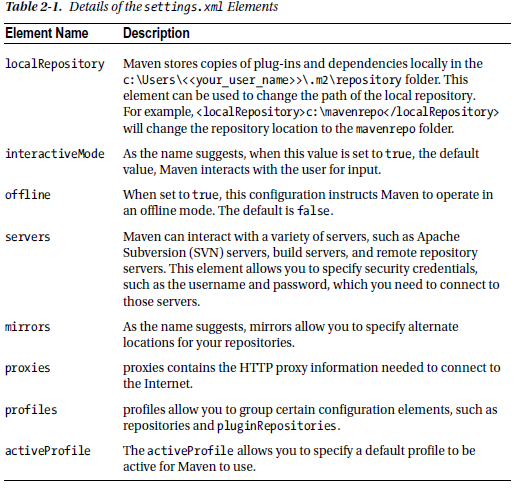
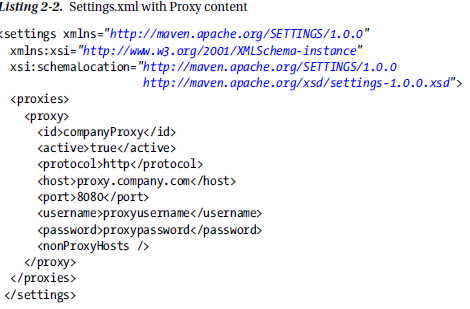
* Apache Maven is an open source, standards-based project management framework that simplifies the building, testing, reporting, and packaging of projects
* ***Desire for developing a standard way to build projects and to share generated artifacts easily across projects***
* Maven provides recommendations 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.
* With Maven, your projects follow a consistent structure and become IDE agnostic.
* 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 whatand not the *how*. The pom.xml file can also serve as a documentation tool, conveying your project dependencies and their versions.
* 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. 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.
* Maven provides a uniform interface for building projects. This frees developers from having to learn build idiosyncrasies so they can focus more on development.
* Maven is fully integrated with today’s continuous integration products such as Jenkins, Bamboo, and Hudson.
* *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.
* 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.
* 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. 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.

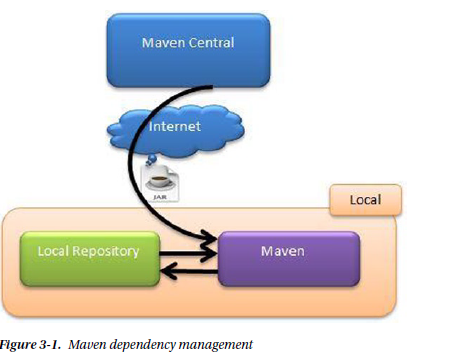




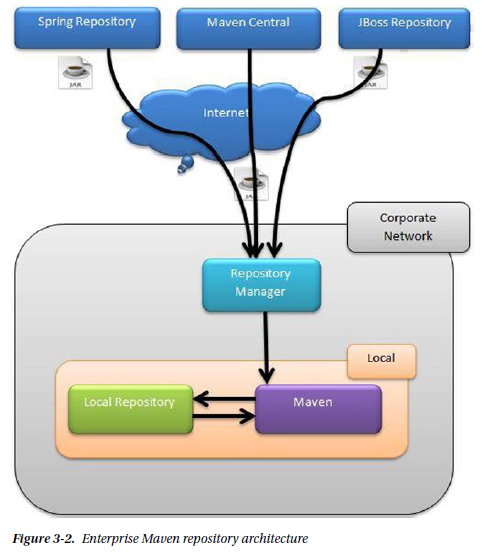
Some companies employ HTTP proxies to restrict access to the Internet. In those scenarios, running Maven will result in *Unable to download artefact* errors. To address this, edit the settings.xml file and add the proxy information specific to your company. A sample configuration is shown below:



* 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.

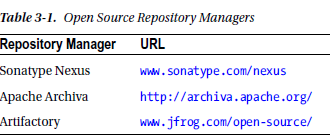


* Maven interacts with repositories that hold artifacts and related metadata. Repositories that are typically accessed over the web are considered remote and are maintained by a third party. The default remote repository with which Maven interacts is called *Maven Central*, and it is located at repo.maven.apache.org and uk.maven.org. Maven places the downloaded artifacts in the local repository.



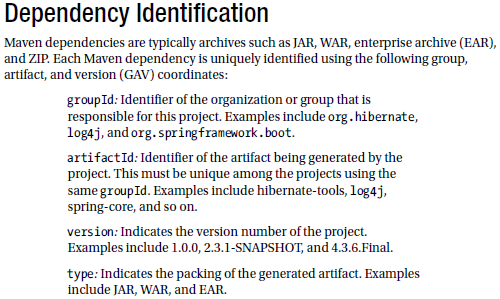
* The internal repository manager acts as a proxy to remote repositories. Because you have full control over the internal repository, you can regulate the types of artifacts allowed in your company. Additionally, you can also push your organization’s artifacts onto the server, thereby enabling collaboration. There are several open source repository managers. **The repository manager is used for following reasons:**
  + The first problem is that sharing company-related artifacts between teams is 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.

