Maven by Example

Ed. 0.7

Contents

Preface

Maven is a build tool, a project management tool, an abstract container for running build tasks. It is a tool that has shown itself indispensable for projects that graduate beyond the simple and need to start finding consistent ways to manage and build large collections of interdependent modules and libraries which make use of tens or hundreds of third-party components. It is a tool that has removed much of the burden of third-party dependency management from the daily work schedule of millions of engineers, and it has enabled many organizations to evolve beyond the toil and struggle of build management into a new phase where the effort required to build and maintain software is no longer a limiting factor in software design.

This work is the first attempt at a comprehensive title on Maven. It builds upon the combined experience and work of the authors of all previous Maven titles, and you should view it not as a finished work but as the first edition in a long line of updates to follow. While Maven has been around for a few years, the authors of this book believe that it has just begun to deliver on the audacious promises it makes. The authors, and company behind this book, Sonatype, believe that the publishing of this book marks the beginning of a new phase of innovation and development surrounding Maven and the software ecosystem that surrounds it.

Acknowledgements

Sonatype would like to thank the following contributors. The people listed below have provided feedback which has helped improve the quality of this book. Thanks to Raymond Toal, Steve Daly, Paul Strack, Paul Reinerfelt, Chad Gorshing, Marcus Biel, Brian Dols, Mangalaganesh Balasubramanian, Marius Kruger, Chris Maki, Matthew McCollough, Matt Raible, and Mark Stewart. Special thanks to Joel Costigliola for helping to debug and correct the Spring web chapter. Stan Guillory was practically a contributing author given the number of corrections he posted to the book's Get Satisfaction. Thank you Stan. Special thanks to Richard Coasby of Bamboo for acting as the provisional grammar consultant.

Thanks to our contributing authors including Eric Redmond.

Thanks to the following contributors who reported errors or even contributed fixes: Paco Soberón, Ray Krueger, Steinar Cook, Henning Saul, Anders Hammar, “george_007”, “ksangani”, Niko Mahle, Arun Kumar, Harold Shinsato, “mimil”, “-thrawn-”, Matt Gumbley, Andrew Janke.

If you see your Get Satisfaction username in this list, and you would like it replaced with your real name, send an email to book@sonatype.com.

Special thanks to Grant Birchmeier for taking the time to proofread portions of the book and file extremely detailed feedback.

How to Contribute

The source code for this book can be found on the official GitHub repository and we accept pull requests with improvements.

Chapter 1

Introducing Apache Maven

This book is an introduction to Apache Maven which uses a set of examples to demonstrate core concepts. Starting with a simple Maven project which contains a single class and a single unit test, this book slowly develops an enterprise multi-module project which interacts with a database, interacts with a remote API, and presents a simple web application.

1.1 Maven... What is it?

The answer to this question depends on your own perspective. The great majority of Maven users are going to call Maven a “build tool”: a tool used to build deployable artifacts from source code. Build engineers and project managers might refer to Maven as something more comprehensive: a project management tool. What is the difference? A build tool such as Ant is focused solely on preprocessing, compilation, packaging, testing, and distribution. A project management tool such as Maven provides a superset of features found in a build tool. In addition to providing build capabilities, Maven can also run reports, generate a web site, and facilitate communication among members of a working team.

A more formal definition of Apache Maven: Maven is a project management tool which encompasses a project object model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle. When you use Maven, you describe your project using a well-defined project object model, Maven can then apply cross-cutting logic from a set of shared (or custom) plugins.

Don't let the fact that Maven is a "project management" tool scare you away. If you were just looking for a build tool, Maven will do the job. In fact, the first few chapters of this book will deal with the most common use case: using Maven to build and distribute your project.

1.2 Convention Over Configuration

Convention over configuration is a simple concept: Systems, libraries, and frameworks should assume reasonable defaults. Without requiring unnecessary configuration, systems should "just work". Popular frameworks such as Ruby on Rails and EJB3 have started to adhere to these principles in reaction to the configuration complexity of frameworks such as the initial EJB 2.1 specifications. An illustration of convention over configuration is something like EJB3 persistence: all you need to do to make a particular bean persistent is to annotate that class with @Entity. The framework assumes table and column names based on the name of the class and the names of the properties. Hooks are provided for you to override these default, assumed names if the need arises, but, in most cases, you will find that using the framework-supplied defaults results in a faster project execution.

Maven incorporates this concept by providing sensible default behavior for projects. Without customization, source code is assumed to be in `${basedir}/src/main/java` and resources are assumed to be in `${basedir}/src/main/resources`. Tests are assumed to be in `${basedir}/src/test`, and a project is assumed to produce a JAR file. Maven assumes that you want the compile byte-code to `${basedir}/target/classes` and then create a distributable JAR file in `${basedir}/target`. While this might seem trivial, consider the fact that most Ant-based builds have to define the locations of these directories. Ant doesn't ship with any built-in idea of where source code or resources might be in a project; you have to supply this information. Maven's adoption of convention over configuration goes farther than just simple directory locations. Maven's core plugins apply a common set of conventions for compiling source code, packaging distributions, generating web sites, and many other processes. Maven's strength comes from the fact that it is "opinionated"; it has a defined life-cycle and a set of common plugins that know how to build and assemble software. If you follow the conventions, Maven will require almost zero effort - just put your source in the correct directory, and Maven will take care of the rest.

One side effect of using systems that follow "convention over configuration" is that end-users might feel that they are forced to use a particular methodology or approach. While it is certainly true that Maven has some core opinions that shouldn't be challenged, most of the defaults can be customized. For example, the location of a project's source code and resources can be customized, names of JAR files can be customized, and through the development of custom plugins, almost any behavior can be tailored to your specific environment's requirements. If you don't care to follow convention, Maven will allow you to customize defaults in order to adapt to your specific requirements.

1.3 A Common Interface

Before Maven provided a common interface for building software, every single project had someone dedicated to managing a fully customized build system. Developers had to take time away from developing software to learn about the idiosyncrasies of each new project they wanted to contribute to. In 2001, you'd have a completely different approach to building a project like Turbine than you would to building a project like Tomcat. If a new source code analysis tool came out that would perform static analysis on source code, or if someone developed a new unit testing framework, everybody would have to drop what they were doing and figure out how to fit it into each project's custom build environment. How do you run unit tests? There were a thousand different answers. This environment was characterized by a thousand endless arguments about tools and build procedures. The age before Maven was an age of inefficiency, the age of the "Build Engineer".

Today, most open source developers have used or are currently using Maven to manage new software projects. This transition is less about developers moving from one build tool to another and more about developers starting to adopt a common interface for project builds. As software systems have become more modular, build systems have become more complex, and the number of projects has skyrocketed. Before Maven, when you wanted to check out a project like Apache ActiveMQ or Apache ServiceMix from Subversion and build it from source, you really had to set aside about an hour to figure out the build system for each particular project. What does the project need to build? What libraries do I need to download? Where do I put them? What goals can I execute in the build? In the best case, it took a few minutes to figure out a new project's build, and in the worst cases (like the old Servlet API implementation in the Jakarta Project), a project's build was so difficult it would take multiple hours just to get to the point where a new contributor could edit source and compile the project. These days, you check it out from source, and you run `mvn install`.

While Maven provides an array of benefits including dependency management and reuse of common build logic through plugins, the core reason why it has succeeded is that it has defined a common interface for building software. When you see that a project like Apache ActiveMQ uses Maven, you can assume that you'll be able to check it out from source and build it with `mvn install` without much hassle. You know where the ignition keys goes, you know that the gas pedal is on the right side, and the brake is on the left.

1.4 Universal Reuse through Maven Plugins

Plugins are more than just a trick to minimize the download size of the Maven distribution. Plugins add new behavior to your project's build. Maven retrieves both dependencies and plugins from the remote repository, allowing for universal reuse of build logic.

The Maven Surefire plugin is the plugin that is responsible for running unit tests. Somewhere between version 1.0 and the version that is in wide use today someone decided to add support for the TestNG unit testing framework in addition to the support for JUnit. This upgrade happened in a way that didn't break backwards compatibility. If you were using the Surefire plugin to compile and execute JUnit 3 unit tests, and you upgraded to the most recent version of the Surefire plugin, your tests continued to execute without fail. But, you gained new functionality; if you want to execute unit tests in TestNG you now have that ability. You also gained the ability to run annotated JUnit 4 unit tests. You gained all of these capabilities without having to upgrade your Maven installation or install new software. Most importantly, nothing about your project had to change aside from a version number for a plugin in a single Maven configuration file called the Project Object Model (POM).

It is this mechanism that affects much more than the Surefire plugin. Maven has plugins for everything from compiling Java code, to generating reports, to deploying to an application server. Maven has abstracted common build tasks into plugins which are maintained centrally and shared universally. If the state-of-the-art changes in any area of the build, if some new unit testing framework is released or if some new tool is made available, you don't have to be the one to hack your project's custom build system to support it. You benefit from the fact that plugins are downloaded from a remote repository and maintained centrally. This is what is meant by universal reuse through Maven plugins.

1.5 Conceptual Model of a “Project”

Maven maintains a model of a project. You are not just compiling source code into bytecode, you are developing a description of a software project and assigning a unique set of coordinates to a project. You are describing the attributes of the project. What is the project's license? Who develops and contributes to the project? What other projects does this project depend upon? Maven is more than just a “build tool”, it is more than just an improvement on tools like make and Ant, it is a platform that encompasses a new semantics related to software projects and software development. This definition of a model for every project enables such features as:

Dependency Management
: Because a project is defined by a unique set of coordinates consisting of a group identifier, an artifact identifier, and a version, projects can now use these coordinates to declare dependencies.

Remote Repositories
: Related to dependency management, we can use the coordinates defined in the Maven Project Object Model (POM) to create repositories of Maven artifacts.

Universal Reuse of Build Logic
: Plugins contain logic that works with the descriptive data and configuration parameters defined in Project Object Model (POM); they are not designed to operate upon specific files in known locations.

Tool Portability / Integration
Tools like Eclipse, NetBeans, and IntelliJ now have a common place to find information about a project. Before the advent of Maven, every IDE had a different way to store what was essentially a custom Project Object Model (POM). Maven has standardized this description, and while each IDE continues to maintain custom project files, they can be easily generated from the model.

Easy Searching and Filtering of Project Artifacts
Tools like Nexus allow you to index and search the contents of a repository using the information stored in the POM.

1.6 Is Maven an alternative to XYZ?

So, sure, Maven is an alternative to Ant, but Apache Ant continues to be a great, widely-used tool. It has been the reigning champion of Java builds for years, and you can integrate Ant build scripts with your project's Maven build very easily. This is a common usage pattern for a Maven project. On the other hand, as more and more open source projects move to Maven as a project management platform, working developers are starting to realize that Maven not only simplifies the task of build management, it is helping to encourage a common interface between developers and software projects. Maven is more of a platform than a tool, while you could consider Maven an alternative to Ant, you are comparing apples to oranges. “Maven” includes more than just a build tool.

This is the central point that makes all of the Maven vs. Ant, Maven vs. Buildr, Maven vs. Gradle arguments irrelevant. Maven isn't totally defined by the mechanics of your build system. It isn't about scripting the various tasks in your build as much as it is about encouraging a set of standards, a common interface, a life-cycle, a standard repository format, a standard directory layout, etc. It certainly isn't about what format the POM happens to be in (XML vs. YAML vs. Ruby). Maven is much larger than that, and Maven refers to much more than the tool itself. When this book talks of Maven, it is referring to the constellation of software, systems, and standards that support it. Buildr, Ivy, Gradle, all of these tools interact with the repository format that Maven helped create, and you could just as easily use a repository manager like Nexus to support a build written entirely in Ant.

While Maven is an alternative to many of these tools, the community needs to evolve beyond seeing technology as a zero-sum game between unfriendly competitors in a competition for users and developers. This might be how large corporations relate to one another, but it has very little relevance to the way that open source communities work. The headline “Who's winning? Ant or Maven?” isn't very constructive. If you force us to answer this question, we're definitely going to say that Maven is a superior alternative to Ant as a foundational technology for a build; at the same time, Maven's boundaries are constantly shifting and the Maven community is constantly trying to seek out new ways to become more ecumenical, more inter-operable, more cooperative. The core tenets of Maven are declarative builds, dependency management, repository managers, and universal reuse through plugins, but the specific incarnation of these ideas at any given moment is less important than the sense that the open source community is collaborating to reduce the inefficiency of “enterprise-scale builds”.

1.7 Comparing Maven with Ant

The authors of this book have no interest in creating a feud between Apache Ant and Apache Maven, but we are also cognizant of the fact that most organizations have to make a decision between the two standard solutions: Apache Ant and Apache Maven. In this section, we compare and contrast the tools.

Ant excels at build process; it is a build system modeled after `make` with targets and dependencies. Each target consists of a set of instructions which are coded in XML. There is a copy task and a javac task as well as a jar task. When you use Ant, you supply Ant with specific instructions for compiling and packaging your output. Look at the following example of a simple `build.xml` file:

A Simple Ant build.xml File

```
<project name="my-project" default="dist" basedir=".">
  <description>simple example build file</description>

  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist"  location="target"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Ouput into ${build} into a MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
```

```
        basedir="${build}"/>
  </target>

  <target name="clean"
          description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

In this simple Ant example, you can see how you have to tell Ant exactly what to do. There is a `compile` goal which includes the `javac` task that compiles the source in the `src/main/java` directory to the *target/classes* directory. You have to tell Ant exactly where your source is, where you want the resulting bytecode to be stored, and how to package this all into a JAR file. While there are some recent developments that help make Ant less procedural, a developer's experience with Ant is in coding a procedural language written in XML.

Contrast the previous Ant example with a Maven example. In Maven, to create a JAR file from some Java source, all you need to do is create a simple `pom.xml`, place your source code in `${basedir}/src/main/java` and then run `mvn install` from the command line. The example Maven `pom.xml` that achieves the same results as the simple Ant file listed in A Simple Ant build.xml File is shown in A Sample Maven pom.xml.

A Sample Maven pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```

That's all you need in your `pom.xml`. Running `mvn install` from the command line will process resources, compile source, execute unit tests, create a JAR, and install the JAR in a local repository for reuse in other projects. Without modification, you can run `mvn site` and then find an `index.html` file in `target/site` that contains links to JavaDoc and a few reports about your source code.

Admittedly, this is the simplest possible example project containing nothing more than some source code and producing a simple JAR. It is a project which closely follows Maven conventions and doesn't require any dependencies or customization. If we wanted to start customizing the behavior, our `pom.xml` is going to grow in size, and in the largest of projects you can see collections of very complex Maven POMs which contain a great deal of plugin customization and dependency declarations. But, even when your project's POM files become more substantial, they hold an entirely different kind of information from the build file of a similarly sized project using Ant. Maven POMs contain declarations: "This is a JAR project", and "The source code is in `src/main/java`". Ant build files contain explicit instructions: "This is project", "The source is in `src/main/java`", "Run `javac` against this directory", "Put the results in `target/classes`", "Create a JAR from the . . . ", etc. Where Ant had to be explicit about the process, there was something "built-in" to Maven that just knew where the source code was and how it should be processed.

The differences between Ant and Maven in this example are:

Apache Ant

- Ant doesn't have formal conventions like a common project directory structure or default behavior. You have to tell Ant exactly where to find the source and where to put the output. Informal conventions have emerged over time, but they haven't been codified into the product.
- Ant is procedural. You have to tell Ant exactly what to do and when to do it. You have to tell it to compile, then copy, then compress.
- Ant doesn't have a lifecycle. You have to define goals and goal dependencies. You have to attach a sequence of tasks to each goal manually.

Apache Maven

- Maven has conventions. It knows where your source code is because you followed the convention. Maven's Compiler plugin put the bytecode in `target/classes`, and it produces a JAR file in target.
- Maven is declarative. All you had to do was create a `pom.xml` file and put your source in the default directory. Maven took care of the rest.
- Maven has a lifecycle which was invoked when you executed `mvn install`. This command told Maven to execute a series of sequential lifecycle phases until it reached the install lifecycle phase. As a side effect of this journey through the lifecycle, Maven executed a number of default plugin goals which did things like compile and create a JAR.

Maven has built-in intelligence about common project tasks in the form of Maven plugins. If you wanted to write and execute unit tests, all you would need to do is write the tests, place them in `${basedir}/src/test/java`, add a test-scoped dependency on either TestNG or JUnit, and run `mvn test`. If you wanted to deploy a web application and not a JAR, all you would need to do is change your project type to `war` and put your docroot in `${basedir}/src/main/webapp`. Sure, you can do all of this with Ant, but you will be writing the instructions from scratch. In Ant, you would first have to figure out where the JUnit JAR file should be. Then you would have to create a classpath that includes the JUnit JAR file. Then you would tell Ant where it should look for test source code, write a goal that compiles the test source to bytecode, and execute the unit tests with JUnit.

Without supporting technologies like antlibs and Ivy (and even with these supporting technologies), Ant has the feeling of a custom procedural build. An efficient set of Maven POMs in a project which adheres to Maven's assumed conventions has surprisingly little XML compared to the Ant alternative. Another benefit of Maven is the reliance on widely-shared Maven plugins. Everyone uses the Maven Surefire plugin for unit testing, and if someone adds support for a new unit testing framework, you can gain new capabilities in your own build by just incrementing the version of a particular Maven plugin in your project's POM.

The decision to use Maven or Ant isn't a binary one, and Ant still has a place in a complex build. If your current build contains some highly customized process, or if you've written some Ant scripts to complete a specific process in a specific way that cannot be adapted to the Maven standards, you can still use these scripts with Maven. Ant is made available as a core Maven plugin. Custom Maven plugins can be implemented in Ant, and Maven projects can be configured to execute Ant scripts within the Maven project lifecycle.

Chapter 2

Installing Maven

The process of installing Apache Maven is very simple. This chapter covers it in detail. Your only prerequisite is an installed Java Development Kit (JDK). If you are just interested in installation, you can move on to the rest of the book after reading through Section 2.2 and Section 2.3. If you are interested in the details of your Maven installation, this entire chapter will give you an overview of what you've installed and the meaning of the Apache Software License, Version 2.0.

2.1 Verify your Java Installation

The latest version of Maven currently requires the usage of Java 7 or higher. While older Maven versions can run on older Java versions, this book assumes that you are running at least Java 7. Go with the most recent stable Java Development Kit (JDK) available for your operating system.

```
% java -version
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b14)
Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)
```

Tip
More details about Java version required for different Maven versions can be found on the Maven site.

Maven works with all certified Java™ compatible development kits, and a few non-certified implementations of Java. The examples in this book were written and tested against the official Java Development Kit releases downloaded from the Oracle web site.

2.2 Downloading Maven

You can download Apache Maven from the project website at http://maven.apache.org/download.html.

When downloading Maven, make sure you choose the latest version of Apache Maven from the Maven website. The latest version of Maven when this book was written was Maven 3.3.3. If you are not familiar with the Apache Software License, you should familiarize yourself with the terms of the license before you start using the product. More information on the Apache Software License can be found in Section 2.8.

2.3 Installing Maven

There are wide differences between operating systems such as Mac OS X and Microsoft Windows, and there are subtle differences between different versions of Windows. Luckily, the process of installing Maven on all of these operating systems is relatively painless and straightforward. The following sections outline the recommended best-practice for installing Maven on a variety of operating systems.

2.3.1 Installing Maven on Linux, BSD and Mac OS X

Download the current release of Maven from http://maven.apache.org/download.html. Choose a format that is convenient for you to work with. Pick an appropriate place for it to live, and expand the archive there. If you expanded the archive into the directory `/usr/local/apache-maven-3.0.5`, you may want to create a symbolic link to make it easier to work with and to avoid the need to change any environment configuration when you upgrade to a newer version:

```
/usr/local % cd /usr/local
/usr/local % ln -s apache-maven-3.0.5 maven
/usr/local % export PATH=/usr/local/maven/bin:$PATH
```

Once Maven is installed, you need to add its `bin` directory in the distribution (in this example, `/usr/local/maven/bin`) to your command path.

You'll need to add the `PATH` configuration to a script that will run every time you login. To do this, add the following lines to `.bash_login` or `.profile`.

```
export PATH=/usr/local/maven/bin:${PATH}
```

Once you've added these lines to your own environment, you will be able to run Maven from the command line.

Note
These installation instructions assume that you are running bash.

2.3.2 Installing Maven on Microsoft Windows

Installing Maven on Windows is very similar to installing Maven on Mac OS X, the main differences being the installation location and the setting of an environment variable. This book assumes a Maven installation directory of `C:\Program Files\apache-maven-3.0.5`, but it won't make a difference if you install Maven in another directory as long as you configure the proper environment variable. Once you've unpacked Maven to the installation directory, you will need to update the `PATH` environment variable:

```
C:\Users\tobrien > set PATH="c:\Program Files\apache-maven-3.0.5\bin";% ↩
    PATH%
```

Setting this environment variable on the command line will allow you to run Maven in your current session. Unless you add them to the System or User environment variables through the Control Panel, you'll have to execute these two lines every time you log into your system. You should modify both of these variables through the Control Panel in Microsoft Windows.

2.3.2.1 Setting Environment Variables

- Go into the `Control Panel`
- Select `System`

- Go in `Advanced` tab and click on `Environment Variables`.
- Click on the Path variable in the lower System variables section and click the Edit button.
- Add the string "`C:\Program Files\apache-maven-3.0.5\bin;`" in the Variable value field to the front of the existing value and click on the OK button in this and the following dialogs.

2.4 Testing a Maven Installation

Once Maven is installed, you can check the version by running `mvn -v` from the command line. If Maven has been installed, you should see something resembling the following output.

```
$ mvn -v
Apache Maven 3.0.5 (r01de14724cdef164cd33c7c8c2fe155faf9602da; 2013-02-19 ↩
    05:51:28-0800)
Maven home: /usr/local/maven
Java version: 1.7.0_75, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.7.0_75.jdk/Contents/Home ↩
    /jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.8.5", arch: "x86_64", family: "mac"
```

If you see this output, you know that Maven is available and ready to be used. If you do not see this output, and your operating system cannot find the `mvn` command, make sure that your `PATH` environment variable and `M2_HOME` environment variable have been properly set.

2.5 Maven Installation Details

Maven's download measures in at a few megabyte only. It has attained such a slim download size because the core of Maven has been designed to retrieve plugins and dependencies from a remote repository on-demand. When you start using Maven, it will start to download plugins to a local repository described in Section 2.5.1. In case you are curious, let's take a quick look at what is in Maven's installation directory.

