THE EXPERT'S VOICE® IN JAVA

Introducing Maven

Balaji Varanasi and Sudha Belida

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Introducing Maven provides a concise introduction to Maven, the de facto standard for building, managing, and automating Java and JEE-based projects in enterprises throughout the world. The book starts by explaining the fundamental concepts of Maven and showing you how to set up and test Maven on your local machine. It then delves deeply into concepts such as dependency management, life cycle phases, plug-ins, and goals. It also discusses project structure conventions, jump-starting project creation using archetypes, and documentation and report generation. Finally, it concludes with a discussion of Nexus and Maven's release process.

How This Book Is Structured

Chapter 1 starts with a gentle introduction to Maven. It discusses reasons for adopting Maven, and it provides an overview of its two alternatives: Ant and Gradle.

Chapter 2 focuses on setting up Maven on your machine and testing the installation. It also provides an overview of Maven's `settings.xml` file, and it shows you how to run Maven in a HTTP proxy-enabled environment.

Chapter 3 delves deeply into Maven's dependency management. It then discusses the GAV coordinates Maven uses for uniquely identifying its artifacts. Finally, it covers transitive dependencies and the impact they have on builds.

Chapter 4 discusses the organization of a basic Maven project and covers the important elements of a `pom.xml` file. Then you learn about testing the project using JUnit.

Chapter 5 provides detailed coverage of Maven's life cycle, plug-ins, build phases, and goals. It then walks you through the process of creating and using a simple Maven plug-in.

Chapter 6 introduces archetypes' project templates that enable you to bootstrap new projects quickly. The built-in archetypes are used to generate a Java project, a web project, and a multimodule project. You will then create a custom archetype from scratch and use it to generate a new project.

Chapter 7 covers the basics of site generation using Maven. It then discusses report generation and documentation such as Javadocs, test coverage reports, and FindBugs reports and how to integrate them into a Maven site.

Chapter 8 begins with a discussion of the Nexus repository manager and shows you how it can be integrated with Maven. It then provides complete coverage of Maven's release process and its different phases.

Target Audience

Introducing Maven is intended for developers and automation engineers who would like to get started quickly with Apache Maven. This book assumes basic knowledge of Java. No prior experience with Maven is required.

Downloading the Source Code

The source code for the examples in this book can be downloaded from www.apress.com/9781484208427. The source code is also available on GitHub at https://github.com/bava/gswm-book.

Once downloaded, unzip the code and place the contents in the `C:\apress\gswm-book folder`. The source code is organized by individual chapters. Where applicable, the chapter folders contain the `gswm` project with the bare minimum files to get you started on that chapter's code listings. The chapter folders also contain a folder named `final`, which holds the expected end state of the project(s).

Questions

We welcome reader feedback. If you have any questions or suggestions, you can contact the authors at Balaji@inflinx.com or Sudha@inflinx.com.

CHAPTER 1

Getting Started with Maven

Like other craftsmen, software developers rely on their tools to build applications. Developer's integrated development environments (IDEs), bug-tracking tools, build tools, frameworks, and debug tools, such as memory analyzers, play a vital role in day-to-day development and maintenance of quality software. This book will discuss and explore the features of Maven, which we know will become an important tool in your software development arsenal.

Apache Maven is an open source, standards-based project management framework that simplifies the building, testing, reporting, and packaging of projects. Maven's initial roots were in the Apache Jakarta Alexandria project that took place in early 2000. It was subsequently used in the Apache Turbine project. Like many other Apache projects at that time, the Turbine project had several subprojects, each with its own Ant-based build system. Back then, there was a strong desire for developing a standard way to build projects and to share generated artifacts easily across projects. This desire gave birth to Maven. Maven version 1.0 was released in 2004, followed by version 2.0 in 2005. At the time of writing this book, 3.0.5 is the current version of Maven.

Maven has become one of the most widely used open source software programs in enterprises around the world. Let's look at some of the reasons why Maven is so popular.

Standardized Directory Structure

Often, when we start work on a new project, a considerable amount of time is spent deciding on the project layout and folder structure needed to store code and configuration files. These decisions can vary vastly across projects and teams, which can make it difficult for new developers to understand and adopt other teams' projects. It can also make it hard for existing developers to jump between projects and find what they are seeking.

Maven addresses the above problems by standardizing the folder structure and organization of a project. Maven provides recommendations on where different parts of a project, such as source code, test code, and configuration files, should reside. For example, Maven suggests that all of the Java source code should be placed in the `src\main\java` folder. This makes it easier to understand and navigate any Maven project.

Additionally, these conventions make it easy to switch to and start using a new IDE. Historically, IDEs varied with project structure and folder names. A dynamic web project in Eclipse might use the WebContent folder to store web assets, whereas NetBeans might use Web Pages for the same purpose. With Maven, your projects follow a consistent structure and become IDE agnostic.

Declarative Dependency Management

Most Java projects rely on other projects and open source frameworks to function properly. It can be cumbersome to download these *dependencies* manually and keep track of their versions as you use them in your project.

Maven provides a convenient way to declare these project dependencies in a separate, external `pom.xml` file. It then automatically downloads those dependencies and allows you to use them in your project. This simplifies project dependency management greatly. It is important to note that in the `pom.xml` file you specify the *what* and not the *how*. The `pom.xml` file can also serve as a documentation tool, conveying your project dependencies and their versions.

Plug-ins

Maven follows a *plug-in–based architecture*, making it easy to augment and customize its functionality. These plug-ins encapsulate reusable build and task logic. Today, there are hundreds of Maven plug-ins available that can be used to carry out tasks ranging from code compilation to packaging to project documentation generation.

Maven also makes it easy to create your own plug-ins, thereby enabling you to integrate tasks and workflows that are specific to your organization.

Uniform Build Abstraction

Maven provides a uniform interface for building projects. You can build a Maven project by using just a handful of commands. Once you become familiar with Maven's build process, you can easily figure out how to build other Maven projects. This frees developers from having to learn build idiosyncrasies so they can focus more on development.

Tools Support

Maven provides a powerful command-line interface to carry out different operations. All major IDEs today provide excellent tool support for Maven. Additionally, Maven is fully integrated with today's continuous integration products such as Jenkins, Bamboo, and Hudson.

Archetypes

As we already mentioned, Maven provides a standard directory layout for its projects. When the time comes to create a new Maven project, you need to build each directory manually, and this can easily become tedious. This is where Maven archetypes come to rescue. *Maven archetypes* are predefined project templates that can be used to generate new projects. Projects created using archetypes will contain all of the folders and files needed to get you going.

Archetypes is also a valuable tool for bundling best practices and common assets that you will need in each of your projects. Consider a team that works heavily on Spring framework-based web applications. All Spring-based web projects share common dependencies and require a set of Spring configuration files. It is also highly possible that all of these web projects have similar `Log4j/Logback` configuration files, `CSS/Images`, and Apache Tile layouts or SiteMesh decorators. Maven lets this team bundle these common assets into an archetype. When new projects get created using this archetype, they will automatically have the common assets included. No more copy and pastes or drag and drops required.

Open Source

Maven is *open source* and costs nothing to download and use. It comes with rich online documentation and the support of an active community. Additionally, companies such as Sonatype offer commercial support for the Maven ecosystem.

CONVENTION OVER CONFIGURATION

Convention over configuration (CoC) or *coding by convention* is one of the key tenants of Maven. Popularized by the Ruby on Rails community, CoC emphasizes sensible defaults, thereby reducing the number of decisions to be made. It saves time and also results in a simpler end product, as the amount of configuration required is drastically reduced.

As part of its CoC adherence, Maven provides several sensible defaults for its projects. It lays out a standard directory structure and provides defaults for the generated artifacts. Imagine looking at a Maven artifact with the name `log4j-1.4.3.jar`. At a glance, you can easily see that you are looking at a `log4j` JAR file, version 1.4.3.

One drawback of Maven's CoC is the rigidness that end users experience when using it. To address this, you can customize most of Maven's defaults. For example, it is possible to change the location of the Java source code in your project. As a rule of thumb, however, such changes to defaults should be minimized.

Maven Alternatives

Although the emphasis of this book is on Maven, let's look at a couple of its alternatives: Ant + Ivy and Gradle.

Ant + Ivy

Apache Ant (http://ant.apache.org) is a popular open source tool for scripting builds. Ant is Java based, and it uses Extensible Markup Language (XML) for its configuration. The default configuration file for Ant is the `build.xml` file.

Using Ant typically involves defining tasks and targets. As the name suggests, an *Ant task* is a unit of work that needs to be completed. Typical tasks involve creating a directory, running a test, compiling source code, building a web application archive (WAR) file, and so forth. A *target* is simply a set of tasks. It is possible for a target to depend on other targets. This dependency lets us sequence target execution. Listing 1-1 demonstrates a simple `build.xml` file with one target called *compile*. The `compile` target has two `echo` tasks and one `javac` task.

Listing 1-1. Sample Ant `build.xml` File

```
<project name="Sample Build File" default="compile" basedir=".">

   <target name="compile" description="Compile Source Code">
      <echo message="Starting Code Compilation"/>
      <javac srcdir="src" destdir="dist"/>
      <echo message="Completed Code Compilation"/>
   </target>

</project>
```

Ant doesn't impose any conventions or restrictions on your project and it is known to be extremely flexible. This flexibility has sometimes resulted in complex, hard-to-understand and maintain `build.xml` files.

Apache Ivy (http://ant.apache.org/ivy/) provides automated dependency management, making Ant more joyful to use. With Ivy, you declare the dependencies in an XML file called `ivy.xml`, as shown in Listing 1-2. Integrating Ivy with Ant involves declaring new targets in the `build.xml` file to retrieve and resolve dependencies.

Listing 1-2. Sample Ivy Listing

```
<ivy-module version="2.0">
  <info organisation="com.apress" module="gswm-ivy" />

  <dependencies>
        <dependency org="org.apache.logging.log4j" name="log4j-api"
rev="2.0.2" />
  </dependencies>
</ivy-module>
```

Gradle

Gradle (http://gradle.org/) is the newest addition to the Java build project automation tool family. Unlike Ant and Maven, which use XML for configuration, Gradle uses a Groovy-based *Domain Specific Language* (DSL).

Gradle provides the flexibility of Ant, and it uses the same notion of tasks. It also follows Maven's conventions and dependency management style. Listing 1-3 shows a default `build.gradle` file.

Listing 1-3. Default `build.gradle` File

```
apply plugin: 'java'

version = '1.0'

repositories {
   mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.10'
}
```

Gradle's DSL and its adherence to CoC results in compact build files. The first line in Listing 1-3 includes a Java plug-in for `build`'s use. Plug-ins in Gradle provide preconfigured tasks and dependencies to the project. The Java plug-in, for example, provides tasks for building source files, running unit tests, and installing artifacts. The dependencies section in the `default.build` file instructs Gradle to use `JUnit` dependency during the compilation of test source files. Gradle's flexibility, like that of Ant, can be abused, which results in difficult and complex builds.

Summary

Apache Maven greatly simplifies the build process and automates project management tasks. This chapter provided a gentle introduction to Maven and described the main reasons for adopting it. We also looked at Maven's close peers: Ant + Ivy and Gradle.

In the next chapter, you will learn about the set up required to get up and running with Maven.

CHAPTER 2

Setting Up Maven

Maven installation is an easy and straightforward process. This chapter will explain how to install and set up Maven using the Windows 7 operating system. You can follow the same procedure with other operating systems.

Note Maven is a Java-based application and requires the Java Development Kit (JDK) to function properly. Maven version 3.2 requires JDK 1.6 or above and versions 3.0/3.1 can be run using JDK 1.5 or above. Before proceeding with Maven installation, make sure that you have Java installed. If not, install the JDK (not just Java Runtime Environment [JRE]) from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. In this book, we will be using JDK 1.7.

You will begin the installation process by downloading the latest version of Maven from the Apache Maven web site (`http://maven.apache.org/download.html`). At the time of this writing, the latest version is 3.2.3. Download the Maven 3.2.3 binary `.zip` file as shown in Figure 2-1.

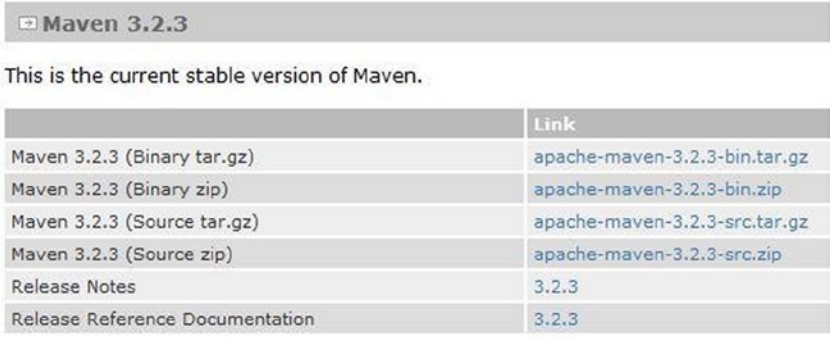


Figure 2-1. *Maven download page*

Once the download is complete, unzip the distribution to a local directory on your computer. It will create a folder named `apache-maven-3.2.3-bin`. This book assumes that you have placed the contents of `apache-maven-3.2.3-bin` folder under `c:\tools\maven directory`, as shown in Figure 2-2.

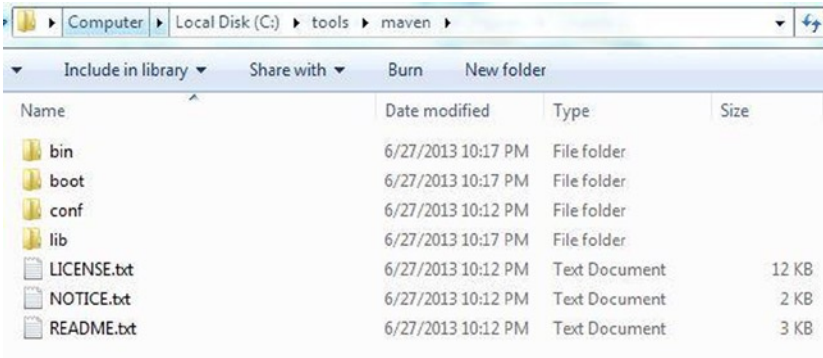


Figure 2-2. *Maven install location*

The next step in the installation process is to add the `M2_HOME` environment variable pointing to the Maven installation directory, in our case `c:\tools\maven`. Launch the Start menu, and right-click the Computer option. Next select System Properties followed by the Advanced system settings. This will launch the window shown in Figure 2-3.

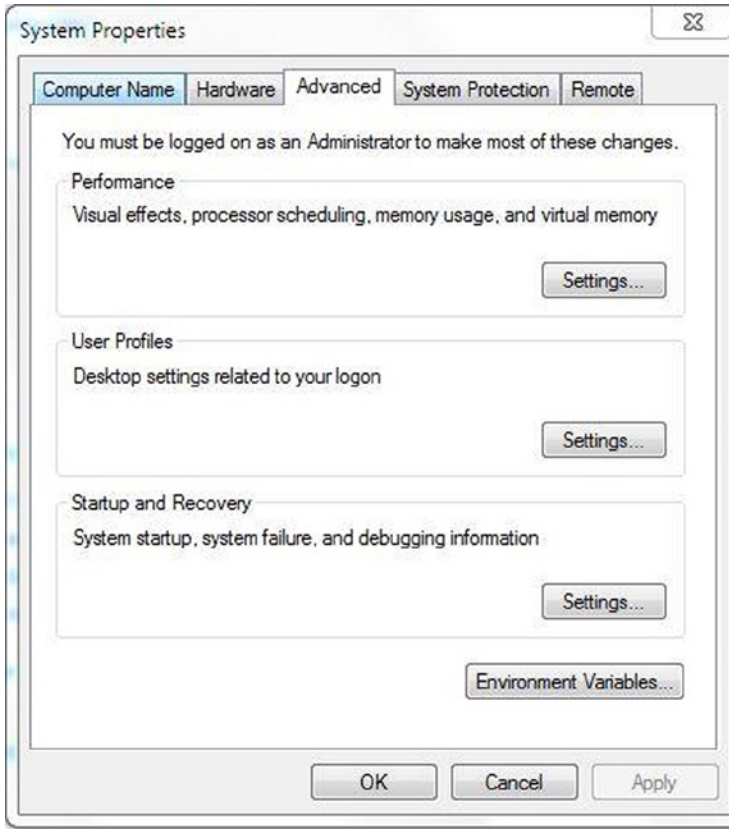


Figure 2-3. *System Properties window*

Click the Environment Variables button, and then click New under System variables. Enter the values shown in Figure 2-4 and click OK.

Figure 2-4. *Maven Home system variable*

The final step in the process is to modify the Path Environment variable so that you can run Maven commands from the command line. Select the Path variable and click Edit. Add `%M2_HOME%/bin` at the beginning of the path value, as shown in Figure 2-5. Click OK. This completes the Maven installation. If you have any open command-line windows, close them and reopen a new command-line window. When environment variables are added or modified, new values are not propagated to open command-line windows automatically.

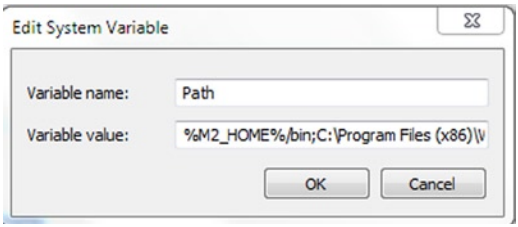


Figure 2-5. *Adding Maven Home to the path variable*

MAVEN_OPTS ENVIRONMENT VARIABLE

When using Maven, especially in a complex project, chances are that you will run into OutOfMemory errors. This may happen, for example, when you are running a large number of JUnit tests or when you are generating a large number of reports. To address this error, increase the heap size of the Java virtual machine (JVM) used by Maven. This is done globally by creating a new environment variable called `MAVEN_OPTS`. To begin, we recommend using the value `-Xmx512m`.

Testing Installation

Now that Maven is installed, it's time to test and verify the installation. Open a Command Prompt and run the following command:

```
mvn –v
```

This command should output information similar to the following:

```
C:\Windows\System32>mvn -v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4;
2014-08-11T14:58:10-06:00)
Maven home: c:\tools\maven
Java version: 1.7.0_25, vendor: Oracle Corporation
Java home: C:\Java\jdk1.7.0_25\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "x86", family: "windows"
```

The -v command-line option tells the path where Maven is installed and what Java version it is using. You would also get the same results by running the expanded command `mvn --version`.

Getting Help

You can get a list of Maven's command-line options by using the -h or `--help` options. Running the command below will produce output similar to that shown in Figure 2-6.

```
mvn -h
```

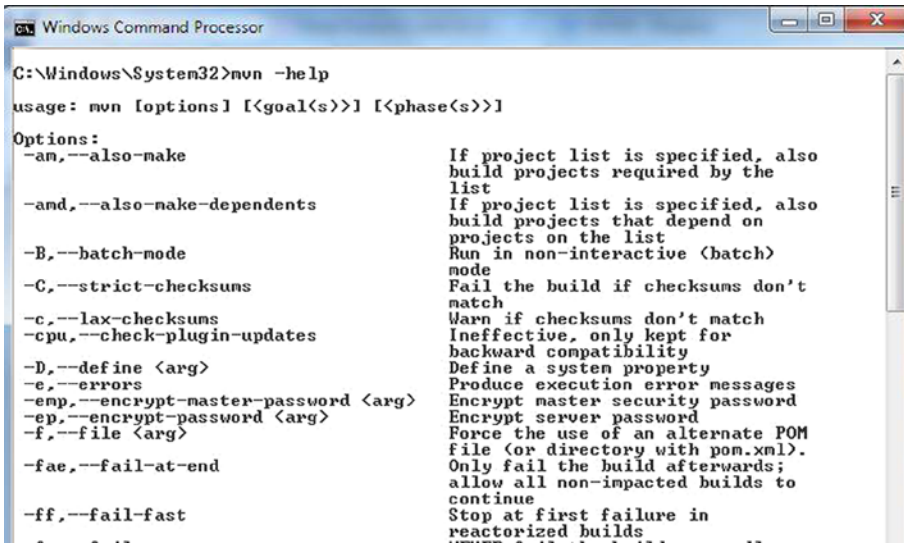


Figure 2-6. *Results of running Maven Help command*

Additional Settings

The installation steps we have provided so far are enough to get you started with Maven. However, for most enterprise uses, you need to provide additional configuration information. This user-specific configuration is provided in a `settings.xml` file located in the `c:\Users\<<user_name>>\.m2` folder.

