



Introducing Maven

A Build Tool for Today's Java
Developers

—

Second Edition

—

Balaji Varanasi

Apress®

Introducing Maven

**A Build Tool for Today's
Java Developers**

Second Edition

Balaji Varanasi

Apress®

Introducing Maven: A Build Tool for Today's Java Developers

Balaji Varanasi
Salt Lake City, UT, USA

ISBN-13 (pbk): 978-1-4842-5409-7

ISBN-13 (electronic): 978-1-4842-5410-3

<https://doi.org/10.1007/978-1-4842-5410-3>

Copyright © 2019 by Balaji Varanasi

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

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 editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles 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 Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers via the book's product page, located at www.apress.com/978-1-4842-5409-7. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

For my Vedha

Table of Contents

- About the Authorix**
- About the Technical Reviewerxi**
- Acknowledgmentsxiii**
- Introductionxv**

- Chapter 1: Getting Started with Maven 1**
 - Standardized Directory Structure 2
 - Declarative Dependency Management 2
 - Plug-ins..... 3
 - Uniform Build Abstraction 3
 - Tools Support 3
 - Archetypes 4
 - Open Source..... 4
 - Maven Alternatives 5
 - Ant + Ivy 5
 - Gradle 7
 - Maven Components 9
 - Maven SCM 9
 - Maven Wagon 9
 - Maven Doxia 10
 - Summary..... 10

TABLE OF CONTENTS

- Chapter 2: Setting Up Maven11**
 - Installing on Windows 13
 - Installing on Mac..... 13
 - Testing Installation 14
 - Getting Help 15
 - Additional Settings..... 16
 - Setting Up a Proxy..... 19
 - Securing Passwords 20
 - IDE Support 21
 - Summary..... 21
- Chapter 3: Maven Dependency Management23**
 - Using New Repositories 26
 - Dependency Identification 28
 - Transitive Dependencies 29
 - Dependency Scope 32
 - Manual Dependency Installation..... 33
 - Summary..... 35
- Chapter 4: Maven Project Basics.....37**
 - Basic Project Organization 37
 - Understanding the pom.xml File 40
 - Building a Project..... 42
 - Testing the Project 44
 - Properties in pom.xml 49
 - Implicit Properties 50
 - User-Defined Properties 50
 - Summary..... 51

Chapter 5: Maven Lifecycle	53
Goals and Plug-ins	53
Lifecycle and Phases	57
Plug-in Development.....	61
Summary.....	68
Chapter 6: Maven Archetypes.....	69
Built-in Archetypes.....	69
Generating a Web Project.....	71
Multimodule Project.....	74
Creating an Archetype.....	80
Using the Archetype.....	86
Summary.....	87
Chapter 7: Documentation and Reporting	89
Using the Site Lifecycle.....	89
Advanced Site Configuration.....	94
Generating Javadoc Reports	99
Generating Unit Test Reports.....	101
Generating Code Coverage Reports	102
Generating the SpotBugs Report.....	104
Summary.....	105
Chapter 8: Maven Release	107
Integration with Nexus.....	107
Project Release	113
Git Client Installation.....	115
Creating a GitHub Repository	115
Checking in Source Code	116

TABLE OF CONTENTS

Maven Release..... 118

Prepare Goal 118

Clean Goal 123

Perform Goal..... 123

Summary..... 126

Chapter 9: Continuous Integration.....127

 Installing Jenkins 128

 Maven Project 129

 Configuring Jenkins 130

 Triggering Build Job 133

 Summary..... 135

Index.....137

About the Author



Balaji Varanasi is a seasoned technology leader, author, and speaker. He has over 18 years of experience managing, architecting, and delivering high-performance, scalable enterprise 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. He shares his insights and experiments at <http://blog.inflinx.com>.

About the Technical Reviewer

Germán González-Morris is a polyglot software architect/engineer with 20+ years in the field, with knowledge in Java(EE), Spring, Haskell, C, Python, and Javascript, among others. He works with web distributed applications. Germán loves math puzzles (including reading Knuth) and swimming. He has tech-reviewed several books, including an application container book (WebLogic), as well as titles covering various programming languages (Haskell, TypeScript, WebAssembly, Math for coders, and regexp). You can find more details at his blog site (<https://devwebcl.blogspot.com/>) or twitter account (@devwebcl).

Acknowledgments

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

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

Finally, I want to thank my wife, Sudha, for her constant support and encouragement. Thank you so much, dear!

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, lifecycle 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 Maven's release process and integration with Jenkins.

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.

INTRODUCTION

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 lifecycle, 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 SpotBugs 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.

Chapter 9 introduces continuous integration (CI) concepts and discusses installation and configuration on Jenkins, an open source tool for continuous integration.

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 via the Download Source Code button located at www.apress.com/978-1-4842-5409-7.

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 author at Balaji@inflinx.com.

CHAPTER 1

Getting Started with Maven

Like other craftsmen, software developers rely on their tools to build applications. Developers' integrated development environments (IDEs), bug-tracking tools, build tools, frameworks, containers, 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 and version 3.0 in 2010. At the time of writing this book, 3.6.1 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 preceding 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 IntelliJ IDEA might use `web` folder 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 and Bamboo.

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 are 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 configuration files. It is also highly possible that all of these web projects have similar Log4j/Logback configuration files, CSS/Images, and Thymeleaf page layouts. 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 of code or drag and drops of files 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.

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-core-2.11.2.jar`. At a glance, you can easily see that you are looking at a `log4j-core` JAR file, version 2.11.2.

One drawback of Maven's CoC is the rigidity 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 hard-to-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="2.11.2" />
    </dependencies>
</ivy-module>

```

Gradle

Gradle (<http://gradle.org/>) is an open source build, project automation tool that can be used for Java and non-Java projects. 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. Listing 1-3 shows a default `build.gradle` file.

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

```
plugins {
    id 'java'
}

version = '1.0.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompileOnly 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 and performance have contributed to its growing popularity. However, the learning curve for

Gradle and Groovy DSL could be steep. IDE and plug-in support for Gradle is also less mature when compared to Maven.

Despite growing competition from other tools, Maven continues to dominate the build tool space. This is evident in Figure 1-1 that shows the results of a 2018 survey by Synk.io (<https://synk.io/blog/jvm-ecosystem-report-2018-tools/>) on the build tools used by Java developers.

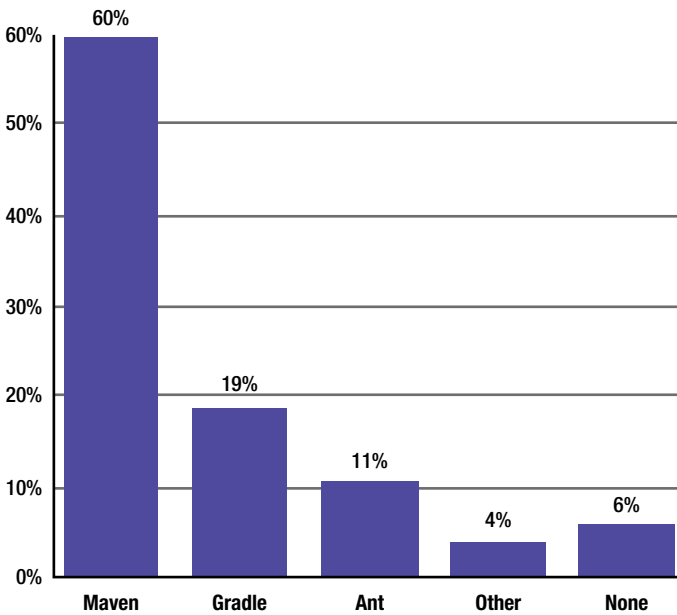


Figure 1-1. Survey results of build tool usage

The *Polyglot for Maven* Project (<https://github.com/takari/polyglot-maven>) allows you to create Maven's POM files in dialects other than XML. Supported languages include Groovy, Ruby, Scala, YAML, and Java.

Maven Components

Maven relies on several components to get its job done. Though you might not interact with these components directly, an overview of these components helps understand the internal workings of Maven and better equip you to troubleshoot Maven errors.

Maven SCM

Maven interacts with several source control/code management (SCM) systems to perform operations such as checking out code or creating a branch or a tag. The Maven SCM (<http://maven.apache.org/scm/>) project provides a common API to perform such operations. The Maven release plug-in discussed in Chapter 8 heavily relies on Maven SCM components. Maven SCM currently supports several popular code management systems such as Git, Subversion, and Perforce.

Maven Wagon

As discussed earlier, Maven automatically downloads project dependencies such as JAR files from different locations such as FTPs, file systems, and web sites. The Maven Wagon (<https://maven.apache.org/wagon/>) project provides a transport abstraction that enables Maven to interact with different transport protocols easily and retrieve dependencies. Maven Wagon supports some of the popular protocols such as

- a) *File*: Allows retrieval of dependencies using file system protocol.
- b) *HTTP*: Allows retrieval of dependencies using HTTP/HTTPS protocols. Two implementation variations are provided – one that uses Apache HttpClient and the other that uses standard Java library.
- c) *FTP*: Allows retrieval of dependencies using File Transfer Protocol.

Maven Doxia

Maven Doxia (<https://maven.apache.org/doxia/>) is a content generation framework that can be used to generate static/dynamic content such as PDF files and web pages. Doxia supports several popular markup languages such as Markdown, Apt, XHTML, and Confluence. Maven relies heavily on Doxia to generate project documentation and reports (more on this in Chapter 7).

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 setup 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 10 and Mac operating systems. You can follow similar procedure with other operating systems.

Note Maven is a Java-based application and requires the Java Development Kit (JDK) to function properly. Maven version 3.6 requires JDK 1.7 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 www.oracle.com/technetwork/java/javase/downloads/index.html. Ensure that you have the `JAVA_HOME` environment variable set and pointing to the JDK installation. In this book, we will be using JDK 8.

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 3.6.1. Download the Maven 3.6.1 binary .zip file as shown in Figure 2-1.

	Link	Checksums	Signature
Binary tar.gz archive	apache-maven-3.6.1-bin.tar.gz	apache-maven-3.6.1-bin.tar.gz.sha512	apache-maven-3.6.1-bin.tar.gz.asc
Binary zip archive	apache-maven-3.6.1-bin.zip	apache-maven-3.6.1-bin.zip.sha512	apache-maven-3.6.1-bin.zip.asc
Source tar.gz archive	apache-maven-3.6.1-src.tar.gz	apache-maven-3.6.1-src.tar.gz.sha512	apache-maven-3.6.1-src.tar.gz.asc
Source zip archive	apache-maven-3.6.1-src.zip	apache-maven-3.6.1-src.zip.sha512	apache-maven-3.6.1-src.zip.asc

Figure 2-1. *Maven archive download*

Once the download is complete, unzip the distribution to a local directory on your computer. It will create a folder named `apache-maven-3.6.1-bin` and the contents of the folder are shown in Figure 2-2.