```
/usr/local/maven $ ls -p1
LICENSE.txt
NOTICE
README.txt
bin/
boot/
```

```
conf/
lib/
```

`LICENSE.txt` contains the software license for Apache Maven. The `lib/` directory contains a the JAR files that contains the core of Maven.

Note

Unless you are working in a shared Unix environment, you should avoid customizing the `settings.xml` in `conf`. Altering the global `settings.xml` file in the Maven installation itself is usually unnecessary and it tends to complicate the upgrade procedure for Maven as you'll have to remember to copy the customized `settings.xml` from the old Maven installation to the new installation. If you need to customize `settings.xml`, you should be editing your own `settings.xml` in `~/.m2/settings.xml`.

2.5.1 User-Specific Configuration and Repository

Once you start using Maven extensively, you'll notice that Maven has created some local user-specific configuration files and a local repository in your home directory. In `~/.m2` there will be:

~/.m2/settings.xml
A file containing user-specific configuration for authentication, repositories, and other information to customize the behavior of Maven.

~/.m2/repository/
This directory contains your local Maven repository. When you download a dependency from a remote Maven repository, Maven stores a copy of the dependency in your local repository.

Note

In Unix (and OS X), your home directory will be referred to using a tilde (i.e. `~/bin` refers to `/home/tobrien/bin`). In Windows, we will also be using ~ to refer to your home directory. In Windows XP, your home directory is `C:\Documents and Settings\tobrien`, and in Windows Vista, your home directory is `C:\Users\tobrien`. From this point forward, you should translate paths such as `~/m2` to your operating system's equivalent.

2.5.2 Upgrading a Maven Installation

If you've installed Maven on a Mac OS X or Unix machine according to the details in Section 2.3.1, it should be easy to upgrade to newer versions of Maven when they become available. Simply install the newer version of Maven (*/usr/local/maven-3.future*) next to the existing version of Maven (*/usr/local/maven-3.0.3*). Then switch the symbolic link `/usr/local/maven` from `/usr/local/maven-3.0.3` to `/usr/local/maven-3.future`. Since you've already set your `PATH` variable to point to `/usr/local/maven`, you won't need to change any environment variables.

If you have installed Maven on a Windows machine, simply unpack Maven to `C:\Program Files\maven-3.future` and update your `PATH` variable.

Note
If you have any customizations to the global `settings.xml` in `conf`, you will need to copy this `settings.xml` to the `conf` directory of the new Maven installation.

2.6 Uninstalling Maven

Most of the installation instructions involve unpacking of the Maven distribution archive in a directory and setting of various environment variables. If you need to remove Maven from your computer, all you need to do is delete your Maven installation directory and remove the environment variables. You will also want to delete the `~/.m2` directory as it contains your local repository.

2.7 Getting Help with Maven

While this book aims to be a comprehensive reference, there are going to be topics we will miss and special situations and tips which are not covered. While the core of Maven is very simple, the real work in Maven happens in the plugins, and there are too many plugins available to cover them all in one book. You are going to encounter problems and features which have not been covered in this book; in these cases, we suggest searching for answers at the following locations:

http://maven.apache.org
This will be the first place to look. The Maven web site contains a wealth of information and

documentation. Every plugin has a few pages of documentation and there is a series of "quick start" documents which will be helpful in addition to the content of this book. While the Maven site contains a wealth of information, it can also be frustrating, confusing, and overwhelming. There is a custom Google search box on the main Maven page that will search known Maven sites for information. This provides better results than a generic Google search.

Maven User Mailing List
The Maven User mailing list is the place for users to ask questions. Before you ask a question on the user mailing list, you will want to search for any previous discussion that might relate to your question. It is bad form to ask a question that has already been asked without first checking to see if an answer already exists in the archives. There are a number of useful mailing list archive browsers; we've found Nabble to the be the most useful. You can browse the User mailing list archives at http://mail-archives.apache.org/mod_mbox/maven-users/. You can join the user mailing list by following the instructions available at http://maven.apache.org/mail-lists.html.

http://books.sonatype.com
Sonatype maintains an online copy of this book and other tutorials related to Apache Maven.

2.8 About the Apache Software License

Apache Maven is released under the Apache Software License, Version 2.0. If you want to read this license, you can read `${M2_HOME}/LICENSE.txt` or read this license on the Open Source Initiative's web site at http://www.opensource.org/licenses/apache2.0.php.

There's a good chance that, if you are reading this book, you are not a lawyer. If you are wondering what the Apache License, Version 2.0 means, the Apache Software Foundation has assembled a very helpful Frequently Asked Questions (FAQ) page about the license available at http://www.apache.org/-foundation/licence-FAQ.html.

Chapter 3

A Simple Maven Project

3.1 Introduction

In this chapter, we introduce a simple project created from scratch using the Maven Archetype plugin. This elementary application provides us with the opportunity to discuss some core Maven concepts while you follow along with the development of the project.

Before you can start using Maven for complex, multi-module builds, we have to start with the basics. If you've used Maven before, you'll notice that it does a good job of taking care of the details. Your builds tend to "just work," and you only really need to dive into the details of Maven when you want to customize the default behavior or write a custom plugin. However, when you do need to dive into the details, a thorough understanding of the core concepts is essential. This chapter aims to introduce you to the simplest possible Maven project and then presents some of the core concepts that make Maven a solid build platform. After reading it, you'll have a fundamental understanding of the build lifecycle, Maven repositories, dependency management, and the Project Object Model (POM).

3.1.1 Downloading this Chapter's Example

This chapter develops a very simple example which will be used to explore core concepts of Maven. If you follow the steps described in this chapter, you shouldn't need to download the examples to recreate the code produced by the Maven. We will be using the Maven Archetype plugin to create this simple project and this chapter doesn't modify the project in any way. If you would prefer to read this chapter with the final example source code, this chapter's example project may be downloaded with the book's example code at:

```
http://books.sonatype.com/mvnex-book/mvnex-examples.zip
```

Unzip this archive in any directory, and then go to the `ch-simple/` directory. There you will see a directory named `simple` that contains the source code for this chapter.

3.2 Creating a Simple Project

To start a new Maven project, use the Maven Archetype plugin from the command line. Run the `archetype:generate` goal, select default archetype suggested by pressing "Enter". This will use the archetype `org.apache.maven.archetypes:maven-archetype-quickstart`. Press "Enter" again to confirm the latest version of the archetype and then "Enter" to confirm the supplied parameters.

Warning
At the time of publication, the default `maven-archetype-quickstart` was item #312 in a list of 860 available archetypes. As more and more projects release Maven archetypes, this list will change and the number for the default archetype may change. When you run `archetype:generate` as shown below, the default `maven-archetype-quickstart` will be selected by default.

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook \
-DartifactId=simple \
-Dpackage=org.sonatype.mavenbook \
-Dversion=1.0-SNAPSHOT
[INFO]
[INFO] ------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) @ standalone- ↩
    pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone- ↩
    pom <<<
[INFO]
```

```
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone- ↩
    pom ---
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart (org.apache. ↩
    maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
...
312: remote -> org.apache.maven.archetypes:maven-archetype-quickstart (An  ↩
    archetype which contains a sample Maven project.)
Choose a number or apply filter (format: [groupId:]artifactId, case  ↩
    sensitive contains): 312:
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
Choose a number: 6:
[INFO] Using property: groupId = org.sonatype.mavenbook
[INFO] Using property: artifactId = simple
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = org.sonatype.mavenbook
Confirm properties configuration:
groupId: org.sonatype.mavenbook
artifactId: simple
version: 1.0-SNAPSHOT
package: org.sonatype.mavenbook
 Y: :
[INFO] ---------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x)  ↩
    Archetype: maven-archetype-quickstart:1.1
[INFO] ---------------------------------------------------------
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: artifactId, Value: simple
[INFO] Parameter: basedir, Value: /Volumes/mac-data/dev/github/sonatype
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /Volumes/mac-data/ ↩
    dev/github/sonatype/simple
[INFO] BUILD SUCCESS
...
```

`mvn` is the Maven command. `archetype:generate` is called a Maven goal. An archetype is defined as “an original model or type after which other similar things are patterned; a prototype.” A number of archetypes are available in Maven for anything from a simple application to a complex web application, and the `archetype:generate` offers a list of archetypes to choose from. In this chapter, we are going to use the most basic archetype to create a simple skeleton starter project. The plugin is the prefix `archetype`, and the goal is `generate`.

Once we've generated a project, take a look at the directory structure Maven created under the simple directory:

```
simple/❶
simple/pom.xml❷
/src/
/src/main/❸
/main/java
/src/test/❹
/test/java
```

This generated directory adheres to the Maven Standard Directory Layout. We'll get into more details later in this chapter, but for now, let's just try to understand these few basic directories:

1. The Maven Archetype plugin creates a directory `simple` that matches the `artifactId`. This is known as the project's base directory.
2. Every Maven project has what is known as a Project Object Model (POM) in a file named `pom.xml`. This file describes the project, configures plugins, and declares dependencies.
3. Our project's source code and resources are placed under `src/main`. In the case of our simple Java project this will consist of a few Java classes and some properties file. In another project, this could be the document root of a web application or configuration files for an application server. In a Java project, Java classes are placed in `src/main/java` and classpath resources are placed in `src/main/resources`.
4. Our project's test cases are located in `src/test`. Under this directory, Java classes such as JUnit or TestNG tests are placed in `src/test/java`, and classpath resources for tests are located in `src/test/resources`.

The Maven Archetype plugin generated a single class `org.sonatype.mavenbook.App`, which is a 13-line Java class with a static main function that prints out a message:

```
package org.sonatype.mavenbook;

/**
 * Hello world!
 *
 */
public class App
```

```
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

The simplest Maven archetype generates the simplest possible program: a program which prints "Hello World!" to standard output.

3.3 Building a Simple Project

The created directory `simple` contains the `pom.xml` and you can easily build the project:

```
$ cd simple
$ mvn install
[INFO] Scanning for projects...
[INFO] -----------------------------------------
[INFO] Building simple
[INFO]task-segment: [install]
[INFO] -----------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /simple/target/classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Compiling 1 source file to /simple/target/test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: /simple/target/surefire-reports
```

T E S T S

```
Running org.sonatype.mavenbook.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.105 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar]
```

```
[INFO] Building jar: /simple/target/simple-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing /simple/target/simple-1.0-SNAPSHOT.jar to \
~/.m2/repository/com/sonatype/maven/simple/simple/1.0-SNAPSHOT/ \
simple-1.0-SNAPSHOT.jar
```

You've just created, compiled, tested, packaged, and installed the simplest possible Maven project. To prove to yourself that this program works, run it from the command line.

```
$ java -cp target/simple-1.0-SNAPSHOT.jar org.sonatype.mavenbook.App
Hello World!
```

3.4 Simple Project Object Model

Simple Project's 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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook.simple</groupId>
    <artifactId>simple</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>simple</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>
</project>
```

This `pom.xml` file is the most basic POM you will ever deal with for a Maven project, usually a POM file is considerably more complex: defining multiple dependencies and customizing plugin behavior. The first few elements—groupId, artifactId, packaging, version—are what is known as the Maven coordinates which uniquely identify a project. name and url are descriptive elements of the POM providing a human readable name and associating the project with a web site. The dependencies element defines a single, test-scoped dependency on a unit testing framework called JUnit. These topics will be further introduced in Section 3.5, all you need to know, at this point, is that the pom.xml is the file that makes Maven go.

Maven always executes against an effective POM, a combination of settings from this project's pom.xml, all parent POMs, a super-POM defined within Maven, user-defined settings, and active profiles. All projects ultimately extend the super-POM, which defines a set of sensible default configuration settings. While your project might have a relatively minimal pom.xml, the contents of your project's POM are interpolated with the contents of all parent POMs, user settings, and any active profiles. To see this "effective" POM, run the following command in the simple project's base directory.

```
$ mvn help:effective-pom
```

When you run this, you should see a much larger POM which exposes the default settings of Maven. This goal can come in handy if you are trying to debug a build and want to see how all of the current project's ancestor POMs are contributing to the effective POM.

3.5 Core Concepts

Having just run Maven for the first time, it is a good time to introduce a few of the core concepts of Maven. In the previous example, you generated a project which consisted of a POM and some code assembled in the Maven standard directory layout. You then executed Maven with a lifecycle phase as an argument, which prompted Maven to execute a series of Maven plugin goals. Lastly, you installed a Maven artifact into your local repository. Wait? What is a "lifecycle"? What is a "local repository"? The following section defines some of Maven's central concepts.

3.5.1 Maven Plugins and Goals

To execute a single Maven plugin goal, we used the syntax `mvn archetype:generate`, where `archetype` is the identifier of a plugin and `generate` is the identifier of a goal. When Maven executes a plugin goal, it prints out the plugin identifier and goal identifier to standard output:

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.simple
...
[INFO] [archetype:generate]
...
```

A Maven Plugin is a collection of one or more goals. Examples of Maven plugins can be simple core plugins like the Jar plugin, which contains goals for creating JAR files, Compiler plugin, which contains goals for compiling source code and unit tests, or the Surefire plugin, which contains goals for executing unit tests and generating reports. Other, more specialized Maven plugins include plugins like the Hibernate3 plugin for integration with the popular persistence library Hibernate, the JRuby plugin which allows you to execute ruby as part of a Maven build or to write Maven plugins in Ruby. Maven also provides for the ability to define custom plugins. A custom plugin can be written in Java, or a plugin can be written in any number of languages including Ant, Groovy, beanshell, and, as previously mentioned, Ruby.

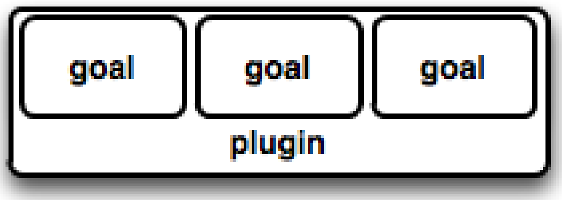

Figure 3.1: A Plugin Contains Goals

A goal is a specific task that may be executed as a standalone goal or along with other goals as part of a larger build. A goal is a “unit of work” in Maven. Examples of goals include the `compile` goal in the Compiler plugin, which compiles all of the source code for a project, or the `test` goal of the Surefire plugin, which can execute unit tests. Goals are configured via configuration properties that can be used to customize behavior. For example, the `compile` goal of the Compiler plugin defines a set of configuration parameters. When running the `archetype:generate` goal earlier in Section 3.2 we passed the `package` parameter to the `generate` goal as `org.sonatype.mavenbook`. If we had omitted the `package` parameter, the package name would have defaulted to `org.sonatype.mavenbook.simple`.

> **Note**
> When referring to a plugin goal, we frequently use the shorthand notation: pluginId:goalId. For example, when referring to the `generate` goal in the Archetype plugin, we write `archetype:generate`.

Goals define parameters that can define sensible default values. In the `archetype:generate` example, we did not specify what kind of archetype the goal was to create on our command line; we simply passed in a `groupId` and an `artifactId`. Not passing in the type of artifact we wanted to create caused the `generate` goal to prompt us for input, the generate goal stopped and asked us to choose an archetype from a list. If you had run the `archetype:create` goal instead, Maven would have assumed that you wanted to generate a new project using the default `maven-archetype-quickstart` archetype. This is our first brush with convention over configuration. The convention, or default, for the `create` goal is to create a simple project called Quickstart. The `create` goal defines a configuration property `archetypeArtifactId` that has a default value of `maven-archetype-quickstart`. The Quickstart archetype generates a minimal project shell that contains a POM and a single class. The Archetype plugin is far more powerful than this first example suggests, but it is a great way to get new projects started fast. Later in this book, we'll show you how the Archetype plugin can be used to generate more complex projects such as web applications, and how you can use the Archetype plugin to define your own set of projects.

The core of Maven has little to do with the specific tasks involved in your project's build. By itself, Maven doesn't know how to compile your code or even how to make a JAR file. It delegates all of this work to Maven plugins like the Compiler plugin and the Jar plugin, which are downloaded on an as-needed basis and periodically updated from the central Maven repository. When you download Maven, you are getting the core of Maven, which consists of a very basic shell that knows only how to parse the command line, manage a classpath, parse a POM file, and download Maven plugins as needed. By keeping the Compiler plugin separate from Maven's core and providing for an update mechanism, Maven makes it easier for users to have access to the latest options in the compiler. In this way, Maven plugins allow for universal reusability of common build logic. You are not defining the compile task in a build file; you are using a Compiler plugin that is shared by every user of Maven. If there is an improvement to the Compiler plugin, every project that uses Maven can immediately benefit from this change. (And, if you don't like the Compiler plugin, you can override it with your own implementation.)

3.5.2 Maven Lifecycle

The second command we ran in the previous section included an execution of the Maven lifecycle. It begins with a phase to validate the basic integrity of the project and ends with a phase that involves deploying a project to production. Lifecycle phases are intentionally vague, defined solely as validation, testing, or deployment, and they may mean different things to different projects. For example, in a project that produces a Java archive, the `package` phase produces a JAR; in a project that produces a web application, the `package` phase produces a WAR.

Plugin goals can be attached to a lifecycle phase. As Maven moves through the phases in a lifecycle, it will execute the goals attached to each particular phase. Each phase may have zero or more goals bound to it. In the previous section, when you ran `mvn install`, you might have noticed that more than one goal was executed. Examine the output after running `mvn install` and take note of the various goals that are executed. When this simple example reached the `package` phase, it executed the `jar` goal in the Jar plugin. Since our simple Quickstart project has (by default) a `jar` packaging type, the `jar:jar` goal is bound to the `package` phase.

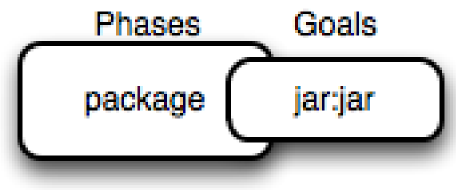

Figure 3.2: A Goal Binds to a Phase

We know that the `package` phase is going to create a JAR file for a project with `jar` packaging. But what of the goals preceding it, such as `compiler:compile` and `surefire:test`? These goals are executed as Maven steps in the phases preceding `package` in the Maven lifecycle.

`resources:resources`
: plugin is bound to the `process-resources` phase. This goal copies all of the resources from `src/main/resources` and any other configured resource directories to the output directory.

`compiler:compile`
: is bound to the `compile` phase. This goal compiles all of the source code from `src/main/java` or any other configured source directories to the output directory.

`resources:testResources`
: plugin is bound to the `process-test-resources` phase. This goal copies all of the resources from `src/test/resources` and any other configured test resource directories to a test output directory.

`compiler:testCompile`
: plugin is bound to the `test-compile` phase. This goal compiles test cases from `src/test/java` and any other configured test source directories to a test output directory.

`surefire:test`
: bound to the `test` phase. This goal executes all of the tests and creates output files that capture detailed results. By default, this goal will terminate a build if there is a test failure.

`jar:jar`
: to the `package` phase. This goal packages the output directory into a JAR file.

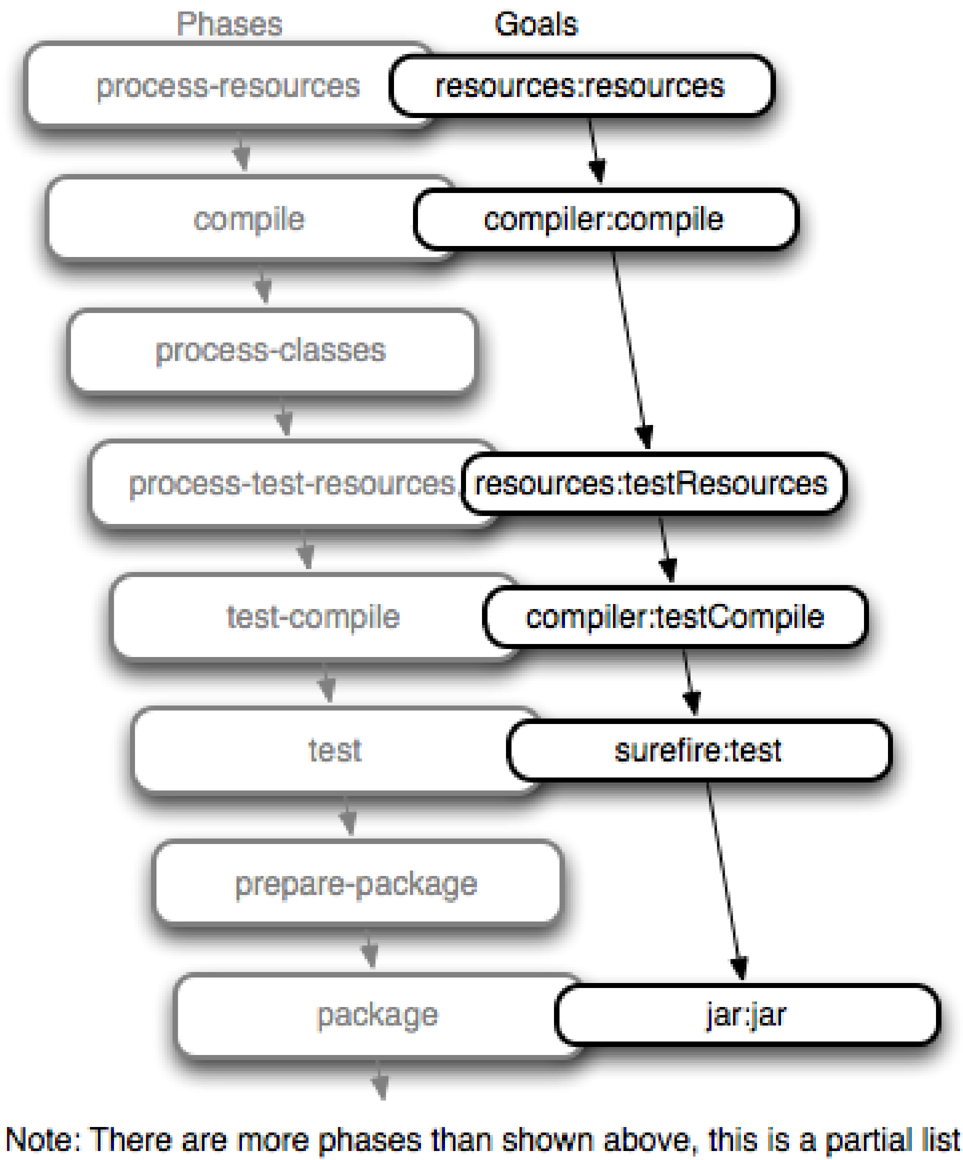

Figure 3.3: Bound Goals are Run when Phases Execute

To summarize, when we executed `mvn install`, Maven executes all phases up to the install phase, and in the process of stepping through the lifecycle phases it executes all goals bound to each phase. Instead of executing a Maven lifecycle goal you could achieve the same results by specifying a sequence of plugin goals as follows:

```
mvn resources:resources \
    compiler:compile \
    resources:testResources \
    compiler:testCompile \
    surefire:test \
    jar:jar \
    install:install
```

It is much easier to execute lifecycle phases than it is to specify explicit goals on the command line, and the common lifecycle allows every project that uses Maven to adhere to a well-defined set of standards. The lifecycle is what allows a developer to jump from one Maven project to another without having to know very much about the details of each particular project's build. If you can build one Maven project, you can build them all.

3.5.3 Maven Coordinates

The Archetype plugin created a project with a file named `pom.xml`. This is the Project Object Model (POM), a declarative description of a project. When Maven executes a goal, each goal has access to the information defined in a project's POM. When the `jar:jar` goal needs to create a JAR file, it looks to the POM to find out what the JAR file's name is. When the `compiler:compile` goal compiles Java source code into bytecode, it looks to the POM to see if there are any parameters for the compile goal. Goals execute in the context of a POM. Goals are actions we wish to take upon a project, and a project is defined by a POM. The POM names the project, provides a set of unique identifiers (coordinates) for a project, and defines the relationships between this project and others through dependencies, parents, and prerequisites. A POM can also customize plugin behavior and supply information about the community and developers involved in a project.

Maven coordinates define a set of identifiers which can be used to uniquely identify a project, a dependency, or a plugin in a Maven POM. Take a look at the following POM.

```
<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>mavenbook</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</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>
</project>
```

coordinates

Figure 3.4: A Maven Project's Coordinates

We've highlighted the Maven coordinates for this project: the `groupId`, `artifactId`, `version` and `packaging`. These combined identifiers make up a project's coordinates. There is a fifth, seldom-used coordinate named `classifier` which we will introduce later in the book. You can feel free to ignore classifiers for now. Just like in any other coordinate system, a set of Maven coordinates is an address for a specific point in "space". Maven pinpoints a project via its coordinates when one project relates to another, either as a dependency, a plugin, or a parent project reference. Maven coordinates are often written using a colon as a delimiter in the following format: `groupId:artifactId:packaging:version`. In the above `pom.xml` file for our current project, its coordinates are represented as `mavenbook:my-app:jar:1.0-SNAPSHOT`.

`groupId`
: The group, company, team, organization, project, or other group. The convention for group identifiers is that they begin with the reverse domain name of the organization that creates the project. Projects from Sonatype would have a `groupId` that begins with `com.sonatype`, and projects in the Apache Software Foundation would have a `groupId` that starts with `org.apache`.

artifactId
: A unique identifier under `groupId` that represents a single project.

version
: A specific release of a project. Projects that have been released have a fixed version identifier that refers to a specific version of the project. Projects undergoing active development can use a special identifier that marks a version as a `SNAPSHOT`.

The packaging format of a project is also an important component in the Maven coordinates, but it isn't a part of a project's unique identifier. A project's `groupId:artifactId:version` make that project unique; you can't have a project with the same three `groupId`, `artifactId`, and `version` identifiers.

packaging
: The type of project, defaulting to `jar`, describing the packaged output produced by a project. A project with packaging `jar` produces a JAR archive; a project with packaging `war` produces a web application.

These four elements become the key to locating and using one particular project in the vast space of other "Mavenized" projects . Maven repositories (public, private, and local) are organized according to these identifiers. When this project is installed into the local Maven repository, it immediately becomes locally available to any other project that wishes to use it. All you must do is add it as a dependency of another project using the unique Maven coordinates for a specific artifact.

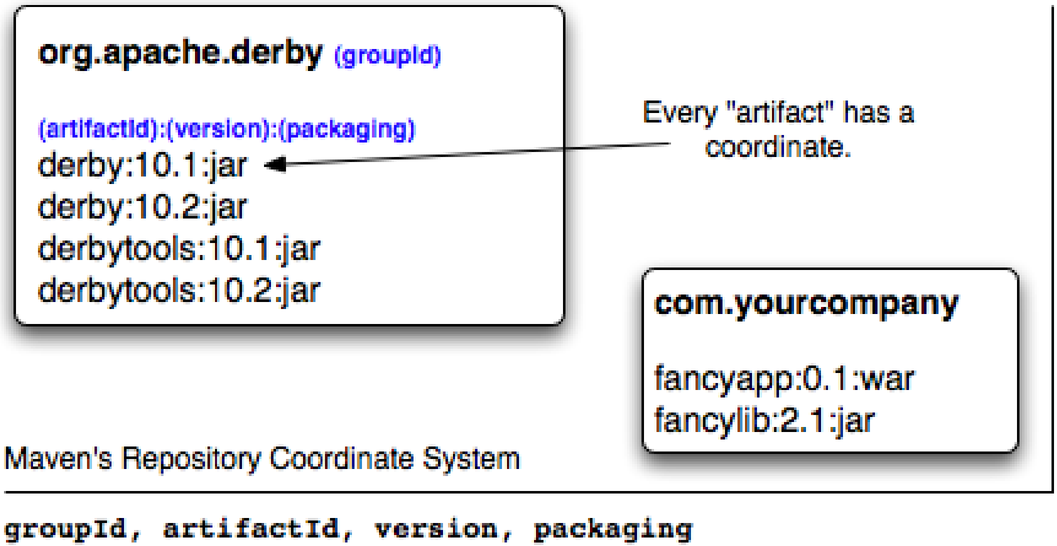

Figure 3.5: Maven Space is a Coordinate System of Projects

3.5.4 Maven Repositories

When you run Maven for the first time, you will notice that Maven downloads a number of files from a remote Maven repository. If the simple project was the first time you ran Maven, the first thing it will do is download the latest release of the Resources plugin when it triggers the `resources:resource` goal. In Maven, artifacts and plugins are retrieved from a remote repository when they are needed. One of the reasons the initial Maven download is so small (1.5 MiB) is due to the fact that Maven doesn't ship with much in the way of plugins. Maven ships with the bare minimum and fetches from a remote repository when it needs to. Maven ships with a default remote repository location (http://repo1.maven.org/maven2) which it uses to download the core Maven plugins and dependencies.

Often you will be writing a project which depends on libraries that are neither free nor publicly distributed. In this case you will need to either setup a custom repository inside your organization's network or download and install the dependencies manually. The default remote repositories can be replaced or augmented with references to custom Maven repositories maintained by your organization. There are multiple products available to allow organizations to manage and maintain mirrors of the public Maven repositories.

What makes a Maven repository a Maven repository? A repository is a collection of project artifacts stored in a directory structure that closely matches a project's Maven coordinates. You can see this structure by opening up a web browser and browsing the central Maven repository at http://repo1.maven.org/maven2/. You will see that an artifact with the coordinates `org.apache.commons:commons-email:1.1` is available under the directory `/org/apache/commons/commons-email/1.1/` in a file named `commons-email-1.1.jar`. The standard for a Maven repository is to store an artifact in the following directory relative to the root of the repository:

```
/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
```

Maven downloads artifacts and plugins from a remote repository to your local machine and stores these artifacts in your local Maven repository. Once Maven has downloaded an artifact from the remote Maven repository it never needs to download that artifact again as Maven will always look for the artifact in the local repository before looking elsewhere. On Windows XP, your local repository is likely in `C:\Documents and Settings\USERNAME\.m2\repository`, and on Windows Vista, your local repository is in `C:\Users\USERNAME\.m2\repository`. On Unix systems, your local Maven repository is available in `~/.m2/repository`. When you build a project like the simple project you created in the previous section, the `install` phase executes a goal which installs your project's artifacts in your local Maven repository.

In your local repository, you should be able to see the artifact created by our simple project. If you run the `mvn install` command, Maven will install our project's artifact in your local repository. Try it.