■ **Note** The `.m2` folder is important to Maven's smooth operation. Among many things, this folder houses a `settings.xml` file and a repository folder. The repository folder contains plug-in `JAR` files and metadata that Maven requires. It also contains the project-dependent JAR files that Maven downloaded from the Internet. We will take a closer look at this folder in Chapter 3.

By default, the `.m2` folder is located in your home directory. In Windows, this directory is usually `c:\Users\<<your_user_name>>`. Maven automatically creates the `.m2` folder. If you don't see this folder on your computer, however, go ahead and create one.

Out of the box, the `.m2` folder does not contain a `settings.xml` file. In the `.m2` folder on your local computer, create a `settings.xml` file and copy the contents of the skeleton `settings.xml` file as shown in Listing 2-1. We will cover some of these elements in the coming chapters. A brief description of the elements is provided in Table 2-1.

Listing 2-1. Skeleton `Settings.xml` Contents

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                          http://maven.apache.org/xsd/settings-1.0.0.xsd">
      <localRepository/>
      <interactiveMode/>
      <usePluginRegistry/>
      <offline/>
      <pluginGroups/>
      <servers/>
      <mirrors/>
      <proxies/>
      <profiles/>
      <activeProfiles/>
</settings>
```

Table 2-1. *Details of the* `settings.xml` *Elements*

Element Name	Description
`localRepository`	Maven stores copies of plug-ins and dependencies locally in the `c:\Users\<<your_user_name>>\.m2\repository` folder. This element can be used to change the path of the local repository. For example, `<localRepository>c:\mavenrepo</localRepository>` will change the repository location to the `mavenrepo` folder.
`interactiveMode`	As the name suggests, when this value is set to `true`, the default value, Maven interacts with the user for input.
`offline`	When set to `true`, this configuration instructs Maven to operate in an offline mode. The default is `false`.
`servers`	Maven can interact with a variety of servers, such as Apache Subversion (SVN) servers, build servers, and remote repository servers. This element allows you to specify security credentials, such as the username and password, which you need to connect to those servers.
`mirrors`	As the name suggests, mirrors allow you to specify alternate locations for your repositories.
`proxies`	proxies contains the HTTP proxy information needed to connect to the Internet.
`profiles`	profiles allow you to group certain configuration elements, such as repositories and `pluginRepositories`.
`activeProfile`	The `activeProfile` allows you to specify a default profile to be active for Maven to use.

Setting Up a Proxy

As we will discuss in detail in Chapter 3, Maven requires an Internet connection to download plug-ins and dependencies. Some companies employ HTTP proxies to restrict access to the Internet. In those scenarios, running Maven will result in *Unable to download artifact* errors. To address this, edit the `settings.xml` file and add the proxy information specific to your company. A sample configuration is shown in Listing 2-2.

Listing 2-2. Settings.xml with Proxy content

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <proxies>
    <proxy>
      <id>companyProxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.company.com</host>
      <port>8080</port>
      <username>proxyusername</username>
      <password>proxypassword</password>
      <nonProxyHosts />
    </proxy>
  </proxies>
 </settings>
```

IDE Support

Throughout this book, we will be using the command line to create and build sample applications. If you are interested in using an IDE, the good news is that all modern IDEs come with full Maven integration without needing any further configuration.

Summary

This chapter walked you through the setup of Maven on your local computer. You learned that Maven downloads the plug-ins and artifacts needed for its operation. These artifacts are stored in the `.m2\repository` folder. The `.m2` folder also contains the `settings.xml` file, which can be used to configure Maven's behavior.

In the next chapter, we will take a deeper look at Maven's dependency management.

CHAPTER 3

Maven Dependency Management

Enterprise-level projects typically depend on a variety of open source libraries. Consider the scenario where you want to use `Log4J` for your application logging. To accomplish this, you would go to the `Log4J` download page, download the `JAR` file, and put it in your project's `lib` folder or add it to the project's class path. As you may know already, there are a couple of problems with this approach:

1. You need to check `JAR` files into SVN so that your projects can be built on a computer other than your own.
2. The `JAR` file you downloaded might depend on a few other libraries. You would now have to hunt down all of those dependencies and add them to your project.
3. When the time comes to upgrade the `JAR` file, you need to start the process all over again.
4. It becomes difficult to share `JAR` files across teams within your organization.

To address these problems, Maven provides declarative dependency management. With this approach, you declare your project's dependencies in an external file called `pom.xml`. Maven will automatically download those dependencies and hand them over to your project for the purpose of building, testing, or packaging.

Figure 3-1 shows a high-level view of Maven's dependency management. As you can see, Maven interacts with repositories that hold artifacts and related metadata. Repositories that are typically accessed over the web are considered remote and are maintained by a third party. The default remote repository with which Maven interacts is called *Maven Central*, and it is located at `repo.maven.apache.org` and `uk.maven.org`. Maven places the downloaded artifacts in the local repository.

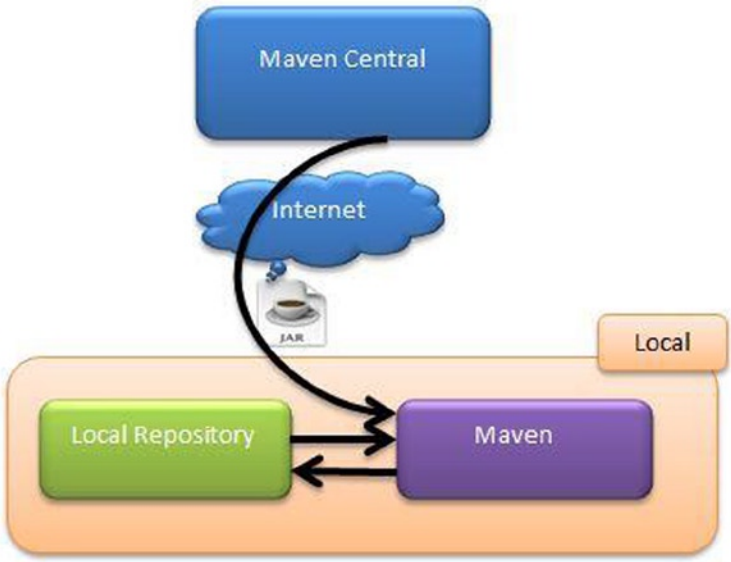


Figure 3-1. *Maven dependency management*

Although the architecture shown in Figure 3-1 works in the majority of cases, it poses a few problems in an enterprise environment. The first problem is that sharing company-related artifacts between teams is not possible. Because of security and intellectual property concerns, you wouldn't want to publish your enterprise's artifacts on Maven Central. Another problem concerns legal and licensing issues. Your company might want the teams only to use officially approved open source software, and this architecture would not fit in that model. The final issue concerns bandwidth and download speeds. In times of heavy load on Maven Central, the download speeds of Maven artifacts are reduced, and this might have a negative impact on your builds. Hence, most enterprises employ the architecture shown in Figure 3-2.

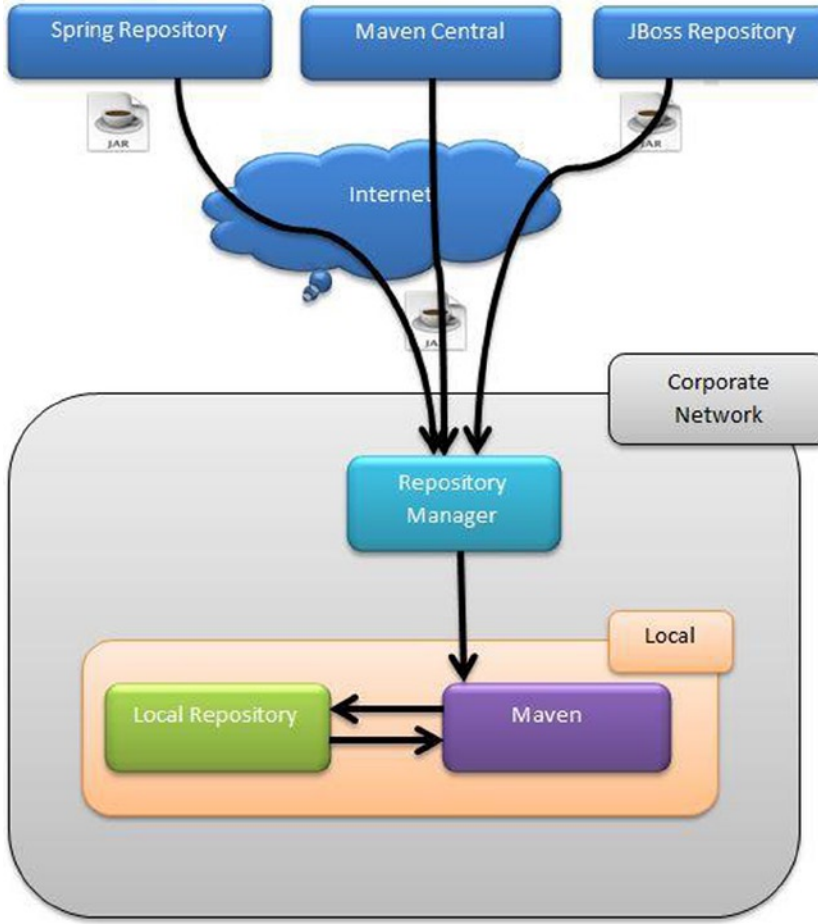


Figure 3-2. *Enterprise Maven repository architecture*

The internal repository manager acts as a proxy to remote repositories. Because you have full control over the internal repository, you can regulate the types of artifacts allowed in your company. Additionally, you can also push your organization's artifacts onto the server, thereby enabling collaboration. There are several open source repository managers. Table 3-1 lists just some of them.

Table 3-1. *Open Source Repository Managers*

Repository Manager	URL
Sonatype Nexus	www.sonatype.com/nexus
Apache Archiva	http://archiva.apache.org/
Artifactory	www.jfrog.com/open-source/

Using Repositories

In order to use a new repository, you need to modify the `settings.xml` file. Listing 3-1 shows Spring and JBoss repositories added to the `settings.xml` file. In this same way, you can add to your company's repository manager.

Listing 3-1. Adding Repositories in `settings.xml`

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

  .......

        <profiles>
            <profile>
              <id>your_company</id>
              <repositories>
                        <repository>
                          <id>spring_repo</id>
                          <url>http://repo.spring.io/release/</url>
                        </repository>
                        <repository>
                          <id>jboss_repo</id>
                          <url>https://repository.jboss.org/</url>
                        </repository>
                    </repositories>
                </profile>
        </profiles>
        <activeProfiles>
            <activeProfile>your_company</activeProfile>
        </activeProfiles>
  .......

</settings>
```

■ **Note** Information regarding repositories can be provided in the `settings.xml` or the `pom.xml` file. There are pros and cons to each approach. Putting repository information in the `pom.xml` file can make your builds portable. It enables developers to download projects and simply build them without any further modifications to their local `settings.xml` file. The problem with this approach is that when artifacts are released, the corresponding `pom.xml` files will have the repository information hard coded in them. If the repository URLs were ever to change, consumers of these artifacts will run into errors due to broken repository paths. Putting repository information in the `settings.xml` file addresses this problem, and because of the flexibility it provides, the `settings.xml` approach is typically recommended in an enterprise setting.

Dependency Identification

Maven dependencies are typically archives such as JAR, WAR, enterprise archive (EAR), and ZIP. Each Maven dependency is uniquely identified using the following group, artifact, and version (GAV) coordinates:

`groupId`*:* Identifier of the organization or group that is responsible for this project. Examples include `org.hibernate`, `log4j`, and `org.springframework.boot`.

`artifactId`*:* Identifier of the artifact being generated by the project. This must be unique among the projects using the same `groupId`. Examples include hibernate-tools, `log4j`, spring-core, and so on.

`version`*:* Indicates the version number of the project. Examples include 1.0.0, 2.3.1-SNAPSHOT, and 4.3.6.Final.

`type`*:* Indicates the packing of the generated artifact. Examples include JAR, WAR, and EAR.

Transitive Dependencies

Dependencies declared in your project's `pom.xml` file often have their own dependencies. Such dependencies are called *transitive dependencies*. Take the example of Hibernate Core. For it to function properly, it requires JBoss Logging, dom4j, javaassist, and so forth. The Hibernate Core declared in your `pom.xml` file is considered a direct dependency, and dependencies such as dom4j and javaassist are considered your project's transitive dependencies. A key benefit of Maven is that it automatically deals with transitive dependencies and includes them in your project.

Figure 3-3 provides an example of transitive dependencies. Notice that transitive dependencies can have their own dependencies. As you might imagine, this can quickly get complex, especially when multiple direct dependencies pull different versions of the same JAR file.

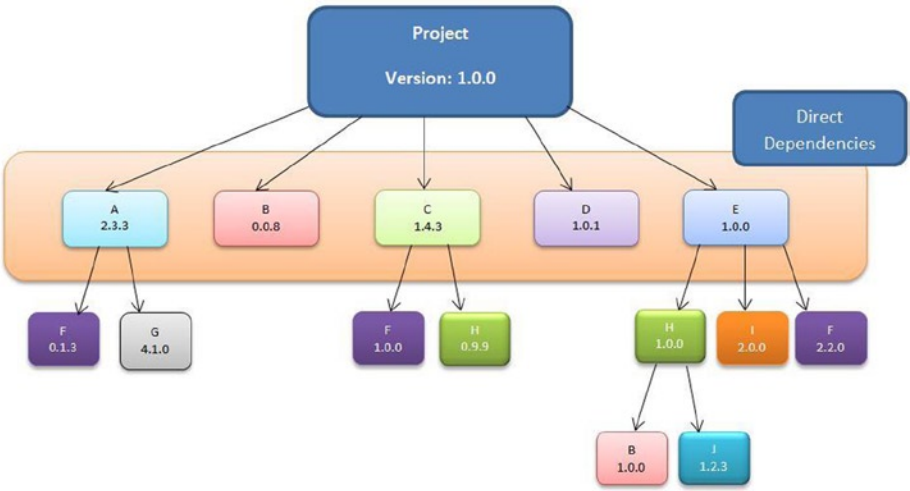


Figure 3-3. *Transitive dependencies*

Maven uses a technique known as *dependency mediation* to resolve version conflicts. Simply stated, dependency mediation allows Maven to pull the dependency that is closest to the project in the tree. In Figure 3-3, there are two versions of dependency B: 0.0.8 and 1.0.0. In this scenario, version 0.0.8 of dependency B is included in the project, because it is a direct dependency and *closest* to the tree. Now look at the three versions of dependency F: 0.1.3, 1.0.0, and 2.2.0. All three dependencies are at the same depth. In this scenario, Maven will use the *first-found dependency*, which would be 0.1.3, and not the latest 2.2.0 version.

■ **Note** Although highly useful, transitive dependencies can cause problems and unpredictable side effects, as you might end up including unwanted JAR files or older versions of JAR files. Always analyze your dependency tree and make sure you are bundling the dependencies you need and excluding the ones you don't require.

Dependency Scope

Consider a project that uses JUnit for its unit testing. The JUnit `JAR` file you included in your project is only needed during testing. You really don't need to bundle the JUnit `JAR` in your final production archive. Similarly, consider the MySQL database driver, `mysql-connector-java.jar` file. You need the dependency when you are running the application inside a container such as Tomcat. Maven uses the concept of scope, which allows you to specify when and where you need a particular dependency.

Maven provides the following six scopes:

`compile`*:* Dependencies with the `compile` scope are available in the class path in all phases on a project build, test, and run. This is the default scope.

`provided`*:* Dependencies with the `provided` scope are available in the class path during the build and test phases. They don't get bundled within the generated artifact. Examples of dependencies that use this scope include Servlet api, JSP api, and so on.

`runtime`*:* Dependencies with the `runtime` scope are not available in the class path during the build phase. Instead they get bundled in the generated artifact and are available during runtime.

`test`*:* Dependencies with the `test` scope are available during the test phase. JUnit and TestNG are good examples of dependencies with the test scope.

`system`*:* Dependencies with the `system` scope are similar to dependencies with the `provided` scope, except that these dependencies are not retrieved from the repository. Instead, a hard-coded path to the file system is specified from which the dependencies are used.

`import`*:* The `import` scope is applicable for `.pom` file dependencies only. It allows you to include dependency management information from a remote `.pom` file. The `import` scope is available only in Maven 2.0.9 or later.

Manual Dependency Installation

Ideally, you will be pulling dependencies in your projects from public repositories or your enterprise repository manager. However, there will be times where you need an archive available in your local repository so that you can continue your development. For example, you might be waiting on your system administrators to add the required `JAR` file to your enterprise repository manager.

Maven provides a handy way of installing an archive into your local repository with the install plug-in. Listing 3-2 installs a `test.jar` file located in the `c:\apress\gswm-book\chapter3` folder.

Listing 3-2. Installing Dependency Manually

```
C:\apress\gswm-book\chapter3>mvn install:install-file -DgroupId=com.apress.
gswmbook -DartifactId=test -Dversion=1.0.0 -Dfile=C:\apress\gswm-book\chapter3\
test.jar -Dpackaging=jar -DgeneratePom=true
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-install-plugin:2.4:install-file (default-cli) @ standalone-
pom---
[INFO] Installing C:\apress\gswm-book\chapter3\test.jar to C:\Users\<<USER_
NAME>>\.m2\repository\com\apress\gswmbook\test\1.0.0\test-1.0.0.jar
[INFO] Installing C:\Users\<<USER_NAME>>\AppData\Local\Temp\
mvninstall2668943665146984418.pom to C:\Users\<<USER_NAME>>\.m2\repository\
com\apress\gswmbook\test\1.0.0\test-1.0.0.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.470 s
[INFO] Finished at: 2014-10-24T20:23:36-06:00
[INFO] Final Memory: 4M/15M
```

After seeing the `BUILD SUCCESS` message, you can verify the installation by going to your local Maven repository, as shown in Figure 3-4.

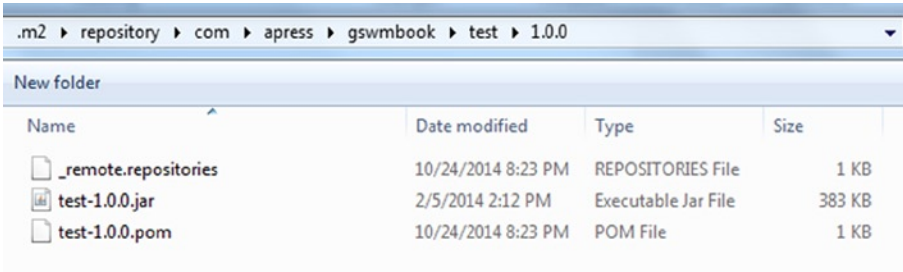


Figure 3-4. *Dependency added to repository*

Summary

Dependency management is at the heart of Maven. Every nontrivial Java project relies on open source or external artifacts, and Maven's dependency management automates the process of retrieving those artifacts and including them at the right stages of the build process. You also learned that Maven uses GAV coordinates to identify its artifacts.

In the next chapter, you will learn about the organization of a basic Maven project.

CHAPTER 4

Maven Project Basics

Maven provides conventions and a standard directory layout for all of its projects. As discussed in Chapter 1, this standardization provides a uniform build interface and it also makes it easy for developers to jump from one project to another. This chapter will explain the basics of a Maven project and the `pom.xml` file.

Basic Project Organization

The best way to understand Maven project structure is to look at one. Figure 4-1 illustrates a bare-bones Maven-based Java project.

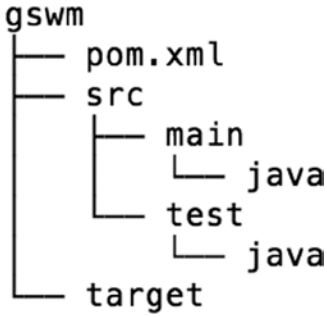


Figure 4-1. *Maven Java project structure*

Now let's look at each of the components in the project:

- The `gswm` is the root folder of the project. Typically, the name of the project matches the name of the generated artifact.
- The `src` folder contains project-related artifacts, which you typically would like to manage in a *source control management* (SCM) system, such as SVN or Git.
- The `src/main/java` folder contains the Java source code.
- The `src/test/java` folder contains the Java unit test code.

- The `target` folder holds generated artifacts, such as `.class` files. Generated artifacts are typically not stored in SCM, so you don't commit the target folder and its contents in to SCM.
- Every Maven project has a `pom.xml` file at the root of the project. It holds project and configuration information, such as dependencies and plug-ins.

In addition to the `src/main` and `src/test` directories, Maven recommends several other directories. Table 4-1 lists those directories along with the content that goes into them.

Table 4-1. *Maven Directories*

Directory Name	Description
`src/main/resources`	Holds resources, such as Spring configuration files and velocity templates, that need to end up in the generated artifact.
`src/main/config`	Holds configuration files, such as Tomcat context files, James Server configuration files, and so on. These files will not end up in the generated artifact.
`src/main/scripts`	Holds any scripts that system administrators and developers need for the application.
`src/test/resources`	Holds configuration files needed for testing.
`src/main/webapp`	Holds web assets such as `.jsp` files, style sheets, and images.
`src/it`	Holds integration tests for the application.
`src/main/db`	Holds database files, such as SQL scripts.
`src/site`	Holds files required during the generation of the project site.

Maven provides for the notion of archetypes (as discussed in Chapter 6) to bootstrap projects quickly. However, in this chapter, you will manually assemble the project. Use the instructions that follow to create the project:

1. Using a command line, go to the folder where you would like to create the project. In this book, we assume that directory to be `c:\apress\gswm-book\chapter4`.
2. Run the command `mkdir gswm`.
3. `cd` into the newly created directory, and create an empty `pom.xml` file.
4. Create the directory `src/main/java`. Create the `src` directory under `gswm`, then create the `main` directory in `src`, and finally create the `java` directory under `main`, as shown in Figure 4-2.

The starting project structure should resemble that shown in Figure 4-2.

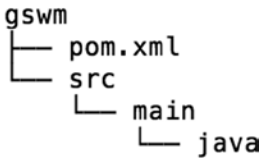


Figure 4-2. *Maven project structure*

Understanding the *pom.xml* File

The `pom.xml` file is the only required artifact in a Maven project. As we have discussed so far in the book, the `pom.xml` file holds the configuration information needed by Maven. Listing 4-1 shows the `pom.xml` file with the basic project information. We start the `pom.xml` file with the `project` element. Then we provide the `groupId`, `artifactId`, and `version` coordinates. The `packaging` element tells Maven that it needs to create a `JAR` archive for this project. Finally, we add information about the developers who are working on this project.

Listing 4-1. `pom.xml` File Configuration

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

  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Getting Started with Maven</name>
  <url>http://apress.com</url>

  <developers>
   <developer>
        <id>balaji</id>
        <name>Balaji Varanasi</name>
        <email>balaji@inflinx.com</email>
       <properties>
          <active>true</active>
       </properties>
   </developer>
```

```
   <developer>
      <id>sudha</id>
      <name>Sudha Belida</name>
      <email>sudha@inflinx.com</email>
      <properties>
          <active>true</active>
      </properties>
   </developer>
 </developers>
</project>
```

