Apache Maven 3 Cookbook

Chapter No.2
"Software Engineering Techniques"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.2 "Software Engineering Techniques"
A synopsis of the book’s content
Information on where to buy this book

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.

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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
Apache Maven 3 Cookbook

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.

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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.

For More Information:  
www.packtpub.com/apache-maven-3-0-cookbook/book
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.

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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:

```bash
$ 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
```

For More Information:

www.packtpub.com/apache-maven-3-0-cookbook/book
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.

For More Information:

www.packtpub.com/apache-maven-3-0-cookbook/book
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
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packtpub</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>Maven Multimodule Project</name>
  <url>http://maven.apache.org</url>
</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
```

For More Information:

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

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

Our overall project structure should look like this:

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

**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.

For More Information:

www.packtpub.com/apache-maven-3-0-cookbook/book
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.

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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:

```xml
<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:

```xml
<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:

```xml
<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>
```

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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.

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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:

```xml
<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:

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

For More Information:

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:

```xml
<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.

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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.

![Diagram of TDD process]

**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`:

```xml
<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.

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

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

For More Information:

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:

```java
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
```

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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:

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

Include the relevant dependencies in the project POM:

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

For More Information:

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:

```xml
<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:

```xml
<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>
  </executions>
</plugin>
```

For More Information:
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
<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

```java
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();
  }
```

For More Information:
www.packtpub.com/apache-maven-3-0-cookbook/book
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.

```xml
<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.

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

For More Information:

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 lifecycle. 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.

```xml
<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 More Information:**

www.packtpub.com/apache-maven-3-0-cookbook/book
For deployment using SSH, note the changes in the extension `artifactId` and the URL.

```xml
<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.
Where to buy this book
You can buy Apache Maven 3 Cookbook from the Packt Publishing website: http://www.packtpub.com/apache-maven-3-0-cookbook/book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.