Init

25 June 2021
16:16

Figure 3-1 shows a high-level view of Maven's dependency management. When you run your Maven project for the first time, Maven connects to the network and downloads artifacts and related metadata from remote repositories. The default remote repository is called *Maven Central*, and it is located at `repo.maven.apache.org` and `uk.maven.org`. Maven places a copy of these downloaded artifacts in its local repository. In subsequent runs, Maven will look for an artifact in its local repository; and upon not finding the artifact, Maven will attempt to download it from remote repository.

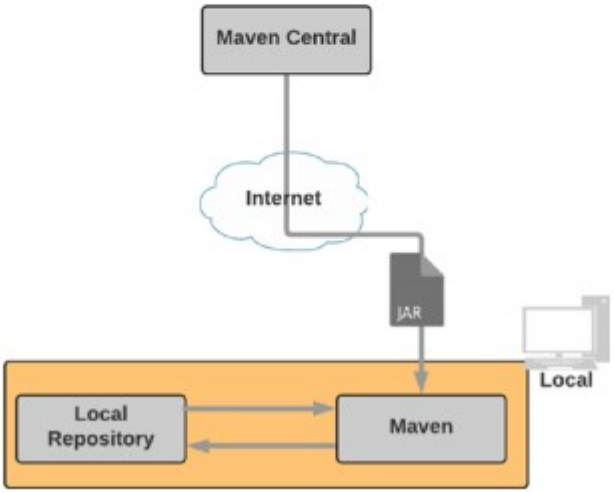


Figure 3-1. *Maven dependency management*

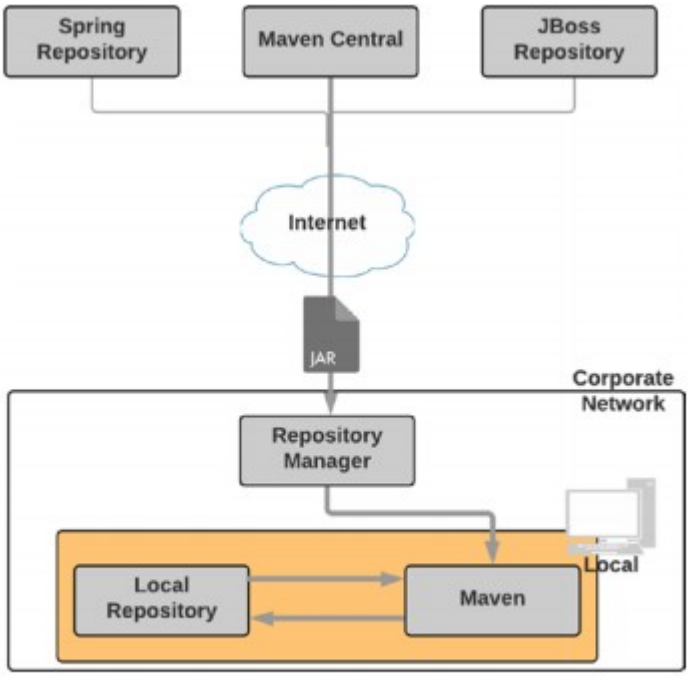


Figure 3-2. *Enterprise Maven repository architecture*

types of artifacts allowed in your company. Additionally, you can also push your organization's artifacts onto the repository manager, thereby enabling collaboration. There are several open source repository managers as shown in Table 3-1.

Table 3-1. *Open Source Repository Managers*

Repository Manager	URL
Nexus Repository OSS	www.sonatype.com/nexus-repository-oss
Apache Archiva	http://archiva.apache.org/
Artifactory Open Source	https://jfrog.com/open-source/#artifactory

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>
```

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`, `org.springframework` and com.companyname.

`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 5.4.2.Final.

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

Artifacts that are still in development are labeled with a SNAPSHOT in their versions. An example version is 1.0-SNAPSHOT. This tells Maven to look for an updated version of the artifact from remote repositories on a daily frequency.

Listing 3-2. Maven Dependency Tree Plug-in

```
[sudha]$mvn dependency:tree
[INFO] Scanning for projects...
[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
```

would conflict with versions loaded by containers. Maven provides an "excludes" tag to exclude a transitive dependency. Listing 3-3 shows the code to exclude the hamcrest library from JUnit dependency. As you can see, the `exclusion` element takes the `groupId` and `artifactId` coordinates of the dependency that you would like to exclude.

Listing 3-3. `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>
```

Dependency Scope

Consider a Java 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 JAR file when you are running the application inside a container such as Tomcat but not during code compilation or testing. 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 3-4 installs a `test.jar` file located in the `c:\apress\gswm-book\chapter3` folder.