We will be looking at other elements in the `pom.xml` file later in this chapter and throughout the rest of the book.

MAVEN VERSIONING

It is recommended that Maven projects use the following conventions for versioning:

```
<major-version>.<minor-version>.<incremental-version>-qualifier
```

The major, minor, and incremental values are numeric, and the qualifier can have values such as RC, alpha, beta, and SNAPSHOT. Some examples that follow this convention are 1.0.0, 2.4.5-SNAPSHOT, 3.1.1-RC1, and so forth.

The *SNAPSHOT* qualifier in the project's version carries a special meaning. It indicates that the project is in a development stage. When a project uses a SNAPSHOT dependency, every time the project is built, Maven will fetch and use the latest SNAPSHOT artifact.

Most repository managers accept release builds only once. However, when you are developing an application in a continuous integration environment, you want to build often and push your latest build to the repository manager. Thus, it is the best practice to suffix your version with SNAPSHOT during development.

Building a Project

Before we look at building a project, let's add the HelloWorld Java class under `src/main/java` folder. Listing 4-2 shows the code for the `HelloWorld` class.

Listing 4-2. Code for `HelloWorld` Java Class

```
public class HelloWorld {
        public void sayHello() {
                System.out.print("Hello World");
        }
}
```

Figure 4-3 shows the project structure after adding the class.

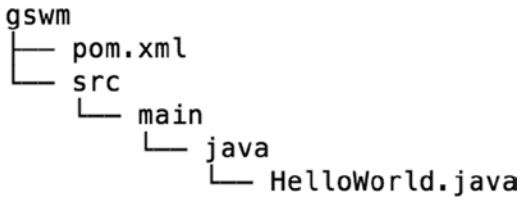


Figure 4-3. *Project structure with Java class added*

Now you're ready to build the application, so let's run the `mvn package` from `gswm`. You should see output similar to that shown in Listing 4-3.

Listing 4-3. Output for Maven Package Command for Building the Application

```
C:\apress\gswm-book\chapter4\gswm>mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ gswm
---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources,i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\apress\gswm-book\chapter4\
gswm\src\main\resources
[INFO]
```

```
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ gswm ---
[WARNING] File encoding has not been set, using platform encoding Cp1252,
i.e. build is platform dependent!
[INFO] Compiling 1 source file to C:\apress\gswm-book\chapter4\gswm\target\
classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)
@ gswm ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources,i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\apress\gswm-book\chapter4\
gswm\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile)
@ gswm---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ gswm ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ gswm ---
[INFO] Building jar: C:\apress\gswm-book\chapter4\gswm\target\gswm-1.0.0-
SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.269s
[INFO] Finished at: Mon Oct 13 21:40:44 MDT 2014
[INFO] Final Memory: 9M/23M
[INFO] ------------------------------------------------------------------------
```

■ **Note** If this is your first time running Maven, it will download the plug-ins and dependencies required for it to run. Thus, your first build might take longer than you would expect.

The *package* suffix after the `mvn` command is a Maven phase that compiles Java code and packages it into the `JAR` file. The packaged `JAR` file ends up in the `gswm\target` folder, as shown in Figure 4-4.

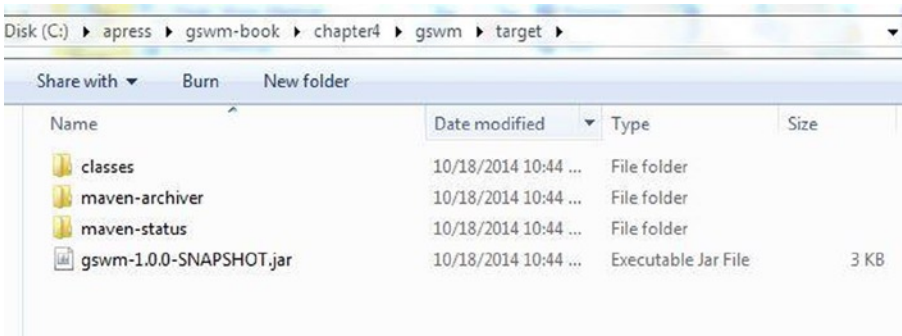


Figure 4-4. *Packaged JAR located under the target folder*

Testing the Project

Now that you have completed the project build, let's add a `JUnit` test that tests the `sayHello()` method. Let's start this process by adding `JUnit` dependency to the `pom.xml` file. You accomplish this by using the `dependencies` element. Listing 4-4 shows the updated `pom.xml` file with `JUnit` dependency.

Listing 4-4. Updated POM with `JUnit` Dependency

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

  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Getting Started with Maven</name>
  <url>http://apress.com</url>

  <developers>
   <developer>
        <id>balaji</id>
        <name>Balaji Varanasi</name>
        <email>balaji@inflinx.com</email>
       <properties>
          <active>true</active>
       </properties>
   </developer>
```

```
   <developer>
      <id>sudha</id>
      <name>Sudha Belida</name>
      <email>sudha@inflinx.com</email>
      <properties>
          <active>true</active>
      </properties>
   </developer>
 </developers>
   <dependencies>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.11</version>
                <scope>test</scope>
        </dependency>
</dependencies>

</project>
```

Notice that you have used the scope `test`, indicating that the `JUnit .jar` is needed only during the testing phase. Let's make sure that this dependency has been successfully added by running `mvn dependency:tree` in the command line. Listing 4-5 shows the output of this operation.

Listing 4-5. Maven Tree Command Output

```
C:\apress\gswm-book\chapter4\gswm>mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ gswm ---
[INFO] com.apress.gswmbook:gswm:jar:1.0.0-SNAPSHOT
[INFO] \- junit:junit:jar:4.11:test
[INFO]    \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 1.646s
[INFO] Finished at: Mon Oct 13 21:54:24 MDT 2014
[INFO] Final Memory: 7M/19M
[INFO] ------------------------------------------------------------
```

The tree goal in the dependency plug-in displays the project's dependencies as `tree`. Notice that the `JUnit` dependency pulled in a transitive dependency named `hamcrest`, which is an open source project that makes it easy to write matcher objects.

Now that you have the `JUnit` dependency in the class path, let's add a unit test `HelloWorldTest.java` to the project. Create the folders `test/java` under `src` and add `HelloWorldTest.java` beneath it. The updated project structure is shown in Figure 4-5.

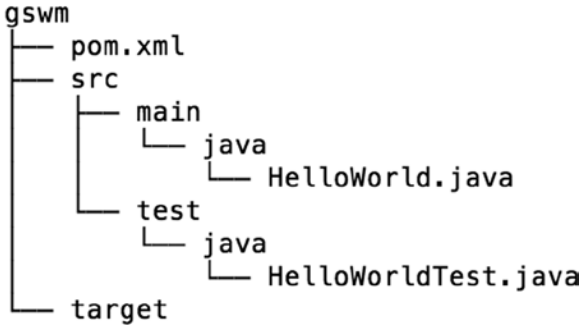


Figure 4-5. *Maven structure with test class*

The source code for `HelloWorldTest` is shown in Listing 4-6.

Listing 4-6. Code for `HelloWorldTest` Java Class

```
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class HelloWorldTest {

        private final ByteArrayOutputStream outStream = new
ByteArrayOutputStream();

        @Before
        public void setUp() {
            System.setOut(new PrintStream(outStream));
        }

        @Test
        public void testSayHello() {
                HelloWorld hw = new HelloWorld();
                hw.sayHello();
                Assert.assertEquals("Hello World", outStream.toString());
        }
```

```
        @After
        public void cleanUp() {
            System.setOut(null);
        }

}
```

You now have everything set up in this project, so you can run the `mvn` package one more time. After you run it, you will see output similar to that shown in Listing 4-7.

Listing 4-7. Output for Maven Command for Building the Project

```
C:\apress\gswm-book\chapter4\gswm>mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ gswm ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources,i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\apress\gswm-book\chapter4\
gswm\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ gswm ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)
@ gswm ---
-------------------------------------------------------
[INFO] Surefire report directory: C:\apress\gswm-book\chapter4\gswm\target\
surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running HelloWorldTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.038 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ gswm ---
[INFO] Building jar: C:\apress\gswmbook\chapter4\gswm\target\gswm-1.0.0-
SNAPSHOT.jar
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.807s
[INFO] Finished at: Mon Oct 13 21:59:57 MDT 2014
[INFO] Final Memory: 9M/22M
[INFO] ------------------------------------------------------------------------
```

Note the Tests section in Listing 4-7. It shows that Maven has run the test and that it has successfully completed.

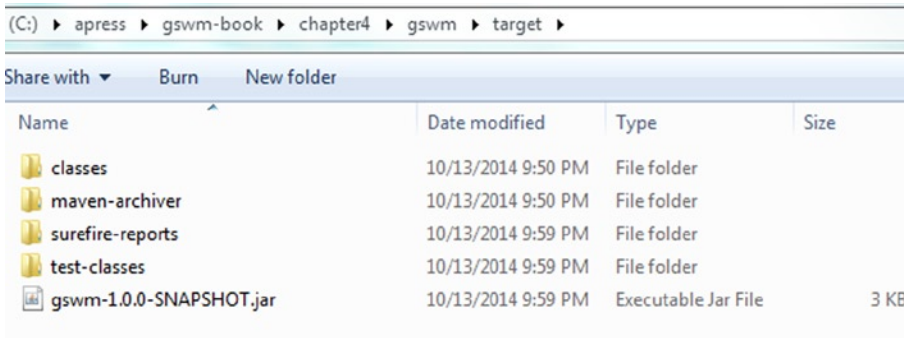


Figure 4-6. *Target folder with test classes*

Figure 4-6 shows the updated target folder. You can see that you now have a `test-classes` folder with their associated reports in that folder.

Properties in *pom.xml*

Maven allows you to declare properties in the `pom.xml` file using the `<properties />` element. These properties are highly useful for declaring dependency versions. Listing 4-8 shows the updated `pom.xml` file with the `JUnit` version declared as a property. Notice the use of `${}` syntax in the version element of `JUnit` dependency. This is especially useful when `pom.xml` has a lot of dependencies and you need to know or change a version of a particular dependency.

Listing 4-8. `pom.xml` File with Properties

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

```
  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Getting Started with Maven</name>
  <url>http://apress.com</url>

  <properties>
   <junit.version>4.11</junit.version>
  </properties>

  <developers>
   <developer>
        <id>balaji</id>
        <name>Balaji Varanasi</name>
        <email>balaji@inflinx.com</email>
       <properties>
          <active>true</active>
       </properties>
   </developer>
   <developer>
      <id>sudha</id>
      <name>Sudha Belida</name>
      <email>sudha@inflinx.com</email>
      <properties>
          <active>true</active>
      </properties>
   </developer>
 </developers>
   <dependencies>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>${junit.version}</version>
                <scope>test</scope>
        </dependency>
</dependencies>
</project>
```

Excluding Dependencies

Chapter 3 discussed transitive dependencies and the occasional need to exclude a particular transitive dependency. The `exclusions` element in the `pom.xml` file allows you to exclude a dependency.

Listing 4-9 shows the updated dependencies element for `JUnit` where the `hamcrest` transitive dependency is excluded. As you can see, the `exclusion` element takes the `groupId` and `artifactId` coordinates of the dependency that you would like to exclude.

Listing 4-9. `JUnit` Dependency with Exclusion

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

Summary

Maven's CoC prescribes a standard directory layout for all of its projects. It provides several sensible directories such as `src\main\java` and `src\it`, along with recommendations on the content that goes into each one of them. You learned about the mandatory `pom.xml` file and some of its elements, which are used to configure Maven project's behavior.

In the next chapter, you will look at Maven's life cycle, plug-ins, build phases, goals, and how to leverage them effectively.

CHAPTER 5

Maven Life Cycle

Goals and Plug-ins

Build processes generating artifacts typically require several steps and tasks to be completed successfully. Examples of such tasks include compiling source code, running a unit test, and packaging of artifacts. Maven uses the concept of *goals* to represent such granular tasks.

To better understand what a goal is, let's look at an example. Listing 5-1 shows the `compile` goal executed on `gswm` project code under `C:\apress\gswm-book\chapter5\gswm`. As the name suggests, the `compile` goal compiles source code. The `compile` goal identifies the Java class `HelloWorld.java` under `src/main/java`, compiles it, and places the compiled class file under the `target\classes` folder.

Listing 5-1. Maven `compile` Goal

```
C:\apress\gswm-book\chapter5\gswm>mvn compiler:compile
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-cli) @ gswm ---
[WARNING] File encoding has not been set, using platform encoding Cp1252,
i.e. build is platform dependent!
[INFO] Compiling 1 source file to C:\apress\gswm-book\chapter5\gswm\target\
classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.197s
[INFO] Finished at: Mon Oct 13 22:11:42 MDT 2014
[INFO] Final Memory: 7M/18M
[INFO] ------------------------------------------------------------------------
```

Goals in Maven are packaged in *plug-ins*, which are essentially a collection of one or more goals. In Listing 5-1, the compiler is the plug-in that provides the goal `compile`. Listing 5-2 introduces a pretty nifty goal called `clean`. As mentioned earlier, the target folder holds Maven-generated temporary files and artifacts. There are times when the target folder becomes huge or when certain files that have been cached need to be cleaned out of the folder. The `clean` goal accomplishes exactly that, as it attempts to delete the target folder and all its contents.

Listing 5-2. Maven `clean` Goal

```
C:\apress\gswm-book\chapter5\gswm>mvn clean:clean
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-cli) @ gswm ---
[INFO] Deleting C:\apress\gswm-book\chapter5\gswm\target
[INFO] ------------------------------------------------------------------------
----
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.233s
[INFO] Finished at: Mon Oct 13 22:14:49 MDT 2014
[INFO] Final Memory: 3M/15M
[INFO] ------------------------------------------------------------------------
```

Notice, the `clean:clean` suffix of the command in Listing 5-2. The `clean` before the colon (`:`) represents the `clean` plug-in, and the `clean` following the colon represents the `clean` goal. By now it should be obvious that running a goal in the command line requires the following syntax:

```
mvn plugin_identifier:goal_identifier
```

Plug-ins and their behavior can be configured using the plug-in section of `pom.xml`. Consider the case where you want to enforce that your project must be compiled with Java 1.6. As of version 3.0, the Maven Compiler Plug-in compiles the code against Java 1.5. Thus, you will need to modify the behavior of this plug-in in the `pom.xml` file, as shown in Listing 5-3.

Listing 5-3. Plug-in Element in the `pom.xml` File

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

```
  <!-- Project details omitted for brevity -->

  <dependencies>
        <!-- Dependency details omitted for brevity -->
  </dependencies>

