven-plug101-plugin Maven Mojo 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
[INFO] --- maven-plug101-plugin:1.0-SNAPSHOT:helloworld (default-
cli) @ maven-plug101-plugin ---
[INFO] Hello, world.
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
```

How it works...

You just executed a five step process to create your first Apache Maven plugin. Here are explanations for each step described in the preceding section:

1. Here we executed an Apache Maven `archetype:generate` command using the `maven-archetype-mojo` archetype as your project template. After a successful execution, this command created a sample MOJO class file embedded inside the Apache Maven project structure. The end result should look like this:

```
└──pom.xml
└──src
    └──main
        └──java
            └──net
                └──srirangan
                    └──packt
                        └──maven
                            └──MyMojo.java
```

2. Here we executed Maven `clean` and `install` on the newly-generated Apache Maven plugin project. This command cleans and builds the project and would have created the `target` folder which contains the artifact (JAR file) along with test reports, package structures, and `.class` files. It also installs the plugin in your local repository.

```
└──target
    ├──maven-plug101-plugin-1.0-SNAPSHOT.jar
    ├──classes
```

```
    |  ├──META-INF
    |  |  └──maven
  |  └──net
    |      └──srirangan
    |          └──packt
    |              └──maven
    ├──maven-archiver
    └──surefire
```

Now, let's have a look at the Maven plugin project's POM file. We see that it has a dependency on the Maven plugin API along with a JUnit dependency and typical project co-ordinate information.

```
<project>

  <modelVersion>4.0.0</modelVersion>

  <groupId>net.srirangan.packt.maven</groupId>
  <artifactId>maven-plug101-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <name> maven-plug101-plugin Maven Mojo</name>

  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>2.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

3. Here we modified the source of your Maven plugin (`MyMojo.java`). Do note that:
   - The `MyMojo` class extends `org.apache.maven.plugin.AbstractMojo`. `AbstractMojo` is part of the Maven plugin API and it provides infrastructure required to implement a MOJO.
   - You are required to implement the `execute()` method.
   - The `@goal` annotation is used inside a comment which is different from typical annotation usage in Java 6.

4. In this step, we included our plugin into a Maven project by appending to the `<Build><Plugins>` element in the `pom.xml`.
5. In the fifth step, we executed our plugin on the command line. Note that we used the fully qualified plugin name, which included the complete package structure.

   You have just created your first plugin, specified a custom goal, included the plugin in a project, and executed it from the command line. Not a bad start! :-)

Making your Java Maven plugin useful

This recipe is a continuation of the preceding recipe in which we saw how to create an Apache Maven plugin in Java.

We ran an `archetype:generate`, a Maven command to create a Maven plugin project and modified the `MyMojo.java` source and the POM file. We then built, installed, and executed the plugin from the command line.

However, in a real-world scenario, in addition to what we explored in the preceding recipe, your plugin may need:

- Support for multiple goals
- External configuration
- Short commands
- Execution in build phase

In this recipe, we take this to its logical conclusion. We have a fully-functional plugin and will understand how to tweak, reconfigure, and enhance it further.

Getting ready

A prerequisite for the recipe is the preceding recipe, *Creating a Maven plugin using Java*. You need to complete that recipe, create the Maven plugin project, and then continue with this recipe.

How to do it...

Follow the ensuing steps to support multiple goals in your Apache Maven plugin:

1. In our `maven-plug101-plugin` project, let us create a new file called `MyMojo2.java`:

```
package net.srirangan.packt.maven;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
```

```
/**
  * @goal wassup
*/
public class MyMojo2 extends AbstractMojo {
    public void execute() throws MojoExecutionException {
        getLog().info("Wassup");
    }
}
```

2. The plugin project can now be built, and plugin goals executed as follows:

```
$ mvn install
$ mvn plug101:wassup
```

3. To enable your Apache Maven plugin to be externally configurable, you need to make use of the parameter annotation (`@parameter`) in the MOJO while defining it, as shown in the following code:

```
import java.util.Properties;

/**
  * @parameter
*/
private Properties myProperties;
```

These are not typical Java 6 annotations and you need to encapsulate them in the `/** **/` comments tag.