Listing 3-4. 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] ------------< org.apache.maven:standalone-pom >---------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------[ pom ]-----------------------
[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\bavara\.m2\repository\com\apress\gswmbook\test\1.0.0\
test-1.0.0.jar
[INFO] Installing C:\Users\bavara\AppData\Local\Temp\
mvninstall5971068007426768105.pom to C:\Users\bavara\.m2\
repository\com\apress\gswmbook\test\1.0.0\test-1.0.0.pom
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time:  0.439 s
[INFO] Finished at: 2019-09-01T00:05:21-06:00
[INFO] ------------------------------------------------------------
```

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.

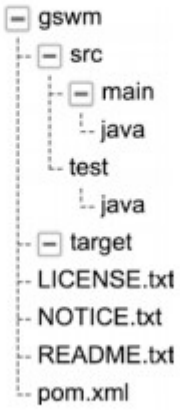


Figure 4-1. *Maven Java project structure*

- The `gswm` is the root folder of the project. Typically, the name of the root folder matches the name of the generated artifact.
- The `src` folder contains project-related artifacts such as source code or property files, 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 into SCM.
- The `LICENSE.txt` file contains license information related to project.
- The `README.txt` file contains information/instructions about the project.
- The `NOTICE.txt` file contains notices required by third-party libraries used by this project.
- 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.

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 Mail 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 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.

Goals and Plug-ins

Build processes generating artifacts such as JAR or WAR files typically require several steps and tasks to be completed successfully in a well-defined order. Examples of such tasks include compiling source code, running unit tests, and packaging of the artifact. 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] --- maven-compiler-plugin:3.1:compile (default-cli)
@ gswm ---
```

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] --- maven-clean-plugin:2.5:clean (default-cli)
@ gswm ---
[INFO] Deleting C:\apress\gswm-book\chapter5\gswm\target
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
```

Notice, the format of the command `clean:clean` 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
```

Listing 5-3. Maven `Help` Plug-in

```
mvn help:describe -Dplugin=compiler
```

```
  <build>
   <plugins>
    <plugin>
       <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
         <source>1.8</source>
         <target>1.8</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 1.8.

Lifecycle and Phases

Maven goals are granular and typically perform one task. Multiple goals need to be executed in an orderly fashion to perform complex operations such as generating artifacts or documentation. Maven simplifies these complex operations via lifecycle and phase abstractions such that build-related operations could be completed with a handful of commands.

Maven's build lifecycle constitutes a series of stages that get executed in the same order, independent of the artifact being produced. Maven refers to the stages in a lifecycle as *phases*. Every Maven project has the following three built-in lifecycles:

default: This lifecycle handles the compiling, packaging, and deployment of a Maven project.

clean: This lifecycle handles the deletion of temporary files and generated artifacts from the target directory.

site: This lifecycle handles the generation of documentation and site generation.

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

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.

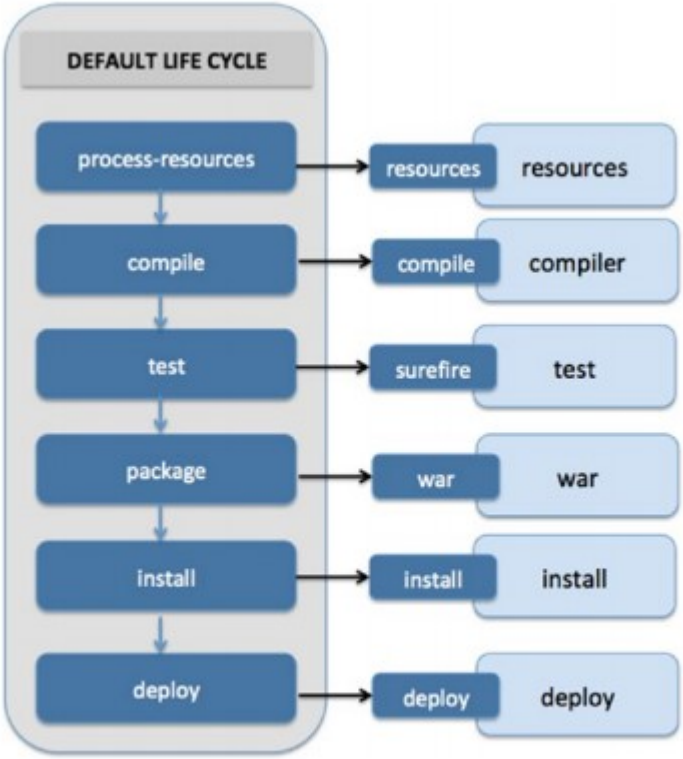


Figure 5-2. *Default lifecycle for WAR project*

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 compiling and 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
```

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.

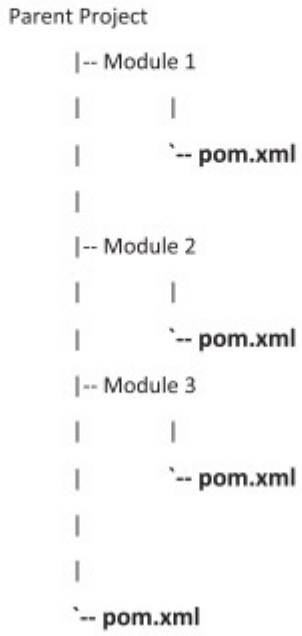


Figure 6-3. *Multimodule project structure*

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>
```

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 all of the 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] ------------------------------------------------------------
```

Using the Site Lifecycle

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

```
mvn site
```

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

```
  <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>
<build>
<plugins>
 <plugin>
```

```
    <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.8.2</version>
      </plugin>
  </plugins>
</build>

</project>
```

```
mvn clean site
```

Upon successful completion of the command, 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
 -----
 Apress
 -----
 08-10-2019
 -----

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

The first three sections contain the document's title, author, and date. The following block of text contains the project description. Running `mvn clean site` results in a new About page, as shown in Figure 7-6.