  <build>
   <plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
         <source>1.6</source>
         <target>1.6</target>
      </configuration>
    </plugin>
    </plugins>
  </build>
</project>
```

Now if you were to run the `mvn compiler:compile` command, the generated class files will be of Java version 1.6.

■ **Note** The `<build />` element in `pom.xml` has a very useful child element called `finalName`. By default, the name of the Maven-generated artifact follows the `<<project_artifiact_id>>-<<project_version>>` format. However, sometimes you might want to change the name of the generated artifact without changing the `artifactId`. You can accomplish this by declaring the `finalName` element as `<finalName>new_name</finalName>`.

Life Cycle and Phases

Maven follows a well-defined *build life cycle* when it builds, tests, and distributes an artifact. The life cycle constitutes a series of stages or steps that get executed in the same order, independent of the artifact being produced. Maven refers to the steps in a life cycle as *phases*. Maven has the following three built-in life cycles:

Default: This life cycle handles the compiling, packaging, and deployment of a Maven project.

Clean: This life cycle handles the deletion of temporary files and generated artifacts from the target directory.

Site: This life cycle handles the generation of documentation and site generation.

■ **Note** Now that you are aware of the clean life cycle, you can clean the target folder simply by running the clean phase using the command `mvn clean`.

To better understand the build life cycle and its phases, let's look at the some of the phases associated with the default life cycle:

Validate: Runs checks to ensure that the project is correct and that all dependencies are downloaded and available.

Compile: Compiles the source code.

Test: Runs unit tests using frameworks. This step doesn't require that the application be packaged.

Package: Assembles compiled code into a distributable format, such as JAR or WAR.

Install: Installs the packaged archive into a local repository. The archive is now available for use by any project running on that machine.

Deploy: Pushes the built archive into a remote repository for use by other teams and team members.

Because the default life cycle clearly defines the ordering of the phases, you can generate an artifact simply by running the command `mvn package`. Maven will automatically execute all of the phases prior to the requested phase. In the provided example, Maven will run phases, such as compile and test, prior to running the package phase. This means developers and configuration managers only have to learn and use a handful of commands.

A number of tasks need to be performed in each phase. For that to happen, each phase is associated with one or more goals. The phase simply delegates those tasks to its associated goals. Figure 5-1 shows the association between life cycle, phases, goals, and plug-ins.

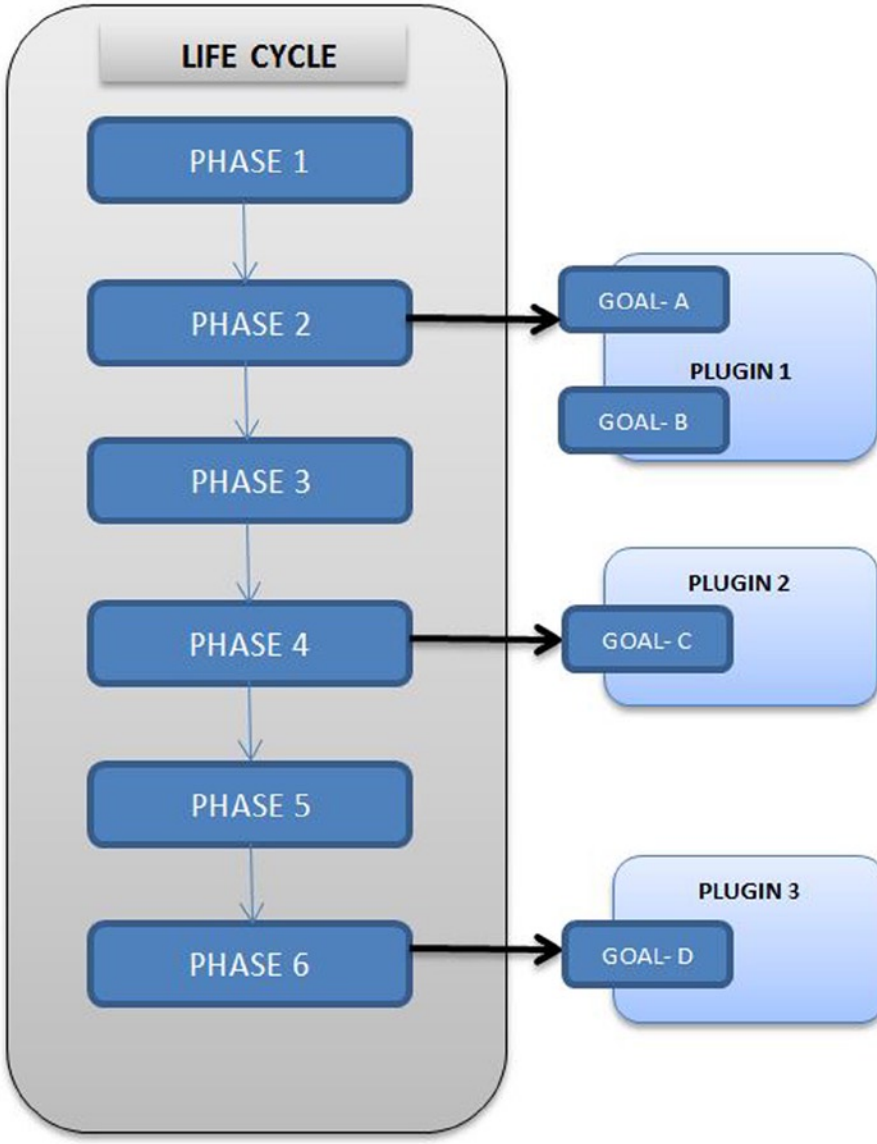


Figure 5-1. *Association between life cycle, phases, goals, and plug-ins*

The `<packaging />` element in the `pom.xml` file will automatically assign the right goals for each of the phases without any additional configuration. Remember that this is a benefit of CoC. If the packaging element contains the value `jar`, then the package phase will be bound to the `jar` goal in the `jar` plug-in. Similarly, for a `WAR` artifact, `pom.xml` will bind the package to a `war` goal in the `war` plug-in.

SKIPPING TESTS

As discussed earlier, when you run the package phase, the test phase is also run and all of the unit tests get executed. If there are any failures in the test phase, the build fails. This is the desired behavior. However, there are times, for example, when dealing with a legacy project, where you would like to skip running the tests so you can build a project successfully. You can achieve this using the `maven.test.skip` property. Here is an example of using this property:

```
mvn package -Dmaven.test.skip=true
```

Plug-in Development

Developing plug-ins for Maven is very straightforward. This section explains how to develop an example `HelloPlugin` that will give you a taste of plug-in development.

As discussed earlier, a plug-in is simply a collection of goals. Thus, when we talk about plug-in development, we are essentially talking about developing goals. In Java, these goals are implemented using MOJOs, which stands for Maven Old Java Object and it is similar to Java's Plain Old Java Object (POJO).

Let's start this plug-in development by creating a Maven Java project, named `gswm-plugin`, as shown in Figure 5-2. We are creating this project under a starter `gswm-plugin` project available in the `C:\apress\gswm-book\chapter5` folder.

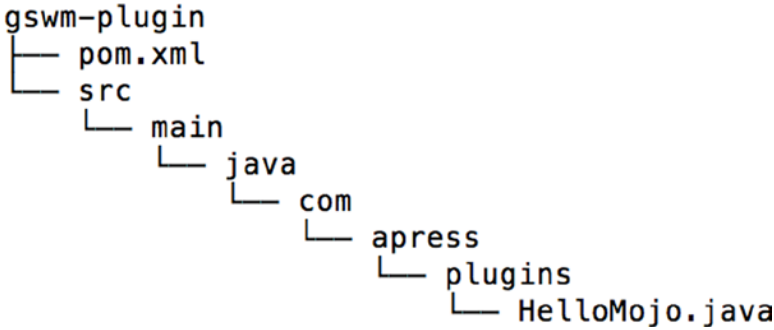


Figure 5-2. *Maven project for plug-in development*

■ **Note** In this chapter we are manually creating the plug-in project. Maven provides a *mavan-archetype-mojo*, which would jumpstart your plug-in development. We will learn about Maven archetypes in Chapter 6.

The content of the `pom.xml` file is shown in Listing 5-4. Notice that the packaging type is `maven-plugin`. We added the `maven-plugin-api` dependency, because it is needed for plug-in development.

Listing 5-4. The `pom.xml` with plug-in api dependency

```
<?xml version="1.0" encoding="UTF-8"?><project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.plugins</groupId>
    <artifactId>gswm-plugin</artifactId>
    <version>1.0.0</version>
    <packaging>maven-plugin</packaging>
    <name>Simple Hello Plugin</name>
    <dependencies>
        <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-api</artifactId>
            <version>3.2.3</version>
        </dependency>
    </dependencies>
</project>
```

The next step in the development process is creating the MOJO. Listing 5-5 shows the code for `HelloMojo`. As you can see, the implementation is straightforward. We are using the `Log` instance to log output to the console. The most important part of this code is actually inside the Java comment section: `@goal hello`. Using the Javadoc tag `@goal`, we are declaring the name of this goal as `hello`. It is also possible to use Java 5 annotations such has `@Mojo` to provide this metadata. However, it requires the `pom.xml` file changes discussed on the Apache Maven web site (http://maven.apache.org/plugin-tools/maven-plugin-plugin/examples/using-annotations.html).

Listing 5-5. `HelloMojo` Java Class

```
package com.apress.plugins;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;

/**
 *
 * @goal hello
 */
public class HelloMojo extends AbstractMojo{

        public void execute() throws MojoExecutionException,
MojoFailureException {
                getLog().info("Hello Maven Plugin");
        }

}
```

The final step in this process is installing the plug-in in the Maven repository. Run the `mvn install` command at the root of the directory and you should get the output shown in Listing 5-6.

Listing 5-6. Maven `install` Command

```
C:\apress\gswm-book\chapter5\gswm-plugin>mvn install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Simple Hello Plugin 1.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-plugin-plugin:3.2:descriptor (default-descriptor)
@ gswm-plugin ---
[INFO] Applying mojo extractor for language: java-annotations
[INFO] Mojo extractor for language: java-annotations found 0 mojo
descriptors.
[INFO] Applying mojo extractor for language: java
[INFO] Mojo extractor for language: java found 1 mojo descriptors.
[INFO] Applying mojo extractor for language: bsh
[INFO] Mojo extractor for language: bsh found 0 mojo descriptors.
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources)
@ gswm-plugin ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\apress\gswm-book\chapter5\
gswm-plugin\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile)
@ gswm-plugin ---
-----------------------------------------
[INFO] Building jar: C:\apress\gswm-book\chapter5\gswm-plugin\target\gswm-
plugin-1.0.0.jar
[INFO]
[INFO] --- maven-plugin-plugin:3.2:addPluginArtifactMetadata (default-
addPluginA
rtifactMetadata) @ gswm-plugin ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install)
@ gswm-plugin ---
```

```
[INFO] Installing C:\apress\gswm-book\chapter5\gswm-plugin\target\gswm-
plugin-1.0.0.jar to C:\Users\<<USER_NAME>>\.m2\repository\com\apress\
plugins\gswm-plugin\1.0.0\gswm-plugin-1.0.0.jar
[INFO] Installing C:\apress\gswm-book\chapter5\gswm-plugin\pom.xml to C:\
Users\<<USER_NAME>>\.m2\repository\com\apress\plugins\gswm-plugin\1.0.0\
gswm-plugin-1.0.0.pom

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.788s
[INFO] Finished at: Mon Oct 13 22:29:55 MDT 2014
[INFO] Final Memory: 13M/32M
[INFO] ------------------------------------------------------------------------
```

Now you're ready to start using this plug-in. Remember that the syntax to run any goal is `mvn pluginId:goalId`. Listing 5-7 shows this plug-in in action. Notice the `Hello Maven Plugin` text on the console.

Listing 5-7. Running the `Hello` Plug-in

```
C:\apress\gswm-book\chapter5\gswm-plugin>mvn com.apress.plugins:gswm-
plugin:hello
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building Simple Hello Plugin 1.0.0
[INFO] ------------------------------------------------------------
[INFO] --- gswm-plugin:1.0.0:hello (default-cli) @ gswm-plugin ---
[INFO] Hello Maven Plugin
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 0.583s
[INFO] Finished at: Mon Oct 13 22:32:55 MDT 2014
[INFO] Final Memory: 4M/15M
[INFO] ------------------------------------------------------------
```

Summary

Maven uses plug-in–based architecture that allows its functionality to be extended easily. Each plug-in is a collection of one or more goals that can be used to execute tasks, such as compiling source code or running tests. Maven ties goals to phases. Phases are typically executed in a sequence as part of a build life cycle. You also learned the basics of creating a plug-in.

In the next chapter, you will be introduced to archetypes and learn about multimodule projects.

CHAPTER 6

Maven Archetypes

Up to this point, you have created a Maven project manually, generating the folders and creating the `pom.xml` files from scratch. This can become tedious, especially when you frequently have to create projects. To address this issue, Maven provides *archetypes*. Maven archetypes are project templates that allow users to generate new projects easily.

Maven archetypes also provide a great platform to share best practices and enforce consistency beyond Maven's standard directory structure. For example, an enterprise can create an archetype with the company's branded cascading style sheet (CSS), approved JavaScript libraries, and reusable components. Developers using this archetype to generate projects will automaticaly conform to the company's standards.

Built-in Archetypes

Maven provides hundreds of out-of-the-box archetypes for developers to use. Additionally, a lot of open source projects provide additional archetypes that you can download and use. Maven also provides an archetype plug-in with goals to create archetypes and generate projects from archetypes.

The archetype plug-in's `generate` goal allows you to view and select an archetype for use. Listing 6-1 shows the results of running the `generate` goal at the command line. As you can see, there are 491 archetypes to choose from. This chapter will look at using a few of these archetypes.

Listing 6-1. Maven `generate` Goal

```
$mvn archetype:generate
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) @ standalone-
pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-
pom <<<
```

```
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-
pom
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart (org.apache.
maven.archetypes:maven-archetype-quickstart:1.0)

.........................
.........................

1176: remote -> ru.yandex.qatools.camelot:camelot-plugin (-)
1177: remote -> se.vgregion.javg.maven.archetypes:javg-minimal-archetype (-)
1178: remote -> sk.seges.sesam:sesam-annotation-archetype (-)
1179: remote -> tk.skuro:clojure-maven-archetype (A simple Maven archetype
for Clojure)
1180: remote -> tr.com.lucidcode:kite-archetype (A Maven Archetype that
allows users to create a Fresh Kite project)
1181: remote -> uk.ac.rdg.resc:edal-ncwms-based-webapp (-)
1182: local -> com.inflinx.book.ldap:practical-ldap-empty-archetype (-)
1183: local -> com.inflinx.book.ldap:practical-ldap-archetype (-)
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 491:
```

Generating a Web Project

Maven provides the `maven-archetype-webapp` archetype for generating a web application. Let's generate the application by running the following command in the `C:\apress\gswm-book\chapter6` folder:

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

The command runs in interactive mode. Enter the following information for the requested inputs:

```
Define value for property 'groupId': : com.apress.gswmbook
Define value for property 'artifactId': : gswm-web
Define value for property 'version':  1.0-SNAPSHOT: :  <<Hit Enter>>
Define value for property 'package':  com.apress.gswmbook: : war

Confirm the properties configuration:
groupId: com.apress.gswmbook
artifactId: gswm-web
version: 1.0-SNAPSHOT
package: war
 Y: <<Hit Enter>>
```

The generated directory structure should resemble the one shown in Figure 6-1.

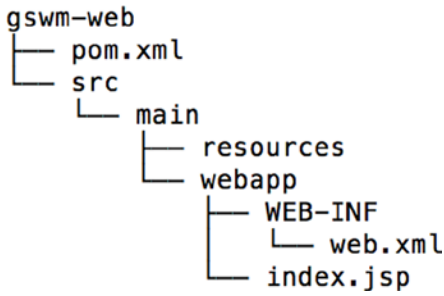


Figure 6-1. *Maven web project structure*

The `pom.xml` file is minimal and only has a JUnit dependency. Maven makes it easier to run your web application using embedded web servers, such as Tomcat and Jetty. Listing 6-2 shows the modified `pom.xml` file with a Tomcat plug-in added.

Listing 6-2. Modified `pom.xml` with Embedded Tomcat Plug-in

```
<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.apress.gswmbook</groupId>
  <artifactId>gswm-web</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gswm-web 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>gswm-web</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```

In order to launch the web application inside the Tomcat server, run the following command at the root directory of the project:

```
mvn tomcat7:run
```

You will see the project deployed and view output similar to that shown in Listing 6-3.

Listing 6-3. Output from the Tomcat `run` Command

```
Oct 11, 2014 12:08:43 PM org.apache.catalina.core.StandardService
startInternal
INFO: Starting service Tomcat
Oct 11, 2014 12:08:43 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.47
Oct 11, 2014 12:08:45 PM org.apache.catalina.util.SessionIdGenerator
createSecureRandom
INFO: Creation of SecureRandom instance for session ID generation using
[SHA1PRNG] took [334] milliseconds.
Oct 11, 2014 12:08:45 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
```

Now launch the browser and navigate to `http://localhost:8080/gswm-web/`. You should see the web page as shown in Figure 6-2.

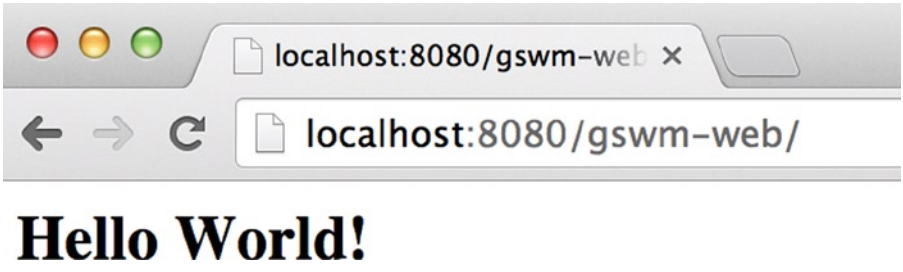


Figure 6-2. *Web project launched in browser*

Multimodule Project

Java Enterprise Edition (JEE) projects are often split into several modules to ease development and maintainability. Each of these modules produces artifacts such as Enterprise JavaBeans (EJBs), web services, web projects, and client `jars`. Maven supports development of such large JEE projects by allowing multiple Maven projects to be nested under a single Maven project. The layout of such a multimodule project is shown in Figure 6-3. The parent project has a `pom.xml` file and individual Maven projects inside it.

```
Parent Project
        |-- Module 1
        |          |
        |          `-- pom.xml
        |
        |-- Module 2
        |          |
        |          `-- pom.xml
        |-- Module 3
        |          |
        |          `-- pom.xml
        |
        |
        `-- pom.xml
```

Figure 6-3. *Multimodule project structure*

In the rest of this section, we will explain how to build a multimodule project for the scenario where you have to split your large project into a web application (`WAR` artifact) that provides a user interface, a service project (`JAR` artifact) that holds service layer code, and a persistence project that holds your repository layer code. Figure 6-4 provides a visual representation of this scenario.

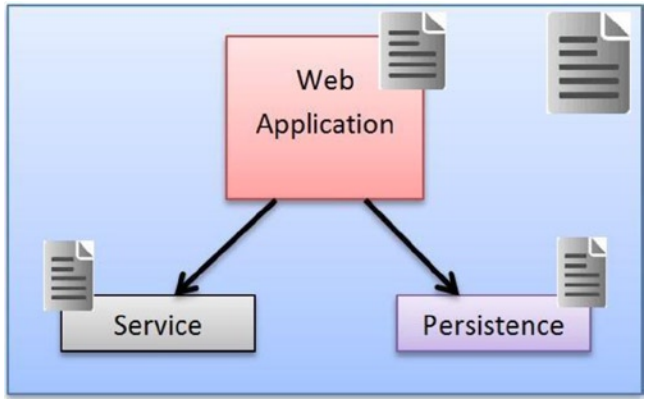


Figure 6-4. *Maven multimodule project*

Let's start the process by generating the parent project. To do this, run the following command at the command line under `C:\apress\gswm-book\chapter6`:

```
mvn archetype:generate -DgroupId=com.apress.gswmbook -DartifactId=gswm-
parent -Dversion=1.0.0-SNAPSHOT -DarchetypeGroupId=org.codehaus.mojo.
archetypes -DarchetypeArtifactId=pom-root
```

The archetype `pom-root` creates the `gswm-parent` folder and a `pom.xml` file underneath it. As you can see in Listing 6-4, the generated `pom.xml` file has minimal content. Notice that the packaging of the parent project is set to type `pom`.

Listing 6-4. Parent `pom.xml` File

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm-parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>gswm-parent</name>
</project>
```

Then create the web project by running the following command in the `C:\apress\gswm-book\chapter6\gswm-parent` folder:

```
mvn archetype:generate -DgroupId=com.apress.gswmbook -DartifactId=gswm-
web -Dversion=1.0.0-SNAPSHOT -Dpackage=war -DarchetypeArtifactId=maven-
archetype-webapp
```

During this web project generation, you are providing Maven coordinates, such as groupId, version, and so on, as parameters to the `generate` goal. This created the `gswm-web` project.

The next step is to create the `service` project. Run the following command under `C:\apress\gswm-book\chapter6\gswm-parent`:

```
mvn archetype:generate -DgroupId=com.apress.gswmbook -DartifactId=gswm-service
-Dversion=1.0.0-SNAPSHOT -DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

Notice that you didn't provide the package parameter, as the `maven-archetype-quickstart` produces a JAR project by default. Also, notice the use of the `interactiveMode` parameter. This instructs Maven to simply run the command without prompting the user for input.

Similar to the previous step, create another Java project `gswm-repository` by running the following command under `C:\apress\gswm-book\chapter6\gswm-parent`:

```
mvn archetype:generate -DgroupId=com.apress.gswmbook -DartifactId=gswm-
repository -Dversion=1.0.0-SNAPSHOT -DarchetypeArtifactId=maven-archetype-
quickstart -DinteractiveMode=false
```

Now that you have all of the projects generated, let's look at the `pom.xml` file under `gswm-parent`. Listing 6-5 shows the `pom.xml` file.

Listing 6-5. Parent `pom.xml` File with Modules

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm-parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>gswm-parent</name>
  <modules>
    <module>gswm-web</module>
    <module>gswm-service</module>
    <module>gswm-repository</module>
  </modules>
</project>
```

The `modules` element allows you to declare child modules in a multimodule project. As you generated each module, Maven intelligently registered them as a child module. Additionally, it modified the individual module's `pom.xml` file and added the parent `pom` information. Listing 6-6 shows `gswm-web` project's `pom.xml` file with the parent `pom` elements.

Listing 6-6. The `pom.xml` File for the Web Module

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.apress.gswmbook</groupId>
    <artifactId>gswm-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>
```

```
  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm-web</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>gswm-web 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>gswm-web</finalName>
  </build>
</project>
```

With the entire infrastructure set up, you are ready to build the next project. To accomplish this, simply run the `mvn package` command under `gswm-project`, as shown in Listing 6-7.