```
$ mvn install
...
[INFO] [install:install]
[INFO] Installing .../simple-1.0-SNAPSHOT.jar to \
~/.m2/repository/com/sonatype/maven/simple/1.0-SNAPSHOT/ \
simple-1.0-SNAPSHOT.jar
...
```

As you can see from the output of this command, Maven installed our project's JAR file into our local Maven repository. Maven uses the local repository to share dependencies across local projects. If you develop two projects—project A and project B—with project B depending on the artifact produced by project A, Maven will retrieve project A's artifact from your local repository when it is building project B. Maven repositories are both a local cache of artifacts downloaded from a remote repository and a mechanism for allowing your projects to depend on each other.

3.5.5 Maven's Dependency Management

In this chapter's simple example, Maven resolved the coordinates of the JUnit dependency `junit:junit:3.8.1` to a path in a Maven repository `/junit/junit/3.8.1/junit-3.8.1.jar`. The ability to locate an artifact in a repository based on Maven coordinates gives us the ability to define dependencies in a project's POM. If you examine the simple project's `pom.xml` file, you will see that there is a section which deals with dependencies, and that this section contains a single dependency—JUnit.

A more complex project would contain more than one dependency, or it might contain dependencies that depend on other artifacts. Support for transitive dependencies is one of Maven's most powerful features. Let's say your project depends on a library that, in turn, depends on 5 or 10 other libraries (Spring or Hibernate, for example). Instead of having to track down all of these dependencies and list them in your `pom.xml` explicitly, you can simply depend on the library you are interested in and Maven will add the dependencies of this library to your project's dependencies implicitly. Maven will also take care of working out conflicts between dependencies, and provides you with the ability to customize the default behavior and exclude certain transitive dependencies.

Let's take a look at a dependency which was downloaded to your local repository when you ran the previous example. Look in your local repository path under `~/.m2/repository/junit/junit/3.8.1/`. If you have been following this chapter's examples, there will be a file named `junit-3.8.1.jar` and a `junit-3.8.1.pom` file in addition to a few checksum files which Maven uses to verify the authenticity of a downloaded artifact. Note that Maven doesn't just download the JUnit JAR file, Maven also downloads a POM file for the JUnit dependency. The fact that Maven downloads POM files in addition to artifacts is central to Maven's support for transitive dependencies.

When you install your project's artifact in the local repository, you will also notice that Maven publishes a slightly modified version of the project's `pom.xml` file in the same directory as the JAR file. Storing a POM file in the repository gives other projects information about this project, most importantly what dependencies it has. If Project B depends on Project A, it also depends on Project A's dependencies. When Maven resolves a dependency artifact from a set of Maven coordinates, it also retrieves the POM and consults the dependencies POM to find any transitive dependencies. These transitive dependencies are then added as dependencies of the current project.

A dependency in Maven isn't just a JAR file; it's a POM file that, in turn, may declare dependencies on other artifacts. These dependencies of dependencies are called transitive dependencies, and they are made possible by the fact that the Maven repository stores more than just bytecode; it stores metadata about artifacts.

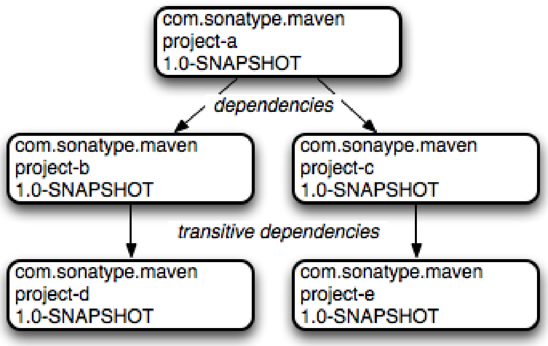

Figure 3.6: Maven Resolves Transitive Dependencies

In the previous figure, project A depends on projects B and C. Project B depends on project D, and project C depends on project E. The full set of direct and transitive dependencies for project A would be projects B, C, D, and E, but all project A had to do was define a dependency on B and C. Transitive dependencies can come in handy when your project relies on other projects with several small dependencies (like Hibernate, Apache Struts, or the Spring Framework). Maven also provides you with the ability to exclude transitive dependencies from being included in a project's classpath.

Maven also provides for different dependency scopes. The simple project's `pom.xml` contains a single dependency — `junit:junit:jar:3.8.1` — with a scope of `test`. When a dependency has a scope of `test`, it will not be available to the `compile` goal of the Compiler plugin. It will be added to the classpath for only the `compiler:testCompile` and `surefire:test` goals.

When you create a JAR for a project, dependencies are not bundled with the generated artifact; they are used only for compilation. When you use Maven to create a WAR or an EAR file, you can configure Maven to bundle dependencies with the generated artifact, and you can also configure it to exclude certain dependencies from the WAR file using the `provided` scope. The `provided` scope tells Maven that a dependency is needed for compilation, but should not be bundled with the output of a build. This scope comes in handy when you are developing a web application. You'll need to compile your code against the Servlet specification, but you don't want to include the Servlet API JAR in your web application's `WEB-INF/lib` directory.

3.5.6 Site Generation and Reporting

Another important feature of Maven is its ability to generate documentation and reports. In your simple project's directory, execute the following command:

```
$ mvn site
```

This will execute the `site` lifecycle phase. Unlike the default build lifecycle that manages generation of code, manipulation of resources, compilation, packaging, etc., this lifecycle is concerned solely with processing site content under the `src/site` directories and generating reports. After this command executes, you should see a project web site in the `target/site` directory. Load `target/site/index.html` and you should see a basic shell of a project site. This shell contains some reports under "Project Reports" in the lefthand navigation menu, and it also contains information about the project, the dependencies, and developers associated with it under "Project Information." The simple project's web site is mostly empty, since the POM contains very little information about itself beyond its Maven coordinates, a name, a URL, and a single test dependency.

On this site, you'll notice that some default reports are available. A unit test report communicates the success and failure of all unit tests in the project. Another report generates Javadoc for the project's API. Maven provides a full range of configurable reports, such as the Clover report that examines unit test coverage, the JXR report that generates cross-referenced HTML source code listings useful for code reviews, the PMD report that analyzes source code for various coding problems, and the JDepend report that analyzes the dependencies between packages in a codebase. You can customize site reports by configuring which reports are included in a build via the `pom.xml` file.

3.6 Summary

In this chapter, we have created a simple project, packaged the project into a JAR file, installed that JAR into the Maven repository for use by other projects, and generated a site with documentation. We accomplished this without writing a single line of code or touching a single configuration file. We also took some time to develop definitions for some of the core concepts of Maven. In the next chapter, we'll start customizing and modifying our project `pom.xml` file to add dependencies and configure unit tests.

Chapter 4

Customizing a Maven Project

4.1 Introduction

This chapter expands on the information introduced in Chapter 3. We're going to create a simple project generated with the Maven Archetype plugin, add some dependencies, add some source code, and customize the project to suit our needs. By the end of this chapter, you will know how to start using Maven to create real projects.

4.1.1 Downloading this Chapter's Example

We'll be developing a useful program that interacts with a Yahoo Weather web service. Although you should be able to follow along with this chapter without the example source code, we recommend that you download a copy of the code to use as a reference. This chapter's example project may be downloaded with the book's example code at:

```
http://books.sonatype.com/mvnex-book/mvnex-examples.zip
```

Unzip this archive in any directory, and then go to the `ch-custom/` directory. There you will see a directory named `simple-weather/`, which contains the Maven project developed in this chapter.

4.2 Defining the Simple Weather Project

Before we start customizing this project, let's take a step back and talk about the Simple Weather project. What is it? It's a contrived example, created to demonstrate some of the features of Maven. It is an application that is representative of the kind you might need to build. The Simple Weather application is a basic command-line-driven application that takes a zip code and retrieves some data from the Yahoo Weather RSS feed. It then parses the result and prints the result to standard output.

We chose this example for a number of reasons. First, it is straightforward. A user supplies input via the command line, the app takes that zip code, makes a request to Yahoo Weather, parses the result, and formats some simple data to the screen. This example is a simple `main()` function and some supporting classes; there is no enterprise framework to introduce and explain, just XML parsing and some logging statements. Second, it gives us a good excuse to introduce some interesting libraries such as Velocity, Dom4J, and Log4J. Although this book is focused on Maven, we won't shy away from an opportunity to introduce interesting utilities. Lastly, it is an example that can be introduced, developed, and deployed in a single chapter.

4.2.1 Yahoo Weather RSS

Before you build this application, you should know something about the Yahoo Weather RSS feed. To start with, the service is made available under the following terms:

```
The feeds are provided free of charge for use by individuals and
nonprofit organizations for personal, noncommercial uses. We ask that
you provide attribution to Yahoo Weather in connection with your use
of the feeds.
```

In other words, if you are thinking of integrating these feeds into your commercial web site, think again—this feed is for personal, noncommercial use. The use we're encouraging in this chapter is personal educational use. For more information about these terms of service, see the Yahoo Weather! API documentation here: http://developer.yahoo.com/weather/.

4.3 Creating the Simple Weather Project

First, let's use the Maven Archetype plugin to create a basic skeleton for the Simple Weather project. Execute the following command to create a new project, press enter to use the default `maven-archet` ype-quickstart and the latest version of the archetype, and then enter "Y" to confirm and generate the new project. Note that the number for the archetype will be different on your execution:

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.custom \
      -DartifactId=simple-weather \
      -Dversion=1.0

[INFO] Preparing archetype:generate
...
[INFO] [archetype:generate {execution: default-cli}]
[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:
...
16: internal -> maven-archetype-quickstart ()
...
Choose a number:  (...) 16: : 16
Confirm properties configuration:
groupId: org.sonatype.mavenbook.custom
artifactId: simple-weather
version: 1.0
package: org.sonatype.mavenbook.custom
Y: : Y
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.custom
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook.custom
[INFO] Parameter: package, Value: org.sonatype.mavenbook.custom
[INFO] Parameter: artifactId, Value: simple-weather
[INFO] Parameter: basedir, Value: /private/tmp
[INFO] Parameter: version, Value: 1.0
[INFO] BUILD SUCCESSFUL
```

Once the Maven Archetype plugin creates the project, go into the simple-weather directory and take a look at the pom.xml file. You should see the XML document that's shown in Initial POM for the simple-weather Project.

Initial POM for the simple-weather Project

```
<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>org.sonatype.mavenbook.custom</groupId>
    <artifactId>simple-weather</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>
    <name>simple-weather</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>
</project>
```

Next, you will need to configure the Maven Compiler plugin to target Java 5. To do this, add the `build` element to the initial POM as shown in POM for the simple-weather Project with Compiler Configuration.

POM for the simple-weather Project with Compiler 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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook.custom</groupId>
    <artifactId>simple-weather</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>
    <name>simple-weather</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

```
</project>
```

Notice that we passed in the `version` parameter to the `archetype:generate` goal. This overrides the default value of `1.0-SNAPSHOT`. In this project, we're developing the `1.0` version of the `simple-weather` project as you can see in the `pom.xml` `version` element.

4.4 Customize Project Information

Before we start writing code, let's customize the project information a bit. We want to add some information about the project's license, the organization, and a few of the developers associated with the project. This is all standard information you would expect to see in most projects. Adding Organizational, Legal, and Developer Information to the pom.xml shows the XML that supplies the organizational information, the licensing information, and the developer information.

Adding Organizational, Legal, and Developer Information to the pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    ...

    <name>simple-weather</name>
    <url>http://www.sonatype.com</url>

    <licenses>
        <license>
            <name>Apache 2</name>
            <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
            <distribution>repo</distribution>
            <comments>A business-friendly OSS license</comments>
        </license>
    </licenses>

    <organization>
        <name>Sonatype</name>
        <url>http://www.sonatype.com</url>
    </organization>

    <developers>
        <developer>
            <id>jason</id>
```

```
            <name>Jason Van Zyl</name>
            <email>jason@maven.org</email>
            <url>http://www.sonatype.com</url>
            <organization>Sonatype</organization>
            <organizationUrl>http://www.sonatype.com</organizationUrl>
            <roles>
                <role>developer</role>
            </roles>
            <timezone>-6</timezone>
        </developer>
    </developers>
    ...
</project>
```

The ellipses in Adding Organizational, Legal, and Developer Information to the pom.xml are shorthand for an abbreviated listing. When you see a `pom.xml` with “...” and “...” directly after the `project` element’s start tag and directly before the `project` element’s end tag, this implies that we are not showing the entire `pom.xml` file. In this case the `licenses`, `organization`, and `developers` elements were all added before the `dependencies` element.

4.5 Add New Dependencies

The Simple Weather application is going to have to complete the following three tasks: retrieve XML data from Yahoo Weather, parse the XML from Yahoo, and then print formatted output to standard output. To accomplish these tasks, we have to introduce some new dependencies to our project’s `pom.xml`. To parse the XML response from Yahoo, we’re going to be using Dom4J and Jaxen, to format the output of this command-line program we are going to be using Velocity, and we will also need to add a dependency for Log4J which we will be using for logging. After we add these dependencies, our `dependencies` element will look like the following example.

Adding Dom4J, Jaxen, Velocity, and Log4J as Dependencies

```
<project>
    [...]
    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
        <dependency>
            <groupId>dom4j</groupId>
```

```
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>velocity</groupId>
            <artifactId>velocity</artifactId>
            <version>1.5</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    [...]
</project>
```

As you can see above, we've added four more `dependency` elements in addition to the existing element which was referencing the `test` scoped dependency on JUnit. If you add these dependencies to the project's `pom.xml` file and then run `mvn install`, you will see Maven downloading all of these dependencies and other transitive dependencies to your local Maven repository.

How did we find these dependencies? Did we just "know" the appropriate `groupId` and `artifactId` values? Some of the dependencies are so widely used (like Log4J) that you'll just remember what the `groupId` and `artifactId` are every time you need to use them. Velocity, Dom4J, and Jaxen were all located using the searching capability on http://repository.sonatype.org. This is a public Sonatype Nexus instance which provides a search interface to various public Maven repositories, you can use it to search for dependencies. To test this for yourself, load http://repository.sonatype.org and search for some commonly used libraries such as Hibernate or the Spring Framework. When you search for an artifact on this site, it will show you an `artifactId` and all of the versions known to the central Maven repository. Clicking on the details for a specific version will load a page that contains the dependency element you'll need to copy and paste into your own project's `pom.xml`. If you need to find a dependency, you'll want to check out repository.sonatype.org, as you'll often find that certain libraries have more than one `groupId`. With this tool, you can make sense of the Maven repository.

4.6 Simple Weather Source Code

The Simple Weather command-line application consists of five Java classes.

org.sonatype.mavenbook.weather.Main
: The `Main` class contains a static `main()` method: the entry point for this system.

org.sonatype.mavenbook.weather.Weather
: The `Weather` class is a straightforward Java bean that holds the location of our weather report and some key facts, such as the temperature and humidity.

org.sonatype.mavenbook.weather.YahooRetriever
: The `YahooRetriever` class connects to Yahoo Weather and returns an `InputStream` of the data from the feed.

org.sonatype.mavenbook.weather.YahooParser
: The `YahooParser` class parses the XML from Yahoo Weather and returns a `Weather` object.

org.sonatype.mavenbook.weather.WeatherFormatter
: The `WeatherFormatter` class takes a `Weather` object, creates a `VelocityContext`, and evaluates a Velocity template.

Although we won't dwell on the code here, we will provide all the necessary code for you to get the example working. We assume that most readers have downloaded the examples that accompany this book, but we're also mindful of those who may wish to follow the example in this chapter step-by-step. The sections that follow list classes in the `simple-weather` project. Each of these classes should be placed in the same package: `org.sonatype.mavenbook.weather`.

Let's remove the `App` and the `AppTest` classes created by `archetype:generate` and add our new package. In a Maven project, all of a project's source code is stored in `src/main/java`. From the base directory of the new project, execute the following commands:

```
$ cd src/test/java/org/sonatype/mavenbook/custom
$ rm AppTest.java
$ cd ../../../../../../..
$ cd src/main/java/org/sonatype/mavenbook/custom
$ cd ..
$ rm App.java
$ mkdir weather
$ cd weather
```

This creates a new package named `org.sonatype.mavenbook.weather`. Now we need to put some classes in this directory. Using your favorite text editor, create a new file named `Weather.java` with the contents shown in Simple Weather's Weather Model Object.

Simple Weather's Weather Model Object

```
package org.sonatype.mavenbook.weather;

public class Weather {
  private String city;
  private String region;
  private String country;
  private String condition;
  private String temp;
  private String chill;
  private String humidity;

  public Weather() {}

  public String getCity() { return city; }
  public void setCity(String city) {
    this.city = city;
  }

  public String getRegion() { return region; }
  public void setRegion(String region) {
    this.region = region;
  }

  public String getCountry() { return country; }
  public void setCountry(String country) {
    this.country = country;
  }

  public String getCondition() { return condition; }
  public void setCondition(String condition) {
    this.condition = condition;
  }

  public String getTemp() { return temp; }
  public void setTemp(String temp) {
    this.temp = temp;
  }

  public String getChill() { return chill; }
  public void setChill(String chill) {
    this.chill = chill;
  }
```

```
  public String getHumidity() { return humidity; }
  public void setHumidity(String humidity) {
    this.humidity = humidity;
  }
}
```

The `Weather` class defines a simple bean that is used to hold the weather information parsed from the Yahoo Weather feed. This feed provides a wealth of information, from the sunrise and sunset times to the speed and direction of the wind. To keep this example as simple as possible, the `Weather` model object keeps track of only the temperature, chill, humidity, and a textual description of current conditions.

Now, in the same directory, create a file named `Main.java`. This `Main` class will hold the static `main()` method—the entry point for this example.

Simple Weather's Main Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.apache.log4j.PropertyConfigurator;


public class Main {

    public static void main(String[] args) throws Exception {
        // Configure Log4J
        PropertyConfigurator
          .configure(Main.class.getClassLoader()
                      .getResource("log4j.properties"));

        // Read the zip code from the command line
        // (if none supplied, use 60202)
        String zipcode = "60202";
        try {
            zipcode = args[0];
        } catch( Exception e ) {}

        // Start the program
        new Main(zipcode).start();
    }

    private String zip;

    public Main(String zip) {
        this.zip = zip;
```

```
    }

    public void start() throws Exception {
        // Retrieve Data
        InputStream dataIn = new YahooRetriever().retrieve( zip );

        // Parse Data
        Weather weather = new YahooParser().parse( dataIn );

        // Format (Print) Data
        System.out.print( new WeatherFormatter().format( weather ) );
    }
}
```

The `main()` method shown above configures Log4J by retrieving a resource from the classpath. It then tries to read a zip code from the command line. If an exception is thrown while it is trying to read the zip code, the program will default to a zip code of 60202. Once it has a zip code, it instantiates an instance of `Main` and calls the `start()` method on an instance of `Main`. The `start()` method calls out to the `YahooRetriever` to retrieve the weather XML. The `YahooRetriever` returns an `InputStream` which is then passed to the `YahooParser`. The `YahooParser` parses the Yahoo Weather XML and returns a `Weather` object. Finally, the `WeatherFormatter` takes a `Weather` object and spits out a formatted `String` which is printed to standard output.

Create a file named `YahooRetriever.java` in the same directory with the contents shown in Simple Weather's YahooRetriever Class.

Simple Weather's YahooRetriever Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

import org.apache.log4j.Logger;

public class YahooRetriever {

    private static Logger log = Logger.getLogger(YahooRetriever.class);

    public InputStream retrieve(String zipcode) throws Exception {
        log.info( "Retrieving Weather Data" );
        String url = "http://weather.yahooapis.com/forecastrss?p="
          + zipcode;
        URLConnection conn = new URL(url).openConnection();
        return conn.getInputStream();
```

```
    }
}
```

This simple class opens a `URLConnection` to the Yahoo Weather API and returns an `InputStream`. To create something to parse this feed, we'll need to create the `YahooParser.java` file in the same directory.

Simple Weather's YahooParser Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import org.apache.log4j.Logger;
import org.dom4j.Document;
import org.dom4j.DocumentFactory;
import org.dom4j.io.SAXReader;

public class YahooParser {

  private static Logger log = Logger.getLogger(YahooParser.class);

  public Weather parse(InputStream inputStream) throws Exception {
    Weather weather = new Weather();

    log.info( "Creating XML Reader" );
    SAXReader xmlReader = createXmlReader();
    Document doc = xmlReader.read( inputStream );

    log.info( "Parsing XML Response" );
    weather.setCity(
      doc.valueOf("/rss/channel/y:location/@city") );
    weather.setRegion(
      doc.valueOf("/rss/channel/y:location/@region") );
    weather.setCountry(
      doc.valueOf("/rss/channel/y:location/@country") );
    weather.setCondition(
      doc.valueOf("/rss/channel/item/y:condition/@text") );
    weather.setTemp(
      doc.valueOf("/rss/channel/item/y:condition/@temp") );
    weather.setChill(
      doc.valueOf("/rss/channel/y:wind/@chill") );
    weather.setHumidity(
      doc.valueOf("/rss/channel/y:atmosphere/@humidity") );
```

```
    return weather;
  }

  private SAXReader createXmlReader() {
    Map<String,String> uris = new HashMap<String,String>();
    uris.put( "y", "http://xml.weather.yahoo.com/ns/rss/1.0" );

    DocumentFactory factory = new DocumentFactory();
        factory.setXPathNamespaceURIs( uris );

    SAXReader xmlReader = new SAXReader();
      xmlReader.setDocumentFactory( factory );
    return xmlReader;
  }
}
```

The YahooParser is the most complex class in this example. We're not going to dive into the details of Dom4J or Jaxen here, but the class deserves some explanation. YahooParser's parse() method takes an InputStream and returns a Weather object. To do this, it needs to parse an XML document with Dom4J. Since we're interested in elements under the Yahoo Weather XML namespace, we need to create a namespace-aware SAXReader in the createXmlReader() method. Once we create this reader and parse the document, we get an org.dom4j.Document object back. Instead of iterating through child elements, we simply address each piece of information we need using an XPath expression. Dom4J provides the XML parsing in this example, and Jaxen provides the XPath capabilities.

Once we've created a Weather object, we need to format our output for human consumption. Create a file named WeatherFormatter.java in the same directory as the other classes.

Simple Weather's WeatherFormatter Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String format( Weather weather ) throws Exception {
        log.info( "Formatting Weather Data" );
```

```
        Reader reader =
            new InputStreamReader( getClass().getClassLoader()
                                   .getResourceAsStream("output.vm"));
        VelocityContext context = new VelocityContext();
        context.put("weather", weather );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }
}
```

The `WeatherFormatter` uses Velocity to render a template. The `format()` method takes a `Weather` bean and spits out a formatted `String`. The first thing the `format()` method does is load a Velocity template from the classpath named `output.vm`. We then create a `VelocityContext` which is populated with a single `Weather` object named `weather`. A `StringWriter` is created to hold the results of the template merge. The template is evaluated with a call to `Velocity.evaluate()` and the results are returned as a `String`.

Before we can run this example, we'll need to add some resources to our classpath.

4.7 Add Resources

This project depends on two classpath resources: the `Main` class that configures Log4J with a classpath resource named `log4j.properties`, and the `WeatherFormatter` that references a Velocity template from the classpath named `output.vm`. Both of these resources need to be in the default package (or the root of the classpath).

To add these resources, we'll need to create a new directory from the base directory of the project: `src/main/resources`. Since this directory was not created by the `archetype:generate` task, we need to create it by executing the following commands from the project's base directory:

```
$ cd src/main
$ mkdir resources
$ cd resources
```

Once the resources directory is created, we can add the two resources. First, add the `log4j.properties` file in the `resources` directory, as shown in Simple Weather's Log4J Configuration File.

Simple Weather's Log4J Configuration File

```
# Set root category priority to INFO and its only appender to CONSOLE.
log4j.rootCategory=INFO, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%-4r %-5p %c{1} %x - %m%n
```

This `log4j.properties` file simply configures Log4J to print all log messages to standard output using a `PatternLayout`. Lastly, we need to create the `output.vm`, which is the Velocity template used to render the output of this command-line program. Create `output.vm` in the `resources` directory.

Simple Weather's Output Velocity Template

```
*********************************
Current Weather Conditions for:
${weather.city}, ${weather.region}, ${weather.country}

Temperature: ${weather.temp}
Condition: ${weather.condition}
Humidity: ${weather.humidity}
Wind Chill: ${weather.chill}
*********************************
```

This template contains a number of references to a variable named `weather`, which is the `Weather` bean that was passed to the `WeatherFormatter`. The `${weather.temp}` syntax is shorthand for retrieving and displaying the value of the `temp` bean property. Now that we have all of our project's code in the right place, we can use Maven to run the example.

4.8 Running the Simple Weather Program

Using the Exec plugin from the Codehaus Mojo project, we can run the Main class:

```
$ mvn install
$ mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main
...
[INFO] [exec:java]
0INFO  YahooRetriever  - Retrieving Weather Data
134  INFO  YahooParser  - Creating XML Reader
333  INFO  YahooParser  - Parsing XML Response
```

```
420  INFO  WeatherFormatter  - Formatting Weather Data
*********************************
Current Weather Conditions for:
Evanston, IL, US

Temperature: 45
Condition: Cloudy
Humidity: 76
Wind Chill: 38
*********************************
...
```

We didn't supply a command-line argument to the `Main` class, so we ended up with the default zip code, 60202. To supply a zip code, we would use the `-Dexec.args` argument and pass in a zip code:

```
$ mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main \
      -Dexec.args="70112"
...
[INFO] [exec:java]
0    INFO  YahooRetriever  - Retrieving Weather Data
134  INFO  YahooParser  - Creating XML Reader
333  INFO  YahooParser  - Parsing XML Response
420  INFO  WeatherFormatter  - Formatting Weather Data
*********************************
Current Weather Conditions for:
New Orleans, LA, US

Temperature: 82
Condition: Fair
Humidity: 71
Wind Chill: 82
*********************************
[INFO] Finished at: Sun Aug 31 09:33:34 CDT 2008
...
```

As you can see, we've successfully executed the Simple Weather command-line tool, retrieved some data from Yahoo Weather, parsed the result, and formatted the resulting data with Velocity. We achieved all of this without doing much more than writing our project's source code and adding some minimal configuration to the `pom.xml`. Notice that no "build process" was involved. We didn't need to define how or where the Java compiler compiles our source to bytecode, and we didn't need to instruct the build system how to locate the bytecode when we executed the example application. All we needed to do to include a few dependencies was locate the appropriate Maven coordinates.

4.8.1 The Maven Exec Plugin

The Exec plugin allows you to execute Java classes and other scripts. It is not a core Maven plugin, but it is available from the Mojo project hosted by Codehaus. For a full description of the Exec plugin, run:

```
$ mvn help:describe -Dplugin=exec -Dfull
```

This will list all of the goals that are available in the Maven Exec plugin. The Help plugin will also list all of the valid parameters for the Exec plugin. If you would like to customize the behavior of the Exec plugin you should use the documentation provided by `help:describe` as a guide. Although the Exec plugin is useful, you shouldn't rely on it as a way to execute your application outside of running tests during development. For a more robust solution, use the Maven Assembly plugin that is demonstrated in the section Section 4.13, later in this chapter.

4.8.2 Exploring Your Project Dependencies

The Exec plugin makes it possible for us to run the Simple Weather program without having to load the appropriate dependencies into the classpath. In any other build system, we would have to copy all of the program dependencies into some sort of `lib/` directory containing a collection of JAR files. Then, we would have to write a simple script that includes our program's bytecode and all of our dependencies in a classpath. Only then could we run `java org.sonatype.mavenbook.weather.Main`. The Exec plugin leverages the fact that Maven already knows how to create and manage your classpath and dependencies.

This is convenient, but it's also nice to know exactly what is being included in your project's classpath. Although the project depends on a few libraries such as Dom4J, Log4J, Jaxen, and Velocity, it also relies on a few transitive dependencies. If you need to find out what is on the classpath, you can use the Maven Dependency plugin to print out a list of dependencies.