4. Our next step is to modify the project's POM file to include the configuration for `myProperties`. You will need to add a configuration element to your existing `<build><plugins><plugin>` structure, as shown in the following code:

```
  <build>
  <plugins>
    <plugin>
    ...
    <configuration>
      <myProperties>
        <property>
        <name>name</name>
        <value>Sri</value>
        <property>
        <property>
        <name>age</name>
        <value>26</value>
        <property>
```

```
      </myProperties>
    </configuration>
    </plugin>
  </plugins>
```

`myProperties` is now available for use in your MOJO and its values are read during execution from the POM file.

For a plugin that needs to be executed from the console, a short command line is preferable. Here's a quick recap of how we executed the plugin from the command line:

```
$ mvn net.srirangan.packt.maven:maven-plug101-plugin:1.0-
SNAPSHOT:helloworld
```

A little too long, don't you think? Remember how we ran the Jetty server through the plugin in earlier chapters?

```
$ mvn jetty:run
```

Short and sweet. How do we do the same, that is, shorten the command-line call with our plugin? (We're off to a good start, so the process of shortening won't be very hard.)

- Firstly, we have the latest version of the plugin installed in our local repository. That is what we're trying to execute. We can directly do away with the version number, and our command becomes:

  ```
  $ mvn net.srirangan.packt.maven:maven-plug101-
  plugin:helloworld
  ```

- We also followed the Maven plugin naming conventions, that is, `maven-{plugin_name}-plugin`. This allows us to shorten it a bit further.
- Finally, add the plugin's `groupId` as a `pluginGroup` in the Maven settings file (`settings.xml`):

  ```
  <pluginGroups>
    <pluginGroup>net.srirangan.packt.maven</pluginGroup>
  </pluginGroups>
  ```

- We should now be able to run the plugin using a short command in the console:

  ```
  $ mvn plug101:helloworld
  ```

5. Here's how you integrate the Apache Maven plugin with the project build lifecycle:

   You will need to modify the project's POM file and include the `executions` element in the `build` and `plugin` description. The following code consists of the integration of the plugin with the `compile` phase:

   ```
   <build>
     <plugins>
       <plugin>
   ```

```
        <groupId>net.srirangan.packt.maven</groupId>
        <artifactId>maven-plug101-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
        <executions>
          <execution>
            <phase>compile</phase>
            <goals>
              <goal>helloworld</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

- Now the `plug101:helloworld` goal is executed as part of the Maven compile phase:

  ```
  $ mvn compile
  ```

How it works...

- Multiple goals: The Maven plugin API and Maven plugin project inherently support multiple goals. A parallel can be drawn as one goal equals to one MOJO. Each MOJO refers to a goal. Therefore, your plugin project can define N number of MOJOs for N number of goals.

  `MyMojo2.java` will complement the original `MyMojo.java` file. Our plugin now supports two goals, namely, `hello world` and `wassup`.

- External configuration: The approach for creating externally configurable plugins is to create parameters in the MOJO and expose them via annotations. These parameters can be individual attributes of the data types like String, Boolean, Integer, Double, Date, File, URL, and so on.

  However, if your plugin needs multiple configurable parameters, it is recommended that you use "Properties", that is, `java.util.Properties`.

- Properties let you define N number of name-value pairs as XML tags which can then be included in your POM file and thereafter be configured as required.

- Execution in build phase: Often you'll find the need to integrate a plugin and goal with the default Maven project lifecycle. Why is this important when you can always run the plugin from the console?

  It is important because a Maven project doesn't live in isolation. Often, you find it integrated with third-party systems in the development ecosystem such as the IDE, SCM, or the continuous integration server. These systems may run only the default build phases and you want them to automatically invoke the plugin goal.

Another reason for integration is that, just like a Maven project, you aren't working in isolation. If you need this plugin executed, say at the test phase, isn't it easier for you and your team to be able to run the plugin as part of your existing workflow and not change the day-to-day workflow to include an additional command?

Documenting your Maven plugin

It is not an exaggeration when it's said that Apache Maven is being used by hundreds and thousands of developers, if not more, everyday.

In addition, Apache Maven has a thriving community and hundreds of individuals, groups, and teams that contribute to Apache Maven in the form of plugins.

However, a common complaint in the Apache Maven community discussions has been the lack of sufficient documentation, especially for Apache Maven plugins.

Thus the Apache Maven community recommends a set of plugin documentation guidelines for plugin developers.

Adhering to these guidelines helps plugin users and thus, in turn, fuels the greater adoption of the plugin.

Getting ready

For this recipe, you need to have an existing Apache Maven plugin project.

In addition, you should also be familiar with the Apache Maven site plugin and the overall reporting/documentation process in an Apache Maven project.

How to do it...

1. Add the Maven plugin to the `reporting` element of the project POM file `pom.xml`, as shown in the following code:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-plugin-plugin</artifactId>
      <version>2.5.1</version>
    </plugin>
  </plugins>
</reporting>
```