Listing 6-7. Maven Package Run on the Parent Project

```
C:\apress\gswm-book\chapter6\gswm-parent>mvn package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] gswm-parent
[INFO] gswm-web Maven Webapp
[INFO] gswm-service
[INFO] gswm-repository
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] gswm-parent ....................................... SUCCESS [0.001s]
[INFO] gswm-web Maven Webapp ............................. SUCCESS [1.033s]
[INFO] gswm-service ...................................... SUCCESS [0.552s]
[INFO] gswm-repository ................................... SUCCESS [0.261s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.949s
[INFO] Finished at: Mon Oct 13 23:09:21 MDT 2014
[INFO] Final Memory: 6M/15M
[INFO] ------------------------------------------------------------------------
```

Creating an Archetype

Maven provides several ways to create a new archetype. Here we will use an existing project to generate an archetype.

Let's start by creating a prototype project that you will use as the seed for archetype creation. This project will be Servlet 3.0 compatible, and it has a Status Servlet that returns a HTTP status code 200. Instead of creating a web project from scratch, copy the previously generated `gswm-web` project code and create `gswm-web-prototype` under `C:\apress\gswm-book\chapter6`. Make the following changes to the newly copied project:

1. Remove all other resources, such as Integrated Development Environment (IDE) specific files (`.project`, `.classpath`, and so forth) that you don't want to end up in the archetype.

2. Replace the contents of the `web.xml` file under the `webapp/WEB-INF` folder. This will upgrade the web application to use Servlet 3.0:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
      version="3.0">
  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

3. Add the Servlet 3.0 dependency to the `pom.xml` file. The updated `pom.xml` is shown in Listing 6-8.

Listing 6-8. The `pom.xml` with Servlet Dependency

```
<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.apress.gswmbook</groupId>
  <artifactId>gswm-web</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gswm-web Maven Webapp</name>
  <url>http://maven.apache.org</url>
```

```
  <dependencies>
        <dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.0.1</version>
  <scope>provided</scope>
</dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>gswm-web</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```

4. Because you will be doing Java web development, create a folder named `java` under `src/main`. Similarly, create `test/java` and `test/resources` folders under `src`.

5. Create the `AppStatusServlet.java` file in the `com.apress.gswmbook.web.servlet` package under `src/main/java`. The package `com.apress.gswmbook.web.servlet` translates to folder structure `com\apress\gswmbook\web\servlet`. The source code for `AppStatusServlet.java` is shown in Listing 6-9.

Listing 6-9. `AppStatusServlet` Java Class Source Code

```
package com.apress.gswmbook.web.servlet;

import javax.servlet.annotation.WebServlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
@WebServlet("/status")
public class AppStatusServlet extends HttpServlet {

        public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException {

        PrintWriter writer = response.getWriter();
        writer.println("OK");
        response.setStatus(response.SC_OK);
    }

}
```

The prototype project will be similar to the structure shown in Figure 6-5.

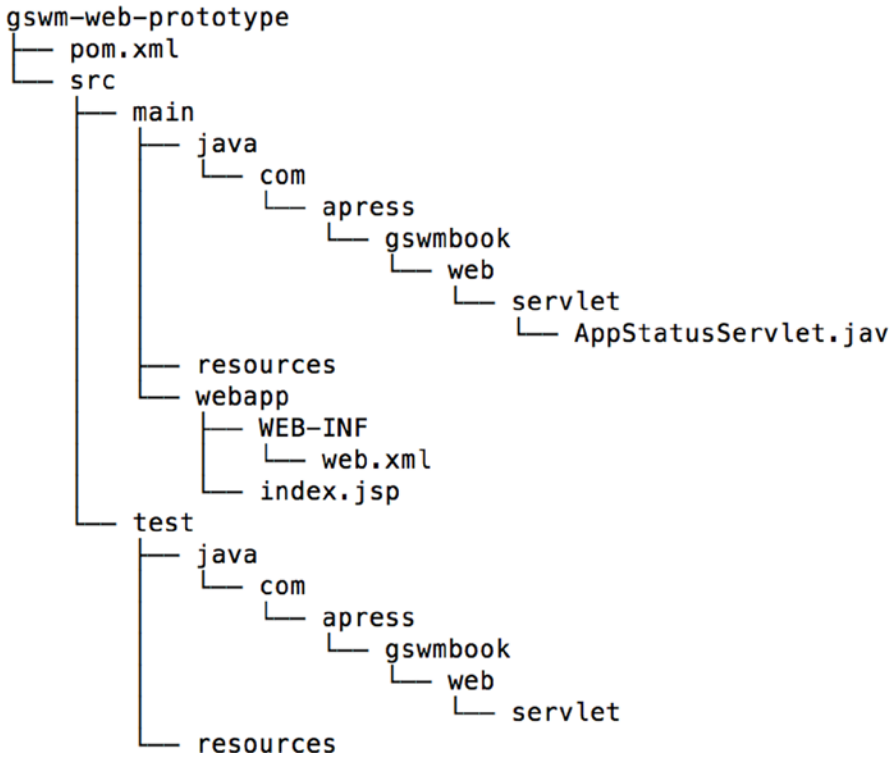


Figure 6-5. *Generated prototype project*

Using the command line, navigate to the project folder `gswm-web-prototype` and run the following command:

```
mvn archetype:create-from-project
```

Upon completion of the command, you should see the message *Archetype created in target/generated-sources/archetype*. The newly created archetype is now under `gswm-web-prototype/target/generated-sources/archetype`.

The next step is to move the newly created archetype into a separate folder `gswm-web-archetype` so that it can be tweaked before it is published. To accomplish this, follow these steps:

1. Create folder `gswm-web-archetype` in the `C:\apress\gswm-book\chapter6` folder.
2. Copy subdirectories and files in the `C:\apress\gswm-book\chapter6\gswm-web-prototype\target\generated-sources\archetype` folder to the `gswm-web-archetype` folder.
3. Delete the `target` subdirectory from the `C:\apress\gswm-book\chapter6\gswm-web-archetype` folder.

The directory structure for `gswm-web-archetype` should be similar to that shown in Figure 6-6.

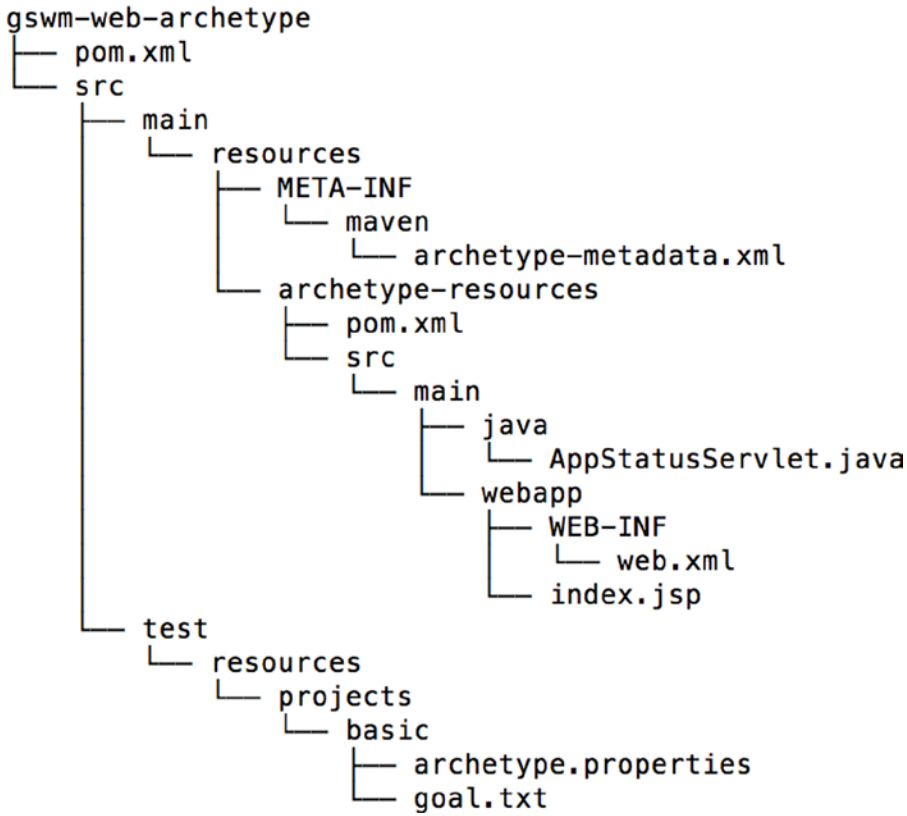


Figure 6-6. *Web archetype project structure*

Let's start the modification process with the `pom.xml` file located at `gswm-web-archetype\src\main\resources\archetype-resources`. Change the final name in the `pom.xml` file from `gswm-web` to `${artifactId}`. During project creation, Maven will replace the `${artifactId}` expression with the user-supplied `artifactId` value.

When a project is created from an archetype, Maven prompts the user for a package name. It will create the directories corresponding to the package under the `src/main/java` folder of the newly created project. It then moves all of the contents under the archetype's `archetype-resources/src/main/java` folder into that package. Because you would like the `AppStatusServlet` under the subpackage `web.servlet`, create the folder `web/servlet` and move `AppStatusServlet.java` under the newly created folder. The new location of the `AppStatusServlet.java` is shown in Figure 6-7.

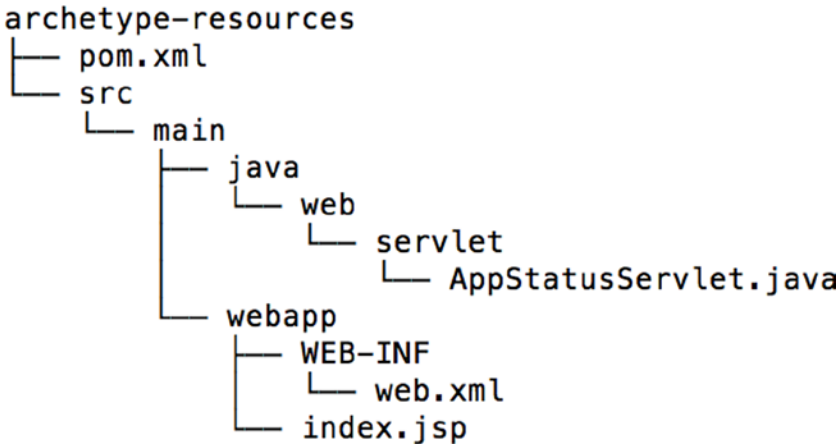


Figure 6-7. *AppStatusServlet under the* `web.servlet` *package*

Open `AppStatusServlet.java` and change the package name from `package ${package};` to `package ${package}.web.servlet;`.

The final step in creating the archetype is to run the following at the command line inside the folder `gswm-web-archetype`:

```
mvn clean install
```

Using the Archetype

Once the archetype is installed, the easiest way to create a project from it is to run the following command under `C:\apress\gswm-book\chapter6`:

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

Enter the values shown in Listing 6-10 for the Maven prompts, and you will see a `test-project` created.

Listing 6-10. Creating a New Project Using Archetype

```
C:\apress\gswm-book\chapter6>mvn archetype:generate -DarchetypeCatalog=local
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart (org.apache.
```

```
maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:1: local -> com.apress.gswmbook:gswm-web-archetype
(gswm-web-archetype)
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): : 1
Define value for property 'groupId': : com.apress.gswmbook
Define value for property 'artifactId': : test-project
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.apress.gswmbook: :
Confirm properties configuration:
groupId: com.apress.gswmbook
artifactId: test-project
version: 1.0-SNAPSHOT
package: com.apress.gswmbook
 Y: :
----------------------------------------------------------------------------
project
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:27.635s
[INFO] Finished at: Mon Oct 13 23:36:01 MDT 2014
[INFO] Final Memory: 9M/22M
[INFO] ------------------------------------------------------------------------
```

Because the `pom.xml` file for the `test-project` already has the embedded Tomcat plug-in, run `mvn tomcat7:run` in the command line under the folder `C:\apress\gswm-book\chapter6\test-project` to launch the project. Open a browser and navigate to `http://localhost:8080/test-project/status`. You will see OK displayed, as shown in Figure 6-8.

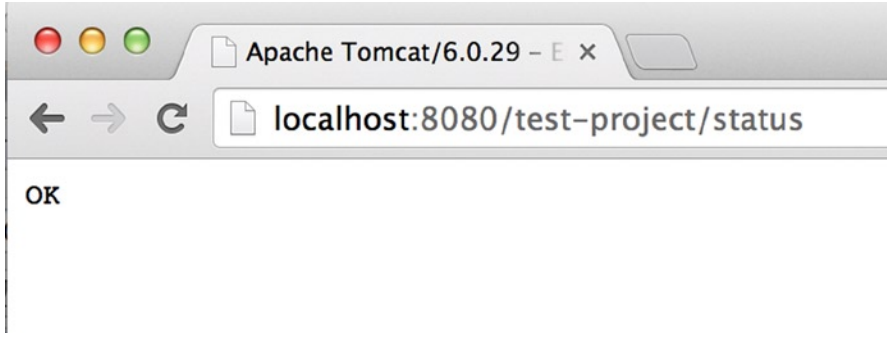


Figure 6-8. *Output of the generated test project*

Summary

Maven archetypes are project templates that allow you to bootstrap new projects quickly. This chapter used built-in archetypes for generating advanced Maven projects, such as web projects and multimodule projects. You also looked at creating and using a custom archetype.

In the next chapter, you will learn the basics of site generation and creating documentation and reports using Maven.

CHAPTER 7

Documentation and Reporting

Documentation and reporting are key aspects of any project. This is especially true for enterprise and open source projects, where many people collaborate to build the project. This chapter looks at some of Maven's tools and plug-ins, which make publishing and maintenance of online documentation a breeze.

This chapter will once again be working with the `gswm` Java project you built in earlier chapters. The `gswm` project is also available in the `C:\apress\gswm-book\chapter7` folder.

Using the Site Life Cycle

As discussed in Chapter 5, Maven provides the *site* life cycle that can be used to generate a project's documentation. Let's run the following command from the `gswm` directory:

```
mvn site
```

The site life cycle uses Maven's site plug-in to generate the site for a single project. Once this command completes, a site folder gets created under the project's target. Figure 7-1 shows the contents of the site folder.

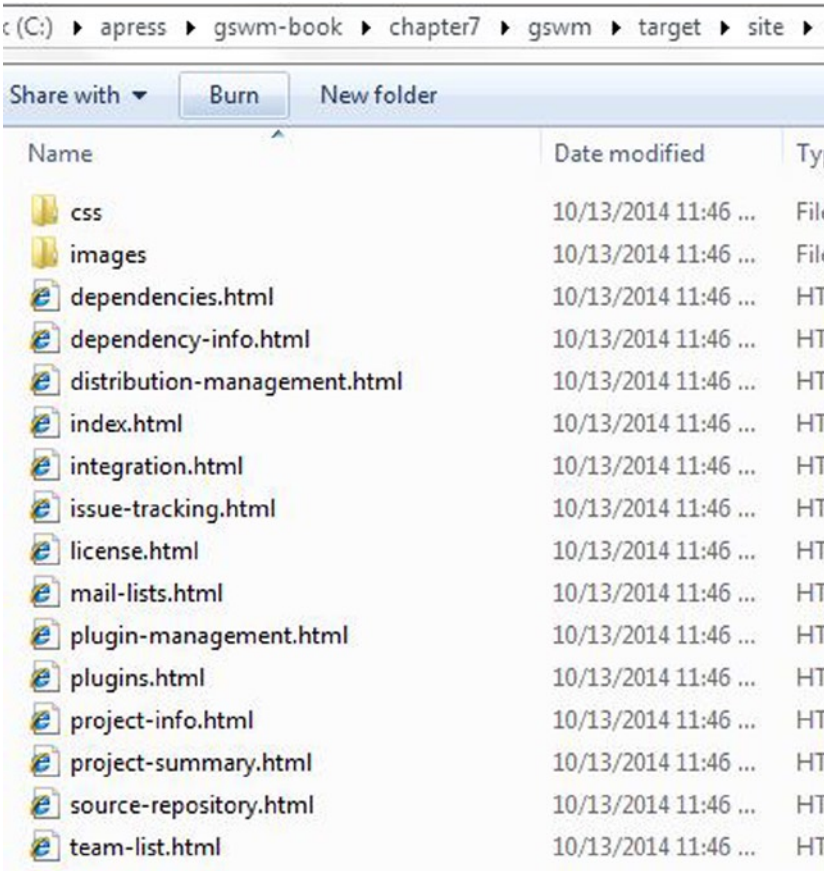


Figure 7-1. *Generated site folder*

Open the `index.html` file to browse the generated site. You will notice that Maven used the information provided in the `pom.xml` file to generate most of the documentation. It also automatically applied the default skin and generated the corresponding images and CSS files. Figure 7-2 shows the generated `index.html` file.

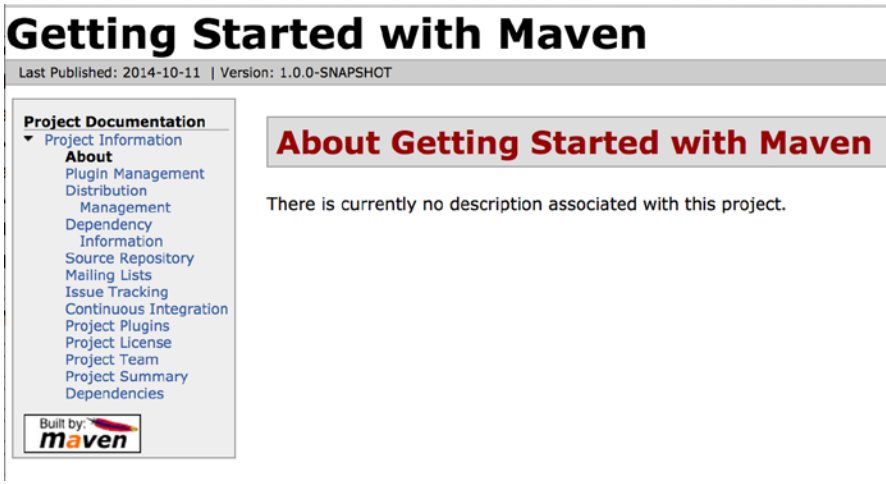


Figure 7-2. *Generated index page*

The Project Dependencies page provides valuable information regarding the project's direct and transitive dependencies. It also provides the licensing information associated with those dependencies, as shown in Figure 7-3.

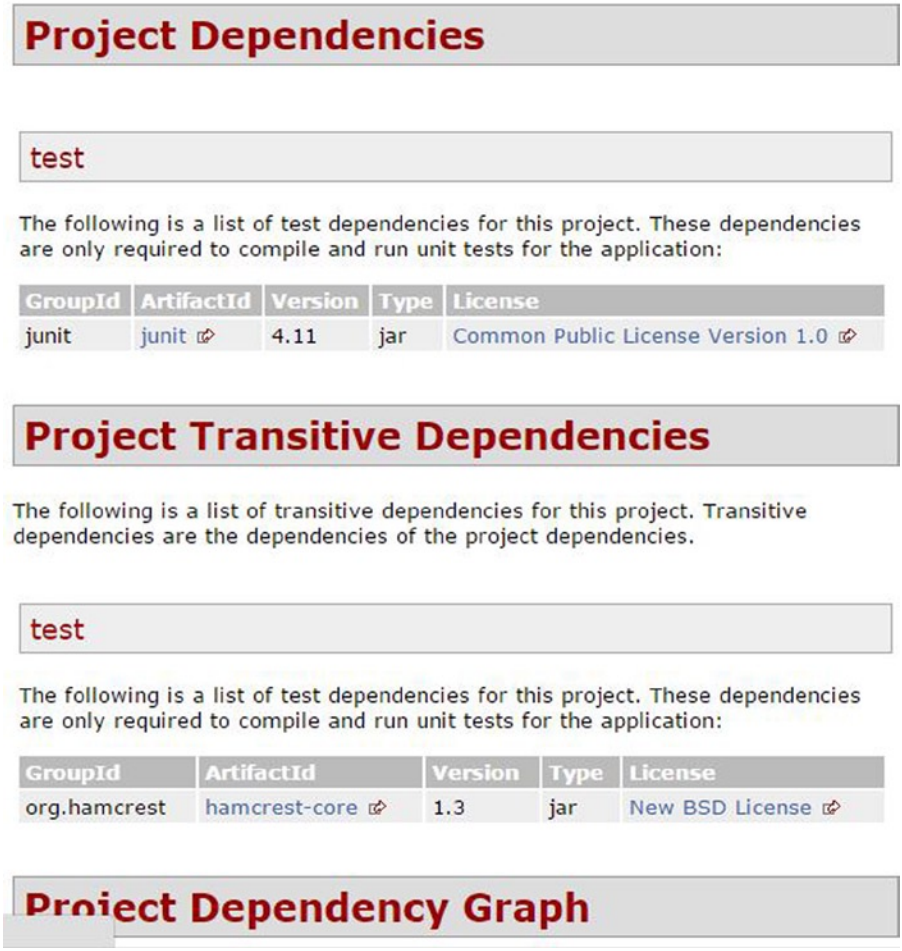


Figure 7-3. *Project dependencies page*

As you browse the generated site, you will notice that pages such as About, Mailing Lists, and Project License are missing information. Let's modify the `pom.xml` file and add the missing information, as provided in Listing 7-1.

Listing 7-1. The `pom.xml` File with Project Information

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