```
$ mvn dependency:resolve
...
[INFO] [dependency:resolve]
[INFO]
[INFO] The following files have been resolved:
[INFO]com.ibm.icu:icu4j:jar:2.6.1 (scope = compile)
[INFO]commons-collections:commons-collections:jar:3.1 (scope = compile)
[INFO]commons-lang:commons-lang:jar:2.1 (scope = compile)
[INFO]dom4j:dom4j:jar:1.6.1 (scope = compile)
[INFO]jaxen:jaxen:jar:1.1.1 (scope = compile)
[INFO]jdom:jdom:jar:1.0 (scope = compile)
[INFO]junit:junit:jar:3.8.1 (scope = test)
[INFO]log4j:log4j:jar:1.2.14 (scope = compile)
```

```
[INFO]oro:oro:jar:2.0.8 (scope = compile)
[INFO]velocity:velocity:jar:1.5 (scope = compile)
[INFO]xalan:xalan:jar:2.6.0 (scope = compile)
[INFO]xerces:xercesImpl:jar:2.6.2 (scope = compile)
[INFO]xerces:xmlParserAPIs:jar:2.6.2 (scope = compile)
[INFO]xml-apis:xml-apis:jar:1.0.b2 (scope = compile)
[INFO]xom:xom:jar:1.0 (scope = compile)
```

As you can see, our project has a very large set of dependencies. While we only included direct dependencies on four libraries, we appear to be depending on 15 dependencies in total. Dom4J depends on Xerces and the XML Parser APIs, and Jaxen depends on Xalan. The Dependency plugin is going to print out the final combination of dependencies under which your project is being compiled. If you would like to know about the entire dependency tree of your project, you can run the `dependency:tree` goal.

```
$ mvn dependency:tree
...
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.custom:simple-weather:jar:1.0
[INFO] +- log4j:log4j:jar:1.2.14:compile
[INFO] +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  \- xml-apis:xml-apis:jar:1.0.b2:compile
[INFO] +- jaxen:jaxen:jar:1.1.1:compile
[INFO] |  +- jdom:jdom:jar:1.0:compile
[INFO] |  +- xerces:xercesImpl:jar:2.6.2:compile
[INFO] |  \- xom:xom:jar:1.0:compile
[INFO] | +- xerces:xmlParserAPIs:jar:2.6.2:compile
[INFO] | +- xalan:xalan:jar:2.6.0:compile
[INFO] | \- com.ibm.icu:icu4j:jar:2.6.1:compile
[INFO] +- velocity:velocity:jar:1.5:compile
[INFO] |  +- commons-collections:commons-collections:jar:3.1:compile
[INFO] |  +- commons-lang:commons-lang:jar:2.1:compile
[INFO] |  \- oro:oro:jar:2.0.8:compile
[INFO] +- org.apache.commons:commons-io:jar:1.3.2:test
[INFO] \- junit:junit:jar:3.8.1:test
...
```

If you're truly adventurous or want to see the full dependency trail, including artifacts that were rejected due to conflicts and other reasons, run Maven with the `-X` debug flag.

```
$ mvn install -X
...
[DEBUG] org.sonatype.mavenbook.custom:simple-weather:jar:1.0 (selected for ↩
    null)
[DEBUG]   log4j:log4j:jar:1.2.14:compile (selected for compile)
[DEBUG]   dom4j:dom4j:jar:1.6.1:compile (selected for compile)
[DEBUG] xml-apis:xml-apis:jar:1.0.b2:compile (selected for compile)
[DEBUG]   jaxen:jaxen:jar:1.1.1:compile (selected for compile)
```

```
[DEBUG] jaxen:jaxen:jar:1.1-beta-6:compile (removed - )
[DEBUG] jaxen:jaxen:jar:1.0-FCS:compile (removed - )
[DEBUG] jdom:jdom:jar:1.0:compile (selected for compile)
[DEBUG] xml-apis:xml-apis:jar:1.3.02:compile (removed - nearer: 1.0.b2)
[DEBUG] xerces:xercesImpl:jar:2.6.2:compile (selected for compile)
[DEBUG] xom:xom:jar:1.0:compile (selected for compile)
[DEBUG]   xerces:xmlParserAPIs:jar:2.6.2:compile (selected for compile)
[DEBUG]   xalan:xalan:jar:2.6.0:compile (selected for compile)
[DEBUG]   xml-apis:xml-apis:1.0.b2.
[DEBUG]   com.ibm.icu:icu4j:jar:2.6.1:compile (selected for compile)
[DEBUG]   velocity:velocity:jar:1.5:compile (selected for compile)
[DEBUG] commons-collections:commons-collections:jar:3.1:compile
[DEBUG] commons-lang:commons-lang:jar:2.1:compile (selected for compile)
[DEBUG] oro:oro:jar:2.0.8:compile (selected for compile)
[DEBUG]   junit:junit:jar:3.8.1:test (selected for test)
```

In the debug output, we see some of the guts of the dependency management system at work. What you see here is the tree of dependencies for this project. Maven is printing out the full Maven coordinates for all of your project's dependencies and the mechanism at work.

4.9 Writing Unit Tests

Maven has built-in support for unit tests, and testing is a part of the default Maven lifecycle. Let's add some unit tests to our simple weather project. First, let's create the `org.sonatype.mavenbook.weather` package under `src/test/java`:

```
$ cd src/test/java
$ cd org/sonatype/mavenbook
$ mkdir -p weather/yahoo
$ cd weather/yahoo
```

At this point, we will create two unit tests. The first will test the `YahooParser`, and the second will test the `WeatherFormatter`. In the `weather` package, create a file named `YahooParserTest.java` with the contents shown in the next example.

Simple Weather's YahooParserTest Unit Test

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;
```

```
import junit.framework.TestCase;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.YahooParser;

public class YahooParserTest extends TestCase {

    public YahooParserTest(String name) {
        super(name);
    }

    public void testParser() throws Exception {
        InputStream nyData = getClass().getClassLoader()
          .getResourceAsStream("ny-weather.xml");
        Weather weather = new YahooParser().parse( nyData );
        assertEquals( "New York", weather.getCity() );
        assertEquals( "NY", weather.getRegion() );
        assertEquals( "US", weather.getCountry() );
        assertEquals( "39", weather.getTemp() );
        assertEquals( "Fair", weather.getCondition() );
        assertEquals( "39", weather.getChill() );
        assertEquals( "67", weather.getHumidity() );
    }
}
```

This `YahooParserTest` extends the `TestCase` class defined by JUnit. It follows the usual pattern for a JUnit test: a constructor that takes a single `String` argument that calls the constructor of the superclass, and a series of public methods that begin with “`test`” that are invoked as unit tests. We define a single test method, `testParser`, which tests the `YahooParser` by parsing an XML document with known values. The test XML document is named `ny-weather.xml` and is loaded from the classpath. We’ll add test resources in Section 4.11. In our Maven project’s directory layout, the `ny-weather.xml` file is found in the directory that contains test resources — `${basedir}/src/test/resources` under `org/sonatype/mavenbook/weather/yahoo/ny-weather.xml`. The file is read as an `InputStream` and passed to the `parse()` method on `YahooParser`. The `parse()` method returns a `Weather` object, which is then tested with a series of calls to `assertEquals()`, a method defined by `TestCase`.

In the same directory, create a file named `WeatherFormatterTest.java`.

Simple Weather’s WeatherFormatterTest Unit Test

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import org.apache.commons.io.IOUtils;
```

```
import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.WeatherFormatter;
import org.sonatype.mavenbook.weather.YahooParser;

import junit.framework.TestCase;

public class WeatherFormatterTest extends TestCase {

  public WeatherFormatterTest(String name) {
    super(name);
  }

  public void testFormat() throws Exception {
    InputStream nyData = getClass().getClassLoader()
      .getResourceAsStream("ny-weather.xml");
    Weather weather = new YahooParser().parse( nyData );
    String formattedResult = new WeatherFormatter().format( weather );
    InputStream expected = getClass().getClassLoader()
      .getResourceAsStream("format-expected.dat");
    assertEquals( IOUtils.toString( expected ).trim(),
                  formattedResult.trim() );
  }
}
```

The second unit test in this simple project tests the `WeatherFormatter`. Like the `YahooParserTest`, the `WeatherFormatterTest` also extends JUnit's `TestCase` class. The single test function reads the same test resource from `${basedir}/src/test/resources` under the `org/sonatype/mavenbook/weather/yahoo` directory via this unit test's classpath. We'll add test resources in Section 4.11. `WeatherFormatterTest` runs this sample input file through the `YahooParser` which spits out a `Weather` object, and this object is then formatted with the `WeatherFormatter`. Since the `WeatherFormatter` prints out a `String`, we need to test it against some expected input. Our expected input has been captured in a text file named `format-expected.dat` which is in the same directory as `ny-weather.xml`. To compare the test's output to the expected output, we read this expected output in as an `InputStream` and use Commons IO's `IOUtils` class to convert this file to a `String`. This `String` is then compared to the test output using `assertEquals()`.

4.10 Adding Test-scoped Dependencies

In `WeatherFormatterTest`, we used a utility from Apache Commons IO—the `IOUtils` class. `IOUtils` provides a number of helpful static methods that take most of the work out of input/output operations. In this particular unit test, we used `IOUtils.toString()` to copy the `format-expec` ted.dat classpath resource to a String. We could have done this without using Commons IO, but it would have required an extra six or seven lines of code to deal with the various InputStreamReader and StringWriter objects. The main reason we used Commons IO was to give us an excuse to add a test-scoped dependency on Commons IO.

A test-scoped dependency is a dependency that is available on the classpath only during test compilation and test execution. If your project has war or ear packaging, a test-scoped dependency would not be included in the project's output archive. To add a test-scoped dependency, add the dependency element to your project's dependencies section, as shown in the following example:

Adding a Test-scoped Dependency

```
<project>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-io</artifactId>
            <version>1.3.2</version>
            <scope>test</scope>
        </dependency>
        ...
    </dependencies>
</project>
```

After you add this dependency to the pom.xml, run mvn dependency:resolve and you should see that commons-io is now listed as a dependency with scope test. We need to do one more thing before we are ready to run this project's unit tests. We need to create the classpath resources these unit tests depend on.

4.11 Adding Unit Test Resources

A unit test has access to a set of resources which are specific to tests. Often you'll store files containing expected results and files containing dummy input in the test classpath. In this project, we're storing a test XML document for YahooParserTest named ny-weather.xml and a file containing expected output from the WeatherFormatter in format-expected.dat.

To add test resources, you'll need to create the src/test/resources directory. This is the default directory in which Maven looks for unit test resources. To create this directory execute the following commands from your project's base directory.

```
$ cd src/test
$ mkdir resources
$ cd resources
```

Once you've create the resources directory, create a file named `format-expected.dat` in the `resources` directory.

Simple Weather's WeatherFormatterTest Expected Output

```
*********************************
Current Weather Conditions for:
New York, NY, US

Temperature: 39
Condition: Fair
Humidity: 67
Wind Chill: 39
*********************************
```

This file should look familiar. It is the same output that was generated previously when you ran the Simple Weather project with the Maven Exec plugin. The second file you'll need to add to the resources directory is `ny-weather.xml`.

Simple Weather's YahooParserTest XML Input

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss ↩
    /1.0"
     xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
  <channel>
    <title>Yahoo Weather - New York, NY</title>
    <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/
      </link>
    <description>Yahoo Weather for New York, NY</description>
    <language>en-us</language>
    <lastBuildDate>Sat, 10 Nov 2007 8:51 pm EDT</lastBuildDate>

    <ttl>60</ttl>
    <yweather:location city="New York" region="NY" country="US" />
    <yweather:units temperature="F" distance="mi" pressure="in"
      speed="mph"/>
    <yweather:wind chill="39" direction="0" speed="0" />
    <yweather:atmosphere humidity="67" visibility="1609"
      pressure="30.18" rising="1" />
```

```
    <yweather:astronomy sunrise="6:36 am" sunset="4:43 pm" />
    <image>
      <title>Yahoo Weather</title>
      <width>142</width>
      <height>18</height>
      <link>http://weather.yahoo.com/</link>
      <url>http://l.yimg.com/us.yimg.com/i/us/nws/th/main_142b.gif</url>
    </image>
    <item>
      <title>Conditions for New York, NY at 8:51 pm EDT</title>
      <geo:lat>40.67</geo:lat>
      <geo:long>-73.94</geo:long>
      <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/
        </link>
      <pubDate>Sat, 10 Nov 2007 8:51 pm EDT</pubDate>
      <yweather:condition text="Fair" code="33" temp="39"
                          date="Sat, 10 Nov 2007 8:51 pm EDT"/>
      <description><![CDATA[
<img src="http://l.yimg.com/us.yimg.com/i/us/we/52/33.gif" /><br />
<b>Current Conditions:</b><br />
Fair, 39 F<BR /><BR />
<b>Forecast:</b><BR />
Sat - Partly Cloudy. High: 45 Low: 32<br />
Sun - Sunny. High: 50 Low: 38<br />
<br />
]]></description>
      <yweather:forecast day="Sat" date="10 Nov 2007" low="32" high="45"
                         text="Partly Cloudy" code="29" />

      <yweather:forecast day="Sun" date="11 Nov 2007" low="38" high="50"
                         text="Sunny" code="32" />
      <guid isPermaLink="false">10002_2007_11_10_20_51_EDT</guid>
    </item>
  </channel>
</rss>
```

This file contains a test XML document for the `YahooParserTest`. We store this file so that we can test the `YahooParser` without having to retrieve an XML response from Yahoo Weather.

4.12 Executing Unit Tests

Now that your project has unit tests, let's run them. You don't have to do anything special to run a unit test; the `test` phase is a normal part of the Maven lifecycle. You run Maven tests whenever you run `mvn` package or mvn install. If you would like to run all the lifecycle phases up to and including the test phase, run mvn test:

```
$ mvn test
...
[INFO] [surefire:test]
[INFO] Surefire report directory:
~/examples/ch-custom/simple-weather/target/surefire-reports

-------------------------------------------------------
T E S T S
-------------------------------------------------------
Running org.sonatype.mavenbook.weather.yahoo.WeatherFormatterTest
0    INFO  YahooParser  - Creating XML Reader
177  INFO  YahooParser  - Parsing XML Response
239  INFO  WeatherFormatter  - Formatting Weather Data
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.547 sec
Running org.sonatype.mavenbook.weather.yahoo.YahooParserTest
475  INFO  YahooParser  - Creating XML Reader
483  INFO  YahooParser  - Parsing XML Response
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Executing mvn test from the command line caused Maven to execute all lifecycle phases up to the test phase. The Maven Surefire plugin has a test goal which is bound to the test phase. This test goal executes all of the unit tests this project can find under src/test/java with filenames matching **/Test*.java, **/*Test.java and **/*TestCase.java. In the case of this project, you can see that the Surefire plugin's test goal executed WeatherFormatterTest and YahooParserTest. When the Maven Surefire plugin runs the JUnit tests, it also generates XML and text reports in the ${basedir}/target/surefire-reports directory. If your tests are failing, you should look in this directory for details like stack traces and error messages generated by your unit tests.

4.12.1 Ignoring Test Failures

You will often find yourself developing on a system that has failing unit tests. If you are practicing Test-Driven Development (TDD), you might use test failure as a measure of how close your project is to completeness. If you have failing unit tests, and you would still like to produce build output, you are going to have to tell Maven to ignore build failures. When Maven encounters a build failure, its default behavior is to stop the current build. To continue building a project even when the Surefire plugin encounters failed test cases, you'll need to set the testFailureIgnore configuration property of the Surefire plugin to `true`.

Ignoring Unit Test Failures

```
<project>
    [...]
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <testFailureIgnore>true</testFailureIgnore>
                </configuration>
            </plugin>
        </plugins>
    </build>
    [...]
</project>
```

The plugin documents (http://maven.apache.org/plugins/maven-surefire-plugin/test-mojo.html) show that this parameter declares an expression:

Plugin Parameter Expressions

```
testFailureIgnore  Set this to true to ignore a failure during
                   testing. Its use is NOT RECOMMENDED, but quite
                   convenient on occasion.

* Type: boolean
* Required: No
* User Property: maven.test.failure.ignore
```

This property can be set from the command line using the `-D` parameter:

```
$ mvn test -Dmaven.test.failure.ignore=true
```

4.12.2 Skipping Unit Tests

You may want to configure Maven to skip unit tests altogether. Maybe you have a very large system where the unit tests take minutes to complete and you don't want to wait for unit tests to complete before producing output. You might be working with a legacy system that has a series of failing unit tests, and instead of fixing the unit tests, you might just want to produce a JAR. Maven provides for the ability to skip unit tests using the `skip` parameter of the Surefire plugin. To skip tests from the command line, simply add the `maven.test.skip` property to any goal:

```
$ mvn install -Dmaven.test.skip=true
...
[INFO] [compiler:testCompile]
[INFO] Not compiling test sources
[INFO] [surefire:test]
[INFO] Tests are skipped.
...
```

When the Surefire plugin reaches the `test` goal, it will skip the unit tests if the `maven.test.skip` properties is set to `true`. Another way to configure Maven to skip unit tests is to add this configuration to your project's `pom.xml`. To do this, you would add a `plugin` element to your build configuration.

Skipping Unit Tests

```
<project>
    [...]
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <skip>true</skip>
                </configuration>
            </plugin>
        </plugins>
    </build>
    [...]
</project>
```

4.13 Building a Packaged Command Line Application

In Section 4.8 earlier in descriptor in the Maven Assembly plugin to produce a distributable JAR file, which contains the project's bytecode and all of the dependencies.

The Maven Assembly plugin is a plugin you can use to create arbitrary distributions for your applications. You can use the Maven Assembly plugin to assemble the output of your project in any format you desire by defining a custom assembly descriptor. In a later chapter we will show you how to create a custom assembly descriptor which produces a more complex archive for the Simple Weather application. In this chapter, we're going to use the predefined `jar-with-dependencies` format. To configure the Maven Assembly Plugin, we need to add the following plugin configuration to our existing build configuration in the `pom.xml`.

Configuring the Maven Assembly Descriptor

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Once you've added this configuration, you can build the assembly by running the `assembly:assembly` goal. In the following screen listing, the `assembly:assembly` goal is executed after the Maven build reaches the `install` lifecycle phase:

```
$ mvn install assembly:assembly
...
[INFO] [jar:jar]
[INFO] Building jar:
~/examples/ch-custom/simple-weather/target/simple-weather-1.0.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Expanding: \
.m2/repository/dom4j/dom4j/1.6.1/dom4j-1.6.1.jar into \
/tmp/archived-file-set.1437961776.tmp
[INFO] Expanding: .m2/repository/commons-lang/commons-lang/2.1/\
commons-lang-2.1.jar
into /tmp/archived-file-set.305257225.tmp
... (Maven Expands all dependencies into a temporary directory) ...
[INFO] Building jar: \
~/examples/ch-custom/simple-weather/target/\
simple-weather-1.0-jar-with-dependencies.jar
```

Once our assembly is assembled in `target/simple-weather-1.0-jar-with-dependencies.jar`, we can run the `Main` class again from the command line. To run the simple weather application's `Main` class, execute the following commands from your project's base directory:

```
$ cd target
$ java -cp simple-weather-1.0-jar-with-dependencies.jar \
       org.sonatype.mavenbook.weather.Main 10002
0    INFO  YahooRetriever  - Retrieving Weather Data
221  INFO  YahooParser  - Creating XML Reader
399  INFO  YahooParser  - Parsing XML Response
474  INFO  WeatherFormatter  - Formatting Weather Data
*********************************
Current Weather Conditions for:
New York, NY, US

Temperature: 44
Condition: Fair
Humidity: 40
Wind Chill: 40
*********************************
```

The `jar-with-dependencies` format creates a single JAR file that includes all of the bytecode from the `simple-weather` project as well as the unpacked bytecode from all of the dependencies. This somewhat unconventional format produces a 9 MiB JAR file containing approximately 5,290 classes, but it does provide for an easy distribution format for applications you've developed with Maven. Later in this book, we'll show you how to create a custom assembly descriptor to produce a more standard distribution.

4.13.1 Attaching the Assembly Goal to the Package Phase

In Maven 1, a build was customized by stringing together a series of plugin goals. Each plugin goal had prerequisites and defined a relationship to other plugin goals. With the release of Maven 2, a lifecycle was introduced and plugin goals are now associated with a series of phases in a default Maven build lifecycle. The lifecycle provides a solid foundation that makes it easier to predict and manage the plugin goals which will be executed in a given build. In Maven 1, plugin goals related to one another directly; in Maven 2, plugin goals relate to a set of common lifecycle stages. While it is certainly valid to execute a plugin goal directly from the command line as we just demonstrated, it is more consistent with the design of Maven to configure the Assembly plugin to execute the `assembly:assembly` goal during a phase in the Maven lifecycle.

The following plugin configuration configures the Maven Assembly plugin to execute the `attached` goal during the `package` phase of the Maven default build lifecycle. The `attached` goal does the same thing as the `assembly` goal. To bind to `assembly:attached` goal to the `package` phase we use the `executions` element under `plugin` in the `build` section of the project's POM.

Configuring Attached Goal Execution During the Package Lifecycle Phase

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <id>simple-command</id>
            <phase>package</phase>
            <goals>
              <goal>attached</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Once you have this configuration in your POM, all you need to do to generate the assembly is run `mvn package`. The execution configuration will make sure that the `assembly:attached` goal is executed when the Maven lifecycle transitions to the `package` phase of the lifecycle. The assembly will also be created if you run `mvn install`, as the `package` phase precedes the `install` phase in the default Maven lifecycle.

Chapter 5

A Simple Web Application

5.1 Introduction

In this chapter, we create a simple web application with the Maven Archetype plugin. We'll run this web application in a Servlet container named Jetty, add some dependencies, write a simple Servlet, and generate a WAR file. At the end of this chapter, you will be able to start using Maven to accelerate the development of web applications.

5.1.1 Downloading this Chapter's Example

The example in this chapter is generated with the Maven Archetype plugin. While you should be able to follow the development of this chapter without the example source code, we recommend downloading a copy of the example code to use as a reference. This chapter's example project may be downloaded with the book's example code at:

```
http://books.sonatype.com/mvnex-book/mvnex-examples.zip
```

Unzip this archive in any directory, and then go to the `ch-simple-web` directory. There you will see a directory named `simple-webapp`, which contains the Maven project developed in this chapter.

5.2 Defining the Simple Web Application

We've purposefully kept this chapter focused on Plain-Old Web Applications (POWA)—a Spring Framework; and the other that uses Plexus.

5.3 Creating the Simple Web Project

To create your web application:

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.simpleweb \
      -DartifactId=simple-webapp \
      -Dpackage=org.sonatype.mavenbook \
      -Dversion=1.0-SNAPSHOT
...
[INFO] [archetype:generate {execution: default-cli}]
Choose archetype:
...
19: internal -> maven-archetype-webapp (A simple Java web application)
...
Choose a number:  (...) 15: : 19
Confirm properties configuration:
groupId: org.sonatype.mavenbook.simpleweb
artifactId: simple-webapp
version: 1.0-SNAPSHOT
package: org.sonatype.mavenbook.simpleweb
Y: : Y
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.simpleweb
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook.simpleweb
[INFO] Parameter: package, Value: org.sonatype.mavenbook.simpleweb
[INFO] Parameter: artifactId, Value: simple-webapp
[INFO] Parameter: basedir, Value: /private/tmp
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
...
[INFO] BUILD SUCCESSFUL
```

Once the Maven Archetype plugin creates the project, change directories into the `simple-webapp` directory and take a look at the `pom.xml`. You should see something close to the following.