**Some examples of open source repository managers is:**

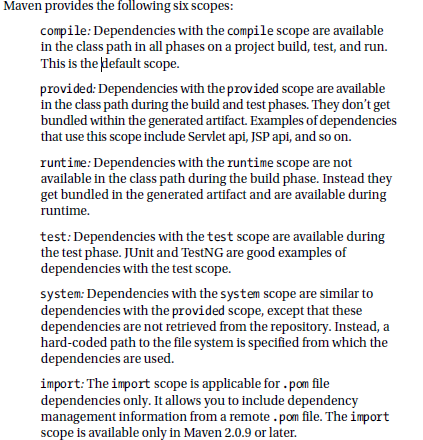
****

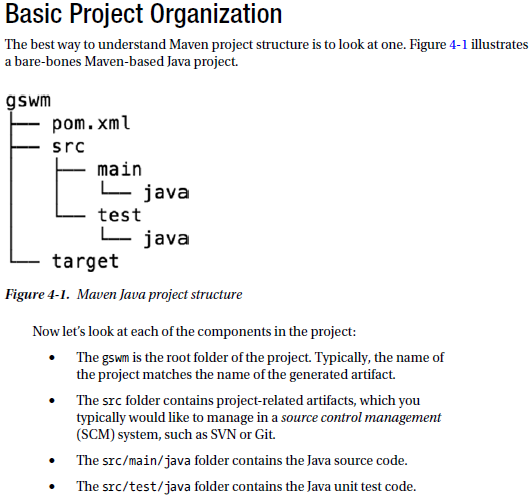
**NOTE:**

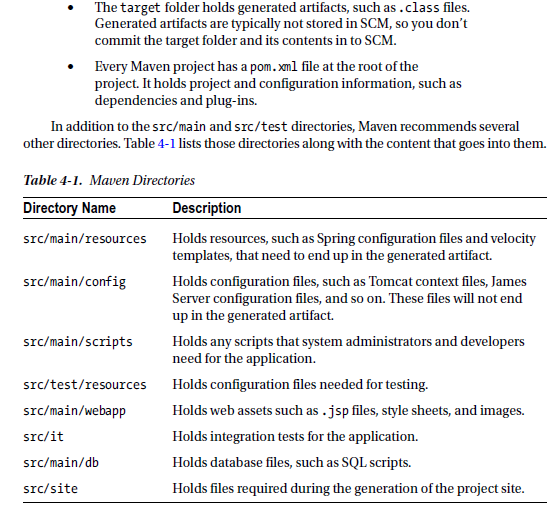
* Information regarding repositories can be provided in the settings.xml or the pom.xml file. There are pros and cons to each approach. Putting repository information in the pom.xml file can make your builds portable. It enables developers to download projects and simply build them without any further 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.

****

* 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.
* **Maven uses the concept of scope, which allows you to specify when and where you need a particular dependency (**For example, It means you only need junit dependencies when you are running tests**).**

****

****

****

* The POM file is the only required artefact in a maven project
* The packaging element in POM file tells Maven that it needs to create a JAR OR WAR archive for this project.
* In a continuous integration environment where you have to build often and push your latest build to the repository manager the use of version in POM file is very good idea

The basic syntax for version is

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

**Example:**

<version>1.0.0-SNAPSHOT</version>

* 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 artefact.
* Maven allows you to declare properties in the pom.xml file using the <properties /> element. These properties are highly useful for declaring dependency versions.

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

**<dependencies>**

**<dependency>**

**<groupId>junit</groupId>**

**<artifactId>junit</artifactId>**

**<version>${junit.version}</version>**

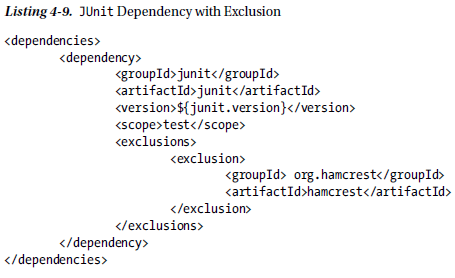
**<scope>test</scope>**

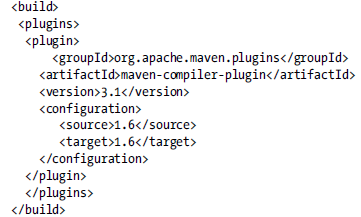
**</dependency>**

**</dependencies>**

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.