```
  <groupId>com.apress.gswmbook</groupId>
  <artifactId>gswm</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Getting Started with Maven</name>
  <url>http://apress.com</url>

  <description>
    This project acts as a starter project for the Introducing Maven book
(http://www.apress.com/9781484208427) published by Apress.
  </description>

  <mailingLists>
    <mailingList>
      <name>GSWM Developer List</name>
      <subscribe>gswm-dev-subscribe@apress.com</subscribe>
      <unsubscribe>gswm-dev-unsubscribe@apress.com</unsubscribe>
      <post>developer@apress.com</post>
    </mailingList>
  </mailingLists>

  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
    </license>
  </licenses>

  <!--- Developer, Dependency and Build information removed for brevity --->

</project>
```

Looking at the code for the `pom.xml` file in Listing 7-1, it is obvious that the `description` element is used to provide the project description. The `mailingList` element holds the mailing list information, and the `license` element includes the project's license. Let's regenerate the site by running the following command:

```
mvn clean site
```

Launch the `index.html` file under the newly generated `target\site` folder. Figure 7-4A and Figure 7-4B shows the new About and Project License pages, respectively. Notice that Maven uses the URL declared in the `license` element to download the license text and include it in the generated web site.

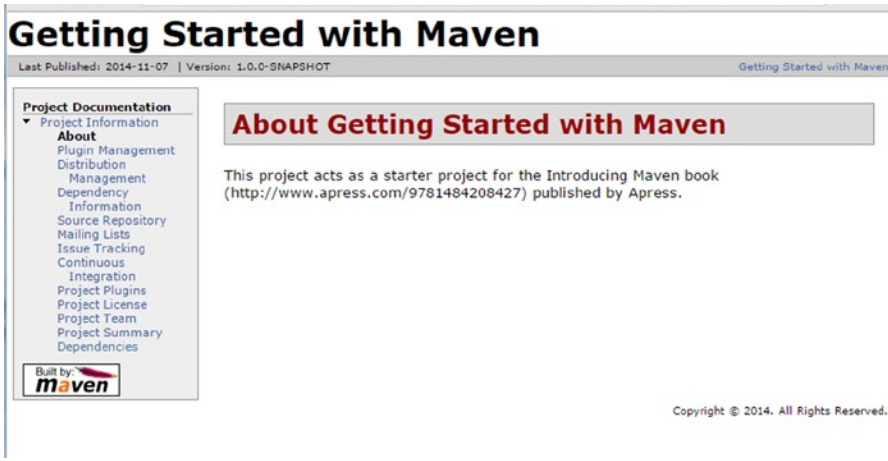


Figure 7-4A. *Generated About page*

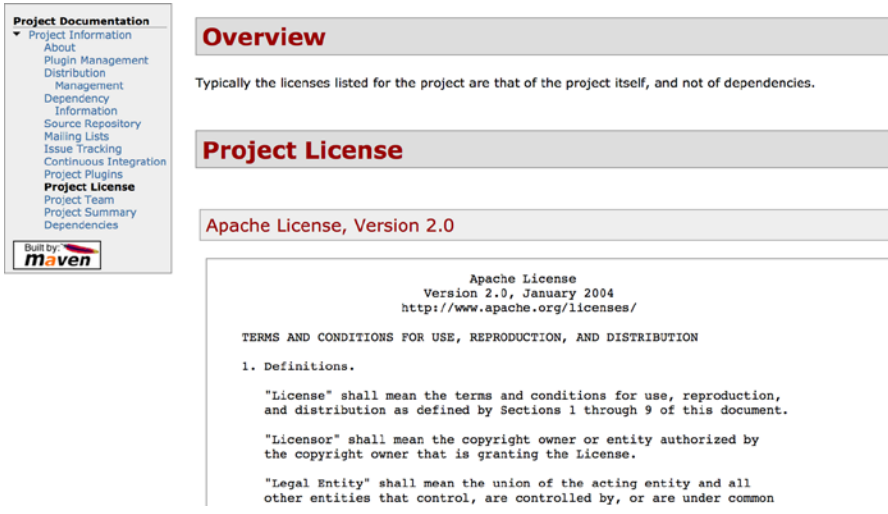


Figure 7-4B. *Generated Project License page*

Advanced Site Configuration

In the preceding section, project information was specified in the `pom.xml` file for Maven to use during site generation. For medium to complex projects, this approach would result in bloated and hard-to-maintain `pom.xml` files. Also, enterprises typically prefer to use their branding and logos in the generated site rather than the default Maven skin. To address these concerns, Maven allows you to specify content and configuration for site generation under the aptly named `src/site` folder. Figure 7-5 shows the directory structure for a simple site folder.

```
site
├── apt
│   └── index.apt
└── site.xml
```

Figure 7-5. *Site folder directory structure*

The `site.xml` file, also known as the *site descriptor*, is used to customize the generated site. We will look at this element in just a second.

The `apt` folder contains site content written in *Almost Plain Text* (APT) format. The APT format allows documentation to be created in a syntax that resembles plain text. More information about the APT format can be found on the Maven web site (`http://maven.apache.org/doxia/references/apt-format.html`). In addition to APT, Maven supports other formats, such as FML, Xdoc, and Markdown.

Maven provides several archetypes that allow you to generate site structure automatically. Because you will be updating the existing `gswm` project, you will use the `create` goal instead of `generate`, as shown in the following code. Run the command in the `C:\apress\gswm-book\chapter7\gswm` folder:

```
mvn archetype:create -DarchetypeArtifactId=maven-archetype-site-simple
```

Upon successful completion, you will see the site folder created under `gswm\src` with the `site.xml` and `apt` folders. Let's start by adding the project description to `index.apt`. Replace the contents of the `index.apt` file with the code from Listing 7-2.

Listing 7-2. The `index.apt` File Contents

```
-----
 Getting Started with Maven Starter
 -----
 Apress
 -----
 10-10-2014
 -----
```

This project acts as a starter project for the *Introducing Maven* book published by Apress. For more information check out the Apress Site `https://www.apress.com`.

Running `mvn clean site` results in a new About page, as shown in Figure 7-6.

Figure 7-6. *About page with new content*

Notice that the left navigation pane for the page has completely disappeared. This is because Maven constructs this navigation using the `site.xml` file, and this `site.xml` file currently lacks navigation information.

Before we look at the information in the `site.xml` file, let's add an image that will serve as the site logo. Static assets, such as images and HTML files, are placed in the `site/resources` folder. When Maven builds the site, it copies the assets in the `resources` folder to the root of the generated site. Copy the company logo `company.png` from the `C:\apress\gswm-book\chapter7` folder and place it in the `gswm/src/site/resources/images` folder.

Now you are ready to modify the `site.xml` file so that the logo and navigation show up. Replace the `site.xml` file with the contents of Listing 7-3. Notice that the `src` element for the logo includes the relative path `images/company.png`. The `menu` element is used to create different navigation links to be displayed on the site.

Listing 7-3. The `site.xml` File Contents

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

<project xmlns="http://maven.apache.org/DECORATION/1.6.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/DECORATION/1.6.0
http://maven.apache.org/xsd/decoration-1.6.0.xsd"
  name="Getting Started With Maven" >

        <bannerLeft>
        <name>Apress</name>
        <src>images/company.png</src>
        <href>http://apress.com</href>
    </bannerLeft>
        <body>
            <links>
              <item name="Maven" href="http://maven.apache.org/"/>
            </links>
            <menu name="Project Information">
            <item name="Introduction" href="index.html"/>
            <item name="Plugin Management" href="plugin-management.html"/>
            <item name="Dependency Information" href="dependency-info.html"/>
            <item name="Source Repository" href="source-repository.html"/>
            <item name="Mailing Lists" href="mail-lists.html"/>
            <item name="Issue Tracking" href="issue-tracking.html"/>
            <item name="Continuous Integration" href="integration.html"/>
            <item name="Project Plugins" href="plugins.html"/>
            <item name="Project License" href="license.html"/>
            <item name="Project Team" href="team-list.html"/>
            <item name="Project Summary" href="project-summary.html"/>
            <item name="Dependencies" href="dependencies.html"/>
        </menu>
```

```
        <menu name="Reports">
        </menu>
        </body>
</project>
```

Running `mvn clean site` generates the site with the new logo and navigation, as shown in Figure 7-7.

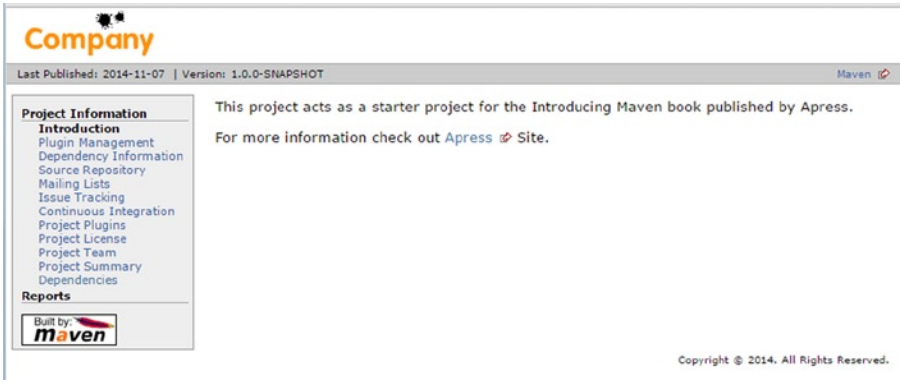


Figure 7-7. *About page with the new logo*

Generating Javadoc Reports

Javadoc is the de facto standard for documenting Java code. It helps developers understand what a class or a method does. Javadoc also highlights deprecated classes, methods, or fields.

Maven provides a Javadoc plug-in, which uses the Javadoc tool for generating Javadocs. Integrating the Javadoc plug-in simply involves declaring it in the `reporting` element of `pom.xml` file, as shown in Listing 7-4. Plug-ins declared in the `pom reporting` element are executed during site generation.

Listing 7-4. The `pom.xml` Snippet with Javadoc Plug-in

```
<project>
        <!—Content removed for brevity-->
 <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.10.1</version>
      </plugin>
    </plugins>
  </reporting>
</project>
```

Now that you have the Javadoc plug-in configured, let's run `mvn clean site` to generate the Javadoc. After the command successfully runs, you will notice the `apidocs` folder created under `gswm /target/site`. Double-click the `index.html` file under `apidocs`, and you will be able to browse the Javadoc. Figure 7-8 shows the Javadoc generated for the `gswm` project.

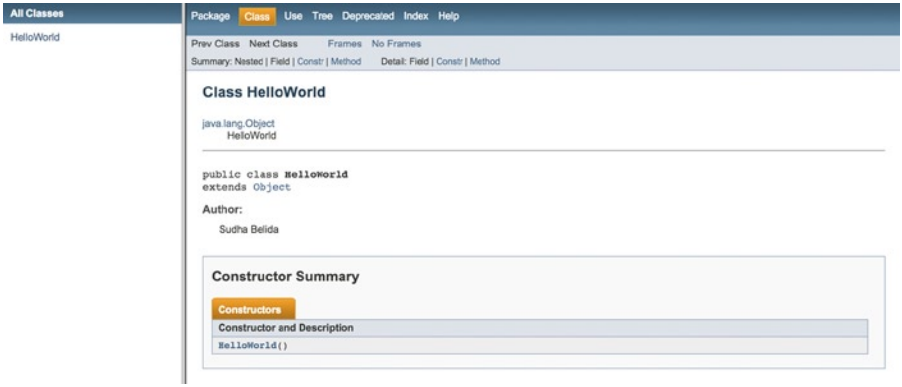


Figure 7-8. *Generated Javadoc page*

Generating Unit Test Reports

Test-driven development has become the norm in enterprises today. Unit tests provide immediate feedback to developers and allow them to build quality code. Considering how important tests are, Maven executes all of the tests for each build. Any test failures result in a failed build.

Maven offers the Surefire plug-in that provides a uniform interface for running tests created by frameworks such as JUnit or TestNG. It also generates execution results in various formats such as XML and HTML. These published results enable developers to find and fix broken tests quickly.

The Surefire plug-in is configured in the same way as the Javadoc plug-in in the `reporting` section of the `pom` file. Listing 7-5 shows the Surefire plug-in configuration.

Listing 7-5. The `pom.xml` Snippet with Surefire Plug-in

```
<project>
        <!—Content removed for brevity-->
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
```

```
        <version>2.17</version>
      </plugin>
    </plugins>
  </reporting>
</project>
```

Now that Surefire is configured, let's generate a Maven site by running mvn clean site command. Upon successful execution of the command, you will see a `Surefire Reports` folder generated under `gswm\target`. It contains the test execution results in XML and TXT formats. The same information will be available in HTML format in the `surefire-report.html` file under `site` folder. Figure 7-9 shows Surefire Report for the `gswm` project.

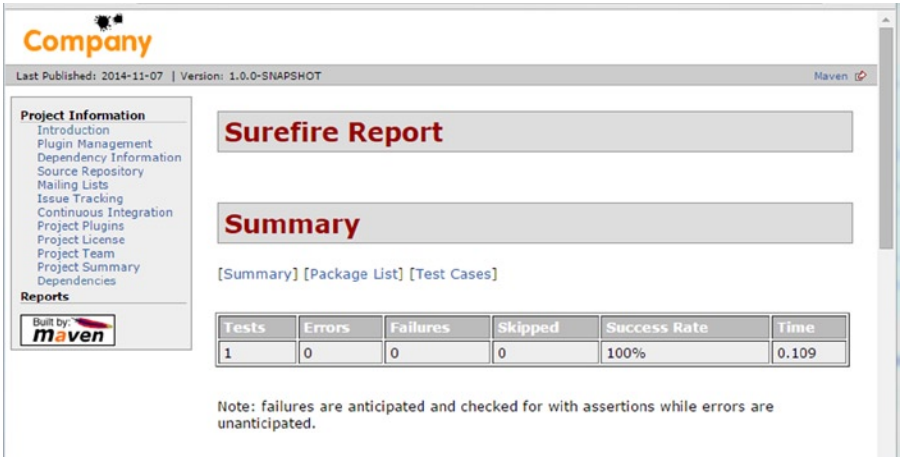


Figure 7-9. *Generated Surefire Report*

Generating Code Coverage Reports

Code coverage is a measurement of how much source code is being exercised by automated tests. Essentially, it provides an indication of the quality of your tests. *Emma* and *Cobertura* are two popular open source code coverage tools for Java.

In this section, you will use Cobertura for measuring this project's code coverage. Configuring Cobertura is similar to other plug-ins, as shown in Listing 7-6.

Listing 7-6. The `pom.xml` Snippet with the Cobertura Plug-in

```
<project>
        <!—Content removed for brevity-->
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
        <version>2.6</version>
      </plugin>
   </plugins>
  </reporting>
</project>
```

Now that the plug-in is configured, let's generate the site using the `mvn clean site` command. Upon successful completion of the command, Cobertura will create a `cobertura` folder under `gswm \target\site`. Launch the report by double-clicking the `index.html` file. The report should be similar to the one shown in Figure 7-10.

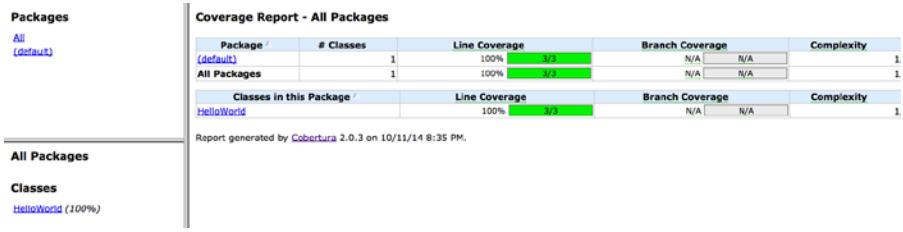


Figure 7-10. *Generated Cobertura report*

Generating the FindBugs Report

FindBugs is a tool for detecting defects in Java code. It uses static analysis to detect bug patterns, such as infinite recursive loops and null pointer dereferences. Listing 7-7 shows the FindBugs configuration.

Listing 7-7. The `pom.xml` Snippet with FindBugs Plug-in

```
<project>
        <!—Content removed for brevity-->
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>findbugs-maven-plugin</artifactId>
```

```
        <version>3.0.0</version>
      </plugin>
    </plugins>
  </reporting>
</project>
```

Once the Maven site gets generated, open the `findbugs.html` file under `C:\apress\gswm-book\chapter7\gswm\target\site` to launch the FindBugs report. It should be similar to the one shown in Figure 7-11.

Figure 7-11. *Generated FindBugs Bug Detector Report*

Summary

The documentation and reporting capabilities provided by Maven play an important role in creating maintainable quality software. This chapter explained the basics of using the site life cycle and the configuration needed to produce documentation. You also looked at generating Javadocs, test coverage, and FindBugs reports.

In the next chapter, we will explain how to integrate Maven with Nexus and SVN. You will also learn about Maven's release process.

CHAPTER 8

Maven Release

Integration with Nexus

Repository managers are a key part of Maven deployment in enterprises. *Repository managers* act as a proxy of public repositories, facilitate artifact sharing and team collaboration, ensure build stability, and enable the governance of artifacts used in the enterprise.

Nexus is a popular open source repository manager from Sonatype. It is a web application that allows you to maintain internal repositories and access external repositories. It allows repositories to be grouped and accessed via a single URL. This enables the repository administrator to add and remove new repositories behind the scenes without requiring developers to change the configuration on their computers. Additionally, it provides hosting capabilities for sites generated using Maven site and artifact search capabilities.

Before we look at integrating Maven with Nexus, you will need to install Nexus on your local machine. Nexus is distributed as an archive, and it comes bundled with a Jetty instance. Download the Nexus distribution (`.zip` version for Windows) from Sonatype's web site at `www.sonatype.org/nexus/go/`. At the time of this writing, version 2.10.0-02 of Nexus is available. Unzip the file, and place the contents on your machine. In this book, we assume the contents to be in the `C:\tools\nexus` folder.

Note Most enterprises typically have repository managers installed and available on a central server. If you already have access to a repository manager, skip this part of the installation.

Launch your command line in *administrator mode* and navigate to the bin folder located under `C:\tools\nexus\nexus-2.10.0-02`. Then run the command `nexus install`. You will see the success message as illustrated in Figure 8-1. This installs the native service wrapper that enables Jetty to run.

```
C:\tools\nexus\nexus-2.10.0-02\bin>nexus install
wrapper  | nexus installed.
```

Figure 8-1. *Success message when installing Nexus*

■ **Note** Nexus 2.10 requires JRE 1.7 to function properly. Make sure you have version 1.7 of JDK/JRE installed on your local machine. Also, make sure that `JAVA_HOME` is pointing to version 1.7 of the JDK.

On the same command line, run the command `nexus start` to launch Nexus. Figure 8-2 shows the result of running this command.

```
C:\tools\nexus\nexus-2.10.0-02\bin>nexus start
wrapper  | Starting the nexus service...
wrapper  | Waiting to start...
wrapper  | Waiting to start...
wrapper  | Waiting to start...
wrapper  | nexus started.
```

Figure 8-2. *Starting Nexus*

By default, Nexus runs on port 8081. Launch a web browser and navigate to Nexus at `http://localhost:8081/nexus`. Figure 8-3 shows the Nexus launch screen. Log in and browse Nexus with the username `admin` and password `admin123`.