Documentation for an Apache Maven plugin project can be automatically generated. The command for doing so is as follows:

```
$ mvn site
```

2. Verify that plugin documentation adheres to the Maven community standards by running the command:

```
$ mvn docck:check
```

How it works...

The `mvn site` command will automatically generate a plugin site for your plugin project based on available information in the project POM file, the `src/site` folder, and available reporting plugins.

The Maven plugin is an important reporting plugin as it leads to the generation of documentation for each MOJO and thus for each plugin goal.

The first step for ensuring good documentation is to achieve completeness of the project's POM file. Make sure all optional elements in the project object model are filled in including organization details, names, descriptions, URLs, mailing lists, licenses, issue/bug trackers, source code repositories, and so on.

There's more

Additional recommended reporting plugins for your Maven plugin project are:

- Maven Javadoc plugin
- Maven JXR plugin

The following example shows how both can be included in your `pom.xml` file in the `reporting` element:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <minmemory>128m</minmemory>
        <maxmemory>512</maxmemory>
      </configuration>
    </plugin>
```

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jxr-plugin</artifactId>
        <version>2.1</version>
      </plugin>
    </plugins>
  </reporting>
```

Creating a Maven plugin using Ant

Apache Ant is a software build automation tool for the Java platform. It uses an XML build configuration file (similar to Apache Maven and dissimilar to Make) for defining project builds.

Ant is extremely popular. Not only is it lightweight and an occasional replacement for Maven itself, but it inherently is easy to use for non-programmers. This is because it consists of instructions written in XML rather than any programming language.

If you want to create an extension for Maven, that is, create a Maven plugin in Apache Ant, then this recipe is for you.

Getting ready

For this recipe, you need Apache Maven 3 installed and set up correctly. You also need to be familiar with Maven concepts such as archetypes, project co-ordinates, dependencies, and so on.

Additionally, you need to be familiar with Apache Ant and the overall reporting/documentation process in an Apache Maven project.

How to do it...

Let's get started.

1. Create a new project named `maven-plugant-plugin` with the following folder tree structure thereafter:

```
└──src
    └──main
        └──scripts
```

2. Create an XML file named `src/main/scripts/hello.build.xml` with the following contents:

```
<project>
  <target name="hello">
    <echo>Hello, World</echo>
  </target>
</project>
```

3. Follow this by creating `src/main/scripts/hello.mojos.xml` with the following contents:

```
<pluginMetadata>
  <mojos>
    <mojo>
      <goal>hello</goal>
      <call>hello</call>
      <description>
        Say Hello, World.
      </description>
    </mojo>
  </mojos>
</pluginMetadata>
```