The exclusions element in the pom.xml file allows you to exclude a dependency, mostly used for transitive dependencies

******

In maven the plugin is defined in POM file to run various different goals at once. The goals are basically the granular tasks which the maven wants to complete successfully***.***

The plugin is used here to define various goals like above the code will be compiled using the java version 1.6

**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 artefacts from the target directory.

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

**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.

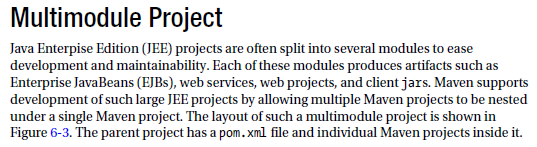
***Test*:** Runs unit tests using frameworks. This step doesn’t require that the application be packaged.

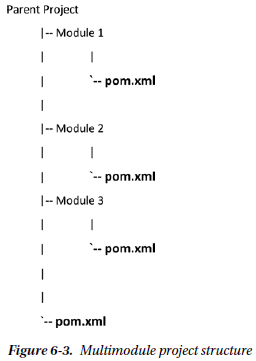
***Package*:** Assembles compiled code into a distributable format, such as JAR or WAR.

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

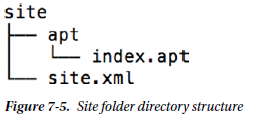
***Deploy*:** Pushes the built archive into a remote repository for use by other teams and team members.

Maven provides *archetypes*. Maven archetypes are project templates that allow users to generate new projects easily.





* 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. Maven provides the *site* life cycle that can be used to generate a project’s documentation. Maven allows you to specify content and configuration for site generation under the aptly named **src/site folder**.

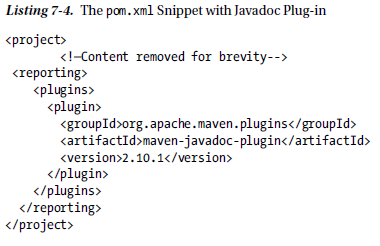


* The site.xml file, also known as the *site descriptor*, is used to customize the generated site. We will look at this element in just a second. The apt folder contains site content written in *Almost Plain Text* (APT) format. The APT format allows documentation to be created in a syntax that resembles 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.

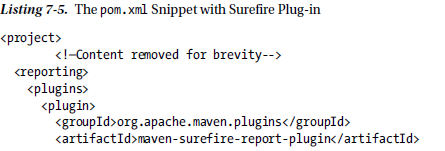


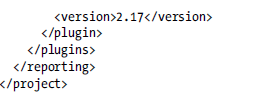


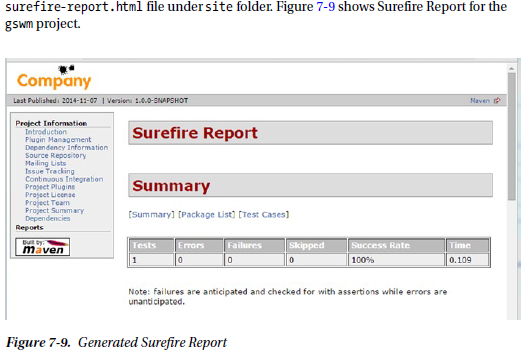
**Javadoc**is the de facto standard for documenting Java code. It helps developers understand what a class or a method does. Javadoc also highlights deprecated classes, methods, or fields. Maven provides a Javadoc plug-in, which uses the Javadoc tool for generating Java docs. Integrating the Javadoc plug-in simply involves declaring it in the reporting element of pom.xml file.

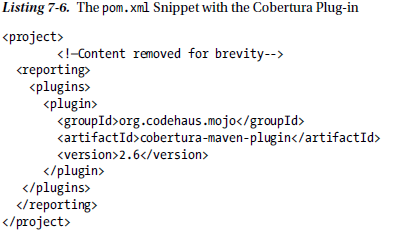


Maven offers the **Surefire plug-in** that provides a uniform interface for running tests created by frameworks such as JUnit or TestNG. It also generates execution results in various formats such as XML and HTML. These published results enable developers to find and fix broken tests quickly. The **Surefire** plug-in is configured in the same way as the Javadoc plug-in in the reporting section of the POM file.

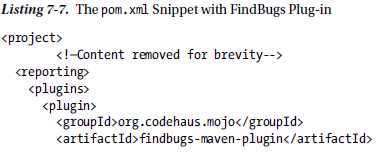


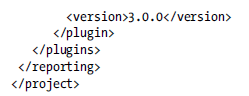




***Code coverage***is a measurement of how much source code is being exercised by automated tests. Essentially, it provides an indication of the quality of your tests. ***Emma* and *Cobertura*** are two popular open source code coverage tools for Java.

***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





**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/](http://www.sonatype.org/nexus/go/).

Once you have repository manager installed like nexus, you can start by adding a distribution Management element in the pom.xml file, 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.

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.