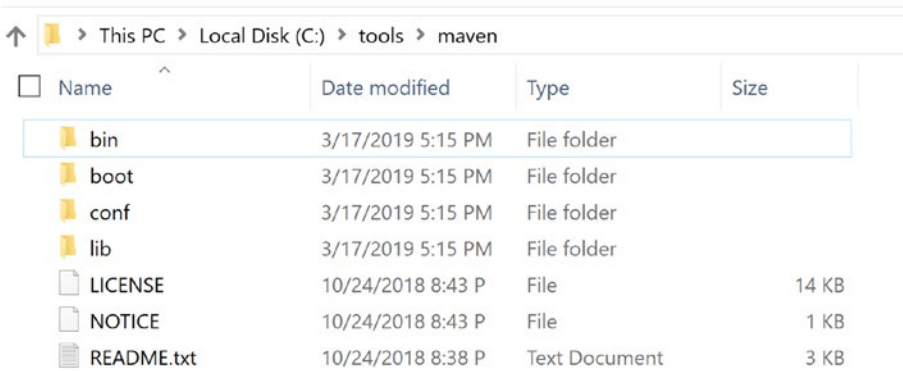


Figure 2-2. *Maven install directory contents*

The `bin` folder contains platform-specific Maven executables – `mvn.cmd` file for Windows and `mvn.sh` for Mac and Unix/Linux platforms that you can use to launch Maven. The debug versions of these executables – `mvnDebug.cmd` and `mvnDebug.sh` – include debugging arguments that allow you to attach an IDE to a running Maven process for remote debugging.

That `boot` folder contains the “`plexus-classworlds-2.5.2.jar`” file. Maven uses Plexus Classworlds framework (<https://codehaus-plexus.github.io/plexus-classworlds/>) to build its classloader graph.

The `conf` folder contains configuration files that you can use to alter Maven's default behaviors. An important file in this folder is the `settings.xml` file which we will cover later in the chapter. Another file is the `simplelogger.properties` file that allows you to control Maven's logging. For example, the log level can be changed to debug by setting the `defaultLogLevel` property to debug. Similarly, you can change the `logFile` property to write log output from "System.out" to a file.

Finally, the `lib` folder contains the libraries that are essential for Maven and its plug-ins to run properly.

Installing on Windows

Move the contents of the `apache-maven-3.6.0-bin` to a new directory `c:\tools\maven`. The next step is to add the Maven executable to the `PATH` Environment variable so that you can run Maven commands from the command line. In Windows Search box, search for "Environment Variable" and select "Edit the system environment variables". In the resulting window, click the Environment Variables button, and select the `PATH` variable and click Edit. Click New and enter the value "`C:/tools/maven/bin`" and click OK.

Installing on Mac

Move the contents of the `apache-maven-3.6.0-bin` folder into `$HOME/tools/maven` where `$HOME` is your home directory on Mac. Edit `.bash_profile` file by running the command `nano ~/.bash_profile`. Add Maven executable to the `PATH` variable by adding the following line to `.bash_profile`:

```
export PATH=$HOME/tools/maven/bin:$PATH
```

This completes the Maven installation. If you have any open command-line windows/terminals, 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.

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
```

On a Windows machine, this command should output information similar to the following:

```
C:\> mvn -v
Apache Maven 3.6.1 (d66c9c0b3152b2e69ee9bac180bb8fcc8e6af555;
2019-04-04T13:00:29-06:00)
Maven home: C:\tools\maven\bin\..
Java version: 1.8.0_144, vendor: Oracle Corporation, runtime:
C:\Java\jdk1.8.0_144\jre
```

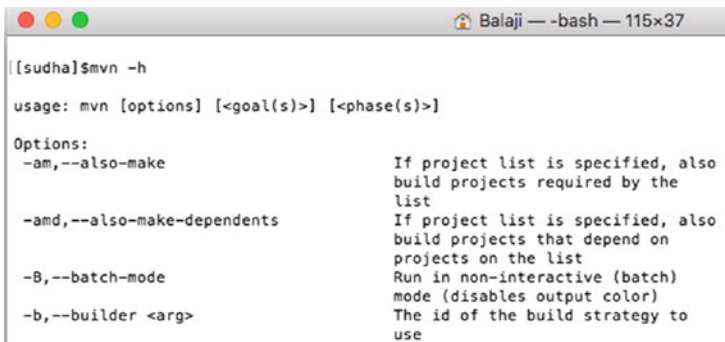
```
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family:
"windows"
```

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 following command will produce output similar to that shown in Figure 2-3.

```
mvn -h
```



```
[[sudha]$mvn -h
usage: mvn [options] [<goal(s)>] [<phase(s)>]

Options:
  -am,--also-make                If project list is specified, also
                                build projects required by the
                                list
  -amd,--also-make-dependents    If project list is specified, also
                                build projects that depend on
                                projects on the list
  -B,--batch-mode                Run in non-interactive (batch)
                                mode (disables output color)
  -b,--builder <arg>            The id of the build strategy to
                                use
```

Figure 2-3. 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. Maven looks for the `settings.xml` file in two locations – in the `conf` folder of Maven’s installation and `.m2` folder in the user’s home directory. The `settings.xml` file under `conf` folder is called `global settings`, and the file under `.m2` folder is referred to as `user settings`. If both files exist, Maven will merge the contents of two files and the `user settings` will take precedence.

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>>`. On Mac, this directory is usually `/Users/<<your_user_name>>/`. You can run the command `mvn help:evaluate -Dexpression=settings.localRepository` to identify the local repository location.

When you run a Maven command, 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 some 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/>
  <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 <code>c:\Users\<<your_user_name>>\.m2\repository</code> folder. The <code>localRepository</code> element can be used to change the path of the local repository. For example, <code><localRepository>c:\mavenrepo</localRepository></code> will change the repository location to the <code>mavenrepo</code> folder.
interactiveMode	As the name suggests, when this value is set to <code>true</code> , Maven interacts with the user for input. If the value is <code>false</code> , Maven will try to use sensible defaults. The default is <code>true</code> .
offline	When set to <code>true</code> , this configuration instructs Maven to not connect to network and operate in an offline mode. With offline mode set to <code>true</code> , Maven will not attempt to download new dependencies or updates to dependencies. The default is <code>false</code> .
servers	Maven can interact with a variety of servers, such as Git 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 downloading dependencies from remote repositories. For example, your organization might have mirrored a public repository on their internal network. The mirror element allows you to force Maven use the internal mirrored repository instead of the public repository.
proxies	Proxies contain the HTTP proxy information needed to connect to the Internet.

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>
```

Securing Passwords

The password to connect to the proxy server in section 2-2 is stored in clear text in the settings.xml file. If you were to accidentally share your settings.xml file, your password will be compromised. To address this, Maven provides a mechanism to encrypt the passwords that get stored in settings.xml file.

We begin the encryption process by creating a master password using the following code:

```
mvn -emp mymasterpassword
{LCWwO+NAqwOHuYH9HNz+1D7aE1XM242PtuyoDXDAue1xjwZC8MyXaACkHSy7
tZwU}
```

Maven requires the newly generated master password to be saved in a settings-security.xml file under .m2 folder. Create a new settings-security.xml file under .m2 folder and copy the following contents into that file.

```
<settingsSecurity>
<master>{LCWwO+NAqwOHuYH9HNz+1D7aE1XM242PtuyoDXDAue1xjwZC8MyXaA
CkHSy7tZwU}</master>
</settingsSecurity>
```

Run the following command to encrypt the “proxypassword” password. Once the command completes, copy the output and replace the clear text password in settings.xml file with it:

```
mvn -ep proxypassword
{i4RnaIHgxqgHyKYySxor+cvshmHweTAvNjuORNYyu5w=}
```

Though the preceding process encrypts the passwords and avoids the need to save passwords in clear text, it is important to remember that anyone that has access to settings-security.xml file can easily decode the

master password and subsequently decrypt the passwords in the settings.xml file. One mechanism to address this is to store the settings-security.xml file in an external device such as USB drive.

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. There are a couple of problems with this approach:

1. The JAR file you downloaded might depend on a few other libraries. You would now have to hunt down all of those dependencies (and their dependencies) and add them to your project.
2. When the time comes to upgrade the JAR file, you need to start the process all over again.
3. You need to add JAR files to source control along with your source code so that your projects can be built on a computer other than your own. This increases project size, checkout, and build time.
4. Sharing JAR files across teams within your organization becomes difficult.

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. 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.

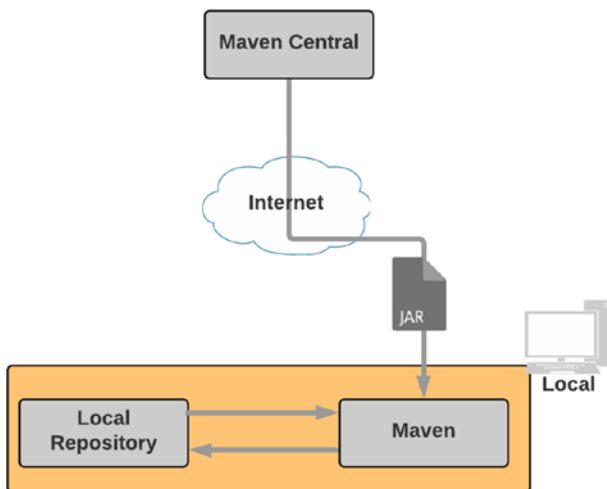


Figure 3-1. *Maven dependency management*

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.

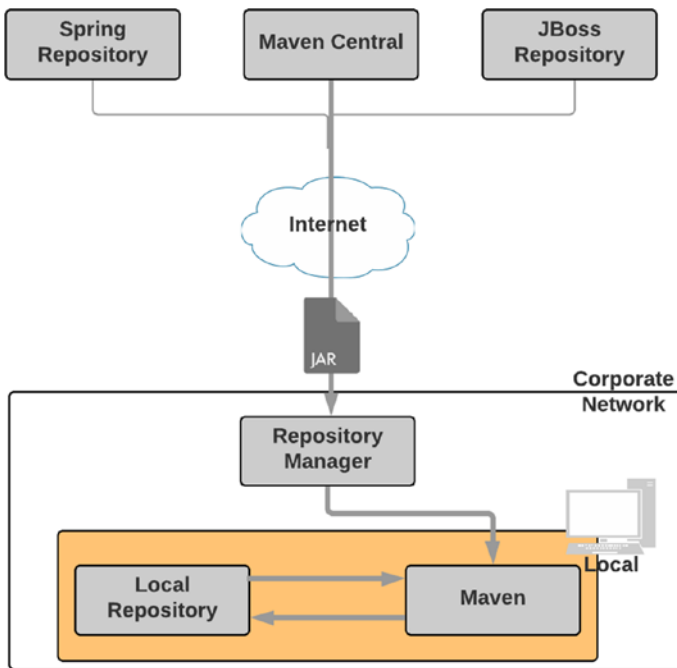


Figure 3-2. Enterprise Maven repository architecture

The internal repository manager acts as a proxy to remote repositories. This allows you to cache artifacts from remote repositories resulting in faster artifact downloads and build performance improvements. 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 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

Using New Repositories

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

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.

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.