4. Finally, in the project's root folder, create a POM file (`pom.xml`) with the following Maven project configuration:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>net.srirangan.packt.maven</groupId>
  <artifactId>maven-plugant-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>
  <name>maven-plugant-plugin Plugin</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-script-ant</artifactId>
      <version>2.0.6</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.6</version>
        <dependencies>
          <dependency>
            <groupId>org.apache.maven.plugin-tools</groupId>
            <artifactId>maven-plugin-tools-ant</artifactId>
            <version>2.5</version>
          </dependency>
        </dependencies>
        <configuration>
          <goalPrefix>hello</goalPrefix>
```

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

5. The next step is to build and install the project into your Maven local repository:

   ```
   $ mvn install
   ```

6. On build success, you can now execute the plugin from the command line:

   ```
   $ mvn net.srirangan.packt.maven:maven-plugant-plugin:hello
   ```

How it works...

`hello.mojos.xml` contains the definition and configuration of the Mojo. It is the one that associates the `hello` goal with the `hello.build.xml` Ant build script.

The project POM file shows a dependency on Ant itself (of course!). It also makes use of `maven-plugin-plugin` with a dependency on `maven-plugin-tools-ant`.

These together allow Maven plugins to be written in Ant and located in the `src` folder to be discovered, packaged, and installed.

If you look at your local repository, you will find that the Ant plugin has been converted into an artifact (`.jar`) at `repository/net/srirangan/packt/maven/maven-plugant-plugin/1.0-SNAPSHOT/maven-plugant-plugin-1.0-SNAPSHOT.jar`.

Creating a Maven plugin using JRuby

JRuby is a 100 percent pure Java implementation of the popular scripting and programming language–Ruby. JRuby shares a number of features from Ruby like being dynamic and reflective. To its audience, JRuby/Ruby is close to being the best thing since sliced bread.

Syntactically, JRuby/Ruby makes it easy to get started and to quickly accomplish programming tasks that would otherwise take you a while with Java.

But since JRuby is based on the Java Virtual Machine, it compiles into bytecode for JVM and can interact seamlessly with existing Java infrastructure, much like Groovy and Scala.

Getting ready

For this recipe, you need Apache Maven 3 installed and set up correctly. You also need to be familiar with Maven concepts such as archetypes, project co-ordinates, dependencies, and so on.

Additionally, you need to be familiar with programming/scripting in JRuby/Ruby.

How to do it...

1. Create a folder called `maven-plugjruby-plugin`. This is your plugin project folder. In this, implement the standard Apache Maven directory structure and create `src|main|scripts`.
2. Inside `scripts`, create the `hello.rb` MOJO:

```
# @goal "hello"