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>
                <artifactId>maven-javadoc-plugin</artifactId>
      </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

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`.

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. *JaCoCo* (open source) and Atlassian's *Clover* are two popular code coverage tools for Java.

In this section, you will use JaCoCo for measuring this project's code coverage. Listing 7-6 shows JaCoCo plugin configuration. The prepare-agent goal sets a property pointing to JaCoCo runtime environment that gets passed as a VM argument when unit tests are run. The report goal generates the code coverage reports after the unit test execution is complete.

```
<project>
        <build>
           <plugins>
               <!--Content removed for brevity-->
               <plugin>
                        <groupId>org.jacoco</groupId>
                        <artifactId>jacoco-maven-plugin
                        </artifactId>
                        <version>0.8.4</version>
                        <executions>
                           <execution>
                          <id>jacoco-init</id>
                                <goals>
                                    <goal>prepare-agent</goal>
                                </goals>
                           </execution>
                           <execution>
                                <id>jacoco-report</id>
                                <phase>test</phase>
                                 <goals>
                                <goal>report</goal>
                                 </goals>
                           </execution>
                        </executions>
                        </plugin>
                   </plugins>
             <build>
</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, JaCoCo will create a `jacoco` folder under `gswm\target\site`. Launch the code coverage report by double-clicking the `index.html` file under jacoco folder. The report should be similar to the one shown in Figure 7-10.

Figure 7-10. *Generated JaCoCo report*

Generating the SpotBugs Report

SpotBugs 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 SpotBugs configuration.

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

```
<project>
        <!—Content removed for brevity-->
  <reporting>
    <plugins>
      <plugin>
        <groupId>com.github.spotbugs</groupId>
  <artifactId>spotbugs-maven-plugin</artifactId>
  <version>3.1.12</version>
      </plugin>

   </plugins>
 </reporting>
</project>
```

Once the Maven site gets generated, open `index.html` file under `site` folder and navigate to Project Reports ➤ SpotBugs to view the SpotBugs report. It should be similar to the one shown in Figure 7-11.

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

Maven Release

Maven provides the release plugin that automates steps involved with releasing software. Before we deep dive into the Maven release process, we will set up and configure Nexus repository and use Maven to publish artifacts to Nexus.

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.

Sonatype *Nexus* repository manager is a popular open source software 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 `https://help.sonatype.com/repomanager3/download`. At the time of this writing,

version 3.18.1-01 of Nexus is available. Unzip the file, and place the contents on your machine. In this book, we assume the contents to be under `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-3.18.1-01`. Then run the command `nexus /install Nexus_Repo_Manager`. You will see the success message as illustrated in Figure 8-1.

```
C:\tools\nexus\nexus-3.18.1-01\bin>nexus /install Nexus_Repo_Manager
Installed service 'Nexus_Repo_Manager'.
```

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

Note Nexus 3.18 requires JRE 8 to function properly. Make sure you have version 8 of JDK/JRE installed on your local machine. Also, make sure that `JAVA_HOME` is pointing to version 8 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-3.18.1-01\bin>nexus /start Nexus_Repo_Manager
Starting service 'Nexus_Repo_Manager'.
```

By default, Nexus runs on port 8081. Launch a web browser and navigate to Nexus at `http://localhost:8081/`. It will take several minutes, but eventually you should see the Nexus launch screen as shown in Figure 8-3.

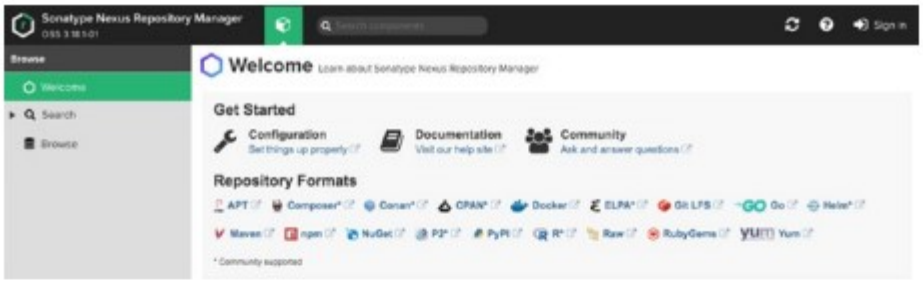


Figure 8-3. *Nexus launch screen*

Click the "Sign In" link on the top-right corner to log in to Nexus. You will be presented with a login modal containing the location to the file with autogenerated admin password as shown in Figure 8-4.

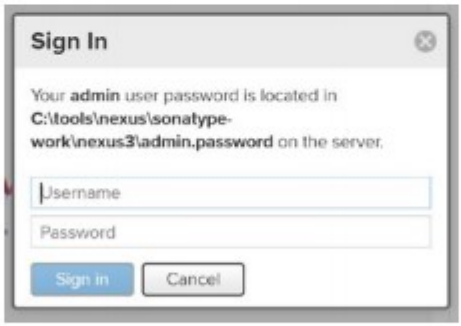


Figure 8-4. *Nexus login modal*

Log in to Nexus with the username admin and password copied from admin.password file. You will be asked to change the password as shown in Figure 8-5. For the exercises in this book, I changed the password to admin123.