Dependencies are declared in `pom.xml` file using the `dependencies` tag as shown in the following:

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-tools</artifactId>
    <version>5.4.2.Final</version>
  </dependency>
</dependencies>
```

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.

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 dependency 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. If you want Maven to use the latest 2.2.0 version of artifact F, you need to explicitly add that version dependency to pom.xml file.

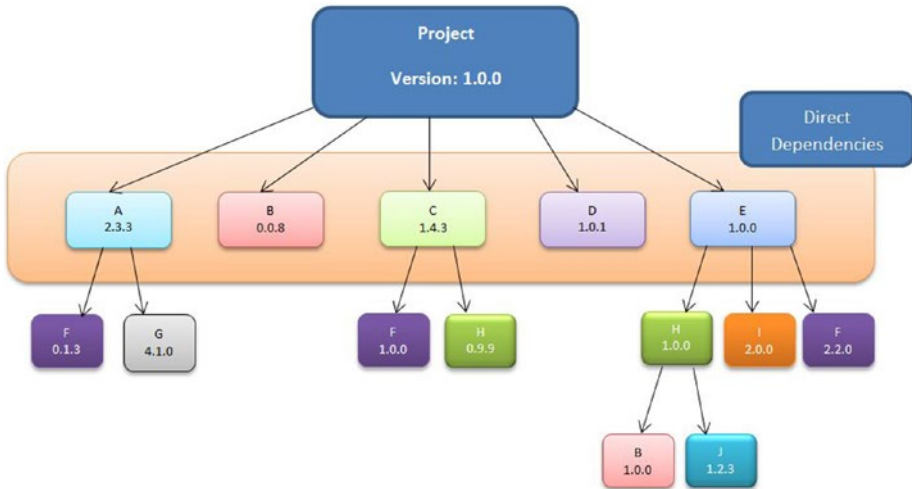


Figure 3-3. *Transitive dependencies*

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. Maven provides a handy dependency plug-in that allows you to visualize project dependency tree. Listing 3-2 shows the output of running the *dependency tree* goal on a sample project. You can see that the project depends on 4.11 version of JUnit JAR file. The JUnit JAR itself depends on 1.3 version of hamcrest JAR file.

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
```

There are times where you don't want to include certain transitive dependency JARs in the final archive. For example, when deploying an application inside a container such as Tomcat or WebLogic, you might want to exclude certain JAR files such as `servlet-api` or `javaee-api` as they 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] -----

```

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

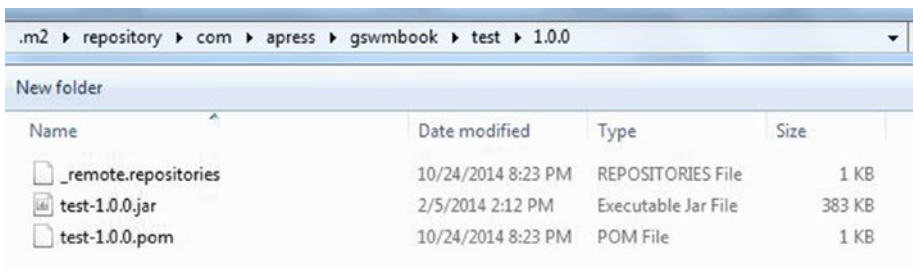


Figure 3-4. Dependency added to repository

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.

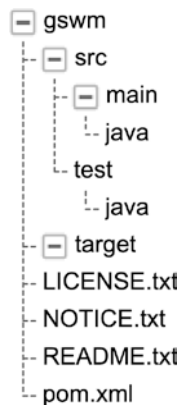


Figure 4-1. *Maven Java project structure*

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 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.

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

Table 4-1. *Maven Directories*

Directory Name	Description
<code>src/main/resources</code>	Holds resources, such as Spring configuration files and velocity templates, that need to end up in the generated artifact.
<code>src/main/config</code>	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.
<code>src/main/scripts</code>	Holds any scripts that system administrators and developers need for the application.
<code>src/test/resources</code>	Holds configuration files needed for testing.
<code>src/main/webapp</code>	Holds web assets such as <code>.jsp</code> files, style sheets, and images.
<code>src/it</code>	Holds integration tests for the application.
<code>src/main/db</code>	Holds database files, such as SQL scripts.
<code>src/site</code>	Holds files required during the generation of the project site.

Maven provides archetypes (as discussed in Chapter 6) to bootstrap projects quickly. However, in this chapter, you will manually assemble a Maven-based Java 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 `src` directory under `gswm`, then create the `main` directory in `src`, and finally create the `java` directory under `main`.

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

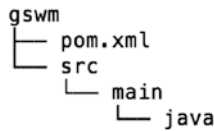


Figure 4-2. Starting project structure

Understanding the `pom.xml` File

The `pom.xml` file is the most important file 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 use the `developers` element to 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.

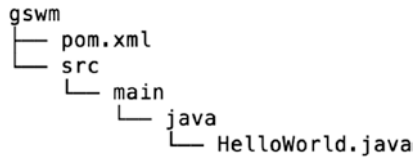


Figure 4-3. Project structure with Java class added

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] Compiling 1 source file to C:\apress\gswm-book\chapter4\
gswm\target\classes
.....
[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

```

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.

Local Disk (C:) > apress > gswm-book > chapter4 > gswm > target		
Name	Type	Date
classes	File folder	9/1/
maven-archiver	File folder	9/1/
maven-status	File folder	9/1/
gswm-1.0.0-SNAPSHOT.jar	Executable Jar File	9/1/

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>4.12</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] --- maven-dependency-plugin:2.8:tree (default-cli)
@ gswm ---
[INFO] com.apress.gswmbook:gswm:jar:1.0.0-SNAPSHOT
[INFO] \- junit:junit:jar:4.12:test
[INFO]    \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] -----
[INFO] BUILD SUCCESS

```

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.

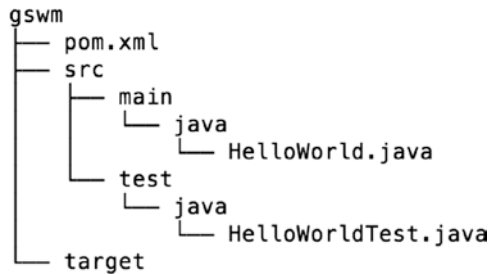


Figure 4-5. Maven structure with test class

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", outputStream.
            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 an 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] --- 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

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







Local Disk (C:) > apress > gswm-book > chapter4 > gswm > target			
Name		Type	Date
 classes		File folder	9/1/:
 maven-archiver		File folder	9/1/:
 maven-status		File folder	9/1/:
 surefire-reports		File folder	9/1/:
 test-classes		File folder	9/1/:
 gswm-1.0.0-SNAPSHOT.jar		Executable Jar File	9/1/:

Figure 4-6. Target folder with test classes

Properties in pom.xml

Maven provides properties AKA placeholders that can be used inside pom.xml file. Maven properties are referenced in pom.xml file using the `${property_name}` notation. There are two types of properties – implicit and user-defined properties.

Implicit Properties

Implicit properties are properties that are available by default to any Maven project. For example, Maven exposes its Project Object Model properties using the “project.” prefix. To access the artifactId value inside the pom.xml file, you can use the `${project.artifactId}` as shown in the following:

```
<build>
    <finalName>${project.artifactId}</finalName>
</build>
```

Similarly, to access properties from settings.xml file, you can use the “settings.” prefix. Finally, the “env.” prefix can be used to access environment variable values. For example, `${env.PATH}` will return the value of PATH environment variable.

User-Defined Properties

Maven allows you to declare custom 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. 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>
  <!-- Removed for brevity -->

<properties>
  <junit.version>4.12</junit.version>
</properties>

<developers>
  <!-- Removed for brevity -->
</developers>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

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\test`, 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 lifecycle, plug-ins, build phases, goals, and how to leverage them effectively.

CHAPTER 5

Maven Lifecycle

Central to Maven is its lifecycle that provides a uniform interface for building and distributing projects. In this chapter, we will review the lifecycle and building blocks that make up the lifecycle.

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 ---
```



```
[INFO] Compiling 1 source file to C:\apress\gswm-book\chapter5\
gswm\target\classes
[INFO] -----
[INFO] BUILD SUCCESS
[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] --- 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
```

Maven provides an out-of-the box Help plug-in that can be used to list available goals in a given plug-in. Listing 5-3 shows the Help plug-in's describe goal to display goals inside the compiler plug-in.

Listing 5-3. Maven Help Plug-in

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

```
[INFO] Scanning for projects...
```

```
Name: Apache Maven Compiler Plugin
```

```
Description: The Compiler Plugin is used to compile the sources
of your project.
```

```
Group Id: org.apache.maven.plugins
```

```
Artifact Id: maven-compiler-plugin
```

```
Version: 3.8.1
```

```
Goal Prefix: compiler
```

```
This plugin has 3 goals:
```

```
compiler:compile
```

```
  Description: Compiles application sources
```

```
compiler:help
```

```
  Description: Display help information on maven-compiler-plugin.
  Call mvn compiler:help -Ddetail=true -Dgoal=<goal-name> to
  display parameter details.
```

```
compiler:testCompile
```

```
  Description: Compiles application test sources.
```

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 8. As of version 3.8, the Maven compiler plug-in compiles the code against Java 1.6. Thus, you will need to modify the behavior of this plug-in in the `pom.xml` file, as shown in Listing 5-4.

Listing 5-4. 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>3.8.1</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.

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_artifact_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>`.

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.

Maven lifecycle is an abstract concept and can't be directly executed. Instead, you execute one or more phases. For example, the command `mvn package` will execute the `package` phase of the default lifecycle. In addition to clearly defining the ordering of phases in a lifecycle, Maven also automatically executes all the phases prior to a requested phase. So, when the `mvn package` command is run, Maven will run all prior phases such as `compile` and `test`.

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

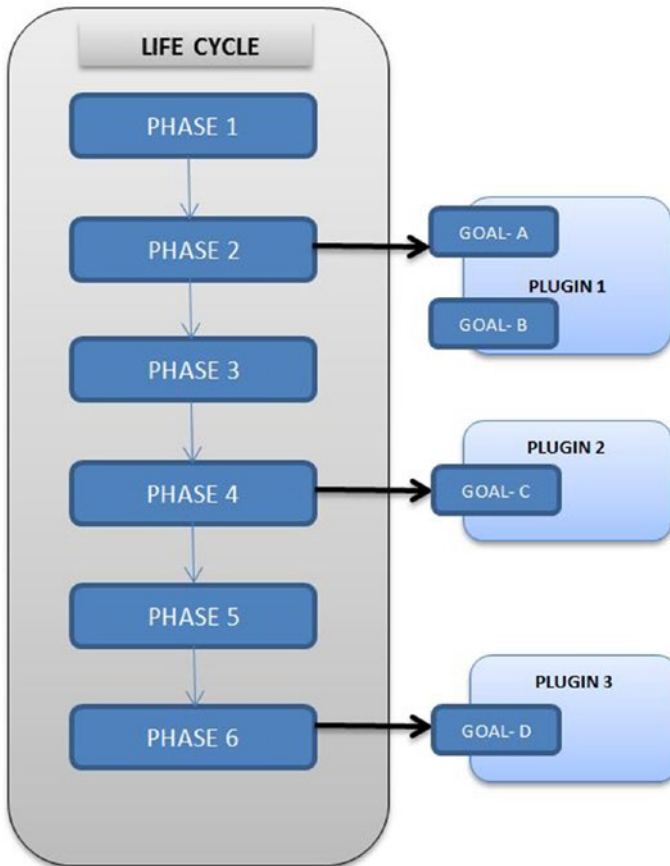


Figure 5-1. Association between lifecycle, phases, goals, and plug-ins

It is valid for a Maven phase to not have any goals associated with it. In that case, Maven will skip the phase execution. Such phases serve as placeholders for users and third-party vendors to associate their custom-built goals.

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. For example, if the packaging element is `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. Figure 5-2 shows a portion of the internal lifecycle associated with a WAR project.