class Hello < Mojo

  def execute
    info "hello world"
  end

end

run_mojo Hello
```

3. The next step is to create the project POM file (`pom.xml`) in the project root folder:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>net.srirangan.packt.maven</groupId>
  <artifactId>maven-plugjruby-plugin</artifactId>
  <name>Example Ruby Mojo</name>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jruby-maven-plugin</artifactId>
      <version>1.0-beta-4</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.4</version>
        <dependencies>
          <dependency>
            <groupId>org.codehaus.mojo</groupId>
```

```
            <artifactId>jruby-maven-plugin</artifactId>
            <version>1.0-beta-4</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

4. You need to build and install the plugin into your Maven local repository:

   ```
   $ mvn install
   ```

5. After build success, the plugin is now available for execution from the command line or for further integration into other projects. To execute from the command line, try the following:

   ```
   $ mvn net.srirangan.packt.maven:maven-plugant-plugin:hello
   ```

How it works...

In the first step, we created the project structure based on the standard Apache Maven project skeleton. We followed it by creating a MOJO `hello.rb`. This MOJO class extends the Abstract Mojo and implements the execute method.

We now have the source code ready for the JRuby plugin. However, to complete the project, we create the project POM file `pom.xml`. Here, we defined the dependency to the `jruby-maven-plugin` and included it in the `<build><plugins>` element of the project object model.

Finally, we installed our Maven JRuby plugin with the `$mvn install` command and saw an example on how to execute the plugin from the command line.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

P

R

S

T

U

W

Thank you for buying

Apache Maven 3 Cookbook

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

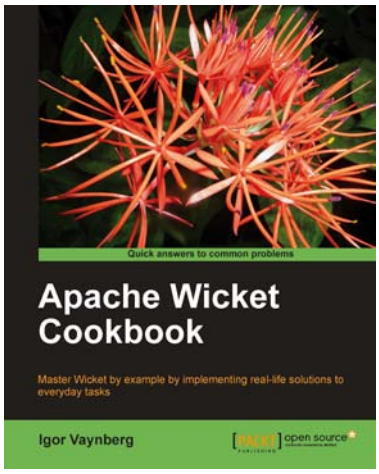
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced 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.

[PACKT] open source
community experience distilled
PUBLISHING



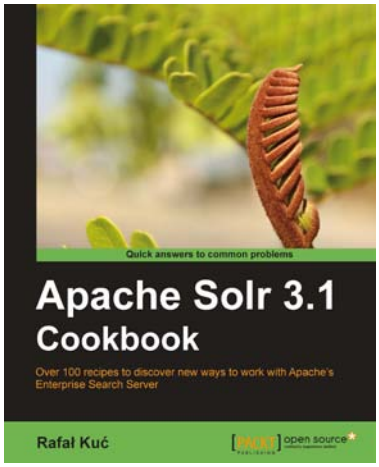
Apache Wicket Cookbook

ISBN: 978-1-84951-160-5 Paperback: 312 pages

Master Wicket by example by implementing real-life solutions to every day tasks

1. The Apache Wicket Cookbook covers the full spectrum of features offered by the Wicket web framework
2. Implement advanced user interactions by following the live examples given in this Cookbook
3. Create reusable components and speed up your web application development

---



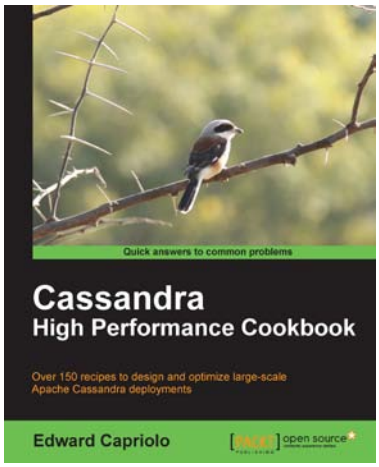
Apache Solr 3.1 Cookbook

ISBN: 978-1-84951-218-3 Paperback: 300 pages

Over 100 recipes to discover new ways to work with Apache's Enterprise Search Server

1. Improve the way in which you work with Apache Solr to make your search engine quicker and more effective
2. Deal with performance, setup, and configuration problems in no time
3. Discover little-known Solr functionalities and create your own modules to customize Solr to your company's needs
4. Part of Packt's Cookbook series; each chapter covers a different aspect of working with Solr