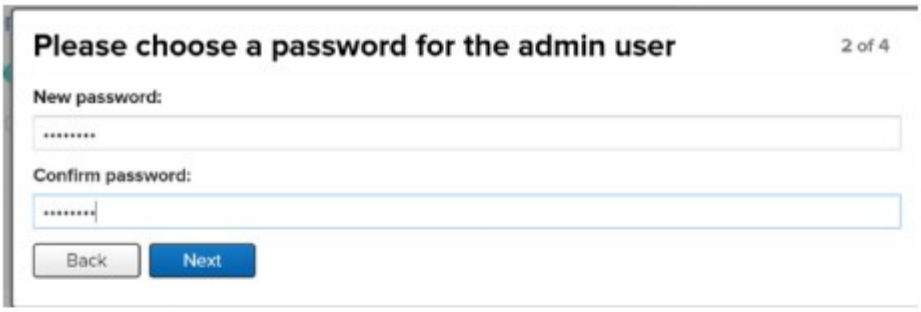


Figure 8-5. *Nexus change password screen*

Now that Nexus is installed and running, 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 provide repository information on where the project's artifacts will be deployed. The `repository` subelement indicates the location where the 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">
        <dependencies>
            <!-- Content removed for brevity -->
        </dependencies>
        <distributionManagement>

         <repository>
            <id>nexusReleases</id>
            <name>Releases</name>
```

```
            <url>http://localhost:8081/repository/maven-releases
            </url>
               </repository>
               <snapshotRepository>
                  <id>nexusSnapshots</id>
                  <name>Snapshots</name>
                  <url>http://localhost:8081/repository/maven-
                  snapshots</url>
               </snapshotRepository>
            </distributionManagement>
            <build>
                <!-- Content removed for brevity -->
            </build>
</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. For Maven to interact and deploy artifacts on Nexus, you need to provide user with the right access roles in the settings.xml file. Listing 8-2 shows the `settings.xml` file with the server information. As you can see, we are using admin user information to connect to Nexus. 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>admin</username>
      <password>admin123</password>
   </server>
   <server>
      <id>nexusSnapshots</id>
      <username>admin</username>
      <password>admin123</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/#browse/browse:maven-snapshots`, as shown in Figure 8-6.

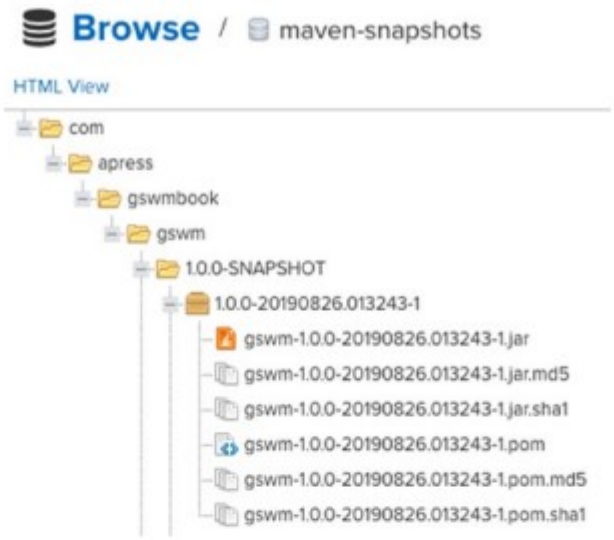


Figure 8-6. *SNAPSHOT artifact under Nexus*

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 `pom.xml` file, and prepare for the next development cycle.

Maven has a release plug-in that provides a standard mechanism for executing the preceding 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 Git as the source controls system and GitHub as the remote server that houses repositories. A typical interaction between Maven and GitHub is shown in Figure 8-7. Maven releases are typically performed on a developer or build machine. Maven requires Git client to be installed on such machines. These command-line tools allow Maven to interact with GitHub and perform operations such as checking out code, creating tags, and so forth.

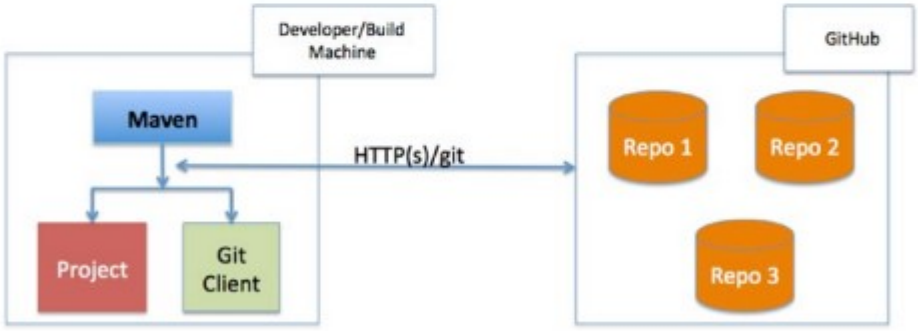


Figure 8-7. *Interaction between Maven and GitHub*

Before we delve deeper into the Maven release process, you need to set up the environment by completing the following steps:

1. Install Git client on your local machine.
2. Create a new remote repository on GitHub.
3. Check the project you will be using into the remote repository.