Initial POM for the simple-webapp Project

```
<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>org.sonatype.mavenbook.simpleweb</groupId>
    <artifactId>simple-webapp</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>simple-webapp 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>simple-webapp</finalName>
    </build>
</project>
```

Next, you will need to configure the Maven Compiler plugin to target Java 5. To do this, add the plugins element to the initial POM as shown in POM for the simple-webapp Project with Compiler Configuration.

POM for the simple-webapp Project with Compiler 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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook.simpleweb</groupId>
    <artifactId>simple-webapp</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>simple-webapp Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

```
    <build>
        <finalName>simple-webapp</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Notice the `packaging` element contains the value `war`. This packaging type is what configures Maven to produce a web application archive in a WAR file. A project with `war` packaging is going to create a WAR file in the `target/` directory. The default name of this file is `${artifactId}-${version}.war`. In this project, the default WAR would be generated in `target/simple-webapp-1.0-SNAPSHOT.war`. In the `simple-webapp` project, we've customized the name of the generated WAR file by adding a finalName element inside of this project's build configuration. With a finalName of `simple-webapp`, the `package` phase produces a WAR file in `target/simple-webapp.war`.

5.4 Configuring the Jetty Plugin

Configuring the Jetty Plugin

```
<project>
    [...]
    <build>
        <finalName>simple-webapp</finalName>
        <plugins>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
    [...]
</project>
```

Once you've configured the Maven Jetty Plugin in your project's `pom.xml`, you can then invoke the Run goal of the Jetty plugin to start your web application in the Jetty Servlet container. Run `mvn jetty:run` from the `simple-webapp/` project directory as follows:

```
~/examples/ch-simple-web/simple-webapp $ mvn jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[INFO] Webapp source directory = \
~/svnw/sonatype/examples/ch-simple-web/simple-webapp/src/main/webapp
[INFO] web.xml file = \
~/svnw/sonatype/examples/ch-simple-web/\
simple-webapp/src/main/webapp/WEB-INF/web.xml
[INFO] Classes = ~/svnw/sonatype/examples/ch-simple-web/\
simple-webapp/target/classes
2007-11-17 22:11:50.532::INFO:  Logging to STDERR via org.mortbay.log. ↩
    StdErrLog
[INFO] Context path = /simple-webapp
[INFO] Tmp directory =  determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] Webapp directory = \
~/svnw/sonatype/examples/ch-simple-web/simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-17 22:11:50.673::INFO:  jetty-6.1.6rc1
2007-11-17 22:11:50.846::INFO:  No Transaction manager found
2007-11-17 22:11:51.057::INFO:  Started SelectChannelConnector@0 ↩
    .0.0.0:8080
[INFO] Started Jetty Server
```

Warning

If you are running the Maven Jetty Plugin on a Windows platform you may need to move your local Maven repository to a directory that does not contain spaces. Some readers have reported issues on Jetty startup caused by a repository that was being stored under `C:\Documents and Settings\<user>`. The solution to this problem is to move your local Maven repository to a directory that does not contain spaces and redefine the location of your local repository in `~/.m2/settings.xml`.

After Maven starts the Jetty Servlet container, load the URL http://localhost:8080/simple-webapp/ in a web browser. The simple `index.jsp` generated by the Archetype is trivial; it contains a second-level heading with the text "Hello World!". Maven expects the document root of the web application to be stored in `src/main/webapp`. It is in this directory where you will find the `index.jsp` file shown in Contents of src/main/webapp/index.jsp.

Contents of src/main/webapp/index.jsp

```
<html>
  <body>
    <h2>Hello World!</h2>
  </body>
</html>
```

In `src/main/webapp/WEB-INF`, we will find the smallest possible web application `web.xml`, shown in this next example:

Contents of src/main/webapp/WEB-INF/web.xml

```
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd" >

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

5.5 Adding a Simple Servlet

A web application with a single JSP page and no configured servlets is next to useless. Let's add a simple servlet to this application and make some changes to the `pom.xml` and `web.xml` to support this change. First, we'll need to create a new package under `src/main/java` named `org.sonatype.mavenbook.web`:

```
$ mkdir -p src/main/java/org/sonatype/mavenbook/web
$ cd src/main/java/org/sonatype/mavenbook/web
```

Once you've created this package, change to the `src/main/java/org/sonatype/mavenbook/web` directory and create a class named `SimpleServlet` in `SimpleServlet.java`, which contains the code shown in the `SimpleServlet` class:

SimpleServlet Class

```
package org.sonatype.mavenbook.web;

import java.io.*;
```

```
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        out.println( "SimpleServlet Executed" );
        out.flush();
        out.close();
    }
}
```

Our `SimpleServlet` class is just that: a servlet that prints a simple message to the response's `Writer`. To add this servlet to your web application and map it to a request path, add the `servlet` and `servlet-mapping` elements shown in the following `web.xml` to your project's `web.xml` file. The `web.xml` file can be found in `src/main/webapp/WEB-INF`.

Mapping the Simple Servlet

```
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>
      org.sonatype.mavenbook.web.SimpleServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
</web-app>
```

Everything is in place to test this servlet; the class is in `src/main/java` and the `web.xml` has been updated. Before we launch the Jetty plugin, compile your project by running `mvn compile`:

```
~/examples/ch-simple-web/simple-webapp $ mvn compile
...
[INFO] [compiler:compile]
```

```
[INFO] Compiling 1 source file to \
~/examples/ch-simple-web/simple-webapp/target/classes
[INFO] ------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------
[INFO] Compilation failure

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[4,0] \
package javax.servlet does not exist

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[5,0] \
package javax.servlet.http does not exist

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[7,35] \
cannot find symbol
symbol: class HttpServlet
public class SimpleServlet extends HttpServlet {

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[8,22] \
cannot find symbol
symbol  : class HttpServletRequest
location: class org.sonatype.mavenbook.web.SimpleServlet

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[9,22] \
cannot find symbol
symbol  : class HttpServletResponse
location: class org.sonatype.mavenbook.web.SimpleServlet

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[10,15] \
cannot find symbol
symbol  : class ServletException
location: class org.sonatype.mavenbook.web.SimpleServlet
```

The compilation fails because your Maven project doesn't have a dependency on the Servlet API. In the next section, we'll add the Servlet API to this project's POM.

5.6 Adding J2EE Dependencies

To write a servlet, we'll need to add the Servlet API as a dependency to the project's POM.

Add the Servlet 2.4 Specification as a Dependency

```
<project>
```

```
    [...]
    <dependencies>
        [...]
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    [...]
</project>
```

It is also worth pointing out that we have used the `provided` scope for this dependency. This tells Maven that the JAR is "provided" by the container and thus should not be included in the WAR. If you were interested in writing a custom JSP tag for this simple web application, you would need to add a dependency on the JSP 2.0 API. Use the configuration shown in this example:

Adding the JSP 2.0 Specification as a Dependency

```
<project>
    [...]
    <dependencies>
        [...]
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    [...]
</project>
```

Once you've added the Servlet specification as a dependency, run `mvn clean install` followed by `mvn jetty:run`.

Note

`mvn jetty:run` will continue to run the Jetty servlet container until you stop the process with CTRL-C. If you started Jetty in Section 5.4, you will need to stop that process before starting Jetty a second time.

```
[tobrien@t1 simple-webapp]$ mvn clean install
...
[tobrien@t1 simple-webapp]$ mvn jetty:run
[INFO] [jetty:run]
...
2007-12-14 16:18:31.305::INFO:  jetty-6.1.6rc1
2007-12-14 16:18:31.453::INFO:  No Transaction manager found
2007-12-14 16:18:32.745::INFO:  Started SelectChannelConnector@0 ←
    .0.0.0:8080
[INFO] Started Jetty Server
```

At this point, you should be able to retrieve the output of the `SimpleServlet`. From the command line, you can use curl to print the output of this servlet to standard output:

```
~/examples/ch-simple-web $ curl http://localhost:8080/simple-webapp/simple
SimpleServlet Executed
```

5.7 Conclusion

After reading this chapter, you should be able to bootstrap a simple web application. This chapter didn't dwell on the million different ways to create a complete web application. Other chapters provide a more comprehensive overview of projects that involve some of the more popular web frameworks and technologies.

Chapter 6

A Multi-Module Project

6.1 Introduction

In this chapter, we create a multi-module project that combines the examples from the two previous chapters. The `simple-weather` code developed in Chapter 4 will be combined with the `simple-webapp` project defined in Chapter 5 to create a web application that retrieves and displays weather forecast information on a web page. At the end of this chapter, you will be able to use Maven to develop complex, multi-module projects.

6.1.1 Downloading this Chapter's Example

The multi-module project developed in this example consists of modified versions of the projects developed in Chapter 4 and Chapter 5, and we are not using the Maven Archetype plugin to generate this multi-module project. We strongly recommend downloading a copy of the example code to use as a supplemental reference while reading the content in this chapter. This chapter's example project may be downloaded with the book's example code at:

```
http://books.sonatype.com/mvnex-book/mvnex-examples.zip
```

Unzip this archive in any directory, and then go to the `ch-multi/` directory. There you will see a directory named `simple-parent`, which contains the multi-module Maven project developed in this chapter. In this directory, you will see a `pom.xml` and the two submodule directories, `simple-weather` and `simple-webapp`.

6.2 The Simple Parent Project

A multi-module project is defined by a parent POM referencing one or more submodules. In the `simple-parent/` directory, you will find the parent POM (also called the top-level POM) in `simple-parent/pom.xml`. See simple-parent Project POM.

simple-parent Project POM

```
<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>org.sonatype.mavenbook.multi</groupId>
    <artifactId>simple-parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0</version>
    <name>Multi Chapter Simple Parent Project</name>

    <modules>
        <module>simple-weather</module>
        <module>simple-webapp</module>
    </modules>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <source>1.5</source>
                        <target>1.5</target>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

    <dependencies>
```

```
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Notice that the parent defines a set of Maven coordinates: the `groupId` is `org.sonatype.mavenbook.multi`, the `artifactId` is `simple-parent`, and the `version` is `1.0`. The parent project doesn't create a JAR or a WAR like our previous projects; instead, it is simply a POM that refers to other Maven projects. The appropriate packaging for a project like `simple-parent` that simply provides a Project Object Model is `pom`. The next section in the `pom.xml` lists the project's submodules. These modules are defined in the `modules` element, and each `module` element corresponds to a subdirectory of the `simple-parent` directory. Maven knows to look in these directories for `pom.xml` files, and it will add submodules to the list of Maven projects included in a build.

Lastly, we define some settings which will be inherited by all submodules. The `simple-parent` build configuration configures the target for all Java compilation to be the Java 5 JVM. Since the compiler plugin is bound to the lifecycle by default, we can use the `pluginManagement` section do to this. We will discuss `pluginManagement` in more detail in later chapters, but the separation between providing configuration to default plugins and actually binding plugins is much easier to see when they are separated this way. The `dependencies` element adds JUnit 3.8.1 as a global dependency. Both the build configuration and the dependencies are inherited by all submodules. Using POM inheritance allows you to add common dependencies for universal dependencies like JUnit or Log4J.

6.3 The Simple Weather Module

The first submodule we're going to look at is the `simple-weather` submodule. This submodule contains all the code from the previous section Chapter 4.

simple-weather Module POM

```
<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>
    <parent>
        <groupId>org.sonatype.mavenbook.multi</groupId>
```

```
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-weather</artifactId>
    <packaging>jar</packaging>

    <name>Multi Chapter Simple Weather API</name>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <configuration>
                        <testFailureIgnore>true</testFailureIgnore>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
        <dependency>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>velocity</groupId>
            <artifactId>velocity</artifactId>
            <version>1.5</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-io</artifactId>
            <version>1.3.2</version>
            <scope>test</scope>
        </dependency>
```

```
    </dependencies>
</project>
```

In simple-weather's pom.xml file, we see this module referencing a parent POM using a set of Maven coordinates. The parent POM for simple-weather is identified by a groupId of org.sonatype.mavenbook.multi, an artifactId of simple-parent, and a version of 1.0.

The WeatherService class shown in The WeatherService Class is defined in src/main/java/org/sonatype/mavenbook/weather, and it simply calls out to the three objects defined in Chapter 4. In this chapter's example, we're creating a separate project that contains service objects that are referenced in the web application project. This is a common model in enterprise Java development; often a complex application consists of more than just a single, simple web application. You might have an enterprise application that consists of multiple web applications and some command-line applications. Often, you'll want to refactor common logic to a service class that can be reused across a number of projects. This is the justification for creating a WeatherService class; by doing so, you can see how the simple-webapp project references a service object defined in simple-weather.

The WeatherService Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

public class WeatherService {

    public WeatherService() {}

    public String retrieveForecast( String zip ) throws Exception {
        // Retrieve Data
        InputStream dataIn = new YahooRetriever().retrieve( zip );

        // Parse Data
        Weather weather = new YahooParser().parse( dataIn );

        // Format (Print) Data
        return new WeatherFormatter().format( weather );
    }
}
```

The retrieveForecast() method takes a String containing a zip code. This zip code parameter is then passed to the YahooRetriever's retrieve() method, which gets the XML from Yahoo Weather. The XML returned from YahooRetriever is then passed to the parse() method on YahooParser which returns a Weather object. This Weather object is then formatted into a presentable String by the WeatherFormatter.

6.4 The Simple Web Application Module

The simple-webapp module is the second submodule referenced in the simple-parent project. This web application project depends upon the simple-weather module, and it contains some simple servlets that present the results of the Yahoo weather service query.

simple-webapp Module POM

```
<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>
    <parent>
        <groupId>org.sonatype.mavenbook.multi</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>

    <artifactId>simple-webapp</artifactId>
    <packaging>war</packaging>
    <name>simple-webapp Maven Webapp</name>
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.sonatype.mavenbook.multi</groupId>
            <artifactId>simple-weather</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>simple-webapp</finalName>
        <plugins>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

This `simple-webapp` module defines a very simple servlet that reads a zip code from an HTTP request, calls the `WeatherService` shown in The WeatherService Class, and prints the results to the response's `Writer`.

simple-webapp WeatherServlet

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.WeatherService;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class WeatherServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String zip = request.getParameter("zip" );
        WeatherService weatherService = new WeatherService();
        PrintWriter out = response.getWriter();
        try {
            out.println( weatherService.retrieveForecast( zip ) );
        } catch( Exception e ) {
            out.println( "Error Retrieving Forecast: " + e.getMessage() );
        }
        out.flush();
        out.close();
    }
}
```

In `WeatherServlet`, we instantiate an instance of the `WeatherService` class defined in `simple-weather`. The zip code supplied in the request parameter is passed to the `retrieveForecast()` method and the resulting test is printed to the response's `Writer`.

Finally, to tie all of this together is the `web.xml` for `simple-webapp` in `src/main/webapp/WEB-INF`. The `servlet` and `servlet-mapping` elements in the `web.xml` shown in simple-webapp web.xml map the request path `/weather` to the `WeatherServlet`.

simple-webapp web.xml

```
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd" >

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

```
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>
      org.sonatype.mavenbook.web.SimpleServlet
    </servlet-class>
  </servlet>
  <servlet>
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.sonatype.mavenbook.web.WeatherServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/weather</url-pattern>
  </servlet-mapping>
</web-app>
```

6.5 Building the Multimodule Project

With the `simple-weather` project containing all WAR file. To do this, you will want to compile and install both projects in the appropriate order; since `simple-webapp` depends on `simple-weather`, the `simple-weather` JAR needs to be created before the `simple-webapp` project can compile. To do this, you will run `mvn clean install` command from the `simple-parent` project:

```
~/examples/ch-multi/simple-parent$ mvn clean install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Simple Parent Project
[INFO]   simple-weather
[INFO]   simple-webapp Maven Webapp
[INFO] -----------------------------------------
[INFO] Building simple-weather
[INFO]task-segment: [clean, install]
[INFO] -----------------------------------------
[...]
[INFO] [install:install]
[INFO] Installing simple-weather-1.0.jar to simple-weather-1.0.jar
[INFO] -----------------------------------------
[INFO] Building simple-webapp Maven Webapp
```

```
[INFO]task-segment: [clean, install]
[INFO] ----------------------------------------
[...]
[INFO] [install:install]
[INFO] Installing simple-webapp.war to simple-webapp-1.0.war
[INFO]
[INFO] ----------------------------------------
[INFO] Reactor Summary:
[INFO] ----------------------------------------
[INFO] Simple Parent Project ................. SUCCESS [3.041s]
[INFO] simple-weather ........................ SUCCESS [4.802s]
[INFO] simple-webapp Maven Webapp ............ SUCCESS [3.065s]
[INFO] ----------------------------------------
```

When Maven is executed against a project with submodules, Maven first loads the parent POM and locates all of the submodule POMs. Maven then puts all of these project POMs into something called the Maven Reactor which analyzes the dependencies between modules. The Reactor takes care of ordering components to ensure that interdependent modules are compiled and installed in the proper order.

Note

The Reactor preserves the order of modules as defined in the POM unless changes need to be made. A helpful mental model for this is to picture that modules with dependencies on sibling projects are "pushed down" the list until the dependency ordering is satisfied. On rare occasions, it may be handy to rearrange the module order of your build — for example if you want a frequently unstable module towards the beginning of the build.

Once the Reactor figures out the order in which projects must be built, Maven then executes the specified goals for every module in the multi-module build. In this example, you can see that Maven builds `simple-weather` before `simple-webapp`, effectively executing `mvn clean install` for each submodule.

Note

When you run Maven from the command line you'll frequently want to specify the `clean` lifecycle phase before any other lifecycle stages. When you specify `clean`, you make sure that Maven is going to remove old output before it compiles and packages an application. Running `clean` isn't necessary, but it is a useful precaution to make sure that you are performing a "clean build".

6.6 Running the Web Application

Once the multi-module project has been installed with `mvn install`, you can run the web application with `mvn jetty:run`.

```
~/examples/ch-multi/simple-parent/simple-webapp $ mvn jetty:run
[INFO] ----------------------------------------
[INFO] Building simple-webapp Maven Webapp
[INFO]task-segment: [jetty:run]
[INFO] ----------------------------------------
[...]
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[...]
[INFO] Webapp directory = ~/examples/ch-multi/simple-parent/\
simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-18 1:58:26.980::INFO:  jetty-6.1.6rc1
2007-11-18 1:58:26.125::INFO:  No Transaction manager found
2007-11-18 1:58:27.633::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Once Jetty has started, load http://localhost:8080/simple-webapp/weather?zip=01201 in a browser and you should see the formatted weather output.

Chapter 7

Multi-Module Enterprise Project

7.1 Introduction

In this chapter, we create a multi-module project that evolves the examples from Chapter 6 and Chapter 5 into a project that uses the Spring Framework and Hibernate to create both a simple web application and a command-line utility to read data from the Yahoo Weather feed. The `simple-weather` code developed in Chapter 4 will be combined with the `simple-webapp` project defined in Chapter 5. In the process of creating this multi-module project, we'll explore Maven and discuss the different ways it can be used to create modular projects that encourage reuse.

7.1.1 Downloading this Chapter's Example

The multi-module project developed in this example consists of modified versions of the projects developed in Chapter 4 and Chapter 5, and we are not using the Maven Archetype plug-in to generate this multi-module project. We strongly recommend downloading a copy of the example code to use as a supplemental reference while reading the content in this chapter. Without the examples, you won't be able to recreate this chapter's example code. This chapter's example project may be downloaded with the book's example code at:

```
http://books.sonatype.com/mvnex-book/mvnex-examples.zip
```

Unzip this archive in any directory, and then go to the `ch-multi-spring` directory. There you will see a directory named `simple-parent` that contains the multi-module Maven project developed in this chapter. In the `simple-parent/` project directory you will see a `pom.xml` and the five submodule directories `simple-model/`, `simple-persist/`, `simple-command/`, `simple-weather/` and `simple-webapp/`.

7.1.2 Multi-Module Enterprise Project

Presenting the complexity of a massive Enterprise-level project far exceeds the scope of this book. Such projects are characterized by multiple databases, integration with external systems, and subprojects which may be divided by departments. These projects usually span thousands of lines of code, and involve the effort of tens or hundreds of software developers. While such a complete example is outside the scope of this book, we can provide you with a sample project that suggests the complexity of a larger Enterprise application. In the conclusion we suggest some possibilities for modularity beyond that presented in this chapter.

In this chapter, we're going to look at a multi-module Maven project that will produce two applications: a command-line query tool for the Yahoo Weather feed and a web application which queries the Yahoo Weather feed. Both of these applications will store the results of queries in an embedded database. Each will allow the user to retrieve historical weather data from this embedded database. Both applications will reuse application logic and share a persistence library. This chapter's example builds upon the Yahoo Weather parsing code introduced in Chapter 4. This project is divided into five submodules shown in Figure 7.1.

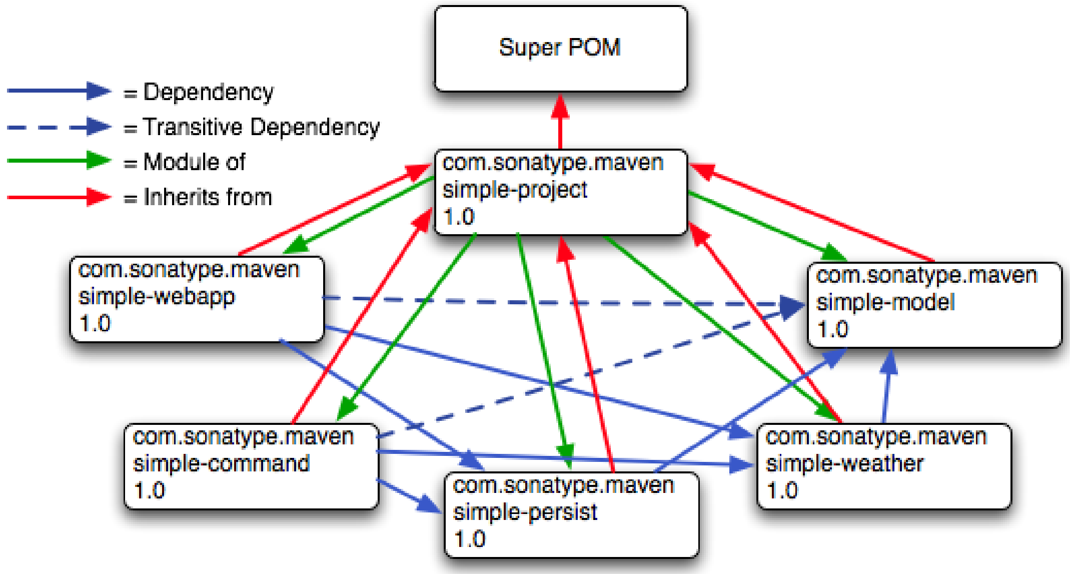

Figure 7.1: Multi-Module Enterprise Application Module Relationships

In Figure 7.1, you can see that there are five submodules of `simple-parent`. They are:

simple-model
: This module defines a simple object model which models the data returned from the Yahoo Weather feed. This object model contains the `Weather`, `Condition`, `Atmosphere`, `Location`, and `Wind` objects. When our application parses the Yahoo Weather feed, the parsers defined in `simple-weather` will parse the XML and create `Weather` objects which are then used by the application. This project contains model objects annotated with Hibernate 3 Annotations. These annotations are used by the logic in `simple-persist` to map each model object to a corresponding table in a relational database.

simple-weather
: This module contains all of the logic required to retrieve data from the Yahoo Weather feed and parse the resulting XML. The XML returned from this feed is converted into the model objects defined in `simple-model`. `simple-weather` has a dependency on `simple-model`. `simple-weather` defines a `WeatherService` object which is referenced by both the `simple-command` and `simple-webapp` projects.

simple-persist
: This module contains some Data Access Objects (DAO) which are configured to store `Weather`

objects in an embedded database. Both of the applications defined in this multi-module project will use the DAOs defined in `simple-persist` to store data in an embedded database. The DAOs defined in this project understand and return the model objects defined in `simple-model`. `simple-persist` has a direct dependency on `simple-model` and it depends upon the Hibernate Annotations present on the model objects.

simple-webapp
The web application project contains two Spring MVC Controller implementations which use the `WeatherService` defined in `simple-weather` and the DAOs defined in `simple-persist`. `simple-webapp` has a direct dependency on `simple-weather` and `simple-persist`; it has a transitive dependency on `simple-model`.

simple-command
This module contains a simple command-line tool which can be used to query the Yahoo Weather feed. This project contains a class with a static `main()` method and interacts with the `WeatherService` defined in `simple-weather` and the DAOs defined in `simple-persist`. `simple-command` has a direct dependency on `simple-weather` and `simple-persist`; it has a transitive dependency on `simple-model`.

This chapter contains a contrived example simple enough to introduce in a book, yet complex enough to justify a set of five submodules. Our contrived example has a model project with five classes, a persistence library with two service classes, and a weather parsing library with five or six classes, but a real-world system might have a model project with a hundred objects, several persistence libraries, and service libraries spanning multiple departments. Although we've tried to make sure that the code contained in this example is straightforward enough to comprehend in a single sitting, we've also gone out of our way to build a modular project. You might be tempted to look at the examples in this chapter and walk away with the idea that Maven encourages too much complexity given that our model project has only five classes. Although using Maven does suggest a certain level of modularity, do realize that we've gone out of our way to complicate our simple example projects for the purpose of demonstrating Maven's multi-module features.

7.1.3 Technology Used in this Example

This chapter's example involves some technology which, while popular, is not directly related to Maven. These technologies are the Spring Framework and Hibernate. The Spring Framework is an Inversion of Control (IoC) container and a set of frameworks that aim to simplify interaction with various J2EE libraries. Using the Spring Framework as a foundational framework for application development gives you access to a number of helpful abstractions that can take much of the meddlesome busywork out of dealing with persistence frameworks like Hibernate or iBatis or enterprise APIs like JDBC, JNDI, and JMS. The Spring Framework has grown in popularity over the past few years as a replacement for the heavy weight enterprise standards coming out of Sun Microsystems. Hibernate is a widely used Object-Relational Mapping framework which allows you to interact with a relational database as if it were a collection of Java objects. This example focuses on building a simple web application and a command-line application that use the Spring Framework to expose a set of reusable components to applications and which also use Hibernate to persist weather data in an embedded database.