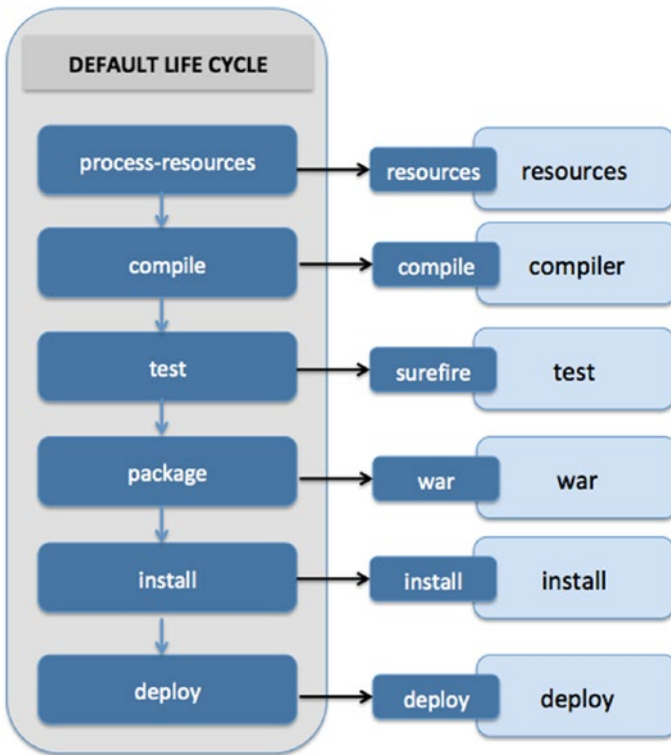


Figure 5-2. Default lifecycle for WAR project

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
```

Plug-in Development

Developing custom plug-ins for Maven is very straightforward. 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).

This section explains how to develop a `SystemInfoPlugin` that displays system information such as Java version, operating system, and the like, on the console running Maven command.

Let's start this plug-in development by creating a Maven Java project, named `gswm-maven-plugin`, as shown in Figure 5-3.

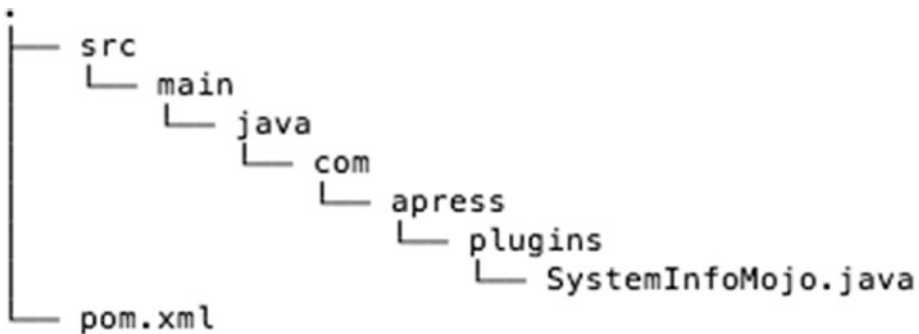


Figure 5-3. Maven project for plug-in development

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 5-5. Notice that the packaging type is `maven-plugin`. We added the `maven-plugin-api` and `maven-plugin-annotations` dependencies, because they are needed for plug-in development. We will be leveraging Apache Commons Lang to get system information. Hence, we have also added the Apache Commons Lang 3 dependency.

Listing 5-5. The `pom.xml` with Dependencies

```

<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.plugins</groupId>
  <artifactId>gswm-maven-plugin</artifactId>
  <version>1.0.0</version>
  <packaging>maven-plugin</packaging>
  <description>System Info Plugin</description>

```

```

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
    <version>3.6.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.maven.plugin-tools</groupId>
    <artifactId>maven-plugin-annotations</artifactId>
    <version>3.6.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.9</version>
  </dependency>
</dependencies>
<!-- Use the latest version of Plugin -->
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-plugin-plugin</artifactId>
      <version>3.6.0</version>
    </plugin>
  </plugins>
</build>

```

```

        </plugins>
    </build>
</project>

```

The next step in the development process is creating the MOJO. Listing 5-6 shows the code for `SystemInfoMojo`. The `@Mojo` annotation marks the `SystemInfoMojo` class as a Mojo with “systeminfo” as the goal name. The `execute` method contains that goal logic. In `SystemInfoMojo`, we simply log several pieces of system information to the console.

Listing 5-6. `SystemInfoMojo` Java Class

```

package com.apress.plugins;

import org.apache.commons.lang3.SystemUtils;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;
import org.apache.maven.plugins.annotations.Mojo;

@Mojo( name = "systeminfo" )
public class SystemInfoMojo extends AbstractMojo {

    @Override
    public void execute() throws MojoExecutionException,
        MojoFailureException {
        getLog().info( "Java Home: " + SystemUtils.JAVA_HOME );
        getLog().info( "Java Version: " + SystemUtils.JAVA_
            VERSION );
        getLog().info( "OS Name: " + SystemUtils.OS_NAME );
        getLog().info( "OS Version: " + SystemUtils.OS_
            VERSION );
        getLog().info( "User Name: " + SystemUtils.USER_NAME );
    }
}

```

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 5-7.

Listing 5-7. Maven install Command

```
C:\apress\gswm-book\chapter5\gswm-maven-plugin>mvn install
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.apress.plugins:gswm-maven-plugin >-----
[INFO] Building gswm-maven-plugin 1.0.0
[INFO] -----[ maven-plugin ]-----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-
resources) @ gswm-maven-plugin
[INFO] java-annotations mojo extractor found 1 mojo descriptor.
[INFO] --- maven-install-plugin:2.4:install (default-install)
@ gswm-maven-plugin ---
[INFO] Installing C:\apress\gswm-book\chapter5\gswm-maven-
plugin\target\gswm-maven-plugin-1.0.0.jar to C:\Users\<<USER_
NAME>>\.m2\repository\com\apress\plugins\gswm-plugin\1.0.0\
gswm-maven-plugin-1.0.0.jar
[INFO] Installing C:\apress\gswm-book\chapter5\gswm-maven-
plugin\pom.xml to C:\Users\<<USER_NAME>>\.m2\repository\com\
apress\plugins\gswm-maven-plugin\1.0.0\gswm-maven-plugin--
1.0.0.pom
[INFO] -----
[INFO] BUILD SUCCESS
```

Now you're ready to start using this plug-in. Remember that the syntax to run any goal is `mvn pluginId:goal-name`. Listing 5-8 shows this plug-in in action. Notice system information displayed on the console.

Listing 5-8. Running the SystemInfoMojo Plug-in

```
C:\apress\gswm-book\chapter5\gswm-plugin>mvn com.apress.
plugins:gswm-maven-plugin:systeminfo
[INFO] Scanning for projects...
[INFO] --- gswm-maven-plugin:1.0.0:systeminfo (default-cli)
@ gswm-maven-plugin ---
[INFO] Java Home: C:\java\jdk-11
[INFO] Java Version: 11.0.1
[INFO] OS Name: Windows
[INFO] OS Version: 10
[INFO] User Name: Balaji
[INFO] -----
```

The newly developed plug-in is also ready to be used in other Maven projects. Listing 5-9 shows a portion of the POM file that attaches `systeminfo` goal to the `validate` phase.

Listing 5-9. POM File Using `systeminfo` Goal

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.plugins</groupId>
  <artifactId>gswm-plugin-test</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <description>Plugin Test</description>
```

```

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies />
<build>
  <plugins>
    <plugin>
      <groupId>com.apress.plugins</groupId>
      <artifactId>gswm-maven-plugin
      </artifactId>
      <version>1.0.0</version>
      <executions>
        <execution>
          <phase>validate</phase>
          <goals>
            <goal>systeminfo
            </goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>

```

When the Maven phase such as compile or package is invoked, you will see the output of the systeminfo goal as shown in Listing 5-10.

Listing 5-10. Compile Phase Output

```
mvn compile
[INFO] Scanning for projects...
[INFO] Building gswm-plugin-test 1.0.0
[INFO] Java Home: C:\java\jdk-11
[INFO] Java Version: 11.0.1
[INFO] OS Name: Windows
[INFO] OS Version: 10
[INFO] User Name: Balaji
[INFO] --- maven-resources-plugin:2.6:resources (default-resources)
@ gswm-plugin-test ---
```

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 lifecycle. 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 in the book, you have created Maven projects 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 sheets (CSS), approved JavaScript libraries, and reusable components. Developers using this archetype to generate projects will automatically 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 new archetypes and generate projects from existing 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. At the time of writing this book, there are couple thousand 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] 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: remote -> am.ik.archetype:elm-spring-boot-blank-archetype
(Blank multi project for Spring Boot + Elm)
2: remote -> am.ik.archetype:maven-reactjs-blank-archetype
(Blank Project for React.js)
3: remote -> am.ik.archetype:msgpack-rpc-jersey-blank-archetype
(Blank Project for Spring Boot + Jersey)

.....
.....

2460: remote -> ws.osiris:osiris-archetype (Maven Archetype for
Osiris)
2461: remote -> xyz.luan.generator:xyz-gae-generator (-)
2462: remote -> xyz.luan.generator:xyz-generator (-)
2463: local -> com.inflinx.book.ldap:practical-ldap-empty-
archetype (-)
2464: local -> com.inflinx.book.ldap:practical-ldap-archetype (-)
2465: local -> com.apress.gswm:gswm-web-archetype (gswm-web-
archetype)
Choose a number or apply filter (format: [groupId:]artifactId,
case sensitive contains): 1398:
```

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](#).

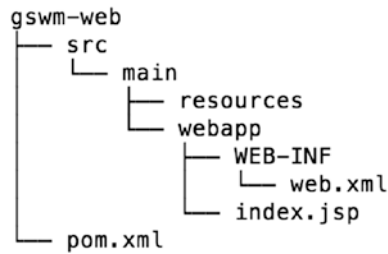


Figure 6-1. Maven web project structure

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 Jetty plug-in added.

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

```

<project>
  <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.eclipse.jetty</groupId>
      <artifactId>jetty-maven-plugin</artifactId>
      <version>9.4.12.RC2</version>
    </plugin>
  </plugins>
</build>
</project>

```

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

```
mvn jetty:run
```

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

Listing 6-3. Output from the Jetty run Command

```

[INFO] Started o.e.j.m.p.JettyWebAppContext@e38f0b7{Archetype
Created Web Application,/,file: C:/apress/gswm-book/chapter6/
gswm-web/src/main/webapp/,AVAILABLE}{file:///C:/apress/gswm-
book/chapter6/gswm-web/src/main/webapp/}
[INFO] Started ServerConnector@5a0e0886{HTTP/1.1,[http/1.1]}
{0.0.0.0:8080}
[INFO] Started @5120ms
[INFO] Started Jetty Server

```

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

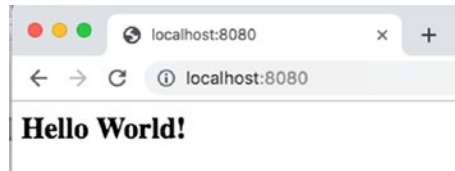


Figure 6-2. Web project launched in browser

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.

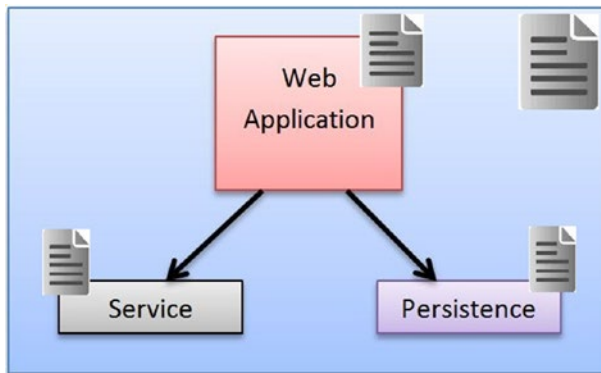


Figure 6-4. *Maven multimodule project*

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 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] -----
[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] -----