Git Client Installation

There are several Git clients that make it easy to interact with Git repositories. Popular ones include SourceTree (www.sourcetreeapp.com/) and GitHub Desktop (https://desktop.github.com/). In this book, we will be using the client that comes with Git SCM distribution. Navigate to https://git-scm.com/downloads and download the Windows version of Git distribution. Double-click the downloaded exe file and accept the default installation options. After the installation is complete, open a new command-line window and type git --version. You should see a message similar to Figure 8-8.

```
C:\>git --version
git version 2.23.0.windows.1
```

Figure 8-8. *Git version*

Creating a GitHub Repository

GitHub is a collaborative development platform that allows you to host public and private Git repositories for free. Before you can create a new repository on GitHub, you need to create an account at https://github.com/join. Once you have logged into GitHub using your credentials, navigate to https://github.com/new and create a new repository as shown in Figure 8-9.

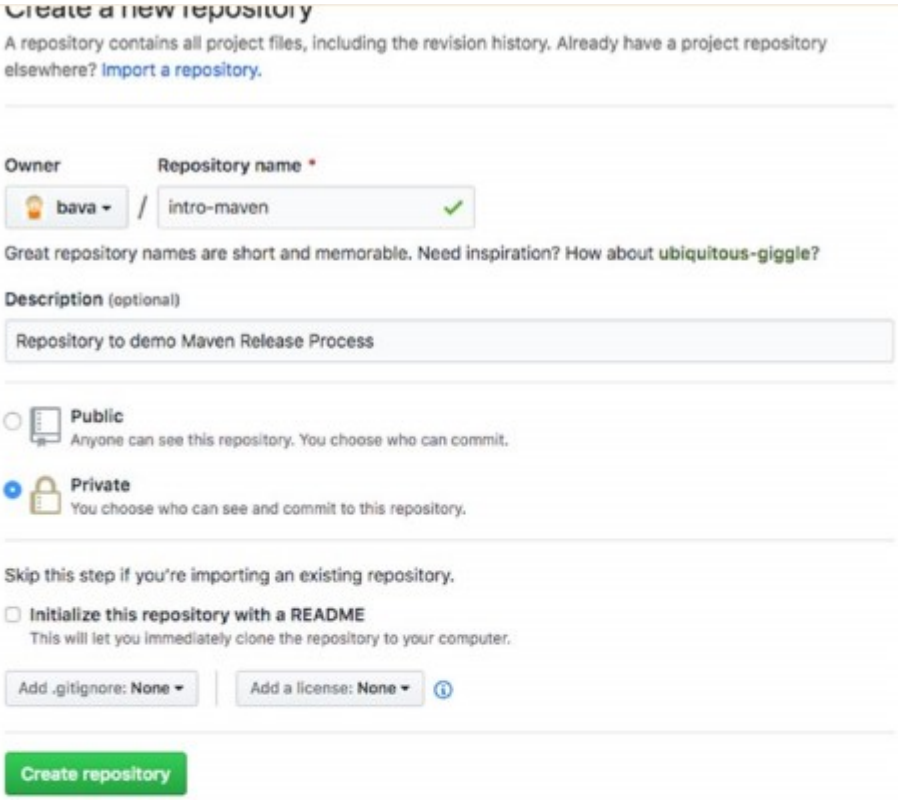


Figure 8-9. *New GitHub repository*

Checking in Source Code

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

repository URL by replacing your GitHub account in the following remote add command:

```
git init

git add .

git commit -m "Initial commit"

git remote add origin https://github.
com/<<your_git_hub_account>>/intro-maven.git

git push -u origin master
```

The Git push command will prompt you for your GitHub username and password. Successful completion of the push command should give the output shown in Figure 8-10.

```
C:\apress\gswm-book\chapter8\gswm>git push -u origin master
Logon failed, use ctrl+c to cancel basic credential prompt.
Username for 'https://github.com': bava
Password for 'https://bava@github.com':
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (10/10), 1.56 KiB | 799.00 KiB/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To https://github.com/bava/intro-maven.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Figure 8-10. *Output from the* `Git` *initial commit*

Using your browser, navigate to your remote repository on GitHub and you will see the checked-in code. Figure 8-11 shows the expected browser screen.

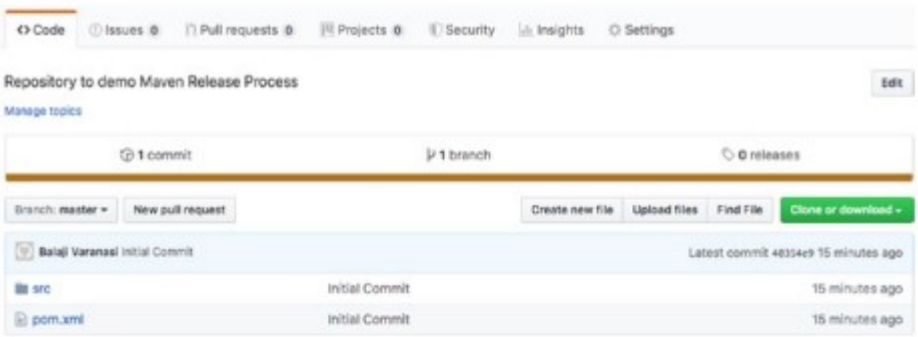


Figure 8-11. *Project checked into GitHub*

The preceding commands have pushed the code into the mater branch on GitHub. However, Maven release plug-in interacts with the code in the release branch. So, the final step in this setup is to create a new local release branch and push it to GitHub by running the following commands:

```
git checkout -b release
git push origin release
```

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 suggests, 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 mode, 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

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

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

    <scm>
      <connection>scm:git:https://github.com/bava/intro-maven.
      git</connection>
      <developerConnection>scm:git:https://github.com/bava/
      intro-maven.git</developerConnection>
      <url>https://github.com/bava/intro-maven</url>
    </scm>
   <!-- Content removed for brevity -->
</project>
```

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

```
git commit . -m "Added SCM Information"
git push origin release
```

In order for Maven to communicate successfully with the GitHub, it needs GitHub credentials. You provide that information in the settings.xml file, as shown in Listing 8-4. The ID for the server element is declared as GitHub, as it must match the hostname.

Listing 8-4. The settings.xml with GitHub 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>admin</username>
      <password>admin123</password>
   </server>
   <server>
      <id>nexusSnapshots</id>
      <username>admin</username>
      <password>admin123</password>
   </server>
   <server>
     <id>github</id>
     <username>[your_github_account_name]</username>
     <password>[your_github_account_password]</password>
   </server>
 </servers>
</settings>
```

You now have all of the 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

Listing 8-5. Maven prepare Command

```
C:\apress\gswm-book\chapter8\gswm>mvn release:prepare
[INFO] Scanning for projects...
[INFO]
[INFO] --------------< com.apress.gswmbook:gswm >--------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
```

```
[INFO] --- maven-release-plugin:2.5.3:prepare (default-cli)
@ gswm ---

[INFO] Verifying that there are no local modifications...

[INFO] Executing: cmd.exe /X /C "git rev-parse --show-toplevel"
[INFO] Working directory: C:\apress\gswm-book\chapter8\gswm
[INFO] Executing: cmd.exe /X /C "git status --porcelain ."

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] Checking in modified POMs...

[INFO] Tagging release with the label gswm-1.0.0...
[INFO] Executing: cmd.exe /X /C "git tag -F C:\Users\bavara\
AppData\Local\Temp\maven-scm-73613791.commit gswm-1.0.0"

[INFO] Executing: cmd.exe /X /C "git push https://github.com/
bava/intro-maven.git refs/tags/gswm-1.0.0"
[INFO] Release preparation complete.
[INFO] BUILD SUCCESS
```

Notice the `Git` commands getting executed as part of the `prepare` goal. Successful completion of the `prepare` goal will result in the creation of a Git tag, as shown in Figure 8-12. The `pom.xml` file in the `gswm` project will now have version 1.0.1-SNAPSHOT.

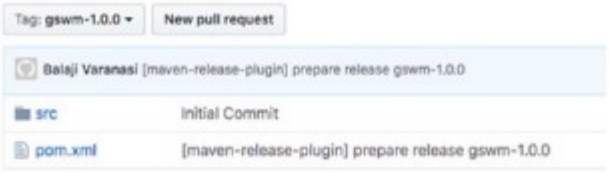


Figure 8-12. *Git tag created upon* `prepare` *execution*

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 (e.g., is unable to connect to Git) 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 release plug-in's `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] -------------< com.apress.gswmbook:gswm >---------------
[INFO] Building Getting Started with Maven 1.0.1-SNAPSHOT
[INFO] --------------------[ jar ]-----------------------------

[INFO] --- maven-release-plugin:2.5.3:perform (default-cli)
@ gswm ---
[INFO] Checking out the project to perform the release ...
[INFO] Executing: cmd.exe /X /C "git clone --branch gswm-1.0.0
https://github.com/bava/intro-maven.git C:\apress\gswm-book\
chapter8\gswm\target\checkout"

[INFO] Invoking perform goals in directory C:\apress\gswm-book\
chapter8\gswm\target\checkout
[INFO] Executing goals 'deploy'...

 [INFO] Building jar: C:\apress\gswm-book\chapter8\gswm\target\
checkout\target\gswm-1.0.0-javadoc.jar


 [INFO] --- maven-install-plugin:2.4:install (default-install)
@ gswm ---
```

```
[INFO] Installing C:\apress\gswm-book\chapter8\gswm\target\
checkout\target\gswm-1.0.0.jar to C:\Users\bavara\.m2\
repository\com\apress\gswmbook\gswm\1.0.0\gswm-1.0.0.jar

[INFO] --- maven-deploy-plugin:2.7:deploy (default-deploy)
@ gswm ---
[INFO] Uploading to nexusReleases: http://localhost:8081/
repository/maven-releases/com/apress/gswmbook/gswm/1.0.0/
gswm-1.0.0.jar

[INFO] Uploaded to nexusReleases: http://localhost:8081/
repository/maven-releases/com/apress/gswmbook/gswm/1.0.0/
gswm-1.0.0.jar (2.4 kB at 14 kB/s)
[INFO] Uploading to nexusReleases: http://localhost:8081/
repository/maven-releases/com/apress/gswmbook/gswm/1.0.0/
gswm-1.0.0.pom

[INFO] Uploaded to nexusReleases: http://localhost:8081/
repository/maven-releases/com/apress/gswmbook/gswm/1.0.0/
gswm-1.0.0-javadoc.jar (22 kB at 84 kB/s)

[INFO] BUILD SUCCESS
```

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 8-13.