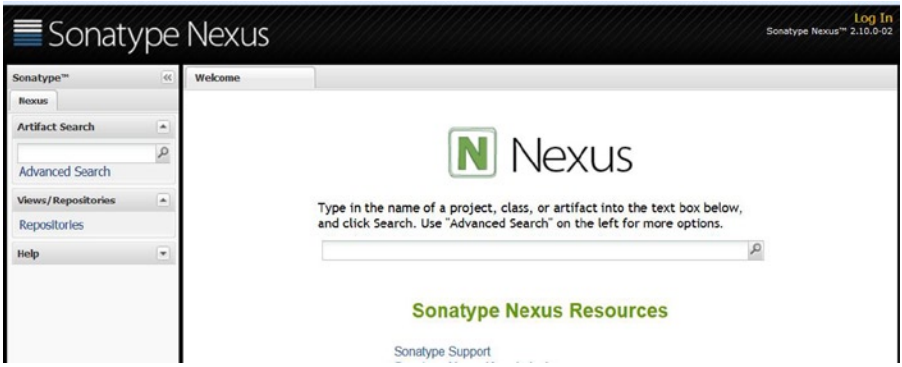


Figure 8-3. *Nexus launch screen*

Now that Nexus is installed, let's modify the gwsm project located under C:\apress\gswm-book\chapter8. You will start by adding a distributionManagement element in the pom.xml file, as shown in Listing 8-1. This element is used to declare the location where the project's artifacts will be when deployed. The repository element indicates the location where released artifacts will be deployed. Similarly, the snapshotRepository element identifies the location where the SNAPSHOT versions of the project will be stored.

Listing 8-1. The pom.xml with distributionManagement

```
<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>

        <!-- Content removed for brevity -->
        <distributionManagement>
           <repository>
                <id>nexusReleases</id>
              <name>Releases</name>
        <url>http://localhost:8081/nexus/content/repositories/releases</url>
           </repository>
           <snapshotRepository>
              <id>nexusSnapshots</id>
              <name>Snapshots</name>
              <url>http://localhost:8081/nexus/content/repositories/
snapshots</url>
           </snapshotRepository>
        </distributionManagement>
<!-- Content removed for brevity -->
</project>
```

■ **Note** Out of the box, Nexus comes with Releases and Snapshots repositories. By default, SNAPSHOT artifacts will be stored in the Snapshots Repository, and release artifacts will be stored in the Releases repository.

Like most repository managers, deployment to Nexus is a protected operation. You provide the credentials needed to interact with Nexus in the settings.xml file.

Listing 8-2 shows the settings.xml file with the server information. The Nexus deployment user with password deployment123 is provided out of the box. Notice that the IDs declared in the server tag — nexusReleases and nexusSnapshots must match the IDs of the repository and snapshotRepository declared in the pom.xml file. Replace the contents of the settings.xml file in the C:\Users\<<USER_NAME>>\.m2 folder with the code in Listing 8-2.

Listing 8-2. Settings.xml File with Server Information

```
<?xml version="1.0" encoding="UTF-8" ?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/
settings-1.0.0.xsd">
<servers>
   <server>
      <id>nexusReleases</id>
      <username>deployment</username>
      <password>deployment123</password>
   </server>
   <server>
      <id>nexusSnapshots</id>
      <username>deployment</username>
      <password>deployment123</password>
   </server>
</servers>
</settings>
```

This concludes the configuration steps for interacting with Nexus. At the command line, run the command `mvn deploy` under the directory `C:\apress\gswm-book\chapter8\gswm`. Upon successful execution of the command, you will see the SNAPSHOT artifact under Nexus at `http://localhost:8081/nexus/content/repositories/snapshots/com/apress/gswmbook/gswm/1.0.0-SNAPSHOT/`, as shown in Figure 8-4.

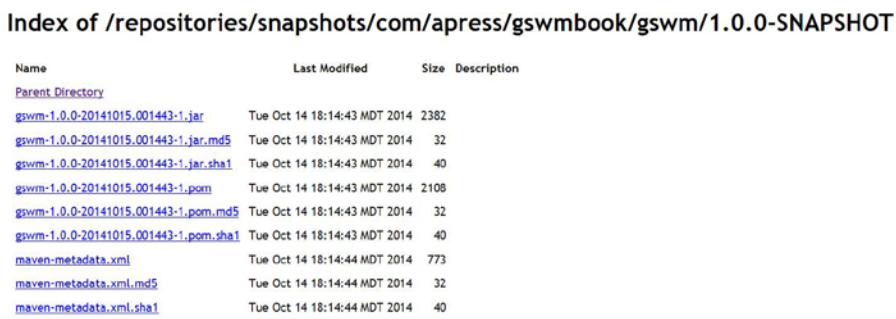


Figure 8-4. *SNAPSHOT artifact under Nexus*

Project Release

Releasing a project is a complex process, and it typically involves the following steps:

- Verify that there are no uncommitted changes on the local machine.
- Remove SNAPSHOT from the version in the `pom.xml` file.
- Make sure that project is not using any SNAPSHOT dependencies.
- Check in the modified `pom.xml` file to your source control.
- Create a source control tag of the source code.
- Build a new version of the artifact, and deploy it to a repository manager.
- Increment the version in the trunk's `pom.xml` file, and prepare for the next development cycle.

Maven has a release plug-in that provides a standard mechanism for executing the above steps and releasing project artifacts. As you can see, as part of its release process, Maven heavily interacts with the source control system. In this section, you will be using *Subversion* (SVN) as the source control system. A typical interaction between Maven and SVN is shown in Figure 8-5. The SVN server houses repositories containing an enterprise's projects. Maven releases are typically performed on a developer or build machine. Maven requires SVN command line tools to be installed on such machines. The SVN command line tools allow Maven to interact with SVN and perform operations such as checking out code, creating tags, and so forth.

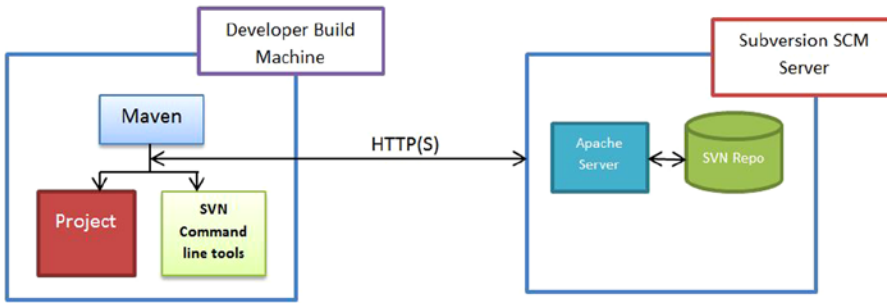


Figure 8-5. *Interaction between Maven and Subversion*

Before we delve deeper into the Maven release process, you need to get your local machine ready by completing the following steps:

1. Install Subversion server and SVN command line tools on your local machine.
2. Create a Subversion repository.
3. Check the project you will be using into the repository.

Subversion Server Command Line Tools Installation

There are several open source projects or commercial companies that provide SVN servers. In this project, you'll be using a Subversion server from VisualSVN.

Start the installation process by downloading the 64-bit VisualSVN Server executable from `www.visualsvn.com/downloads/`. As you can see from Figure 8-6, the server executable comes bundled with SVN command line tools.

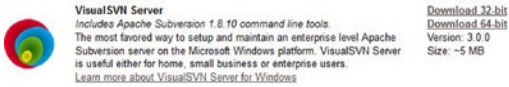


Figure 8-6. *Download for VisualSVN Server*

■ **Note** Enterprises typically have Subversion installed on a centralized server and available for use. If you already have read/write access to a Subversion server, you can skip the Subversion server installation steps. However, you need to have SVN command line tools installed on the machine where you are performing the Maven release. We recommend VisualSVN's "Apache Subversion command line tools," which you can download and install from `www.visualsvn.com/downloads/`.

Once downloaded, double-click the `VisualSVN-Server-3.0.0-x64.exe` install file to launch the install screen. Accept the End-User Licensing Agreement and, on the ensuing screen, make sure that "VisualSVN Server and Management Console" is selected and the "Add Subversion command-line tools to the PATH environment variable" option is checked, as shown in Figure 8-7.

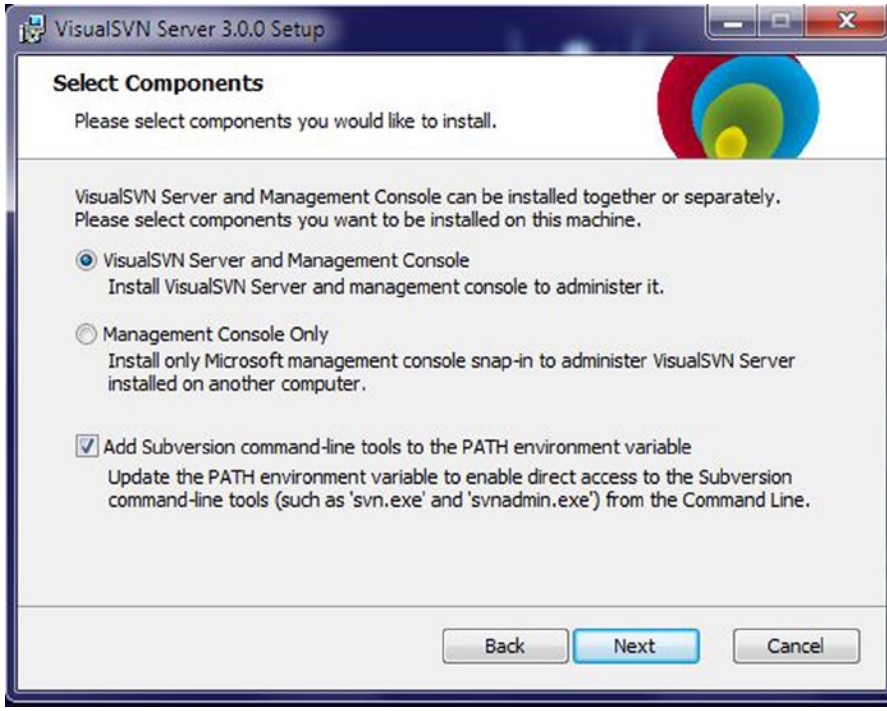


Figure 8-7. *VisualSVN server set up*

VisualSVN Server comes in two flavors: Standard Edition and Enterprise Edition. The features provided by the Standard Edition will satisfy your needs in this chapter. Click the Standard Edition button on VisualSVN Server Editions screen. On the following screen, uncheck the Use secure connection check box as shown in Figure 8-8.

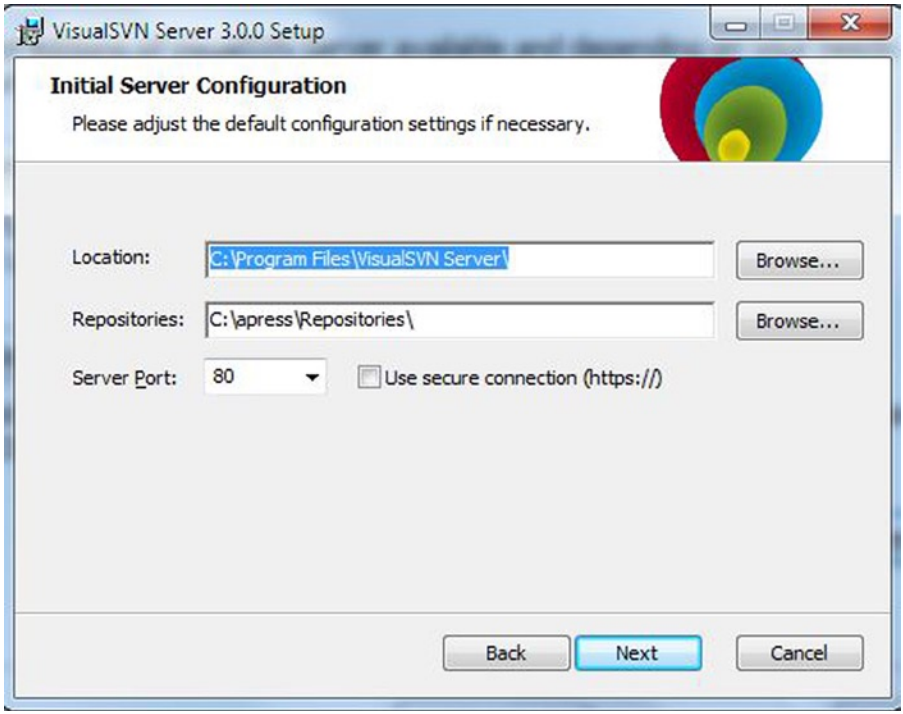


Figure 8-8. *Install and Repository Locations*

On the next screen, click the Install button to start the installation. After successful installation of SVN Server, make sure that the SVN command line tools are properly installed. To do so, open a new command line and run the command `svn help`. You should see output similar to that shown in Figure 8-9.

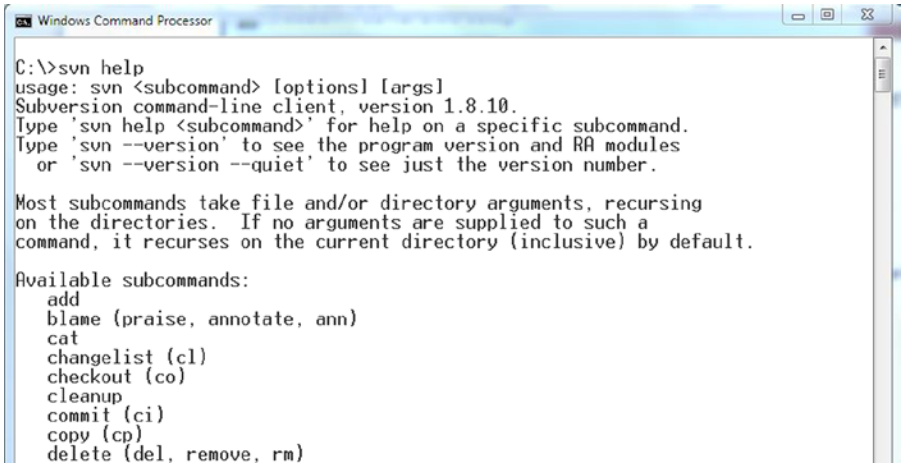


Figure 8-9. *Output after running the* `svn help` *command*

Creating a Repository

Subversion repositories are used to manage files and folders and track any modifications made to those files and folders. VisualSVN provides a great graphical user interface tool called VisualSVN Server Manager, which makes creation and management of repositories a breeze. On your Windows machine, go to All Programs ➤ VisualSVN and launch VisualSVN Server Manager. Follow the steps below to create a new repository:

1. Under the Repositories section of Server Manager, click the Create a new repository link.
2. Leave the Regular FSFS repository option selected in the Repository Type screen. Click Next.
3. On the Figure 8-10, enter **gswm** as the repository name. Click Next.

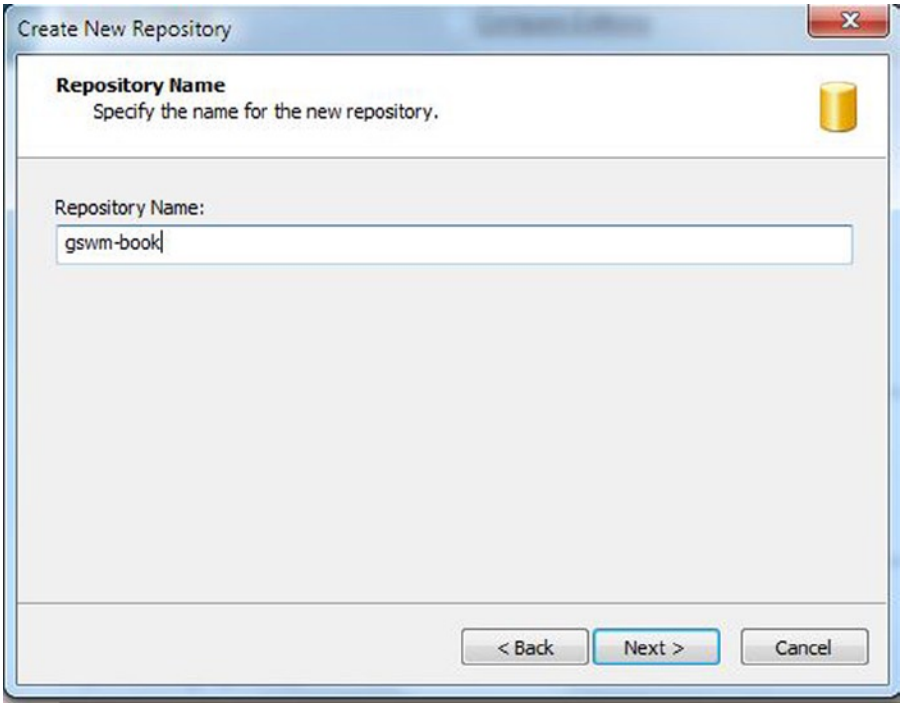


Figure 8-10. *Repository creation*

4. Select the Single-project repository option in the Repository Structure screen, as shown in Figure 8-11. Click Next.

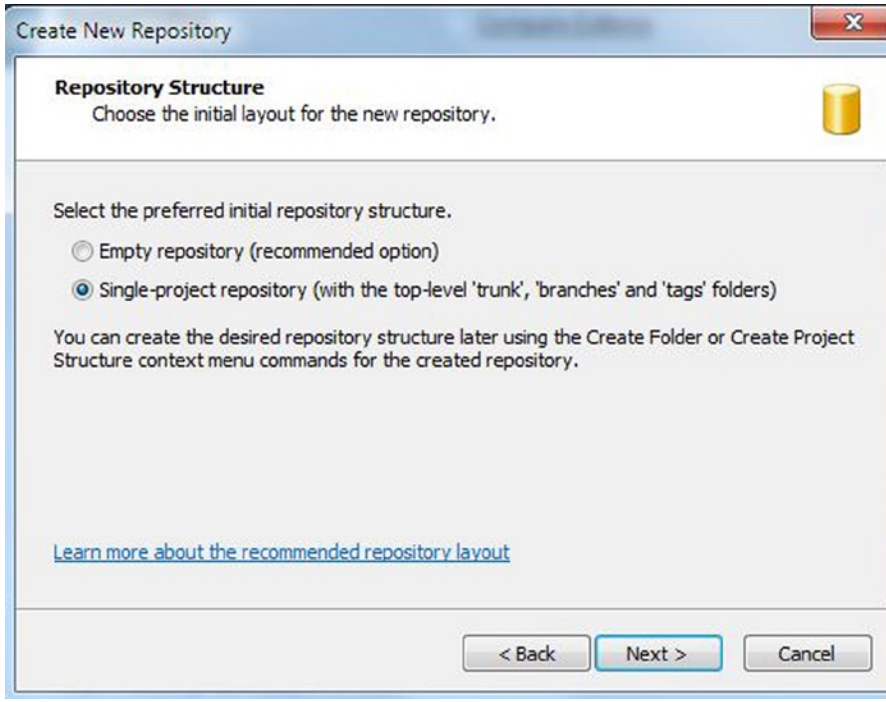


Figure 8-11. *New repository structure*

5. In the Repository Access Permissions screen, accept the default "All Subversion users have Read/Write access." Click Create. A *Repository Created Successfully* message should be displayed. Click Finish.

The final step in getting the repository ready is to create a new user that has read/write access on the `gwsm-book` repository. Follow the steps below to create a new user:

1. On the VisualSVN Service Manager home screen, click the Create a new user link in the Subversion Authentication section, as shown in Figure 8-12.

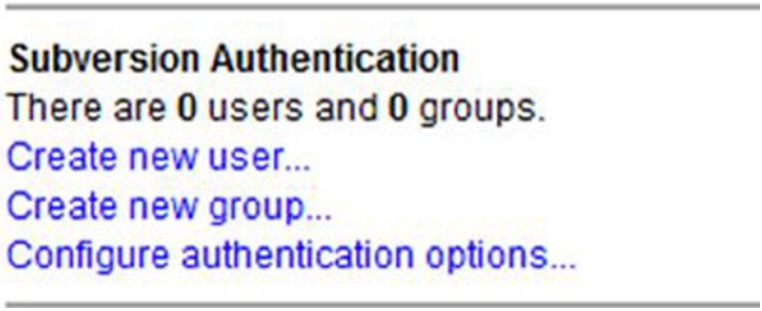


Figure 8-12. *Subversion authentication section*

2. In the Create New User window, enter *gswm* as the username and *gswm* as the password, as shown in Figure 8-13. Click OK.