```

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 4.0 compatible and 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 target folder and 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 with the following code. This will upgrade the web application to use Servlet 4.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/
javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

  <display-name>Archetype Created Web Application
  </display-name>
</web-app>
```

3. Add the Servlet 4.0 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>4.0.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>gswm-web</finalName>
    <plugins>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.12.RC2</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.

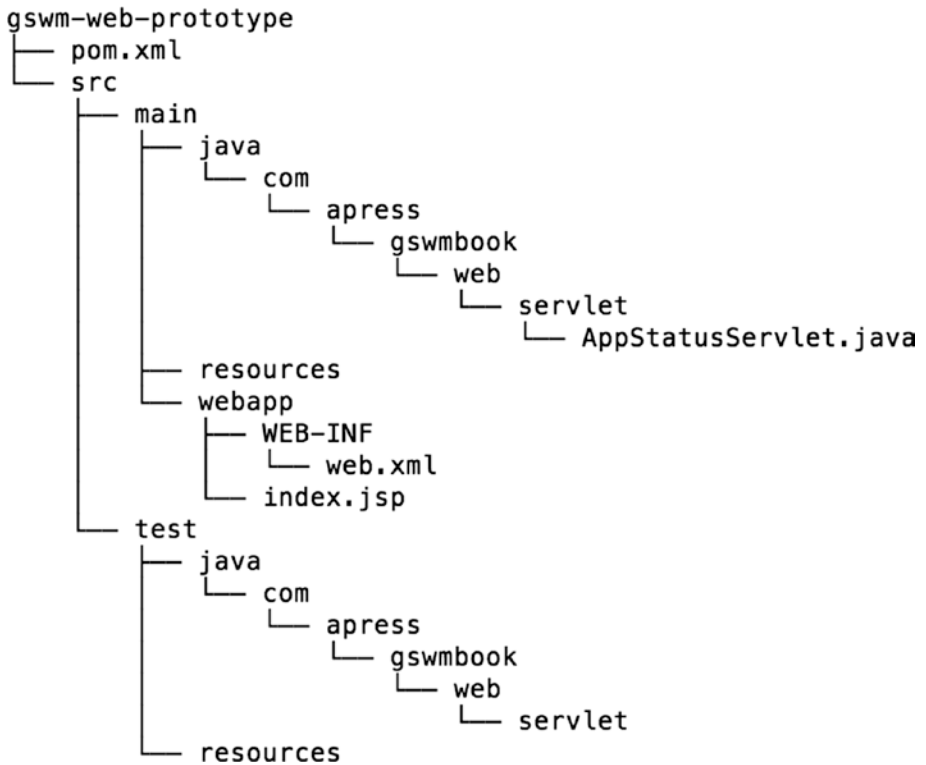


Figure 6-5. *Generated prototype project*

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 `pom.xml` and `src` directory and its files from the `C:\apress\gswm-book\chapter6\gswm-web-prototype\target\generated-sources\archetype` folder to the `gswm-web-archetype` folder.

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

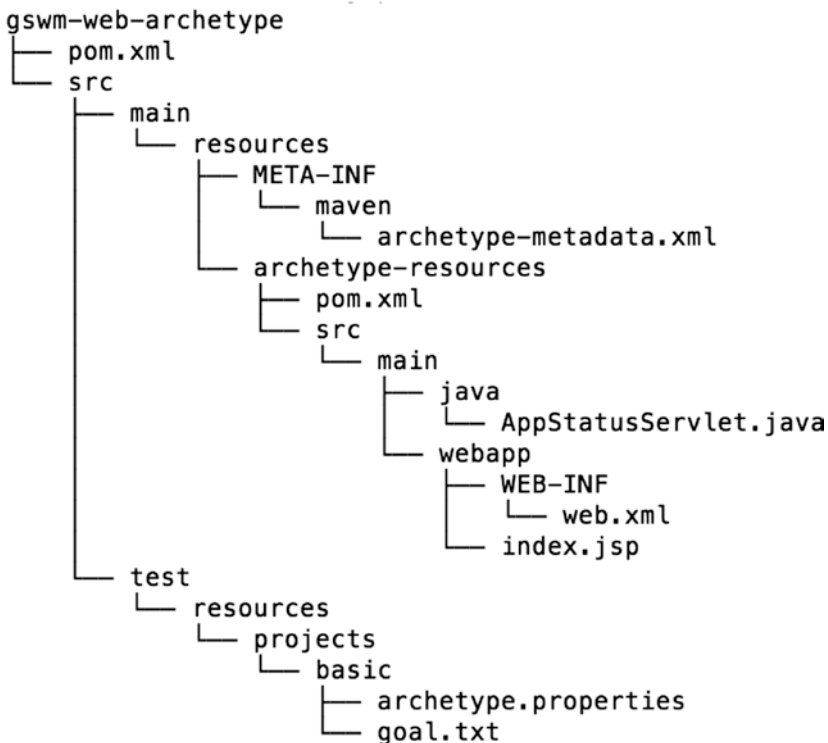


Figure 6-6. Web archetype project structure

Let's start the modification process with the `pom.xml` file located at the root of `gswm-web-archetype` folder. Change the `artifactId` to "gswm-web-archetype" in the `pom` file. Next we will modify the `pom.xml` file located at `gswm-web-archetype\src\main\resources\archetype-resources`. Change the `<finalName>` 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.

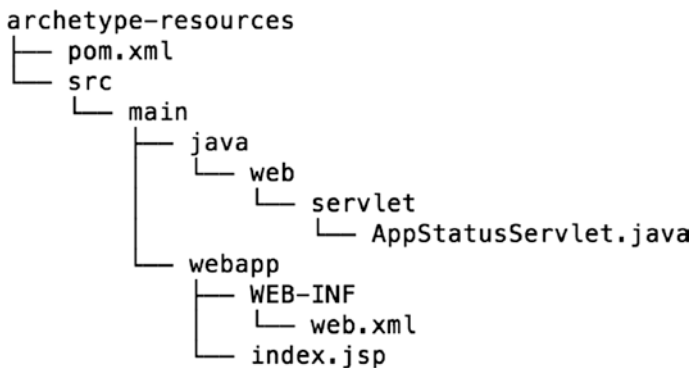


Figure 6-7. *AppStatusServlet under the web.servlet package*

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] -----

```

Because the `pom.xml` file for the `test-project` already has the embedded Jetty plug-in, run `mvn jetty: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/status`. You will see the string `OK` displayed.

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 develop the project. This chapter looks at some of Maven's tools and plug-ins, which make publishing and maintenance of online documentation a breeze.

In this chapter, you 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 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.

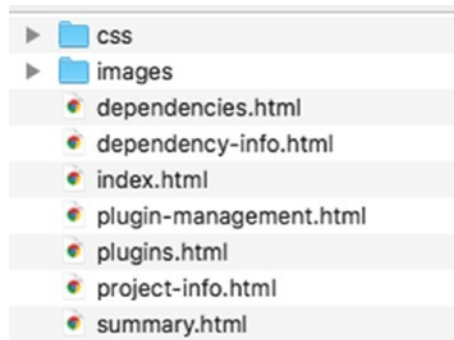


Figure 7-1. *Generated site folder*

Open the `index.html` file in a browser to view the generated site. Maven automatically applies a default skin to the site and generates the corresponding images and CSS files. Figure 7-2 shows the generated `index.html` file.

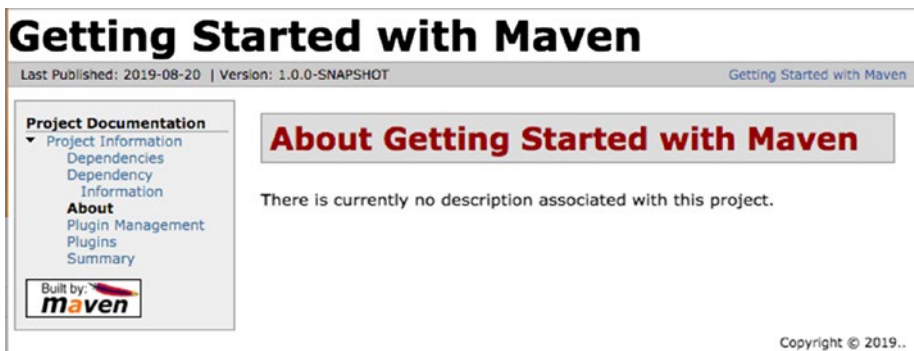


Figure 7-2. *Generated index page*

Clicking the “Dependencies” link at the bottom of the left navigation will take you to the Project Dependencies 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.

Project Dependencies

test

The following is a list of test dependencies for this project. These dependencies are only required to compile and run unit tests for the application:

GroupId	ArtifactId	Version	Type	License
junit	junit	4.11	jar	Common Public License Version 1.0

Project Transitive Dependencies

The following is a list of transitive dependencies for this project. Transitive dependencies are the dependencies of the project dependencies.

test

The following is a list of test dependencies for this project. These dependencies are only required to compile and run unit tests for the application:

GroupId	ArtifactId	Version	Type	License
org.hamcrest	hamcrest-core	1.3	jar	New BSD License

Project Dependency Graph

Figure 7-3. Project dependencies page

Maven allows you to add information to pom.xml file so that the generated site contains useful information. Listing 7-1 shows the updated pom.xml file. For the site to successfully generate, we are explicitly declaring the latest version of the maven-site-plugin.

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 and Dependency removed for brevity -->
<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>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.8.2</version>
</plugin>
</plugins>
</build>

</project>

```

In Listing 7-1, we use the `description` element to provide a description of the project. The `mailingList` element information about different mailing lists associated with the project, and the `license` element includes the project's license. With updated `pom.xml` file in place, 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. Figures 7-4A and 7-4B show 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

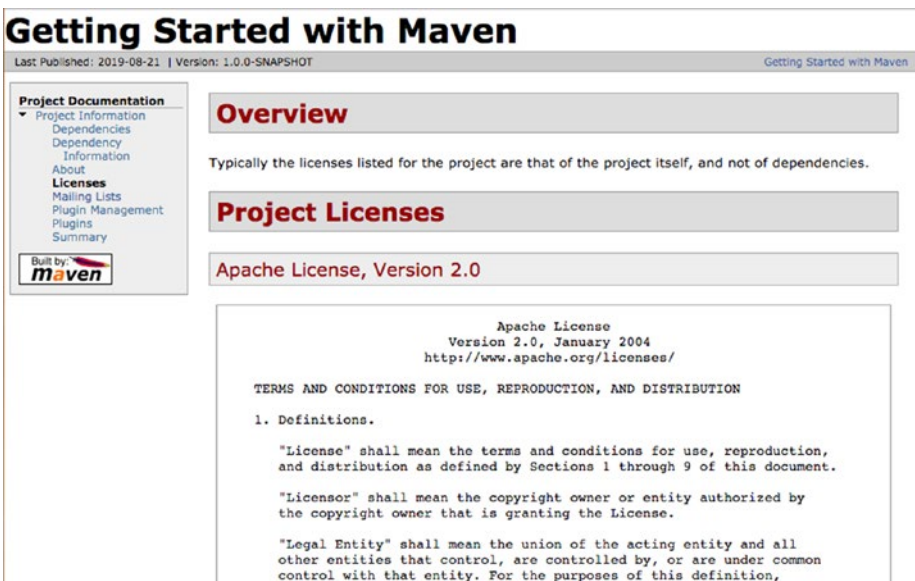


Figure 7-4B. Generated Project Licenses page

Advanced Site Configuration

In the preceding section, project information was specified in the pom.xml file for Maven to use during site generation. For larger 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.

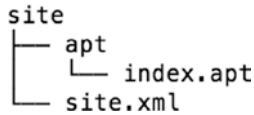


Figure 7-5. Site folder directory structure

The `site.xml` file, also known as the *site descriptor*, is used to customize the generated site. We will look at this file 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. To update the existing `gswm` project, run the following command in the `C:\apress\gswm-book\chapter7\gswm` folder. When prompted, enter the values for `groupId`, `artifactId`, and `package`.