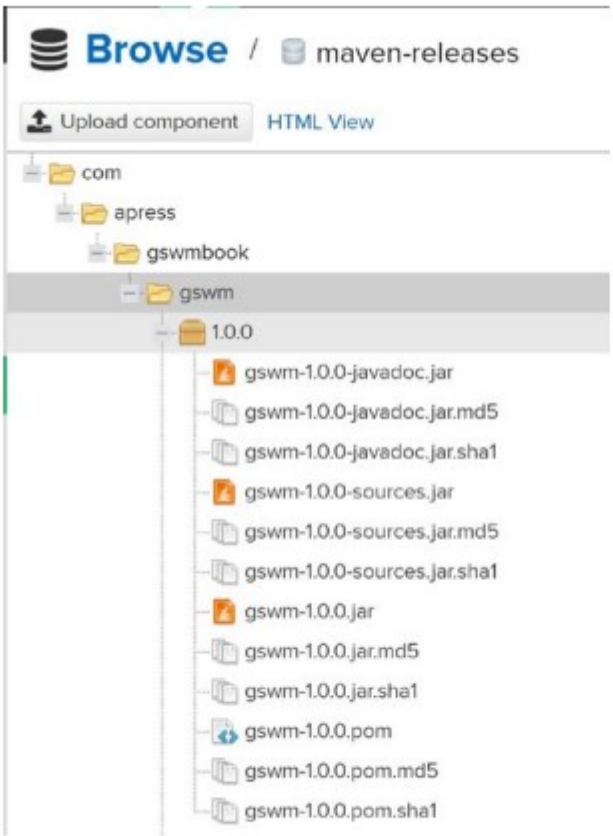


Figure 8-13. *Nexus with released artifact*

Installing Jenkins

Jenkins is distributed in several flavors – native installers, Docker containers, and as an executable WAR file. In this book, we will be using the long-term support (LTS) executable WAR file version that you can download at https://jenkins.io/download/. Save the downloaded version at c:\tools\jenkins.

Once the download is complete, using command line, navigate to the downloaded folder and run the command: ***java –jar jenkins.war.*** Upon successful execution of the command, open a browser and navigate to http://localhost:8080. You will be prompted to locate and enter the autogenerated administrator password from the "initialAdminPassword" file. On the next screen, select "Install Suggested Plugins" and wait for the setup to complete plug-in installation. On the "Create First Admin User" screen, enter "admin" as username and "admin123" as password and fill in the rest of the details on the form. Upon completion of Jenkin's configuration, you should see Jenkins dashboard similar to Figure 9-2.

Figure 9-2. *Jenkins dasboard*

Maven Project

For us to understand Jenkins support for Maven, we need a sample Maven project on a source control server. In this chapter, we will use a gwsm-jenkins project hosted on GitHub at https://github.com/bava/gswm-jenkins. For you to follow along the rest of the chapter, you need to fork the gswm-jenkins repository under your own account. You can do that by logging into GitHub and clicking the fork button as shown in Figure 9-3.

Figure 9-3. *Fork gswm-jenkins repository*

Configuring Jenkins

To begin Jenkins configuration, click the "New Item" link on the dashboard. On the New Item screen, select Freestyle project and enter the name "gswm-jenkins-integration" as shown in Figure 9-4.

Figure 9-4. *New Item screen*

On the next screen, in the General section, select the "GitHub project" checkbox and enter the project URL. This should be the URL to your forked project location on your GitHub account.

Figure 9-5. *New Item - General section*

On the "Source Code Management" section, select the "Git" radio button and enter the URL to your GitHub repository as shown in Figure 9-6. This is the GitHub clone URL that you can find by clicking "Clone or download" under repository name.

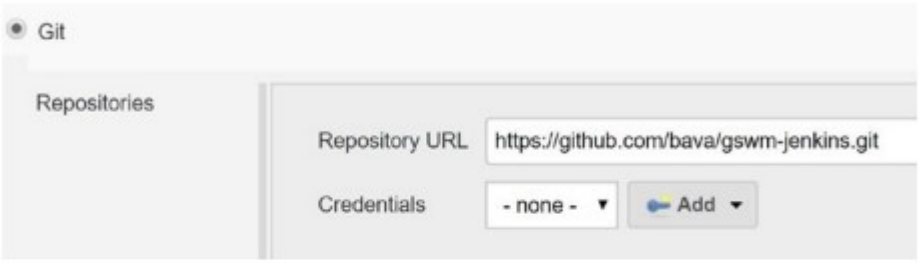
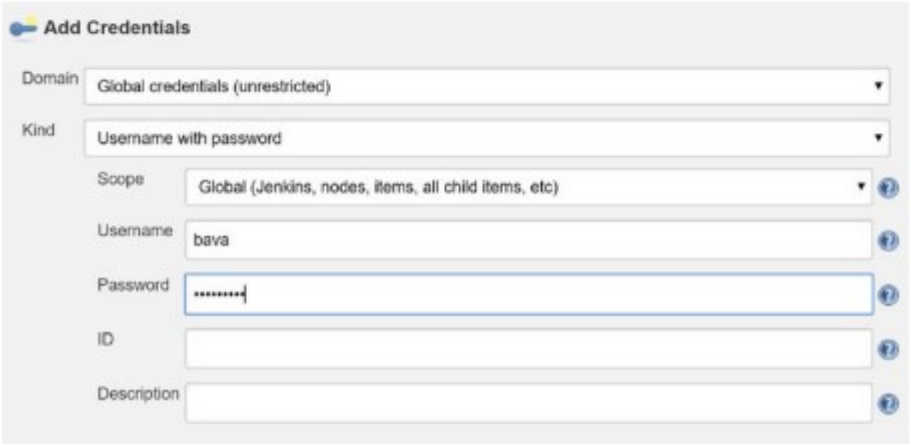