We've decided to include references to these frameworks to demonstrate how one would construct projects using these technologies when using Maven. Although we make brief efforts to introduce these technologies throughout this chapter, we will not go out of our way to fully explain these technologies. For more information about the Spring Framework, please see the project's web site at http://www.springsource.org/documentation. For more information about Hibernate and Hibernate Annotations, please see the project's web site at http://www.hibernate.org. This chapter uses Hyper Structured Query Language Database (HSQLDB) as an embedded database; for more information about this database, see the project's web site at http://hsqldb.org.

7.2 The Simple Parent Project

This `simple-parent` project has a `pom.xml` that references five submodules: `simple-command`, `simple-model`, `simple-weather`, `simple-persist`, and `simple-webapp`. The top-level `pom.xml` is shown in simple-parent Project POM.

simple-parent Project POM

```
<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>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0</version>
    <name>Multi-Spring Chapter Simple Parent Project</name>

    <modules>
        <module>simple-command</module>
        <module>simple-model</module>
        <module>simple-weather</module>
        <module>simple-persist</module>
        <module>simple-webapp</module>
    </modules>

    <build>
        <pluginManagement>
```

```
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <source>1.5</source>
                        <target>1.5</target>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Note
If you are already familiar with Maven POMs, you might notice that this top-level POM does not define a dependencyManagement element. The dependencyManagement element allows you to define dependency versions in a single, top-level POM, and it is introduced in Chapter 8.

Note the similarities of this parent POM to the parent POM defined in simple-parent Project POM. The only real difference between these two POMs is the list of submodules. Where that example only listed two submodules, this parent POM lists five submodules. The next few sections explore each of these five submodules in some detail. Because our example uses Java annotations, we've configured the compiler to target the Java 5 JVM.

7.3 The Simple Model Module

The first thing most enterprise projects need is an object model. An object model captures the core set of domain objects in any system. A banking system might have an object model which consists of `Account`, `Customer`, and `Transaction` objects, or a system to capture and communicate sports scores might have a `Team` and a `Game` object. Whatever it is, there's a good chance that you've modeled the concepts in your system in an object model. It is a common practice in Maven projects to separate this project into a separate project which is widely referenced. In this system we are capturing each query to the Yahoo Weather feed with a `Weather` object which references four other objects. Wind direction, chill, and speed are stored in a `Wind` object. Location data including the zip code, city, region, and country are stored in a `Location` class. Atmospheric conditions such as the humidity, maximum visibility, barometric pressure, and whether the pressure is rising or falling are stored in an `Atmosphere` class. A textual description of conditions, the temperature, and the date of the observation is stored in a `Condition` class.

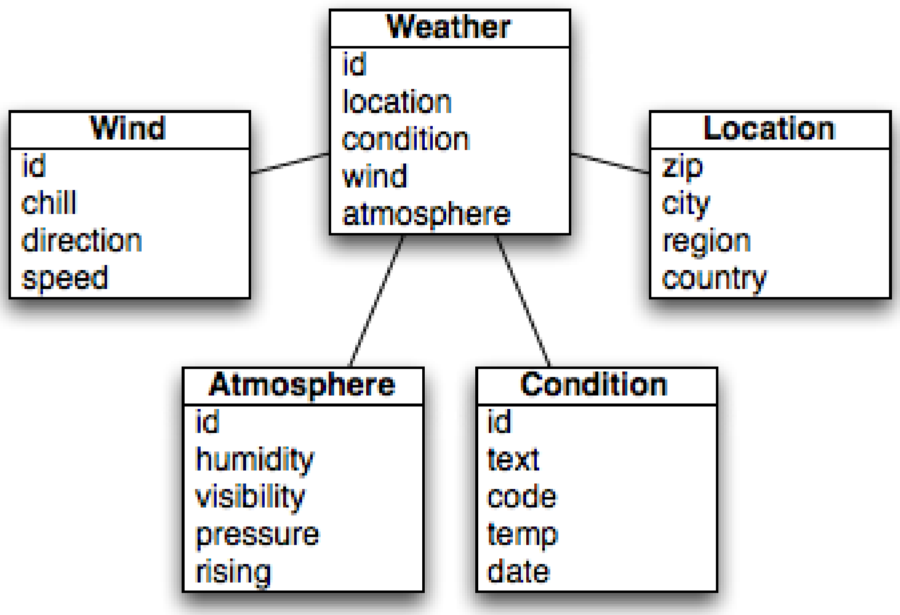

Figure 7.2: Simple Object Model for Weather Data

The `pom.xml` file for this simple object model contains one dependency that bears some explanation. Our object model is annotated with Hibernate Annotations. We use these annotations to map the model objects in this model to tables in a relational database. The dependency is `org.hibernate:hibernate-annotations:3.3.0.ga`. Take a look at the `pom.xml` shown in simple-model pom.xml, and then look at the next few examples for some illustrations of these annotations.

simple-model pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.sonatype.mavenbook.multispring</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-model</artifactId>
    <packaging>jar</packaging>

    <name>Simple Object Model</name>

    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-commons-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
    </dependencies>
</project>
```

In `src/main/java/org/sonatype/mavenbook/weather/model`, we have `Weather.java`, which contains the annotated `Weather` model object. The `Weather` object is a simple Java bean. This means that we have private member variables like `id`, `location`, `condition`, `wind`, `atmosphere`, and `date` exposed with public getter and setter methods that adhere to the following pattern: if a property is named `name`, there will be a public no-arg getter method named `getName()`, and there will be a one-argument setter named `setName(String name)`. Although we show the getter and setter methods for the `id` property, we've omitted most of the getters and setters for most of the other properties to save a few trees. See Annotated Weather Model Object.

Annotated Weather Model Object

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

import java.util.Date;

@Entity
```

```
@NamedQueries({
  @NamedQuery(name="Weather.byLocation",
              query="from Weather w where w.location = :location")
})
public class Weather {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;

  @ManyToOne(cascade=CascadeType.ALL)
  private Location location;

  @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
  private Condition condition;

  @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
  private Wind wind;

  @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
  private Atmosphere atmosphere;

  private Date date;

  public Weather() {}

  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }

  // All getter and setter methods omitted...
}
```

In the `Weather` class, we are using Hibernate annotations to provide guidance to the `simple-persist` project. These annotations are used by Hibernate to map an object to a table in a relational database. Although a full explanation of Hibernate annotations is beyond the scope of this chapter, here is a brief explanation for the curious. The `@Entity` annotation marks this class as a persistent entity. We've omitted the `@Table` annotation on this class, so Hibernate is going to use the name of the class as the name of the table to map `Weather` to. The `@NamedQueries` annotation defines a query that is used by the `WeatherDAO` in `simple-persist`. The query language in the `@NamedQuery` annotation is written in something called Hibernate Query Language (HQL). Each member variable is annotated with annotations that define the type of column and any relationships implied by that column:

Id
: The `id` property is annotated with `@Id`. This marks the `id` property as the property that contains the primary key in a database table. The `@GeneratedValue` controls how new primary key

values are generated. In the case of id, we're using the IDENTITY GenerationType, which will use the underlying database's identity generation facilities.

Location
Each Weather object instance corresponds to a Location object. A Location object represents a zip code, and the @ManyToOne makes sure that Weather objects that point to the same Location object reference the same instance. The cascade attribute of the @ManyToOne makes sure that we persist a Location object every time we persist a Weather object.

Condition, Wind, Atmosphere
Each of these objects is mapped as a @OneToOne with the CascadeType of ALL. This means that every time we save a Weather object, we'll be inserting a row into the Weather table, the Condition table, the Wind table, and the Atmosphere table.

Date
Date is not annotated. This means that Hibernate is going to use all of the column defaults to define this mapping. The column name is going to be date, and the column type is going to be the appropriate time to match the Date object.

Note
If you have a property you wish to omit from a table mapping, you would annotate that property with @Transient.

Next, take a look at one of the secondary model objects, Condition, shown in simple-model's Condition Model Object.. This class also resides in src/main/java/org/sonatype/mavenbook/weather/model.

simple-model's Condition Model Object.

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

@Entity
public class Condition {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;

  private String text;
  private String code;
  private String temp;
  private String date;
```

```
  @OneToOne(cascade=CascadeType.ALL)
  @JoinColumn(name="weather_id", nullable=false)
  private Weather weather;

  public Condition() {}

  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }

  // All getter and setter methods omitted...
}
```

The `Condition` class resembles the `Weather` class. It is annotated as an `@Entity`, and it has similar annotations on the `id` property. The `text`, `code`, `temp`, and `date` properties are all left with the default column settings, and the `weather` property is annotated with a `@OneToOne` annotation and another annotation that references the associated `Weather` object with a foreign key column named `weather_id`.

7.4 The Simple Weather Module

The next module we're going to examine could be considered something of a "service." The Simple Weather module is the module that contains all of the logic necessary to retrieve and parse the data from the Yahoo Weather RSS feed. Although Simple Weather contains three Java classes and one JUnit test, it is going to present a single component, `WeatherService`, to both the Simple Web Application and the Simple Command-Line Utility. Very often an enterprise project will contain several API modules that contain critical business logic or logic that interacts with external systems. A banking system might have a module that retrieves and parses data from a third-party data provider, and a system to display sports scores might interact with an XML feed that presents real-time scores for basketball or soccer. In simple-weather Module POM, this module encapsulates all of the network activity and XML parsing that is involved in the interaction with Yahoo Weather. Other modules can depend on this module and simply call out to the `retrieveForecast()` method on `WeatherService`, which takes a zip code as an argument and which returns a `Weather` object.

simple-weather Module POM

```
<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>
    <parent>
```

```
        <groupId>org.sonatype.mavenbook.multispring</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-weather</artifactId>
    <packaging>jar</packaging>

    <name>Simple Weather API</name>

    <dependencies>
        <dependency>
            <groupId>org.sonatype.mavenbook.multispring</groupId>
            <artifactId>simple-model</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
        <dependency>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-io</artifactId>
            <version>1.3.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

The `simple-weather` POM extends the `simple-parent` POM, sets the packaging to `jar`, and then adds the following dependencies:

org.sonatype.mavenbook.multispring:simple-model:1.0
`simple-weather` parses the Yahoo Weather RSS feed into a `Weather` object. It has a direct dependency on `simple-model`.

log4j:log4j:1.2.14
: simple-weather uses the Log4J library to print log messages.

dom4j:dom4j:1.6.1 and jaxen:jaxen:1.1.1
: Both of these dependencies are used to parse the XML returned from Yahoo Weather.

org.apache.commons:commons-io:1.3.2 (scope=test)
: This test-scoped dependency is used by the YahooParserTest.

Next is the WeatherService class, shown in WeatherService Class. This class is going to look very similar to the WeatherService class from The WeatherService Class. Although the WeatherService is the same, there are some subtle differences in this chapter's example. This version's retrieveForecast() method returns a Weather object, and the formatting is going to be left to the applications that call WeatherService. The other major change is that the YahooRetriever and YahooParser are both bean properties of the WeatherService bean.

WeatherService Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherService {

    private YahooRetriever yahooRetriever;
    private YahooParser yahooParser;

    public WeatherService() {
    }

    public Weather retrieveForecast(String zip) throws Exception {
        // Retrieve Data
        InputStream dataIn = yahooRetriever.retrieve(zip);

        // Parse DataS
        Weather weather = yahooParser.parse(zip, dataIn);

        return weather;
    }

    public YahooRetriever getYahooRetriever() {
        return yahooRetriever;
    }

    public void setYahooRetriever(YahooRetriever yahooRetriever) {
```

```
        this.yahooRetriever = yahooRetriever;
    }

    public YahooParser getYahooParser() {
        return yahooParser;
    }

    public void setYahooParser(YahooParser yahooParser) {
        this.yahooParser = yahooParser;
    }

}
```

Finally, in this project we have an XML file that is used by the Spring Framework to create something called an `ApplicationContext`. First, some explanation: both of our applications, the web application and the command-line utility, need to interact with the `WeatherService` class, and they both do so by retrieving an instance of this class from a Spring `ApplicationContext` using the name `weatherService`. Our web application uses a Spring MVC controller that is associated with an instance of `WeatherService`, and our command-line utility loads the `WeatherService` from an `ApplicationContext` in a static `main()` function. To encourage reuse, we've included an `applicationContext-weather.xml` file in `src/main/resources`, which is available on the classpath. Modules that depend on the `simple-weather` module can load this application context using the `ClasspathXmlApplicationContext` in the Spring Framework. They can then reference a named instance of the `WeatherService` named `weatherService`.

Spring Application Context for the simple-weather Module

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
       default-lazy-init="true">

    <bean id="weatherService"
          class="org.sonatype.mavenbook.weather.WeatherService">
        <property name="yahooRetriever" ref="yahooRetriever"/>
        <property name="yahooParser" ref="yahooParser"/>
    </bean>

    <bean id="yahooRetriever"
          class="org.sonatype.mavenbook.weather.YahooRetriever"/>

    <bean id="yahooParser"
          class="org.sonatype.mavenbook.weather.YahooParser"/>
```

```
</beans>
```

This document defines three beans: yahooParser, yahooRetriever, and weatherService. The weatherService bean is an instance of WeatherService, and this XML document populates the yahooParser and yahooRetriever properties with references to the named instances of the corresponding classes. Think of this applicationContext-weather.xml file as defining the architecture of a subsystem in this multi-module project. Projects like simple-webapp and simple-command can reference this context and retrieve an instance of WeatherService which already has relationships to instances of YahooRetriever and YahooParser.

7.5 The Simple Persist Module

This module defines two very simple Data Access Objects (DAOs). A DAO is an object that provides an interface for persistence operations. In an application that makes use of an Object-Relational Mapping (ORM) framework such as Hibernate, DAOs are usually defined around objects. In this project, we are defining two DAO objects: WeatherDAO and LocationDAO. The WeatherDAO class allows us to save a Weather object to a database and retrieve a Weather object by id, and to retrieve Weather objects that match a specific Location. The LocationDAO has a method that allows us to retrieve a Location object by zip code. First, let's take a look at the simple-persist POM in simple-persist POM.

simple-persist POM

```
<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>
    <parent>
        <groupId>org.sonatype.mavenbook.multispring</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-persist</artifactId>
    <packaging>jar</packaging>

    <name>Simple Persistence API</name>

    <dependencies>
        <dependency>
            <groupId>org.sonatype.mavenbook.multispring</groupId>
            <artifactId>simple-model</artifactId>
```

```
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
            <version>3.2.5.ga</version>
            <exclusions>
                <exclusion>
                    <groupId>javax.transaction</groupId>
                    <artifactId>jta</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-commons-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring</artifactId>
            <version>2.0.7</version>
        </dependency>
    </dependencies>
</project>
```

This POM file references `simple-parent` as a parent POM, and it defines a few dependencies. The dependencies listed in `simple-persist`'s POM are:

org.sonatype.mavenbook.multispring:simple-model:1.0
: Just like the `simple-weather` module, this persistence module references the core model objects defined in `simple-model`.

org.hibernate:hibernate:3.2.5.ga
: We define a dependency on Hibernate version 3.2.5.ga, but notice that we're excluding a depen-

dency of Hibernate. We're doing this because the `javax.transaction:jta` dependency is not available in the public Maven repository. This dependency happens to be one of those Sun dependencies that has not yet made it into the free central Maven repository. To avoid an annoying message telling us to go download these nonfree dependencies, we simply exclude this dependency from Hibernate.

`javax.servlet:servlet-api:2.4`
Since this project contains a Servlet, we need to include the Servlet API version 2.4.

`org.springframework:spring:2.0.7`
This includes the entire Spring Framework as a dependency. It is generally a good practice to depend on only the components of Spring you happen to be using. The Spring Framework project has been nice enough to create focused artifacts such as `spring-hibernate3`.

Why depend on Spring? When it comes to Hibernate integration, Spring allows us to leverage helper classes such as `HibernateDaoSupport`. For an example of what is possible with the help of `HibernateDaoSupport`, take a look at the code for the `WeatherDAO` in simple-persist's WeatherDAO Class.

simple-persist's WeatherDAO Class

```
package org.sonatype.mavenbook.weather.persist;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherDAO extends HibernateDaoSupport { ❶

  public WeatherDAO() {}

  public void save(Weather weather) { ❷
    getHibernateTemplate().save( weather );
  }

  public Weather load(Integer id) { ❸
    return (Weather) getHibernateTemplate().load( Weather.class, id);
  }

  @SuppressWarnings("unchecked")
```

```
  public List<Weather> recentForLocation( final Location location ) {
    return (List<Weather>) getHibernateTemplate().execute(
        new HibernateCallback() { ❹
          public Object doInHibernate(Session session) {
              Query query =
                getSession().getNamedQuery("Weather.byLocation");
              query.setParameter("location", location);
              return new ArrayList<Weather>( query.list() );
        }
    });
  }
}
```

That's it. No really, you are done writing a class that can insert new rows, select by primary key, and find all rows in Weather that join to an id in the Location table. Clearly, we can't stop this book and insert the five hundred pages it would take to get you up to speed on the intricacies of Hibernate, but we can do some very quick explanation:

❶ This class extends `HibernateDaoSupport`. What this means is that the class is going to be associated with a Hibernate `SessionFactory` which it is going to use to create Hibernate `Session` objects. In Hibernate, every operation goes through a `Session` object, a `Session` mediates access to the underlying database and takes care of managing the connection to the JDBC `DataSource`. Extending `HibernateDaoSupport` also means that we can access the `HibernateTemplate` using `getHibernateTemplate()`. For an example of what can be done with the `HibernateTemplate`...

❷ The `save()` method takes an instance of `Weather` and calls the `save()` method on a `HibernateTemplate`. The `HibernateTemplate` simplifies calls to common Hibernate operations and converts any database specific exceptions to runtime exceptions. Here we call out to `save()` which inserts a new record into the `Weather` table. Alternatives to `save()` are `update()` which updates an existing row, or `saveOrUpdate()` which would either save or update depending on the presence of a non-null `id` property in Weather.

❸ The `load()` method, once again, is a one-liner that just calls a method on an instance of `HibernateTemplate`. `load()` on `HibernateTemplate` takes a `Class` object and a `Serializable` object. In this case, the `Serializable` corresponds to the `id` value of the `Weather` object to load.

❹ This last method `recentForLocation()` calls out to a `NamedQuery` defined in the `Weather` model object. If you can think back that far, the `Weather` model object defined a named query `"Weather.byLocation"` with a query of `"from Weather w where w.location =:location"`. We're loading this `NamedQuery` using a reference to a Hibernate `Session` object inside a `HibernateCallback` which is executed by the `execute()` method on `HibernateTemplate`. You can see in this method that we're populating the named parameter `location` with the parameter passed in to the `recentForLocation()` method.

Now is a good time for some clarification. `HibernateDaoSupport` and `HibernateTemplate` are classes from the Spring Framework. They were created by the Spring Framework to make writing Hibernate DAO objects painless. To support this DAO, we'll need to do some configuration in the `simple-persist` Spring `ApplicationContext` definition. The XML document shown in Spring Application Context for simple-persist is stored in `src/main/resources` in a file named `applicationContext-persist.xml`.

Spring Application Context for simple-persist

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
       default-lazy-init="true">

  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation. ↩
        AnnotationSessionFactoryBean">
    <property name="annotatedClasses">
      <list>
        <value>org.sonatype.mavenbook.weather.model.Atmosphere</value>
        <value>org.sonatype.mavenbook.weather.model.Condition</value>
        <value>org.sonatype.mavenbook.weather.model.Location</value>
        <value>org.sonatype.mavenbook.weather.model.Weather</value>
        <value>org.sonatype.mavenbook.weather.model.Wind</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.show_sql">false</prop>
        <prop key="hibernate.format_sql">true</prop>
        <prop key="hibernate.transaction.factory_class">
          org.hibernate.transaction.JDBCTransactionFactory
        </prop>
        <prop key="hibernate.dialect">
          org.hibernate.dialect.HSQLDialect
        </prop>
        <prop key="hibernate.connection.pool_size">0</prop>
        <prop key="hibernate.connection.driver_class">
          org.hsqldb.jdbcDriver
        </prop>
        <prop key="hibernate.connection.url">
          jdbc:hsqldb:data/weather;shutdown=true
        </prop>
        <prop key="hibernate.connection.username">sa</prop>
        <prop key="hibernate.connection.password"></prop>
        <prop key="hibernate.connection.autocommit">true</prop>
        <prop key="hibernate.jdbc.batch_size">0</prop>
```

```
      </props>
    </property>
  </bean>

  <bean id="locationDAO"
        class="org.sonatype.mavenbook.weather.persist.LocationDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <bean id="weatherDAO"
        class="org.sonatype.mavenbook.weather.persist.WeatherDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
</beans>
```

In this application context, we're accomplishing a few things. The `sessionFactory` bean is the bean from which the DAOs retrieve Hibernate `Session` objects. This bean is an instance of `AnnotationSessionFactoryBean` and is supplied with a list of `annotatedClasses`. Note that the list of annotated classes is the list of classes defined in our `simple-model` module. Next, the `sessionFactory` is configured with a set of Hibernate configuration properties (`hibernateProperties`). In this example, our Hibernate properties define a number of settings:

hibernate.dialect
: This setting controls how SQL is to be generated for our database. Since we are using the HSQLDB database, our database dialect is set to `org.hibernate.dialect.HSQLDialect`. Hibernate has dialects for all major databases such as Oracle, MySQL, Postgres, and SQL Server.

hibernate.connection.*
: In this example, we're configuring the JDBC connection properties from the Spring configuration. Our applications are configured to run against a HSQLDB in the `./data/weather` directory. In a real enterprise application, it is more likely you would use something like JNDI to externalize database configuration from your application's code.

Lastly, in this bean definition file, both of the `simple-persist` DAO objects are created and given a reference to the `sessionFactory` bean just defined. Just like the Spring application context in `simple-weather`, this `applicationContext-persist.xml` file defines the architecture of a submodule in a larger enterprise design. If you were working with a larger collection of persistence classes, you might find it useful to capture them in an application context which is separate from your application.

There's one last piece of the puzzle in `simple-persist`. Later in this chapter, we're going to use `hibernate.cfg.xml` in `src/main/resources`. The purpose of this file (which duplicates some of the configuration in `applicationContext-persist.xml`) is to allow us to leverage the Maven

Hibernate3 plugin to generate Data Definition Language (DDL) from nothing more than our annotations. See simple-persist hibernate.cfg.xml.

simple-persist hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- SQL dialect -->
    <property name="dialect">
      org.hibernate.dialect.HSQLDialect
    </property>

    <!-- Database connection settings -->
    <property name="connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="connection.url">jdbc:hsqldb:data/weather</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">true</property>

    <!-- JDBC connection pool (use the built-in one) -->
    <property name="connection.pool_size">1</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache  -->
    <property name="cache.provider_class">
            org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- disable batching so HSQLDB will propagate errors correctly. -->
    <property name="jdbc.batch_size">0</property>

    <!-- List all the mapping documents we're using -->
    <mapping class="org.sonatype.mavenbook.weather.model.Atmosphere"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Condition"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Location"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Weather"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Wind"/>
```

```
  </session-factory>
</hibernate-configuration>
```

The contents of Spring Application Context for simple-persist and simple-parent Project POM are redundant. While the Spring Application Context XML is going to be used by the web application and the command-line application, the `hibernate.cfg.xml` exists only to support the Maven Hibernate3 plugin. Later in this chapter, we'll see how to use this `hibernate.cfg.xml` and the Maven Hibernate3 plugin to generate a database schema based on the annotated object model defined in `simple-model`. This `hibernate.cfg.xml` file is the file that will configure the JDBC connection properties and enumerate the list of annotated model classes for the Maven Hibernate3 plugin.

7.6 The Simple Web Application Module

The web application is defined in a `simple-webapp` project. This simple web application project is going to define two Spring MVC Controllers: `WeatherController` and `simple-weather` and the `applicationContext-persist.xml` file in `simple-persist`. The component architecture of this simple web application is shown in Figure 7.3.

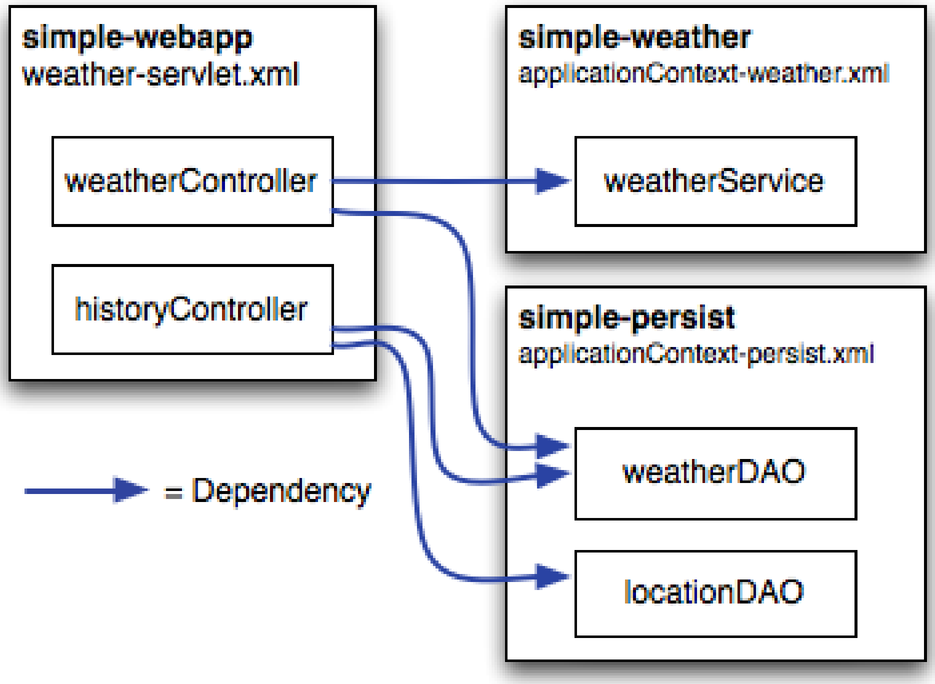

Figure 7.3: Spring MVC Controllers Referencing Components in simple-weather and simple-persist.

The POM for `simple-webapp` is shown in POM for simple-webapp.

POM for simple-webapp

```
<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>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
```

```
  <name>Simple Web Application</name>
  <dependencies>
    <dependency> ❶
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin> ❷
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <dependencies> ❸
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId> ❹
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <components>
```

```
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation> ❺
          </component>
        </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

As this book progresses and the examples become more and more substantial, you'll notice that the `pom.xml` begins to take on some weight. In this POM, we're configuring four dependencies and two plugins. Let's go through this POM in detail and dwell on some of the important configuration points:

1. This `simple-webapp` project defines four dependencies: the Servlet 2.4 specification, the simple-weather service library, the simple-persist persistence library, and the entire Spring Framework 2.0.7.