```

mvn archetype:generate -DarchetypeGroupId=org.apache.maven.
archetypes -DarchetypeArtifactId=maven-archetype-site-simple
-DarchetypeVersion=1.4

```

```

Define value for property 'groupId': : com.apress.gswmbook

```

```

Define value for property 'artifactId': : gswm

```

```

Define value for property 'version': 1.0-SNAPSHOT: :
1.0.0-SNAPSHOT

```

```

Define value for property 'package': com.apress.gswmbook: :

```

```

<<Press Enter>>

```

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.

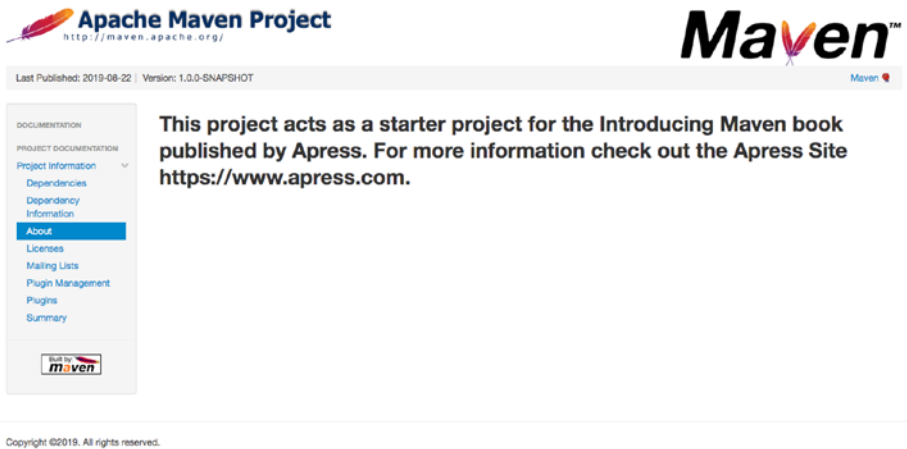


Figure 7-6. *About page with new content*

The `site.xml` file allows you to customize the generated site such as changing the title and overriding default navigation and look and feel. To better understand `site.xml` capability, let's change the generated 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.

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 links to custom web pages/content/Wiki pages you want to include in the site.

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

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="{artifactId}" xmlns="http://maven.apache.org/
DECORATION/1.8.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
```

```
xsi:schemaLocation="http://maven.apache.org/DECORATION/1.8.0
http://maven.apache.org/xsd/decoration-1.8.0.xsd">
<bannerLeft>
  <name>Apress</name>
  <src>images/company.png</src>
  <href>http://apress.com</href>
</bannerLeft>

<skin>
  <groupId>org.apache.maven.skins</groupId>
  <artifactId>maven-fluido-skin</artifactId>
  <version>1.7</version>
</skin>

<body>
  <links>
    <item name="Maven" href="https://maven.apache.org/" />
  </links>

  <menu name="Documentation">
    <item name="Apache Site" href="http://www.apache.org" />
  </menu>

  <menu ref="reports" />
</body>
</project>
```

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

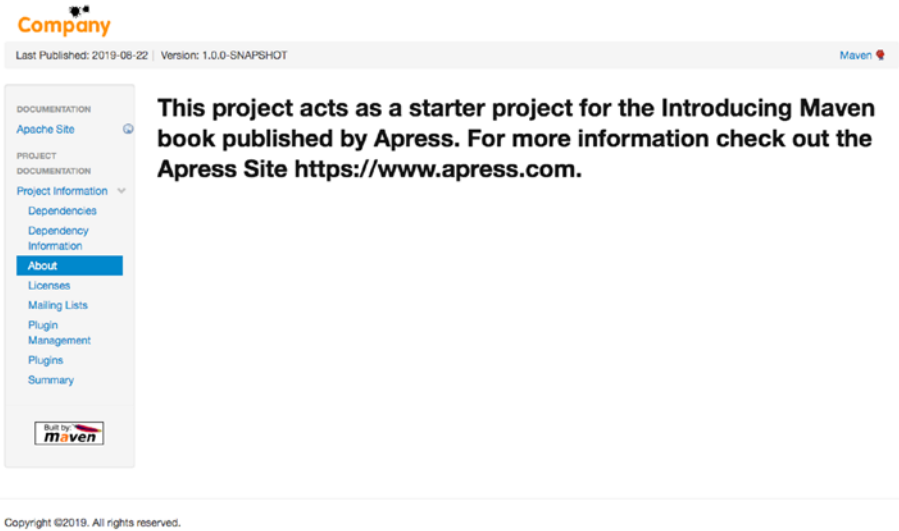


Figure 7-7. About page with the new logo

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>
    <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 will notice the `apidocs` folder created under `gswm /target/site`. Launch `index.html` file under `site`, and navigate to Project Reports ► Javadoc. Figure 7-8 shows the Javadoc generated for the `gswm` project.

The screenshot displays a Javadoc page for the `HelloWorld` class. At the top, there are navigation tabs: PACKAGE, CLASS (selected), USE, TREE, DEPRECATED, INDEX, and HELP. Below the tabs, there is a search bar and a summary section for the class.

The class declaration is shown as:

```
public class HelloWorld
extends Object
```

The **Constructor Summary** section contains a table with the following data:

Constructor	Description
<code>HelloWorld()</code>	

The **Method Summary** section contains a table with the following data:

Modifier and Type	Method	Description
<code>void</code>	<code>sayHello()</code>	

Below the method summary, there is a section for **Methods inherited from class java.lang.Object**, listing methods: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`.

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 failure results 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.

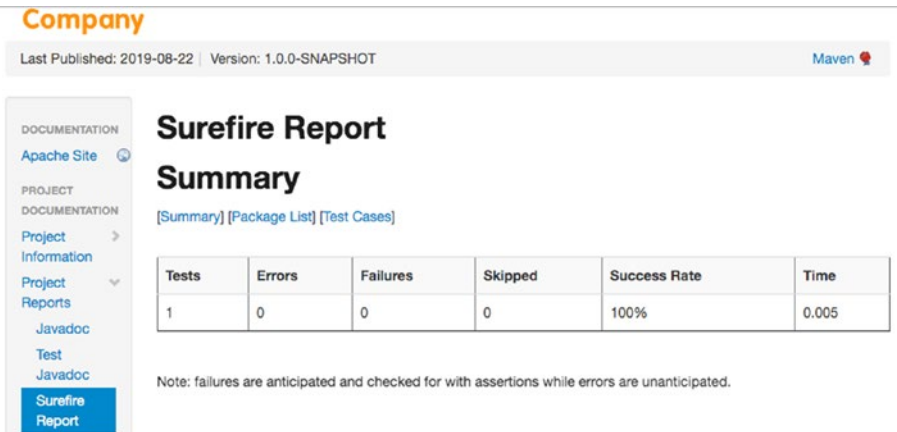
The Surefire plug-in is configured in the same way as the Javadoc plug-in in the reporting section of the pom file. Listing 7-5 shows the Surefire plug-in configuration.

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. Launch `index.html` file under `site`, and navigate to Project Reports ► Surefire Report. Figure 7-9 shows Surefire Report for the `gswm` project.



Company

Last Published: 2019-08-22 | Version: 1.0.0-SNAPSHOT Maven

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
1	0	0	0	100%	0.005

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

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. *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.

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

```
<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.

The screenshot shows a JaCoCo report for a project named "Getting Started with Maven". At the top, there is a progress bar for "default" which is 100% green. Below this is a table with the following data:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		100%	0 of 0	n/a	0	2	0	3	0	2	0	1
Total	0 of 7	100%	0 of 0	n/a	0	2	0	3	0	2	0	1

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.

SpotBugs Bug Detector Report

The following document contains the results of SpotBugs

SpotBugs Version is 3.1.12

Threshold is *medium*

Effort is *default*

Summary

Classes	Bugs	Errors	Missing Classes
1	0	0	0

Files

Class	Bugs
-------	------

Figure 7-11. Generated SpotBugs 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 lifecycle and the configuration needed to produce documentation. You also looked at generating Javadocs, test coverage, and SpotBugs reports.

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

CHAPTER 8

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'.
```

Figure 8-2. Starting Nexus

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.

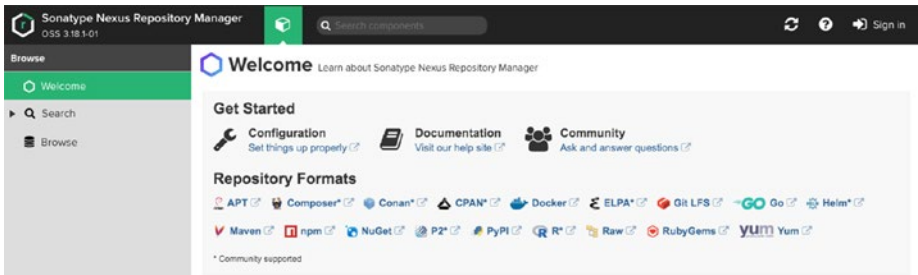


Figure 8-3. Nexus launch screen

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.

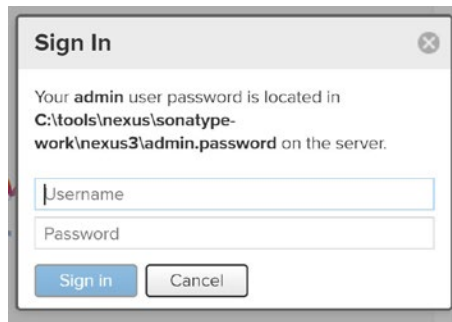


Figure 8-4. Nexus login modal

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.

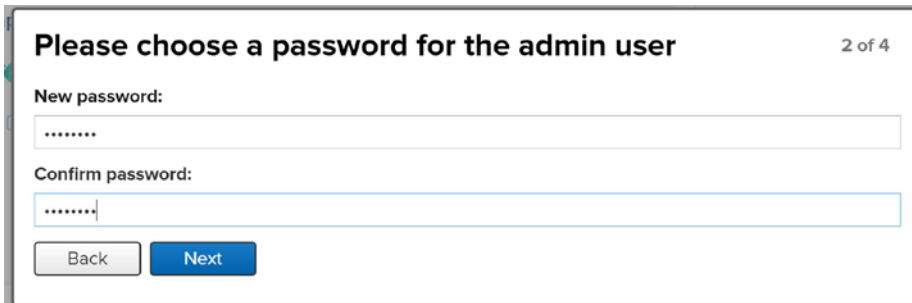


Figure 8-5. Nexus change password screen

Now that Nexus is installed and running, let's modify the `gswm` 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.