Figure 9-6. *New Item Source Code Management section*

For Jenkins to checkout your code, you need to provide your GitHub credentials. You do that by clicking the "Add" button next to Credentials and enter your username and password as shown in Figure 9-7.



In the Build Triggers section, select "Poll SCM" option and enter "H/15 * * * *" as value as shown in Figure 9-8. This indicates that Jenkins need to poll GitHub repo for changes every 15 minutes.

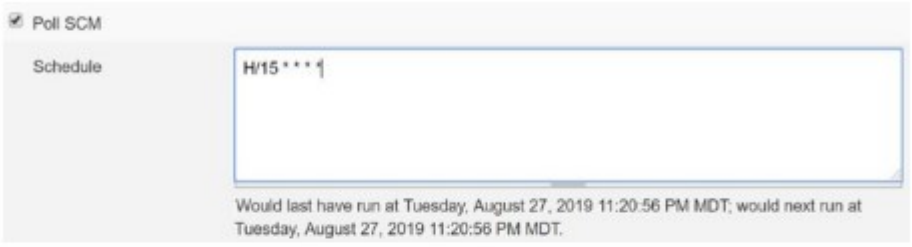


Figure 9-8. *Build Trigger poll schedule*

Under the "Build" section, click "Add build step" and select "Invoke top-level Maven targets". Enter "clean install" as Goals value as shown in Figure 9-9.

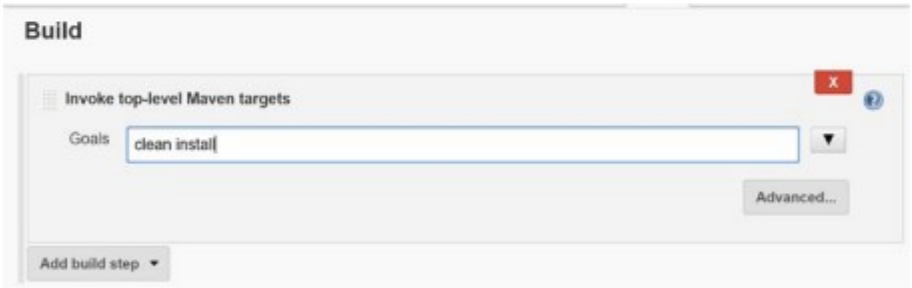


Figure 9-9. *Build step for Maven*

Finally, in the Post-build Actions section, click "Add post-build action" and select "Archive the artifacts". Enter **/*.jar as the value for Files to Archive as shown in Figure 9-10.

Figure 9-10. *Archive artifacts section*

Click the "Add post-build action" button one more time and select "Publish JUnit test result report". Enter "target/surefire-reports/∗.xml" as Test report XMLs value as shown in Figure 9-11. Click Save to save the configuration.

Figure 9-11. *Publish JUnit results*

Triggering Build Job

We now have everything set up to get Jenkins build our project. On the project job page, click "Build Now" link to trigger a new build. This would start a new build with a numerical number that you can access from the Build History section on the bottom-left corner of the page. Click the drop-down arrow next to the Build number and select "Console Output". This will take you to the output screen similar to Figure 9-12.

```
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ gswm-jenkins ---
[INFO] Building jar: C:\Users\bavara\.jenkins\workspace\gswm-jenkins-
integration\target\gswm-jenkins-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ gswm-jenkins ---
[INFO] Installing C:\Users\bavara\.jenkins\workspace\gswm-jenkins-integration\target\gswm-
jenkins-1.0.0-SNAPSHOT.jar to C:\Users\bavara\.m2\repository\com\apress\gswmbook\gswm-
jenkins\1.0.0-SNAPSHOT\gswm-jenkins-1.0.0-SNAPSHOT.jar
[INFO] Installing C:\Users\bavara\.jenkins\workspace\gswm-jenkins-integration\pom.xml to
C:\Users\bavara\.m2\repository\com\apress\gswmbook\gswm-jenkins\1.0.0-SNAPSHOT\gswm-
jenkins-1.0.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  11.685 s
[INFO] Finished at: 2019-08-27T23:25:36-06:00
[INFO] ------------------------------------------------------------------------
Finished: SUCCESS
```

Figure 9-12. *Job console output screen*

Upon successful completion of the job, you will see the built artifact on the project page as shown in Figure 9-13.

Figure 9-13. *Jenkins Project page - Build Artifact*

The test results from the run are also available on the project page under "Latest Test Result".