2. The Maven Jetty plugin couldn't be easier to add to this project; we simply add a `plugin` element that references the appropriate `groupId` and `artifactId`. The fact that this plugin is so trivial to configure means that the plugin developers did a good job of providing adequate defaults that don't need to be overridden in most cases. If you did need to override the configuration of the Jetty plugin, you would do so by providing a `configuration` element.

3. In our build configuration, we're going to be configuring the Maven Hibernate3 Plugin to hit an embedded HSQLDB instance. For the Maven Hibernate 3 plugin to successfully connect to this database using JDBC, the plugin will need to reference the HSQLDB JDBC driver on the classpath. To make a dependency available for a plugin, we add a dependency declaration right inside the plugin declaration. In this case, we're referencing hsqldb:hsqldb:1.8.0.7. The Hibernate plugin also needs the JDBC driver to create the database, so we have also added this dependency to its configuration.

4. The Maven Hibernate plugin is when this POM starts to get interesting. In the next section, we're going to run the `hbm2ddl` goal to generate a HSQLDB database. In this `pom.xml`, we're including a reference to version 2.0 of the `hibernate3-maven-plugin` hosted by the Codehaus Mojo plugin.

5 The Maven Hibernate3 plugin has different ways to obtain Hibernate mapping information that are appropriate for different usage scenarios of the Hibernate3 plugin. If you were using Hibernate Mapping XML (`.hbm.xml`) files, and you wanted to generate model classes using the `hbm2java` goal, you would set your implementation to `configuration`. If you were using the Hibernate3 plugin to reverse engineer a database to produce `.hbm.xml` files and model classes from an existing database, you would use an implementation of `jdbcconfiguration`. In this case, we're simply using an existing annotated object model to generate a database. In other words, we have our Hibernate mapping, but we don't yet have a database. In this usage scenario, the appropriate implementation value is `annotationconfiguration`. The Maven Hibernate3 plugin is discussed in more detail in the later section Section 7.7.

Next, we turn our attention to the two Spring MVC controllers that will handle all of the requests. Both of these controllers reference the beans defined in `simple-weather` and `simple-persist`.

simple-webapp WeatherController

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;
import org.sonatype.mavenbook.weather.WeatherService;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class WeatherController implements Controller {

  private WeatherService weatherService;
  private WeatherDAO weatherDAO;

  public ModelAndView handleRequest(HttpServletRequest request,
                                    HttpServletResponse response)
                                  throws Exception {

    String zip = request.getParameter("zip");
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save(weather);
    return new ModelAndView("weather", "weather", weather);
  }

  public WeatherService getWeatherService() {
    return weatherService;
  }

  public void setWeatherService(WeatherService weatherService) {
    this.weatherService = weatherService;
  }
```

```
  public WeatherDAO getWeatherDAO() {
    return weatherDAO;
  }

  public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
  }
}
```

`WeatherController` implements the Spring MVC Controller interface that mandates the presence of a `handleRequest()` method with the signature shown in the example. If you look at the meat of this method, you'll see that it invokes the `retrieveForecast()` method on the `weatherService` instance variable. Unlike the previous chapter, which had a Servlet that instantiated the `WeatherService` class, the `WeatherController` is a bean with a `weatherService` property. The Spring IoC container is responsible for wiring the controller to the `weatherService` component. Also notice that we're not using the `WeatherFormatter` in this Spring controller implementation; instead, we're passing the `Weather` object returned by `retrieveForecast()` to the constructor of `ModelAndView`. This `ModelAndView` class is going to be used to render a Velocity template, and this template will have references to a `${weather}` variable. The `weather.vm` template is stored in `src/main/webapp/WEB-INF/vm` and is shown in weather.vm Template Rendered by WeatherController.

In the `WeatherController`, before we render the output of the forecast, we pass the `Weather` object returned by the `WeatherService` to the `save()` method on `WeatherDAO`. Here we are saving this `Weather` object—using Hibernate—to an HSQLDB database. Later, in `HistoryController`, we will see how we can retrieve a history of weather forecasts that were saved by the `WeatherController`.

weather.vm Template Rendered by WeatherController

```
<b>Current Weather Conditions for:
${weather.location.city}, ${weather.location.region},
${weather.location.country}</b><br/>

<ul>
<li>Temperature: ${weather.condition.temp}</li>
<li>Condition: ${weather.condition.text}</li>
<li>Humidity: ${weather.atmosphere.humidity}</li>
<li>Wind Chill: ${weather.wind.chill}</li>
<li>Date: ${weather.date}</li>
</ul>
```

The syntax for this Velocity template is straightforward: variables are referenced using `${}` notation. The expression between the curly braces references a property, or a property of a property on the `weather` variable, which was passed to this template by the WeatherController.

The HistoryController is used to retrieve recent forecasts that have been requested by the WeatherController. Whenever we retrieve a forecast from the WeatherController, that controller saves the Weather object to the database via the WeatherDAO. WeatherDAO then uses Hibernate to dissect the Weather object into a series of rows in a set of related database tables. The HistoryController is shown in simple-web HistoryController.

simple-web HistoryController

```
package org.sonatype.mavenbook.web;

import java.util.*;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.sonatype.mavenbook.weather.model.*;
import org.sonatype.mavenbook.weather.persist.*;

public class HistoryController implements Controller {

  private LocationDAO locationDAO;
  private WeatherDAO weatherDAO;

  public ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    String zip = request.getParameter("zip");
    Location location = locationDAO.findByZip(zip);
    List<Weather> weathers = weatherDAO.recentForLocation( location );

    Map<String,Object> model = new HashMap<String,Object>();
    model.put( "location", location );
    model.put( "weathers", weathers );

    return new ModelAndView("history", model);
  }

  public WeatherDAO getWeatherDAO() {
    return weatherDAO;
  }

  public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
  }

  public LocationDAO getLocationDAO() {
    return locationDAO;
```

```
  }

  public void setLocationDAO(LocationDAO locationDAO) {
    this.locationDAO = locationDAO;
  }
}
```

The HistoryController is wired to two DAO objects defined in simple-persist. The DAOs are bean properties of the HistoryController: WeatherDAO and LocationDAO. The goal of the HistoryController is to retrieve a List of Weather objects which correspond to the zip parameter. When the WeatherDAO saves the Weather object to the database, it doesn't just store the zip code, it stores a Location object which is related to the Weather object in the simple-model. To retrieve a List of Weather objects, the HistoryController first retrieves the Location object that corresponds to the zip parameter. It does this by invoking the findByZip() method on LocationDAO.

Once the Location object has been retrieved, the HistoryController will then attempt to retrieve recent Weather objects that match the given Location. Once the List<Weather> has been retrieved, a HashMap is created to hold two variables for the history.vm Velocity template shown in history.vm Rendered by the HistoryController.

history.vm Rendered by the HistoryController

```
<b>
Weather History for: ${location.city}, ${location.region}, ${location. ↩
    country}
</b>
<br/>

#foreach( $weather in $weathers )
<ul>
<li>Temperature: $weather.condition.temp</li>
<li>Condition: $weather.condition.text</li>
<li>Humidity: $weather.atmosphere.humidity</li>
<li>Wind Chill: $weather.wind.chill</li>
<li>Date: $weather.date</li>
</ul>
#end
```

The history.vm template in src/main/webapp/WEB-INF/vm references the location variable to print out information about the location of the forecasts retrieved from the WeatherDAO. This template then uses a Velocity control structure, #foreach, to loop through each element in the weathers variable. Each element in weathers is assigned to a variable named weather and the template between #foreach and #end is rendered for each observation.

You've seen these `Controller` implementations, and you've seen that they reference other beans defined in `simple-weather` and `simple-persist`, they respond to HTTP requests, and they yield control to some mysterious templating system that knows how to render Velocity templates. All of this magic is configured in a Spring application context in `src/main/webapp/WEB-INF/weather-servlet.xml`. This XML configures the controllers and references other Spring-managed beans. It is loaded by a `ServletContextListener` which is also configured to load the `applicationContext-weather.xml` and `applicationContext-persist.xml` from the classpath. Let's take a closer look at the `weather-servlet.xml` shown in Spring Controller Configuration weather-servlet.xml.

Spring Controller Configuration weather-servlet.xml

```
<beans>
  <bean id="weatherController" ❶
        class="org.sonatype.mavenbook.web.WeatherController">
    <property name="weatherService" ref="weatherService"/>
    <property name="weatherDAO" ref="weatherDAO"/>
  </bean>

  <bean id="historyController"
        class="org.sonatype.mavenbook.web.HistoryController">
    <property name="weatherDAO" ref="weatherDAO"/>
    <property name="locationDAO" ref="locationDAO"/>
  </bean>

  <!-- you can have more than one handler defined -->
  <bean id="urlMapping"
        class="org.springframework.web.servlet.handler.
          SimpleUrlHandlerMapping">
    <property name="urlMap">
      <map>
        <entry key="/weather.x"> ❷
          <ref bean="weatherController" />
        </entry>
        <entry key="/history.x">
          <ref bean="historyController" />
        </entry>
      </map>
    </property>
  </bean>

  <bean id="velocityConfig" ❸
    class="org.springframework.web.servlet.view.velocity.
      VelocityConfigurer">
    <property name="resourceLoaderPath" value="/WEB-INF/vm/"/>
  </bean>

  <bean id="viewResolver" ❹
```

```
    class="org.springframework.web.servlet.view.velocity.
      VelocityViewResolver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".vm"/>
    <property name="exposeSpringMacroHelpers" value="true"/>
  </bean>
</beans>
```

1. The `weather-servlet.xml` defines the two controllers as Spring-managed beans. `weatherController` has two properties which are references to `weatherService` and `weatherDAO`. `historyController` references the beans `weatherDAO` and `locationDAO`. When this `ApplicationContext` is created, it is created in an environment that has access to the `ApplicationContexts` defined in both `simple-persist` and `simple-weather`. In web.xml for simple-webapp you will see how Spring is configured to merge components from multiple Spring configuration files.

2. The `urlMapping` bean defines the URL patterns which invoke the `WeatherController` and the `HistoryController`. In this example, we are using the `SimpleUrlHandlerMapping` and mapping `/weather.x` to `WeatherController` and `/history.x` to `HistoryController`.

3. Since we are using the Velocity templating engine, we will need to pass in some configuration options. In the `velocityConfig` bean, we are telling Velocity to look for all templates in the `/WEB-INF/vm` directory.

4. Last, the `viewResolver` is configured with the class `VelocityViewResolver`. There are a number of `ViewResolver` implementations in Spring from a standard ViewResolver to render JSP or JSTL pages to a resolver which can render Freemarker templates. In this example, we're configuring the Velocity templating engine and setting the default prefix and suffix which will be automatically appended to the names of the template passed to `ModelAndView`.

Finally, the `simple-webapp` project was a `web.xml` which provides the basic configuration for the web application. The `web.xml` file is shown in web.xml for simple-webapp.

web.xml for simple-webapp

```
<web-app id="simple-webapp" version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
               http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>Simple Web Application</display-name>
```

```
  <context-param> ❶
    <param-name>contextConfigLocation</param-name>
      <param-value>
        classpath:applicationContext-weather.xml
        classpath:applicationContext-persist.xml
      </param-value>
  </context-param>

  <context-param> ❷
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
  </context-param>

  <listener> ❸
    <listener-class>
      org.springframework.web.util.Log4jConfigListener
    </listener-class>
  </listener>

  <listener>
    <listener-class> ❹
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet> ❺
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping> ❻
    <servlet-name>weather</servlet-name>
    <url-pattern>*.x</url-pattern>
  </servlet-mapping>
</web-app>
```

❶ Here's a bit of magic which allows us to reuse the `applicationContext-weather.xml` and `applicationContext-persist.xml` in this project. The `contextConfigLocation` is used by the `ContextLoaderListener` to create an `ApplicationContext`. When the weather servlet is created, the `weather-servlet.xml` from Spring Controller Configuration weather-servlet.xml is going to be evaluated with the `ApplicationContext` created from this `contextConfigLocation`. In this way, you can define a set of beans in another project and you can reference these beans via the classpath. Since the `simple-persist` and `simple-`

weather JARs are going to be in WEB-INF/lib, all we do is use the classpath: prefix to reference these files. (Another option would have been to copy these files to /WEB-INF and reference them with something like /WEB-INF/applicationContext-persist.xml.)

2 The log4jConfigLocation is used to tell the Log4JConfigListener where to look for Log4J logging configuration. In this example, we tell Log4J to look in /WEB-INF/log4j.properties.

3 This makes sure that the Log4J system is configured when the web application starts. It is important to put this Log4JConfigListener before the ContextLoaderListener; otherwise, you may miss important logging messages which point to a problem preventing application startup. If you have a particularly large set of beans managed by Spring, and one of them happens to blow up on application startup, your application will fail. If you have logging initialized before Spring starts, you might have a chance to catch a warning or an error. If you don't have logging initialized before Spring starts up, you'll have no idea why your application refuses to start.

4 The ContextLoaderListener is essentially the Spring container. When the application starts, this listener will build an ApplicationContext from the contextConfigLocation parameter.

5 We define a Spring MVC DispatcherServlet with a name of weather. This will cause Spring to look for a Spring configuration file in /WEB-INF/weather-servlet.xml. You can have as many DispatcherServlets as you need; a DispatcherServlet can contain one or more Spring MVC Controller implementations.

6 All requests ending in .x will be routed to the weather servlet. Note that the .x extension has no particular meaning; it is an arbitrary choice and you can use whatever URL pattern you like.

7.7 Running the Web Application

To run the web application, you'll first need to build the entire multi-module project and then build the database using the Hibernate3 plugin. First, from the top-level simple-parent project directory, run mvn clean install:

```
$ mvn clean install
```

Running mvn clean install at the top-level of your multi-module project will install all of modules into your local Maven repository. You need to do this before building the database from the simple-webapp project.

> **Warning**
> This plugin version requires Java 6 to work.

To build the database from the simple-webapp project, run the following from the simple-webapp project's directory:

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] ------------------------------------------------------
[INFO] Building Multi-Spring Chapter Simple Web Application
[INFO]task-segment: [hibernate3:hbm2ddl]
[INFO] ------------------------------------------------------
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151  INFO org.hibernate.tool.hbm2ddl.SchemaExport - export  ←
    complete
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
```

Once you've done this, there should be a ${basedir}/data directory which will contain the HSQLDB database. You can then start the web application with:

```
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] ------------------------------------------------------
[INFO] Building Multi-Spring Chapter Simple Web Application
[INFO]task-segment: [jetty:run]
[INFO] ------------------------------------------------------
[INFO] Preparing jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project:
Multi-Spring Chapter Simple Web Application
...
[INFO] Context path = /simple-webapp
[INFO] Tmp directory =  determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] Starting jetty 6.1.7 ...
2008-03-25 10:28:03.639::INFO:  jetty-6.1.7
```

```
...
2147 INFO  DispatcherServlet  - FrameworkServlet 'weather': \
initialization completed in 1654 ms
2008-03-25 10:28:06.341::INFO:  Started SelectChannelConnector@0 ←
    .0.0.0:8080
[INFO] Started Jetty Server
```

Once Jetty is started, you can load http://localhost:8080/simple-webapp/weather.x?zip=60202 and you should see the weather for Evanston, IL in your web browser. Change the ZIP code and you should be able to get your own weather report.

```
Current Weather Conditions for: Evanston, IL, US

* Temperature: 42
* Condition: Partly Cloudy
* Humidity: 55
* Wind Chill: 34
* Date: Tue Mar 25 10:29:45 CDT 2008
```

7.8 The Simple Command Module

The `simple-command` project is a command-line version of the `simple-webapp`. It is a utility that relies on the same dependencies: `simple-persist` and `simple-weather`. Instead of interacting with this application via a web browser, you would run the `simple-command` utility from the command line.

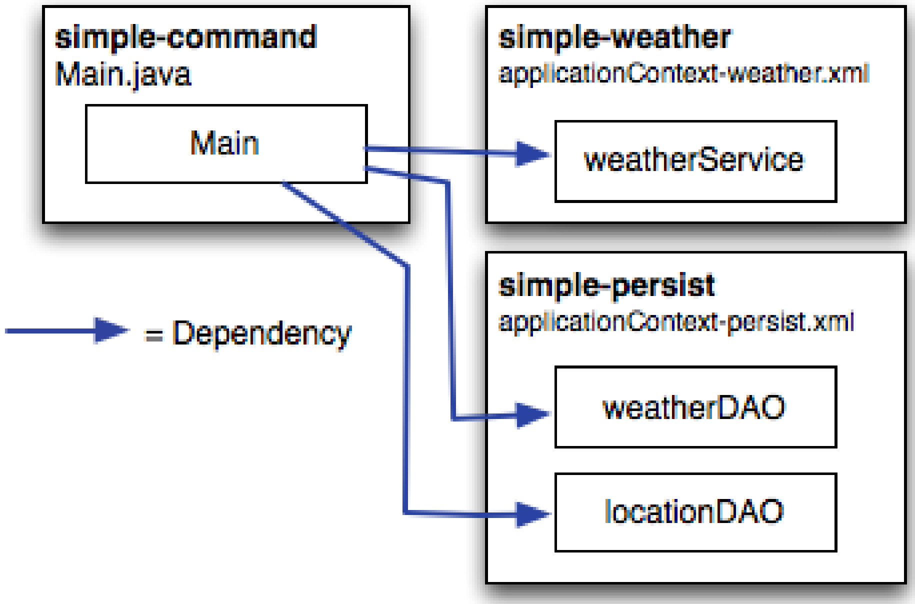

Figure 7.4: Command Line Application Referencing simple-weather and simple-persist

POM for simple-command

```
<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>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Simple Command Line Tool</name>

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

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
```

```
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
  </dependencies>
</project>
```

This POM creates a JAR file which will contain the `org.sonatype.mavenbook.weather.Main` class shown in The Main Class for simple-command. In this POM we configure the Maven Assembly plugin to use a built-in assembly descriptor named `jar-with-dependencies` which creates a single JAR file containing all the bytecode a project needs to execute, including the bytecode from the project you are building and all the bytecode from libraries your project depends upons.

The Main Class for simple-command

```
package org.sonatype.mavenbook.weather;

import java.util.List;

import org.apache.log4j.PropertyConfigurator;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.LocationDAO;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;

public class Main {

    private WeatherService weatherService;
    private WeatherDAO weatherDAO;
    private LocationDAO locationDAO;
```

```
    public static void main(String[] args) throws Exception {
        // Configure Log4J
        PropertyConfigurator.configure(
          Main.class.getClassLoader().getResource("log4j.properties"));

        // Read the zip code from the Command-line
        // (if none supplied, use 60202)
        String zipcode = "60202";
        try {
            zipcode = args[0];
        } catch (Exception e) {
        }

        // Read the Operation from the Command-line
        // (if none supplied use weather)
        String operation = "weather";
        try {
            operation = args[1];
        } catch (Exception e) {
        }

        // Start the program
        Main main = new Main(zipcode);

        ApplicationContext context =
          new ClassPathXmlApplicationContext(
            new String[] { "classpath:applicationContext-weather.xml",
              "classpath:applicationContext-persist.xml" });
        main.weatherService =
          (WeatherService) context.getBean("weatherService");
        main.locationDAO = (LocationDAO) context.getBean("locationDAO");
        main.weatherDAO = (WeatherDAO) context.getBean("weatherDAO");
        if( operation.equals("weather")) {
            main.getWeather();
        } else {
            main.getHistory();
        }
    }

    private String zip;

    public Main(String zip) {
        this.zip = zip;
    }

    public void getWeather() throws Exception {
        Weather weather = weatherService.retrieveForecast(zip);
        weatherDAO.save( weather );
```

```
        System.out.print(new WeatherFormatter().formatWeather(weather));
    }

    public void getHistory() throws Exception {
        Location location = locationDAO.findByZip(zip);
        List<Weather> weathers = weatherDAO.recentForLocation(location);
        System.out.print(
          new WeatherFormatter().formatHistory(location, weathers));
    }
}
```

The `Main` class has a reference to `WeatherDAO`, `LocationDAO`, and `WeatherService`. The static `main()` method in this class:

- Reads the zip code from the first command line argument
- Reads the operation from the second command line argument. If the operation is "weather", the latest weather will be retrieved from the web service. If the operation is "history", the program will fetch historical weather records from the local database.
- Loads a Spring `ApplicationContext` using two XML files loaded from `simple-persist` and `simple-weather`
- Creates an instance of `Main`
- Populates the `weatherService`, `weatherDAO`, and `locationDAO` with beans from the Spring `ApplicationContext`
- Runs the appropriate method `getWeather()` or `getHistory()`, depending on the specified operation

In the web application we use Spring `VelocityViewResolver` to render a Velocity template. In the stand-alone implementation, we need to write a simple class which renders our weather data with a Velocity template. WeatherFormatter Renders Weather Data using a Velocity Template is a listing of the `WeatherFormatter`, a class with two methods that render the weather report and the weather history.

WeatherFormatter Renders Weather Data using a Velocity Template

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;
import java.util.List;
```

```
import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String formatWeather(Weather weather) throws Exception {
        log.info( "Formatting Weather Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                                   getResourceAsStream("weather.vm"));
        VelocityContext context = new VelocityContext();
        context.put("weather", weather );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }

    public String formatHistory(Location location, List<Weather> weathers)
        throws Exception {
        log.info( "Formatting History Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                                   getResourceAsStream("history.vm"));
        VelocityContext context = new VelocityContext();
        context.put("location", location );
        context.put("weathers", weathers );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }
}
```

The `weather.vm` template simply prints the zip code's city, country, and region as well as the current temperature. The `history.vm` template prints the location and then iterates through the weather records stored in the local database. Both of these templates are in `${basedir}/src/main/resources`.

The weather.vm Velocity Template

```
****************************************
Current Weather Conditions for:
${weather.location.city},
```

```
${weather.location.region},
${weather.location.country}
****************************************

* Temperature: ${weather.condition.temp}
* Condition: ${weather.condition.text}
* Humidity: ${weather.atmosphere.humidity}
* Wind Chill: ${weather.wind.chill}
* Date: ${weather.date}
```

The history.vm Velocity Template

```
Weather History for:
${location.city},
${location.region},
${location.country}


#foreach( $weather in $weathers )
****************************************
* Temperature: $weather.condition.temp
* Condition: $weather.condition.text
* Humidity: $weather.atmosphere.humidity
* Wind Chill: $weather.wind.chill
* Date: $weather.date
#end
```

7.9 Running the Simple Command

The `simple-command` project is configured to create a single JAR containing the bytecode of the project and all of the bytecode from the dependencies. To create this assembly, run the `assembly` goal of the Maven Assembly plugin from the `simple-command` project directory:

```
$ mvn assembly:assembly
[INFO] ------------------------------------------------------
[INFO] Building Multi-spring Chapter Simple Command Line Tool
[INFO]task-segment: [assembly:assembly] (aggregator-style)
[INFO] ------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
```

```
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
...
[INFO] [jar:jar]
[INFO] Building jar: .../simple-parent/simple-command/target/simple- ←
    command.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Building jar: .../simple-parent/simple-command/target
/simple-command-jar-with-dependencies.jar
```

The build progresses through the lifecycle compiling bytecode, running tests, and finally building a JAR for the project. Then the `assembly:assembly` goal creates a JAR with dependencies by unpacking all of the dependencies to temporary directories and then collecting all of the bytecode into a single JAR in `target/` named `simple-command-jar-with-dependencies.jar`. This "uber" JAR weighs in at 15 MB.

Before you run the command-line tool, you will need to invoke the `hbm2ddl` goal of the Hibernate3 plugin to create the HSQLDB database. Do this by running the following command from the `simple-command` directory:

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] ------------------------------------------------------
[INFO] Building Multi-spring Chapter Simple Command Line Tool
[INFO]task-segment: [hibernate3:hbm2ddl]
[INFO] ------------------------------------------------------
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151  INFO org.hibernate.tool.hbm2ddl.SchemaExport - export  ←
    complete
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
```

Once you run this, you should see a `data` directory under `simple-command`. This `data` directory holds the HSQLDB database. To run the command-line weather forecaster, run the following from the `simple-command` project directory:

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
       org.sonatype.mavenbook.weather.Main 60202
2321 INFO  YahooRetriever  - Retrieving Weather Data
```

```
2489 INFO  YahooParser  - Creating XML Reader
2581 INFO  YahooParser  - Parsing XML Response
2875 INFO  WeatherFormatter  - Formatting Weather Data
****************************************
Current Weather Conditions for:
Evanston,
IL,
US
****************************************

* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: Wed Aug 06 09:35:30 CDT 2008
```

To run a history query, execute the following command:

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
       org.sonatype.mavenbook.weather.Main 60202 history
2470 INFO  WeatherFormatter  - Formatting History Data
Weather History for:
Evanston, IL, US

****************************************
* Temperature: 39
* Condition: Heavy Rain
* Humidity: 93
* Wind Chill: 36
* Date: 2007-12-02 13:45:27.187
****************************************
* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: 2008-08-06 09:24:11.725
****************************************
* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: 2008-08-06 09:27:28.475
```

7.10 Conclusion

We've spent a great deal of time on topics not directly related to Maven to get this far. We've done this to present a complete and meaningful example project which you can use to implement real-world systems. We didn't take any shortcuts to produce slick, canned results quickly, and we're not going to dazzle you with some Ruby on Rails-esque wizardry and lead you to believe that you can create a finished Java Enterprise application in "10 easy minutes!" There's too much of this in the market; there are too many people trying to sell you the easiest framework that requires zero investment of time or attention. What we're trying to do in this chapter is present the entire picture, the entire ecosystem of a multi-module build. What we've done is present Maven in the context of a application which resembles something you could see in the wild—not the fast-food, 10 minute screen-cast that slings mud at Apache Ant and tries to convince you to adopt Apache Maven.