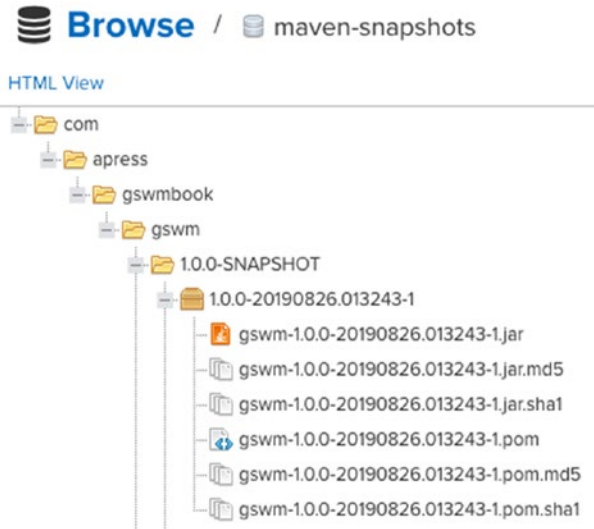


Figure 8-6. SNAPSHOT artifact under Nexus

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.

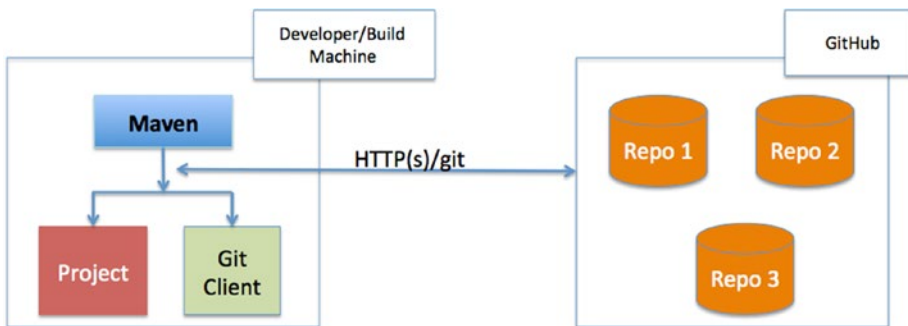


Figure 8-7. *Interaction between Maven and GitHub*

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.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner **Repository name ***

 bava ▾ / intro-maven 

Great repository names are short and memorable. Need inspiration? How about [ubiquitous-giggle?](#)

Description (optional)


Repository to demo Maven Release Process

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾ | Add a license: **None** ▾ 

Create repository

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 remote 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.

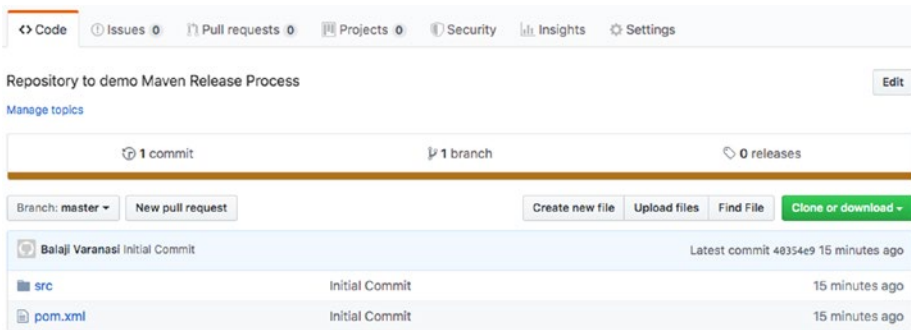


Figure 8-11. Project checked into GitHub

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 SCM information.