Figure 8-13. *New Subversion User*

Checking in Source Code

The final step in getting your machine ready for Maven release is checking in the `gswm` project under `C:\apress\gswm-book\chapter8\gswm` to the newly created repository. Using your command line, navigate to the `C:\apress\gswm-book\chapter8\gswm` folder and run the following commands sequentially:

```
svn checkout http://localhost/svn/gswm/trunk/
C:/apress/gswm-book/chapter8/gswm --username gswm
--password gswm

svn add src

svn add pom.xml

svn commit -m "Inital commit"
```

The output of running the above commands is shown in Figure 8-14.

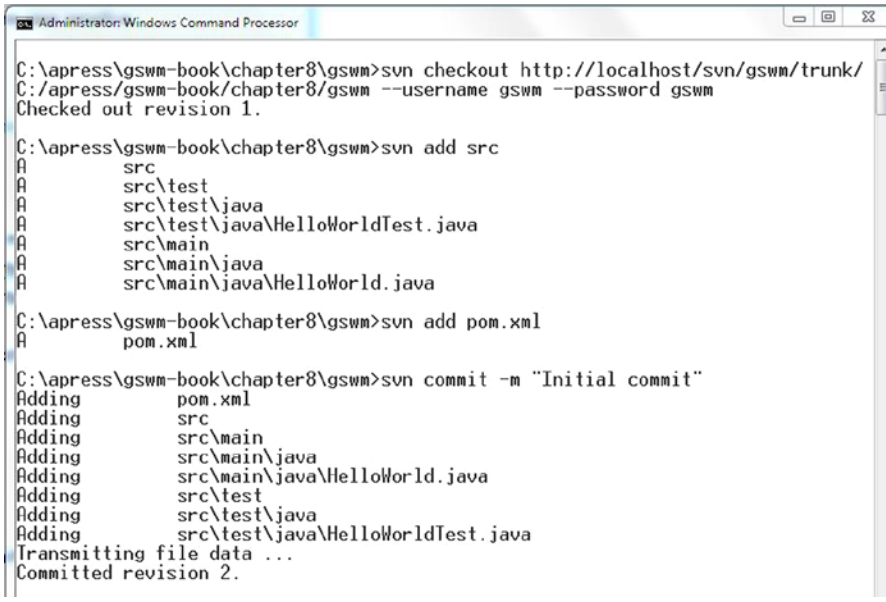


Figure 8-14. *Output from the svn initial commit*

Using your browser, navigate to `http://localhost/svn/gswm/trunk`. When prompted, enter the username *gswm* and password *gswm*, and you will see the checked-in code. Figure 8-15 shows the expected browser screen.

Figure 8-15. *Project checked into SVN*

Maven Release

Releasing an artifact using Maven's release process requires using two important goals: `prepare` and `perform`. Additionally, the release plug-in provides a `clean` goal that comes in handy when things go wrong.

Prepare Goal

The `prepare` goal, as the name suggest, prepares a project for release. As part of this stage, Maven performs the following operations:

- *check-poms*: Checks that the version in the `pom.xml` file has SNAPSHOT in it.
- *scm-check-modifications*: Checks if there are any uncommitted changes.
- *check-dependency-snapshots*: Checks the `pom` file to see if there are any SNAPSHOT dependencies. It is a best practice for your project to use released dependencies. Any SNAPSHOT dependencies found in the `pom.xml` file will result in release failure.
- *map-release-versions*: When `prepare` is run in an interactive mode, the user is prompted for a release version.
- *map-development-versions*: When `prepare` is run in an interactive model, the user is prompted for the next development version.
- *generate-release-poms*: Generates the release `pom` file.
- *scm-commit-release*: Commits the release of the `pom` file to the SCM.

- *scm-tag*: Creates a release tag for the code in the SCM.
- *rewrite-poms-for-development*: The `pom` file is updated for the new development cycle.
- *remove-release-poms*: Deletes the `pom` file generated for the release.
- *scm-commit-development*: Submits the `pom.xml` file with the development version.
- *end-release*: Completes the `prepare` phase of the release.

To facilitate this, you would provide the SCM information in the project's `pom.xml` file. Listing 8-3 shows the `pom.xml` file snippet with the SCM information.

Listing 8-3. The `pom.xml` with SCM Information

```
<project>
  <modelVersion>4.0.0</modelVersion>
        <!-- Content removed for brevity -->

    <scm>
        <connection>scm:svn:http://localhost/svn/gswm/trunk</connection>
        <developerConnection>scm:svn:http://localhost/svn/gswm/trunk</
developerConnection>
        <url>http://localhost/svn/gswm/trunk</url>
   </scm>
   <!-- Content removed for brevity -->
</project>
```

Once you have updated the `pom.xml` file on your local machine, commit the modified file to SVN by running the following command:

```
svn commit -m "Added SVN Information"
```

The output of running this command is shown in Figure 8-16.

```
C:\apress\gswm-book\chapter8\gswm>svn commit -m "Added SVN Information"
Sending        pom.xml
Transmitting file data .
Committed revision 3.
```

Figure 8-16. *Output from running the* `svn` *commit*

In order for Maven to communicate successfully with the SVN server, it needs credentials with write access on the server. You provide that information in the `settings.xml` file, as shown in Listing 8-4. The ID for the `server` element is declared as `localhost`, as it must match the SVN host name.

Listing 8-4. The pom.xml with SVN Server Details

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

<servers>
   <server>
      <id>nexusReleases</id>
      <username>deployment</username>
      <password>deployment123</password>
   </server>
   <server>
      <id>nexusSnapshots</id>
      <username>deployment</username>
      <password>deployment123</password>
   </server>
   <server>
     <id>localhost</id>
     <username>gswm</username>
     <password>gswm</password>
   </server>
 </servers>
</settings>
```

You now have the entire configuration required for Maven's prepare goal. Listing 8-5 shows the results of running the prepare goal. Because the prepare goal was run in interactive mode, Maven will prompt you for the release version, release tag or label, and the new development version. Accept Maven's proposed default values by pressing Enter for each prompt.

Listing 8-5. Maven prepare Command

```
C:\apress\gswm-book\chapter8\gswm>mvn release:prepare
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-release-plugin:2.3.2:prepare (default-cli) @ gswm ---
[INFO] Verifying that there are no local modifications...
[INFO]   ignoring changes on: **\release.properties, **\pom.xml.next, **\
pom.xml.releaseBackup, **\pom.xml.backup, **\pom.xml.branch, **\pom.xml.tag
[INFO] Executing: cmd.exe /X /C "svn --username gswm --password ***** --no-
auth-cache --non-interactive status"
```

```
[INFO] Working directory: C:\apress\gswm-book\chapter8\gswm
[INFO] Checking dependencies and plugins for snapshots ...
What is the release version for "Getting Started with Maven"? (com.apress.
gswmbook:gswm) 1.0.0: :
What is SCM release tag or label for "Getting Started with Maven"? (com.
apress.gswmbook:gswm) gswm-1.0.0: :
What is the new development version for "Getting Started with Maven"? (com.
apress.gswmbook:gswm) 1.0.1-SNAPSHOT: :
[INFO] Transforming 'Getting Started with Maven'...
------------------------------------------------------------------------
[INFO] [INFO] Building jar: C:\apress\gswm-book\chapter8\gswm\target\gswm-
1.0.0.jar
[INFO] [INFO] ------------------------------------------------------------------------
[INFO] [INFO] BUILD SUCCESS
[INFO] [INFO] ------------------------------------------------------------------------
[INFO] [INFO] Total time: 1.654 s
[INFO] [INFO] Finished at: 2014-10-22T23:10:44-06:00
[INFO] [INFO] Final Memory: 11M/27M
[INFO] [INFO] ------------------------------------------------------------------------
[INFO] Checking in modified POMs...
[INFO] Executing: cmd.exe /X /C "svn --username gswm --password ***** --no-
auth-cache --non-interactive commit --file C:\Users\<<USER_NAME>>\AppData\
Local\Temp\maven-scm-203076178.commit --targets C:\Users\<<USER_NAME>>\
AppData\Local\Temp\maven-scm-5496549062663519106-targets"
[INFO] Working directory: C:\apress\gswm-book\chapter8\gswm
[INFO] Tagging release with the label gswm-1.0.0...
[INFO] Executing: cmd.exe /X /C "svn --username gswm --password ***** --no-
auth-cache --non-interactive copy --file C:\Users\<<USER_NAME>>\AppData\
Local\Temp\maven-scm-85876759.commit --revision 6 http://localhost/svn/gswm/
trunk http://localhost/svn/gswm/tags/gswm-1.0.0"
[INFO] Working directory: C:\apress\gswm-book\chapter8\gswm
[INFO] Transforming 'Getting Started with Maven'...
[INFO] Not removing release POMs
[INFO] Checking in modified POMs...
[INFO] Executing: cmd.exe /X /C "svn --username gswm --password ***** --no-auth-
cache --non-interactive commit --file C:\Users\<<USER_NAME>>\AppData\Local\
Temp\maven-scm-112170711.commit --targets C:\Users\<<USER_NAME>>\AppData\
Local\Temp\maven-scm-244
0605286339680080-targets"
[INFO] Working directory: C:\apress\gswm-book\chapter8\gswm
[INFO] Release preparation complete.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 33.711 s
```

```
[INFO] Finished at: 2014-10-22T23:10:44-06:00
[INFO] Final Memory: 7M/17M
[INFO] ------------------------------------------------------------------------
```

Notice the `svn` commands getting executed as part of the `prepare` goal. Successful completion of the `prepare` goal will result in the creation of an SVN tag, as shown in Figure 8-17. The `pom.xml` file in the `gswm` project will now have version 1.0.1-SNAPSHOT.

Figure 8-17. *SVN tag created upon* `prepare` *execution*

Clean Goal

The `prepare` goal performs a lot of activities and generates temporary files, such as `release.properties` and `pom.xml.releaseBackup`, as part of its execution. Upon successful completion, it cleans up those temporary files. Sometimes the `prepare` goal might fail (is unable to connect to SVN, for example) and leave the project in a *dirty* state. This is where the release plug-in's `clean` goal comes into the picture. As the name suggests, it deletes any temporary files generated as part of release execution.

■ **Note** The `clean` goal must be used only when the `prepare` goal fails.

Perform Goal

The `perform` goal is responsible for checking out code from the newly created tag and builds and deploys the released code into the remote repository.

The following phases are executed as part of `perform` goal:

- *verify-completed-prepare-phases*: This validates that a `prepare` phase has been executed prior to running the `perform` goal.
- *checkout-project-from-scm*: Checks out the released code from the SCM tag.
- *run-perform-goal*: Executes the goals associated with `perform`. The default goal is deploy.

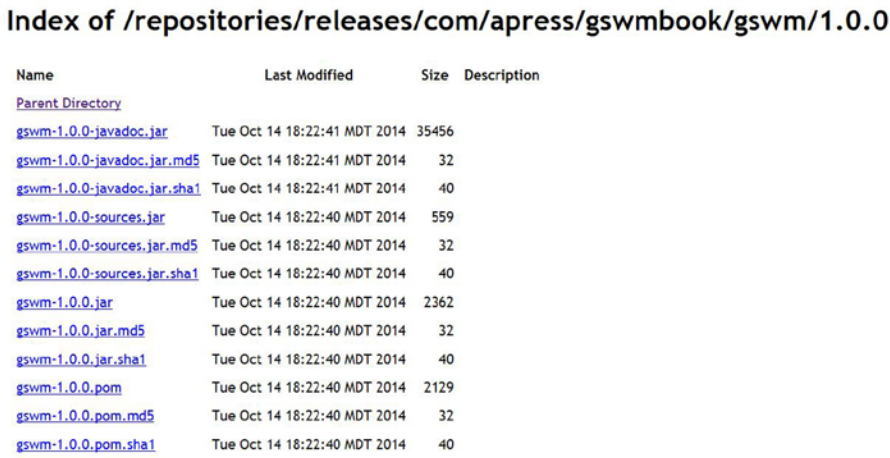
The output of running the `perform` goal on `gswm` project is shown in Listing 8-6.

Listing 8-6. Maven `perform` Command

```
C:\apress\gswm-book\chapter8\gswm>mvn release:perform
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building Getting Started with Maven 1.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------
[INFO]
[INFO] --- maven-release-plugin:2.3.2:perform (default-cli) @ gswm ---
[INFO] Checking out the project to perform the release ...
[INFO] Executing: cmd.exe /X /C "svn --username gswm --password ***** --no-
auth-cache --non-interactive checkout http://localhost/svn/gswm/tags/gswm-
1.0.0 C:\apress\gswm-book\chapter8\gswm\target\checkout"
[INFO] Working directory: C:\apress\gswm-book\chapter8\gswm\target
--------------------------------
[INFO] [INFO] Installing C:\apress\gswm-book\chapter8\gswm\target\checkout\
target\gswm-1.0.0.jar to C:\Users\<<USER_NAME>>\.m2\repository\com\apress\
gswmbook\gswm\1.0.0\gswm-1.0.0.jar
[INFO] [INFO] Installing C:\apress\gswm-book\chapter8\gswm\target\checkout\
pom.xml to C:\Users\<<USER_NAME>>\.m2\repository\com\apress\gswmbook\
gswm\1.0.0\gswm-1.0.0.pom
[INFO] [INFO] Installing C:\apress\gswm-book\chapter8\gswm\target\checkout\
target\gswm-1.0.0-sources.jar to C:\Users\<<USER_NAME>>\.m2\repository\com\
apress\gswmbook\
gswm\1.0.0\gswm-1.0.0-sources.jar
[INFO] [INFO] Installing C:\apress\gswm-book\chapter8\gswm\target\checkout\
target\gswm-1.0.0-javadoc.jar to C:\Users\<<USER_NAME>>\.m2\repository\com\
apress\gswmbook\gswm\1.0.0\gswm-1.0.0-javadoc.jar
[INFO] [INFO]
[INFO] [INFO] --- maven-deploy-plugin:2.7:deploy (default-deploy) @ gswm ---
[INFO] Uploading: http://localhost:8081/nexus/content/repositories/releases/
```

```
com/apress/gswmbook/gswm/1.0.0/gswm-1.0.0.jar
[INFO] 2/3 KB
[INFO] 3/3 KB
[INFO]
[INFO] Uploaded: http://localhost:8081/nexus/content/repositories/releases/
com/apress/gswmbook/gswm/1.0.0/gswm-1.0.0.jar (3 KB at 13.4 KB/sec)
[INFO] Uploading: http://localhost:8081/nexus/content/repositories/releases/
com/apress/gswmbook/gswm/1.0.0/gswm-1.0.0.pom
[INFO] 2/3 KB
[INFO] 3/3 KB
[INFO]
[INFO] Uploaded: http://localhost:8081/nexus/content/repositories/releases/
com/apress/gswmbook/gswm/1.0.0/gswm-1.0.0.pom (3 KB at 14.5 KB/sec)
[INFO] Downloading: http://localhost:8081/nexus/content/repositories/
releases/com/apress/gswmbook/gswm/maven-metadata.xml
[INFO]
[INFO] Uploaded: http://localhost:8081/nexus/content/repositories/releases/
com/apress/gswmbook/gswm/1.0.0/gswm-1.0.0-javadoc.jar (35 KB at 368.5 KB/sec)
[INFO] [INFO] BUILD SUCCESS
[INFO] [INFO] ------------------------------------------------------
[INFO] [INFO] Total time: 3.807 s
[INFO] [INFO] Finished at: 2014-10-22T23:26:36-06:00
[INFO] [INFO] Final Memory: 17M/42M
[INFO] [INFO] ------------------------------------------------------
[INFO] Cleaning up after release...
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 5.867 s
[INFO] Finished at: 2014-10-22T23:26:36-06:00
[INFO] Final Memory: 8M/19M
[INFO] ------------------------------------------------------------
```

This completes the release of the 1.0.0 version of the `gswm` project. The artifact ends up in the Nexus repository manager, as shown in Figure 8-18.


Index of /repositories/releases/com/apress/gswmbook/gswm/1.0.0

Name	Last Modified	Size	Description
Parent Directory			
gswm-1.0.0-javadoc.jar	Tue Oct 14 18:22:41 MDT 2014	35456	
gswm-1.0.0-javadoc.jar.md5	Tue Oct 14 18:22:41 MDT 2014	32	
gswm-1.0.0-javadoc.jar.sha1	Tue Oct 14 18:22:41 MDT 2014	40	
gswm-1.0.0-sources.jar	Tue Oct 14 18:22:40 MDT 2014	559	
gswm-1.0.0-sources.jar.md5	Tue Oct 14 18:22:40 MDT 2014	32	
gswm-1.0.0-sources.jar.sha1	Tue Oct 14 18:22:40 MDT 2014	40	
gswm-1.0.0.jar	Tue Oct 14 18:22:40 MDT 2014	2362	
gswm-1.0.0.jar.md5	Tue Oct 14 18:22:40 MDT 2014	32	
gswm-1.0.0.jar.sha1	Tue Oct 14 18:22:40 MDT 2014	40	
gswm-1.0.0.pom	Tue Oct 14 18:22:40 MDT 2014	2129	
gswm-1.0.0.pom.md5	Tue Oct 14 18:22:40 MDT 2014	32	
gswm-1.0.0.pom.sha1	Tue Oct 14 18:22:40 MDT 2014	40	

Figure 8-18. *Nexus with released artifact*

Summary

Internal repository managers such as Nexus allow enterprises to adopt Maven completely. In addition to serving as public repository proxies, they enable component sharing and governance. This chapter looked at integrating Maven with Nexus and walked you through the process of deploying an artifact to Nexus. You also learned Maven's release process and its different phases.

This discussion brings us to the end of our journey. Throughout the book, you have learned the key concepts behind Maven. We hope you will use your newly found Maven knowledge to automate and improve your existing build and project management processes.

Index

M

N, O

P, Q, R

S, T

U, V, W, X, Y, Z

Introducing Maven

Balaji Varanasi
Sudha Belida

Apress®

Introducing Maven

ISBN-13 (pbk): 978-1-4842-0842-7

ISBN-13 (electronic): 978-1-4842-0841-0

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Deepak Vohra
Developmental Editor: Gary Schwartz
Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade
Coordinating Editor: Mark Powers
Copy Editor: Mary Bearden
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781484208427. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

To Our Parents

Contents

About the Authors

Balaji Varanasi is a software development manager, author, speaker, and technology entrepreneur. He has over 14 years of experience architecting and developing high-performance, scalable Java and .NET mobile applications. During this period, he has worked in the areas of security, web accessibility, search, and enterprise portals. He has a master's degree in computer science from Utah State University and serves as adjunct faculty at the University of Phoenix, teaching programming and information system courses. He shares his insights and experiments at http://blog.inflinx.com.

Sudha Belida is a senior software engineer and technology enthusiast. She has more than seven years of experience working with Java and JEE technologies and frameworks, such as Spring, Hibernate, Struts, and AngularJS. Her interests lie in entrepreneurship and agile methodologies for software design and development. She has a master's degree in computational science from the University of Utah. In her free time, she likes to travel and enjoy the outdoor environment that Utah has to offer.

About the Technical Reviewer

Deepak Vohra is a consultant and a principal member of the NuBean.com software company. Deepak is a Sun-certified Java programmer and web component developer, and he has worked in the fields of XML, Java programming, and Java EE for over five years. Deepak is the coauthor of *Pro XML Development with Java Technology* (Apress, 2006). Deepak is also the author of the *JDBC 4.0* and *Oracle JDeveloper for J2EE Development, Processing XML Documents with Oracle JDeveloper 11g, EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g,* and *Java EE Development in Eclipse IDE* (Packt Publishing). He also served as the technical reviewer on *WebLogic*: *The Definitive Guide* (O'Reilly Media, 2004) and *Ruby Programming for the Absolute Beginner* (Cengage Learning PTR, 2007).

Acknowledgments

This book would not have been possible without the support of several people, and we take this opportunity to sincerely thank them.

Thanks to the amazing folks at Apress: Steve Anglin, Mark Powers, Matthew Moodie, and many others. We also owe a huge thank you to Deepak Vohra for his technical review and for the valuable feedback he provided.

Finally, we would thank our parents for their constant support and encouragement. Without them, this book wouldn't have been possible.