If you walk away from this chapter wondering what it has to do with Maven, we've succeeded. We present a complex set of projects, using popular frameworks, and we tie them together using declarative builds. The fact that more than 60% of this chapter was spent explaining Spring and Hibernate should tell you that Maven, for the most part, stepped out of the way. It worked. It allowed us to focus on the application itself, not on the build process. Instead of spending time discussing Maven, and the work you would have to do to "build a build" that integrated with Spring and Hibernate, we talked almost exclusively about the technologies used in this contrived project. If you start to use Maven, and you take the time to learn it, you really do start to benefit from the fact that you don't have to spend time coding up some procedural build script. You don't have to spend your time worrying about mundane aspects of your build.

You can use the skeleton project introduced in this chapter as the foundation for your own, and chances are that when you do, you'll find yourself creating more and more modules as you need them. For example, the project on which this chapter was based has two distinct model projects, two persistence projects which persist to dramatically different databases, several web applications, and a Java mobile application. In total, the real world system I based this on contains at least 15 interrelated modules. The point is that you've seen the most complex multi-module example we're going to include in this book, but you should also know that this example just scratches the surface of what is possible with Maven.

7.10.1 Programming to Interface Projects

This chapter explored a multi-module project which was more complex than the simple example presented in Chapter 6, yet it was still a simplification of a real-world project. In a larger project, you might find yourself building a system resembling Figure 7.5.

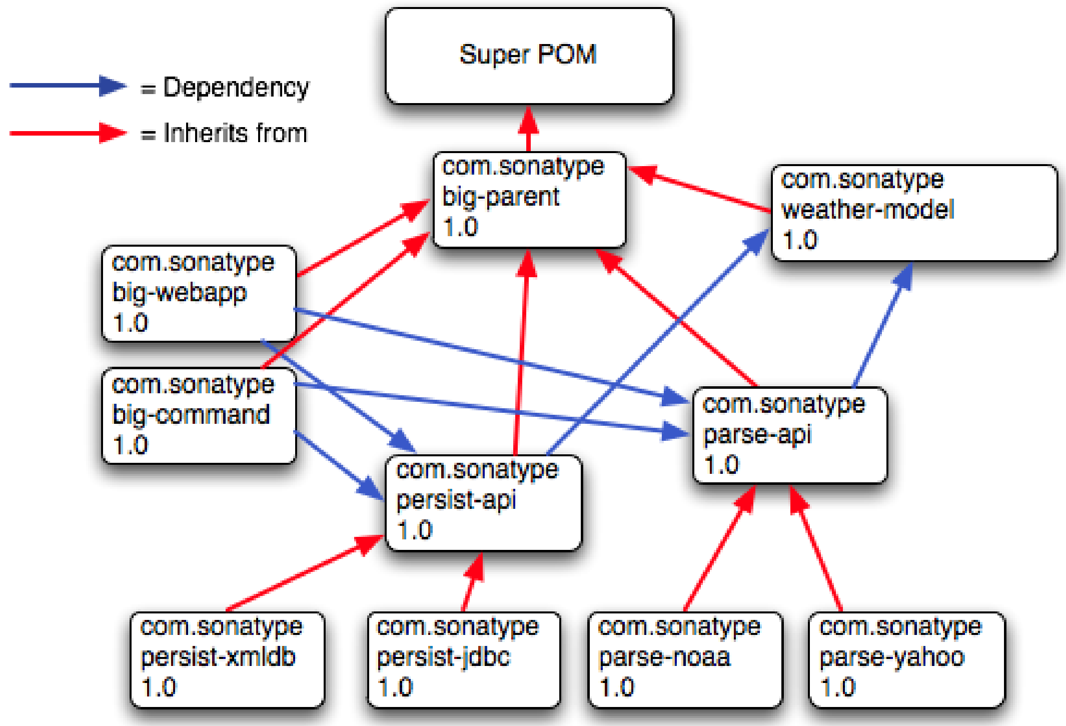

Figure 7.5: Programming to Interface Projects

When we use the term *interface project* we are referring to a Maven project which contains interfaces and constants only. In Figure 7.5 the interface projects would be `persist-api` and `parse-api`. If `big-command` and `big-webapp` are written to the interfaces defined in `persist-api`, then it is very easy to just swap in another implementation of the persistence library. This particular diagram shows two implementations of the `persist-api` project, one which stores data in an XML database, and the other which stores data in a relational database. If you use some of the concepts in this chapter, you can see how you could just pass in a flag to the program that swaps in a different Spring application context XML file to swap out data sources of persistence implementations. Just like the OO design of the application itself, it is often wise to separate the interfaces of an API from the implementation of the API into separate Maven projects.

Chapter 8

Optimizing and Refactoring POMs

8.1 Introduction

In Chapter 7, we showed how many pieces of Maven come together to produce a fully functional multi-module build. Although the example from that chapter suggests a real application—one that interacts with a database, a web service, and that itself presents two interfaces: one in a web application, and one on the command line—that example project is still contrived. To present the complexity of a real project would require a book far larger than the one you are now reading. Real-life applications evolve over years and are often maintained by large, diverse groups of developers, each with a different focus. In a real-world project, you are often evaluating decisions and designs made and created by others. In this chapter, we take a step back from the examples you've seen in the previous chapters, and we ask ourselves if there are any optimizations that might make more sense given what we now know about Maven. Maven is a very capable tool that can be as simple or as complex as you need it to be. Because of this, there are often a million ways to accomplish the same task, and there is often no one "right" way to configure your Maven project.

Don't misinterpret that last sentence as a license to go off and ask Maven to do something it wasn't designed for. While Maven allows for a diversity of approach, there is certainly "A Maven Way", and you'll be more productive using Maven as it was designed to be used. All this chapter is trying to do is communicate some of the optimizations you can perform on an existing Maven project. Why didn't we just introduce an optimized POM in the first place? Designing POMs for pedagogy is a very different requirement from designing POMs for efficiency. While it is certainly much easier to define a certain setting in your `~/.m2/settings.xml` than to declare a profile in a `pom.xml`, writing a book, and reading a book is mostly about pacing and making sure we're not introducing concepts before you are ready. In the previous chapters, we've made an effort not to overwhelm the reader with too much information, and, in doing so, we've skipped some core concepts like the `dependencyManagement` element introduced in this chapter.

There are many instances in the previous chapters when the authors of this book took a shortcut or glossed over an important detail to shuffle you along to the main point of a specific chapter. You learned how to create a Maven project, and you compiled and installed it without having to wade through hundreds of pages introducing every last switch and dial available to you. We've done this because we believe it is important to deliver the new Maven user to a result faster rather than meandering our way through a very long, seemingly interminable story. Once you've started to use Maven, you should know how to analyze your own projects and POMs. In this chapter, we take a step back and look at what we are left with after the example from Chapter 7.

8.2 POM Cleanup

Optimizing a multimodule project's POM is best done in several passes, as there are many areas to focus on. In general, we are looking for repetition within a POM and across the sibling POMs. When you are starting out, or when a project is still evolving rapidly, it is acceptable to duplicate some dependencies and plugin configurations here and there, but as the project matures and as the number of modules increases, you will want to take some time to refactor common dependencies and configuration points. Making your POMs more efficient will go a long way to helping you manage complexity as your project grows. Whenever there is duplication of some piece of information, there is usually a better way.

8.3 Optimizing Dependencies

If you look through the various POMs you notice a lot of duplication that you can remove by moving parts into a parent POM.

Just as in your project's source code, any time you have duplication in your POMs, you open the door a bit for trouble down the road. Duplicated dependency declarations make it difficult to ensure consistent versions across a large project. When you only have two or three modules, this might not be a primary issue, but when your organization is using a large, multimodule Maven build to manage hundreds of components across multiple departments, one single mismatch between dependencies can cause chaos and confusion. A simple version mismatch in a project's dependency on a bytecode manipulation package called ASM three levels deep in the project hierarchy could throw a wrench into a web application maintained by a completely different group of developers who depend on that particular module. Unit tests could pass because they are being run with one version of a dependency, but they could fail disastrously in production where the bundle (WAR, in this case) was packaged up with a different version. If you have tens of projects using something like Hibernate Annotations, each repeating and duplicating the dependencies and exclusions, the mean time between someone screwing up a build is going to be very short. As your Maven projects become more complex, your dependency lists are going to grow, and you are going to want to consolidate versions and dependency declarations in parent POMs.

The duplication of the sibling module versions can introduce a particularly nasty problem that is not directly caused by Maven and is learned only after you've been bitten by this bug a few times. If you use the Maven Release plugin to perform your releases, all these sibling dependency versions will be updated automatically for you, so maintaining them is not the concern. If `simple-web` version `1.3-SNAPSHOT` depends on `simple-persist` version `1.3-SNAPSHOT`, and if you are performing a release of the 1.3 version of both projects, the Maven Release plugin is smart enough to change the versions throughout your multimodule project's POMs automatically. Running the release with the Release plugin will automatically increment all of the versions in your build to `1.4-SNAPSHOT`, and the release plugin will commit the code change to the repository. Releasing a huge multimodule project couldn't be easier, until…

Problems occur when developers merge changes to the POM and interfere with a release that is in progress. Often a developer merges and occasionally mishandles the conflict on the sibling dependency, inadvertently reverting that version to a previous release. Since the consecutive versions of the dependency are often compatible, it does not show up when the developer builds, and won't show up in any continuous integration build system as a failed build. Imagine a very complex build where the trunk is full of components at `1.4-SNAPSHOT`, and now imagine that Developer A has updated Component A deep within the project's hierarchy to depend on version `1.3-SNAPSHOT` of Component B. Even though most developers have `1.4-SNAPSHOT`, the build succeeds if version `1.3-SNAPSHOT` and `1.4-SNAPSHOT` of Component B are compatible. Maven continues to build the project using the `1.3-SNAPSHOT` version of Component B from the developer's local repositories. Everything seems to be going quite smoothly—the project builds, the continuous integration build works fine, and so on. Someone might have a mystifying bug related to Component B, but she chalks it up to malevolent gremlins and moves on. Meanwhile, a pump in the reactor room is steadily building up pressure, until something blows….

Someone, let's call them Mr. Inadvertent, had a merge conflict in component A, and mistakenly pegged component A's dependency on component B to `1.3-SNAPSHOT` while the rest of the project marches on. A bunch of developers have been trying to fix a bug in component B all this time and they've been mystified as to why they can't seem to fix the bug in production. Eventually someone looks at component A and realizes that the dependency is pointing to the wrong version. Hopefully, the bug wasn't large enough to cost money or lives, but Mr. Inadvertent feels stupid and people tend to trust him a little less than they did before the whole sibling dependency screw-up. (Hopefully, Mr. Inadvertent realizes that this was user error and not Maven's fault, but more than likely he starts an awful blog and complains about Maven endlessly to make himself feel better.)

Fortunately, dependency duplication and sibling dependency mismatch are easily preventable if you make some small changes. The first thing we're going to do is find all the dependencies used in more than one project and move them up to the parent POM's dependencyManagement section. We'll leave out the sibling dependencies for now. The `simple-parent` pom now contains the following:

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring</artifactId>
                <version>2.0.7</version>
            </dependency>
            <dependency>
                <groupId>org.apache.velocity</groupId>
                <artifactId>velocity</artifactId>
                <version>1.5</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-annotations</artifactId>
                <version>3.3.0.ga</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-commons-annotations</artifactId>
                <version>3.3.0.ga</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate</artifactId>
                <version>3.2.5.ga</version>
                <exclusions>
                    <exclusion>
                        <groupId>javax.transaction</groupId>
                        <artifactId>jta</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
        </dependencies>
    </dependencyManagement>
    ...
</project>
```

Once these are moved up, we need to remove the versions for these dependencies from each of the POMs; otherwise, they will override the dependencyManagement defined in the parent project. Let's look at only `simple-model` for brevity's sake:

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
        </dependency>
    </dependencies>
    ...
</project>
```

The next thing we should do is fix the replication of the `hibernate-annotations` and `hibernate-commons-annotations` version since these should match. We'll do this by creating a property called `hibernate.annotations.version`. The resulting `simple-parent` section looks like this:

```
<project>
    ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga
      </hibernate.annotations.version>
  </properties>

  <dependencyManagement>
    ...
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    ...
  </dependencyManagement>
  ...
</project>
```

The last issue we have to resolve is with the sibling dependencies and define the versions of sibling projects in the top-level parent project. This is certainly a valid approach, but we can also solve the version problem just by using two built-in properties — `${project.groupId}` and `${project.version}`. Since they are sibling dependencies, there is not much value to be gained by enumerating them in the parent, so we'll rely on the built-in `${project.version}` property. Because they all share the same group, we can further future-proof these declarations by referring to the current POM's group using the built-in `${project.groupId}` property. The `simple-command` dependency section now looks like this:

```
<project>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>simple-weather</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>simple-persist</artifactId>
            <version>${project.version}</version>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

Here's a summary of the two optimizations we completed that reduce duplication of dependencies:

Pull-up common dependencies to `dependencyManagement`
: If more than one project depends on a specific dependency, you can list the dependency in `dependencyManagement`. The parent POM can contain a version and a set of exclusions; all the child POM needs to do to reference this dependency is use the `groupId` and `artifactId`. Child projects can omit the version and exclusions if the dependency is listed in `dependencyManagement`.

Use built-in project `version` and `groupId` for sibling projects
: Use `${project.version}` and `${project.groupId}` when referring to a sibling project. Sibling projects almost always share the same `groupId`, and they almost always share the same release version. Using `${project.version}` will help you avoid the sibling version mismatch problem discussed previously.

8.4 Optimizing Plugins

If we take a look at the various plugin configurations, we can see the HSQLDB dependencies duplicated in several places. Unfortunately, dependencyManagement doesn't apply to plugin dependencies, but we can still use a property to consolidate the versions. Most complex Maven multimodule projects tend to define all versions in the top-level POM. This top-level POM then becomes a focal point for changes that affect the entire project. Think of version numbers as string literals in a Java class; if you are constantly repeating a literal, you'll likely want to make it a variable so that when it needs to be changed, you have to change it in only one place. Rolling up the version of HSQLDB into a property in the top-level POM yields the following properties element:

```
<project>
  ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga
      </hibernate.annotations.version>
    <hsqldb.version>1.8.0.7</hsqldb.version>
  </properties>
  ...
</project>
```

The next thing we notice is that the hibernate3-maven-plugin configuration is duplicated in the simple-webapp and simple-command modules. We can manage the plugin configuration in the top-level POM just as we managed the dependencies in the top-level POM with the dependencyManagement section. To do this, we use the pluginManagement element in the top-level POM's build element:

```
<project>
  ...
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>hibernate3-maven-plugin</artifactId>
          <version>2.1</version>
          <configuration>
            <components>
```

```
                <component>
                  <name>hbm2ddl</name>
                  <implementation>annotationconfiguration</implementation>
                </component>
              </components>
            </configuration>
            <dependencies>
              <dependency>
                <groupId>hsqldb</groupId>
                <artifactId>hsqldb</artifactId>
                <version>${hsqldb.version}</version>
              </dependency>
            </dependencies>
          </plugin>
        </plugins>
      </pluginManagement>
  </build>
  ...
</project>
```

8.5 Optimizing with the Maven Dependency Plugin

On larger projects, additional dependencies often tend to creep into a POM as the number of dependencies grow. As dependencies change, you are often left with dependencies that are not being used, and just as often, you may forget to declare explicit dependencies for libraries you require. Because Maven 2.x includes transitive dependencies in the compile scope, your project may compile properly but fail to run in production. Consider a case where a project uses classes from a widely used project such as Jakarta Commons BeanUtils. Instead of declaring an explicit dependency on BeanUtils, your project simply relies on a project like Hibernate that references BeanUtils as a transitive dependency. Your project may compile successfully and run just fine, but if you upgrade to a new version of Hibernate that doesn't depend on BeanUtils, you'll start to get compile and runtime errors, and it won't be immediately obvious why your project stopped compiling. Also, because you haven't explicitly listed a dependency version, Maven cannot resolve any version conflicts that may arise.

A good rule of thumb in Maven is to always declare explicit dependencies for classes referenced in your code. If you are going to be importing Commons BeanUtils classes, you should also be declaring a direct dependency on Commons BeanUtils. Fortunately, via bytecode analysis, the Maven Dependency plugin is able to assist you in uncovering direct references to dependencies. Using the updated POMs we previously optimized, let's look to see if any errors pop up:

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

```
[INFO] Reactor build order:
[INFO]   Chapter 8 Simple Parent Project
[INFO]   Chapter 8 Simple Object Model
[INFO]   Chapter 8 Simple Weather API
[INFO]   Chapter 8 Simple Persistence API
[INFO]   Chapter 8 Simple Command Line Tool
[INFO]   Chapter 8 Simple Web Application
[INFO]   Chapter 8 Parent Project
[INFO] Searching repository for plugin with prefix: 'dependency'.

...

[INFO] ------------------------------------------------------
[INFO] Building Chapter 8 Simple Object Model
[INFO]task-segment: [dependency:analyze]
[INFO] ------------------------------------------------------
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[WARNING]org.hibernate:hibernate:jar:3.2.5.ga:compile
[WARNING]junit:junit:jar:3.8.1:test

...

[INFO] ------------------------------------------------------
[INFO] Building Chapter 8 Simple Web Application
[INFO]task-segment: [dependency:analyze]
[INFO] ------------------------------------------------------
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [dependency:analyze]
```

```
[WARNING] Used undeclared dependencies found:
[WARNING]org.sonatype.mavenbook.optimize:simple-model:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]org.apache.velocity:velocity:jar:1.5:compile
[WARNING]javax.servlet:jstl:jar:1.1.2:compile
[WARNING]taglibs:standard:jar:1.1.2:compile
[WARNING]junit:junit:jar:3.8.1:test
```

In the truncated output just shown, you can see the output of the `dependency:analyze` goal. This goal analyzes the project to see whether there are any indirect dependencies, or dependencies that are being referenced but are not directly declared. In the `simple-model` project, the Dependency plugin indicates a "used undeclared dependency" on `javax.persistence:persistence-api`. To investigate further, go to the `simple-model` directory and run the `dependency:tree` goal, which will list all of the project's direct and transitive dependencies:

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------
[INFO] Building Chapter 8 Simple Object Model
[INFO]task-segment: [dependency:tree]
[INFO] ------------------------------------------------------
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.optimize:simple-model:jar:1.0
[INFO] +- org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[INFO] |  \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- org.hibernate:hibernate:jar:3.2.5.ga:compile
[INFO] |  +- net.sf.ehcache:ehcache:jar:1.2.3:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] |  +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  +- antlr:antlr:jar:2.7.6:compile
[INFO] |  +- cglib:cglib:jar:2.1_3:compile
[INFO] |  +- asm:asm:jar:1.5.3:compile
[INFO] |  \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
```

From this output, we can see that the `persistence-api` dependency is coming from `hibernate`. A cursory scan of the source in this module will reveal many `javax.persistence` import statements confirming that we are, indeed, directly referencing this dependency. The simple fix is to add a direct reference to the dependency. In this example, we put the dependency version in `simple-parent`'s `dependencyManagement` section because the dependency is linked to Hibernate, and the Hibernate version is declared here. Eventually you are going to want to upgrade your project's version of Hibernate.

Listing the persistence-api dependency version near the Hibernate dependency version will make it more obvious later when your team modifies the parent POM to upgrade the Hibernate version.

If you look at the dependency:analyze output from the simple-web module, you will see that we also need to add a direct reference to the simple-model dependency. The code in simple-webapp directly references the model objects in simple-model, and the simple-model is exposed to simple-webapp as a transitive dependency via simple-persist. Since this is a sibling dependency that shares both the version and groupId, the dependency can be defined in simple-webapp's pom.xml using the ${project.groupId} and ${project.version}.

How did the Maven Dependency plugin uncover these issues? How does dependency:analyze know which classes and dependencies are directly referenced by your project's bytecode? The Dependency plugin uses the ObjectWeb ASM (http://asm.objectweb.org/) library to produce a list of "used, undeclared dependencies" dependencies

In contrast, the list of unused, declared dependencies is a little trickier to validate, and less useful than the "used, undeclared dependencies." For one, some dependencies are used only at runtime or for tests, and they won't be found in the bytecode. These are pretty obvious when you see them in the output; for example, JUnit appears in this list, but this is expected because it is used only for unit tests. You'll also notice that the Velocity and Servlet API dependencies are listed in this list for the simple-web module. This is also expected because, although the project doesn't have any direct references to the classes of these artifacts, they are still essential during runtime.

Be careful when removing any unused, declared dependencies unless you have very good test coverage, or you might introduce a runtime error. A more sinister issue pops up with bytecode optimization. For example, it is legal for a compiler to substitute the value of a constant and optimize away the reference. Removing this dependency will cause the compile to fail, yet the tool shows it as unused. Future versions of the Maven Dependency plugin will provide better techniques for detecting and/or ignoring these types of issues.

You should use the dependency:analyze tool periodically to detect these common errors in your projects. It can be configured to fail the build if certain conditions are found, and it is also available as a report.

8.6 Final POMs

As an overview, the final POM files are listed as a reference for this chapter. Final POM for simple-parent shows the top-level POM for simple-parent.

Final POM for simple-parent

```
<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>org.sonatype.mavenbook.optimize</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 8 Simple Parent Project</name>

  <modules>
    <module>simple-command</module>
    <module>simple-model</module>
    <module>simple-weather</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>hibernate3-maven-plugin</artifactId>
          <version>2.1</version>
          <configuration>
            <components>
              <component>
                <name>hbm2ddl</name>
                <implementation>annotationconfiguration</implementation>
              </component>
            </components>
          </configuration>
          <dependencies>
            <dependency>
              <groupId>hsqldb</groupId>
              <artifactId>hsqldb</artifactId>
```

```
          <version>${hsqldb.version}</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
 </pluginManagement>
</build>

<properties>
  <hibernate.annotations.version>3.3.0.ga
    </hibernate.annotations.version>
  <hsqldb.version>1.8.0.7</hsqldb.version>
</properties>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.5.ga</version>
      <exclusions>
        <exclusion>
          <groupId>javax.transaction</groupId>
          <artifactId>jta</artifactId>
        </exclusion>
```

```
        </exclusions>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-command captures the POM for `simple-command`, the command-line version of the tool.

Final POM for simple-command

```
<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>
  <parent>
      <groupId>org.sonatype.mavenbook.optimize</groupId>
      <artifactId>simple-parent</artifactId>
      <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Chapter 8 Simple Command Line Tool</name>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <archive>
              <manifest>
                <mainClass>org.sonatype.mavenbook.weather.Main</mainClass>
                <addClasspath>true</addClasspath>
              </manifest>
```

```
            </archive>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <configuration>
            <testFailureIgnore>true</testFailureIgnore>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-weather</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-persist</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
    </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-model is the `simple-model` project's POM. The `simple-model` project contains all of the model objects used throughout the application.

Final POM for simple-model

```
<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>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
    </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-persist is the `simple-persist` project's POM. The `simple-persist` project contains all of the persistence logic that is implemented using Hibernate.

Final POM for simple-persist

```
<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>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
```

```
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Persistence API</name>

  <dependencies>
      <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>simple-model</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-commons-annotations</artifactId>
      </dependency>
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.4</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
      </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-weather is the `simple-weather` project's POM. The `simple-weather` project is the project that contains all of the logic to parse the Yahoo Weather RSS feed. This project depends on the `simple-model` project.

Final POM for simple-weather

```
<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>
```

```
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Weather API</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Finally, the POM shown in Final POM for simple-webapp is the `simple-webapp` project's POM. The `simple-webapp` project contains a web application that stores retrieved weather forecasts in an HSQLDB database and that also interacts with the libraries generated by the `simple-weather` project.

Final POM for simple-webapp

```
<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>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>Chapter 8 Simple Web Application</name>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-weather</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-persist</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
    </dependency>
    <dependency>
      <groupId>taglibs</groupId>
      <artifactId>standard</artifactId>
      <version>1.1.2</version>
    </dependency>
```

```
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <version>6.1.9</version>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>${hsqldb.version}</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

8.7 Conclusion

This chapter has shown you several techniques for improving the control of your dependencies and plugins to ease future maintenance of your builds. We recommend periodically reviewing your builds in this way to ensure that duplication is reduced and kept at a minimum. This will ensure that your build performance does not degrade and you produce high quality outputs.

Chapter 9

Creative Commons License


```
<div xmlns:cc="http://creativecommons.org/ns#"
     about="http://creativecommons.org/license/results-one?q_1=2&q_1=1\
            &field_commercial=n&field_derivatives=n&field_jurisdiction=us\
            &field_format=StillImage&field_worktitle=Repository%3A+\ ↩
                Management\
            &field_attribute_to_name=Sonatype%2C+Inc.\
            &field_attribute_to_url=http%3A%2F%2Fwww.sonatype.com\
            &field_sourceurl=http%3A%2F%2Fwww.sonatype.com%2Fbook\
            &lang=en_US&language=en_US&n_questions=3">
    <a rel="cc:attributionURL" property="cc:attributionName"
       href="http://www.sonatype.com">Sonatype, Inc.</a> /
    <a rel="license"
       href="http://creativecommons.org/licenses/by/3.0/us/">
        CC BY 3.0</a>
</div>
```

Chapter 10

Copyright