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

```

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`.

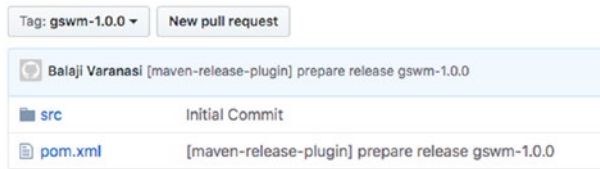


Figure 8-12. *Git tag created upon prepare execution*

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.

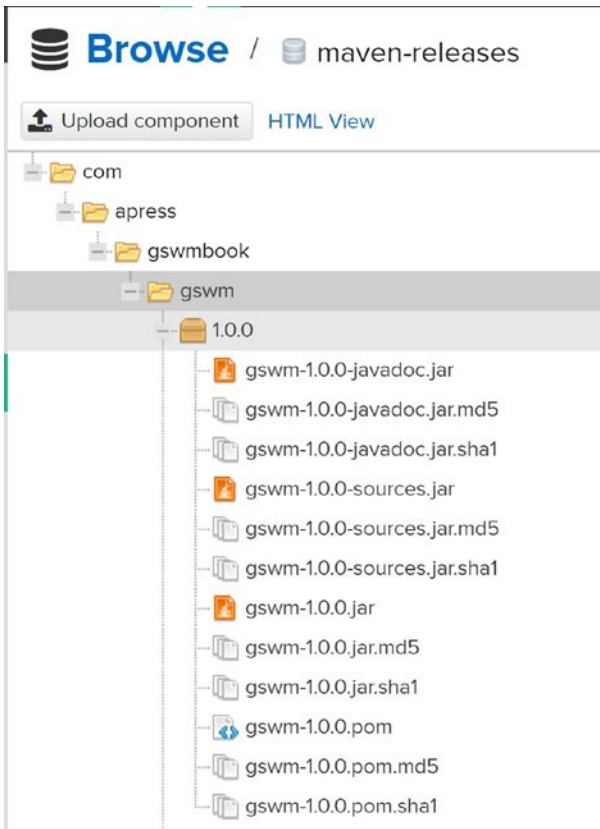


Figure 8-13. Nexus with released artifact

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.

In the next chapter, we will learn the concepts of continuous integration (CI) and install and configure Jenkins – a popular open source CI tool.

CHAPTER 9

Continuous Integration

Continuous integration or CI is a software development best practice where developers integrate changes to their code into a common repository several times a day. Each committed change would result in an automatic build that would compile the code, run tests, and generate a new version of the artifact. Any errors during the build process will be immediately reported to the development team. This frequent code integration allows developers to catch and resolve integration issues early in the development cycle.

A visual representation of continuous integration along with the components involved is shown in Figure 9-1. The CI flow gets kicked off with a developer submitting her changes to a source control system such as Git or SVN. A CI server gets notified or watches/polls for new code changes and upon finding a change will check out the source code and starts the build process. On a successful build, the CI server can publish the artifact to a repository or to a test server. As the last step, notifications on build status get sent to the development team.

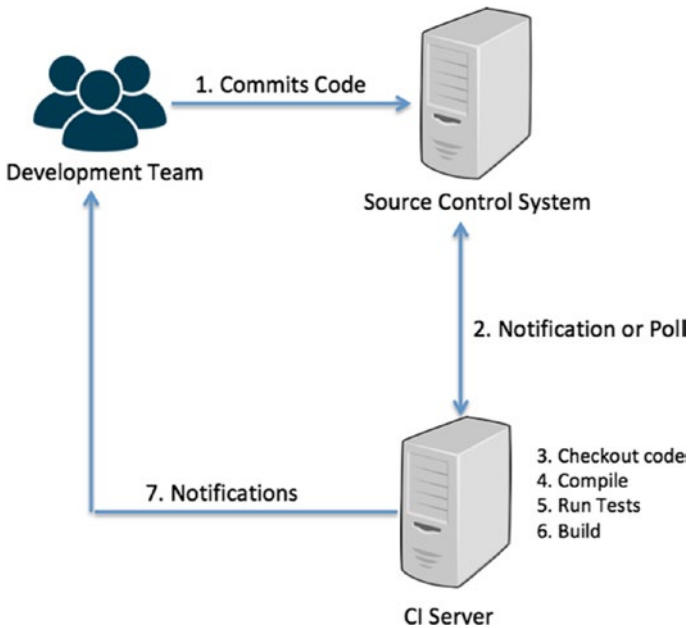


Figure 9-1. CI components

Jenkins is a popular open source CI server that integrates well with Maven. Other popular CI servers include Bamboo, TeamCity, and GitLab. In this chapter, we will install and configure Jenkins to trigger automatic builds for a Maven-based project.

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.

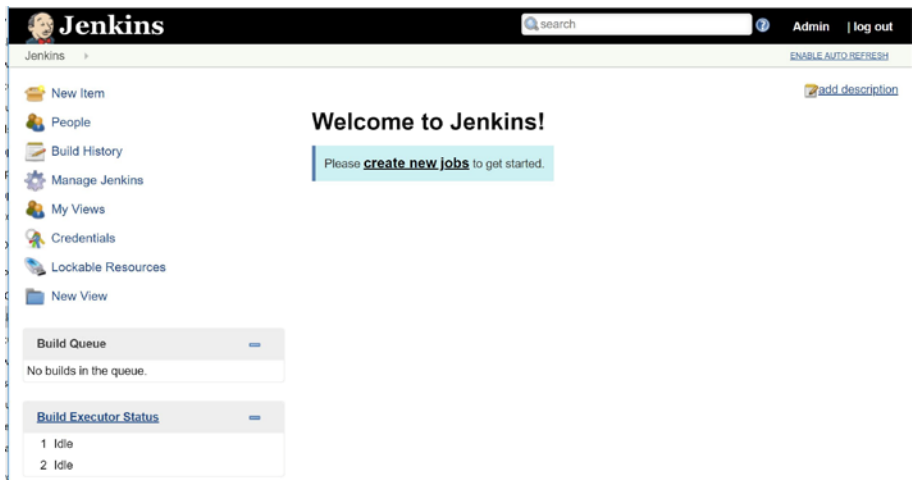


Figure 9-2. Jenkins dashboard

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 `gswm-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.

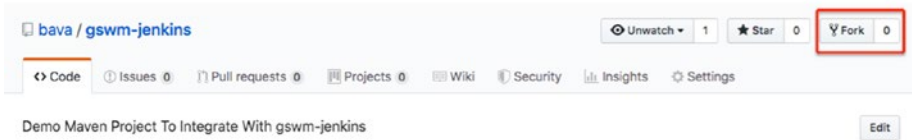


Figure 9-3. Fork gswm-jenkins repository

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.

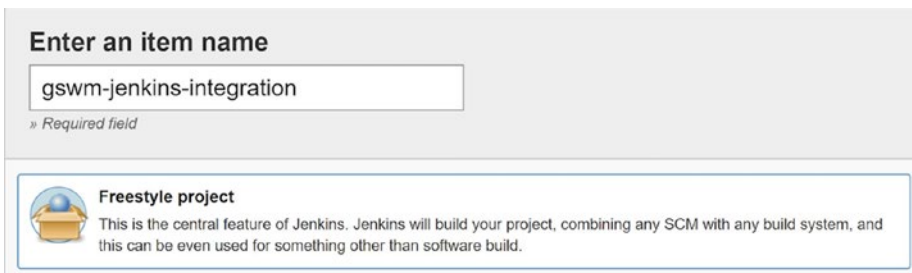


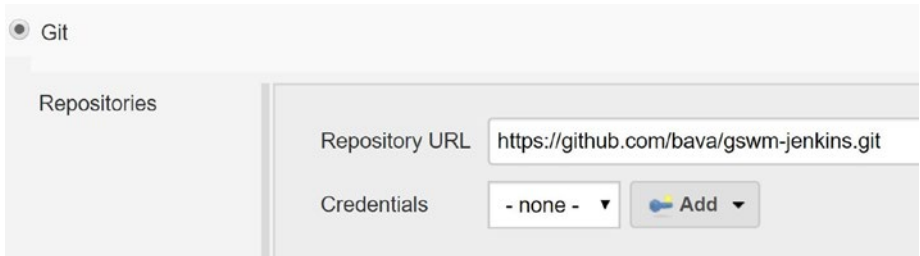
Figure 9-4. New Item screen

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

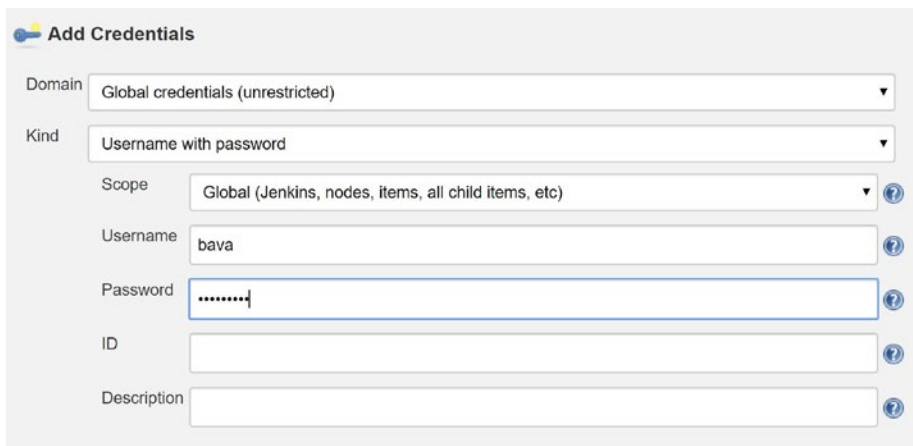
On the “Source Code Management” section, select the “Git” radio button and enter the URL to your GitHub repository as shown in Figure 9-6. This is the GitHub clone URL that you can find by clicking “Clone or download” under repository name.



The screenshot shows the Jenkins configuration interface for a new item. Under the 'Source Code Management' section, the 'Git' radio button is selected. The 'Repository URL' field is populated with 'https://github.com/bava/gswm-jenkins.git'. Below this, the 'Credentials' dropdown menu is set to '- none -', and there is an 'Add' button with a key icon next to it.

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.



The screenshot shows the 'Add Credentials' dialog box in Jenkins. The 'Domain' dropdown is set to 'Global credentials (unrestricted)'. The 'Kind' dropdown is set to 'Username with password'. The 'Scope' dropdown is set to 'Global (Jenkins, nodes, items, all child items, etc)'. The 'Username' field contains the text 'bava'. The 'Password' field is masked with dots. The 'ID' and 'Description' fields are empty. There are help icons (question marks) next to the 'Scope', 'Password', 'ID', and 'Description' fields.

Figure 9-7. *GitHub credential input*

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.

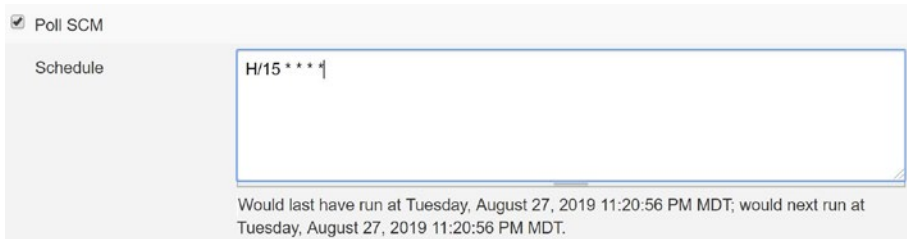


Figure 9-8. Build Trigger poll schedule

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.

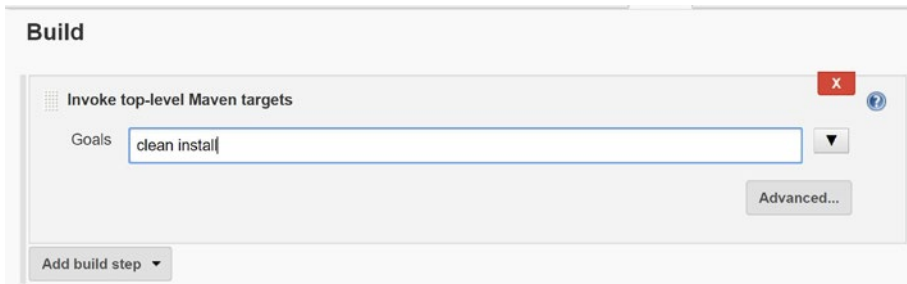


Figure 9-9. Build step for Maven

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.

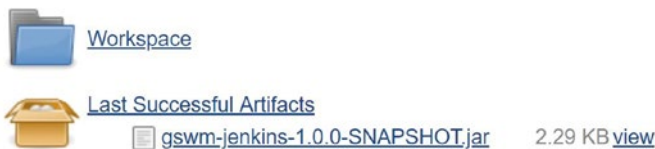


Figure 9-13. Jenkins Project page - Build Artifact

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

Summary

In this chapter, you learned about continuous integration and configured Jenkins to interact with a Maven project.

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 software development processes.

Index

A, B

Ant + Ivy

Archetypes

built-in, [69, 70](#)

creation

AppStatusServlet

java file, [82, 85](#)

gswm-web-archetype, [84](#)

gswm-web-prototype, [80, 83](#)

pom.xml, [80, 81](#)

project structure, [82, 83](#)

Servlet 4.0, [80](#)

defined, [69](#)

test-project, [86, 87](#)

C

Code coverage, [102–104](#)

Continuous integration (CI)

components, [127, 128](#)

Jenkins

archive artifacts, [132](#)

build trigger, [132](#)

configuration, item

screen, [130](#)

GitHub credentials, [131, 132](#)

GitHub project, [130](#)

installation, [128, 129](#)

JUnit test, [133](#)

Maven, [129](#)

triggering build job, [133, 134](#)

Convention over configuration

(CoC), [4, 5](#)

D

Dependency scope, [32, 33](#)

Dependency installation

add, repository, [34](#)

manual, [33](#)

Dependency management

architecture, [24](#)

dependency

identification, [28, 29](#)

Maven central, [24](#)

repository manager, [25, 26](#)

Spring/JBoss repositories, [27, 28](#)

transitive dependencies, [29–31](#)

E, F

Enterprise archive (EAR), [28](#)

Enterprise JavaBeans (EJBs), [74](#)

Extensible markup

language (XML), [5](#)

G, H

Global settings, [16](#)

Goal, Maven Lifecycle

- clean, [54](#)

- compile, [53](#), [54](#)

- Help plug-in, [55](#)

- plug-ins, [54](#)

- pom.xml file, [55](#), [56](#)

Gradle, [7](#)

Group, artifact, and zversion

- (GAV), [28](#)

I

Integrated development

- environments (IDEs), [1](#)

J, K

Java Development Kit (JDK), [11](#)

Javadoc, [99](#), [100](#)

Java Enterprise Edition (JEE), [74](#)

Java Runtime Environment (JRE), [11](#)

Java's Plain Old Java

- Object (POJO), [61](#)

Java virtual machine (JVM), [14](#)

Jenkins, [128](#), [129](#)

JUnit/TestNG, [101](#)

L

Lifecycle

- built-in, [57](#), [58](#)

- goals, [58](#), [59](#)

- phases, [57-59](#)

- WAR project, [60](#)

- packaging element, [60](#)

M, N, O

Maven, [129](#)

- Ant + Ivy, [5](#)

 - Ant build.xml File, [6](#)

 - compile, [5](#)

 - Ivy listing, [6](#)

 - target, [5](#)

 - task, [5](#)

- archetypes, [4](#)

- archive download, [12](#)

- artifact errors, [19](#)

- CoC, [4](#), [5](#)

- dependency management, [2](#)

- directory structure, [2](#)

- Doxia, [10](#)

- Gradle, [7](#)

 - default build.gradle file, [7](#)

 - build tool usage, [8](#)

 - DSL, [7](#)

- help command, [15](#)

- IDE, [21](#)

- installation

 - directory contents, [12](#)

 - Mac, [13](#)

 - testing, [14](#)

 - windows, [13](#)

- open source, [4](#)

- plug-ins, [3](#)

- proxy, setup, [19](#)

- securing passwords, 20
 - settings.xml file, 16–18
 - skeleton settings.xml file, 17
 - source control/code
 - management (SCM), 9
 - tool support, 3
 - wagon, 9
 - Maven Doxia, 10
 - Maven Old Java Object (MOJOs), 61
 - Maven release
 - clean goal, 123
 - Git client installation, 115
 - GitHub repositories, 115, 116
 - interacting with GitHub, 114
 - Nexus
 - distributionManagement
 - element, 110
 - installation, 108
 - login modal, 109
 - repository managers, 107
 - server information, 111
 - SNAPSHOT artifact, 112
 - perform goal
 - command, 124, 125
 - Nexus, 126
 - phases, 123
 - prepare goal
 - command, 121
 - execution, 122
 - GitHub details, 120
 - operations, 118
 - SCM information, 119
 - source code checking, 116–118
 - maven.test.skip property, 61
 - mkdir gswm command, 39
 - Multimodule project
 - interactiveMode
 - parameter, 77
 - mvn package command, 79
 - Parent pom.xml file, 76, 77
 - service project, 76
 - structure, 74
 - visual representation, 75
 - web module, 78, 79
 - mvn package command, 58
- ## P, Q, R
- Plug-ins
 - Apache Commons Lang, 62
 - Maven Java project, 61, 62
 - mvn install command, 65
 - pom.xml, 62–64
 - systeminfo goal, 66–68
 - SystemInfoMojo, 64, 66
 - SystemInfoPlugin, 61
 - pom.xml file, 24, 40, 41
 - implicit properties, 50
 - user-defined
 - properties, 50, 51
 - Project
 - building
 - HelloWorld class, 42
 - packaged JAR file, 44
 - Maven Package, 43
 - project structure, 43

INDEX

Project (*cont.*)

testing

HelloWorldTest, [47, 49](#)

JUnit dependency, [44, 46](#)

structure, [47](#)

target folder, [49](#)

Tree command, [46](#)

organization

components, [38](#)

create, project, [39, 40](#)

directories, [39](#)

structure, [37](#)

S

Site lifecycle

configuration

archetypes, [95](#)

directory structure, [94, 95](#)

file contents, [96, 97](#)

logo, [98](#)

contents, [89](#)

dependencies, [90](#)

description

element, [92, 93](#)

generated web site, [93](#)

index page, [90](#)

SNAPSHOT qualifier, [42](#)

Source control/code management
(SCM) systems, [9, 38](#)

SpotBugs, [104, 105](#)

T, U, V

Test driven development

Surefire plug-in

configuration, [101](#)

Surefire report, [101, 102](#)

Transitive dependencies, [29](#)

dependency mediation, [29](#)

JUnit, [31](#)

tree plug-in, [30](#)

W, X, Y, Z

Web application archive (WAR), [5](#)

Web project

browser, [73, 74](#)

inputs, information, [71](#)

Jetty plug-in, [72, 73](#)

Jetty run Command, [73](#)

maven-archetype-webapp, [71](#)

Jetty run Command, [73](#)

structure, [71, 72](#)