[PACKT] open source
community experience distilled
PUBLISHING

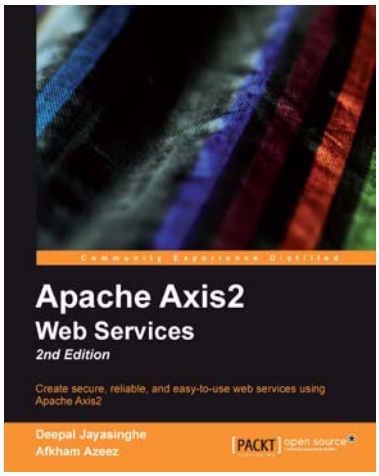


Cassandra High Performance Cookbook

ISBN: 978-1-84951-512-2 Paperback: 310 pages

Over 150 recipes to design and optimize large scale Apache Cassandra deployments

1. Get the best out of Cassandra using this efficient recipe bank
2. Configure and tune Cassandra components to enhance performance
3. Deploy Cassandra in various environments and monitor its performance
4. Well illustrated, step-by-step recipes to make all tasks look easy!

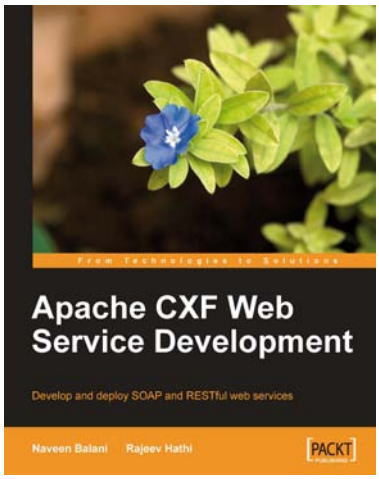


Apache Axis2 Web Services, 2nd Edition

ISBN: 978-1-84951-156-8 Paperback: 308 pages

Create secure, reliable, and easy-to-use web services using Apache Axis2

1. Extensive and detailed coverage of the enterprise ready Apache Axis2 Web Services / SOAP / WSDL engine.
2. Attain a more flexible and extensible framework with the world class Axis2 architecture.
3. Learn all about AXIOM - the complete XML processing framework, which you also can use outside Axis2.
4. Covers advanced topics like security, messaging, REST and asynchronous web services.



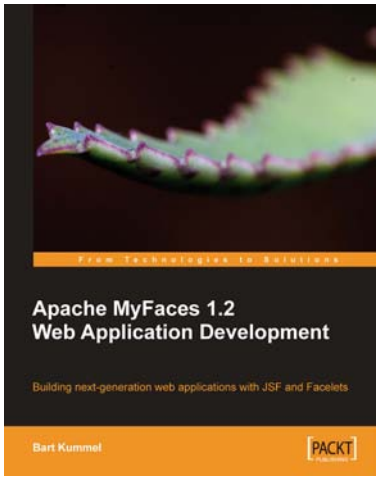
Apache CXF Web Service Development

ISBN: 978-1-847195-40-1 Paperback: 336 pages

Develop and deploy SOAP and RESTful Web Services

1. Design and develop web services using contract-first and code-first approaches
2. Publish web services using various CXF frontends such as JAX-WS and Simple frontend
3. Invoke services by configuring CXF transports
4. Create custom interceptors by implementing advanced features such as CXF Interceptors, CXF Invokers, and CXF Features

---



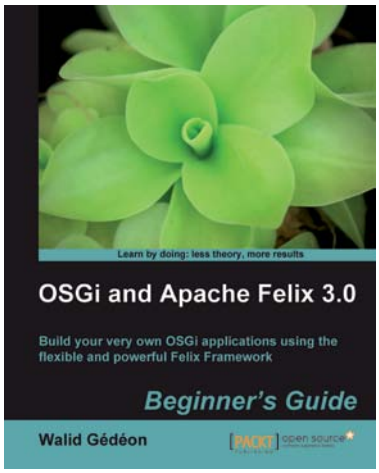
Apache MyFaces 1.2 Web Application Development

ISBN: 978-1-847193-25-4 Paperback: 408 pages

Building next-generation web applications with JSF and Facelets

1. Build powerful and robust web applications with Apache MyFaces
2. Reduce coding by using sub-projects of MyFaces like Trinidad, Tobago, and Tomahawk
3. Update the content of your site daily with ease by using Facelets

[PACKT] open source
community experience distilled
PUBLISHING

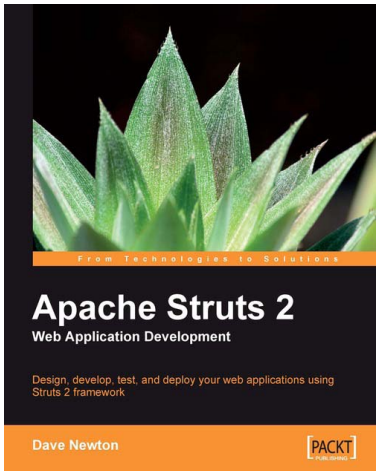


OSGi and Apache Felix 3.0 Beginner's Guide

ISBN: 978-1-84951-138-4 Paperback: 336 pages

Build your very own OSGi applications using the flexible and powerful Felix Framework

1. Build a completely operational real-life application composed of multiple bundles and a web front end using Felix
2. Get yourself acquainted with the OSGi concepts, in an easy-to-follow progressive manner
3. Learn everything needed about the Felix Framework and get familiar with Gogo, its command-line shell to start developing your OSGi applications
4. Simplify your OSGi development experience by learning about Felix iPOJO



Apache Struts 2 Web Application Development

ISBN: 978-1-847193-39-1 Paperback: 384 pages

Apache Struts 2 Web Application Development

1. Design, develop, test, and deploy your web applications using Struts 2 framework
2. No prior knowledge of JavaScript and CSS is required
3. Apply the best of agile development techniques and TDD techniques
4. Step-by-step instructions and careful explanations with lots of code examples