Table of Contents

CHAPTER 1
Introduction

Life can only be understood backwards; but it must be lived forwards. -- Soren Kierkegaard

Why a Book?

You may ask "Why a Maven book? There are plenty of documents online, right?". The problem of diving into any new software project is the problem of where to begin. Yes, there is a growing wealth of information pertaining to the Maven project - but it is scattered and piecemeal. They make great reference materials (we even used some docs for this book) - but many developers, myself included, desire a narrative - a jolly stroll through the growing Maven metropolis - a place to see the grand sites without being overwhelmed - where does the yellow-brick road begin? Here.

Maven... What is it?

This is a complex question, but a good one. Maven is a lot of things to a lot of people. If you use Maven to its fullest extent, it is a build and deployment tool vis-a-vis Ant, a dependency management tool like Ivy, a metric reporting tool, a documentation generator, a software project manager, a parent project, a build lifecycle, a project repository, a convention, a concept, and a community. A user may utilize one or many of these pieces, or use it for purposes hitherto undiscovered. At its core Maven is a framework for managing various aspects of a project, providing its own conventions and tools to meet this end and acting as glue for making existing disparate tools work in a tractable and orderly manner.

Convention over Configuration

Convention is at the heart of Maven. Convention over configuration is a popular aphorism these days, and Maven fully embraces this concept. Convention over configuration is at the central philosophy of frameworks such as Ruby on Rails, and more

recently, the EJB3 specification. In its most basic sense it means that, while configuration is certainly necessary, the majority of users will never utilize such edge-cases those complex configurations provide. Although a powerful framework certainly needs to have the power to configure when necessary, it is certainly reasonable to create defaults to allow the 95% of similar use-cases to work without defining anything at all... the system can assume these defaults. In other words, the system has its own convention. Because of this, the monstrous configurations required of build tools like Ant (where a majority of Ant scripts are cut-and-pasted from existing projects) are non-existent for those projects that follow Maven's conventions.

Another driving force behind the popularity of convention over configuration is the speed at which new users may pick up a new technology, or the speed by which a seasoned user may begin using the tool without concerning him/herself with details that need not come up until later in the development process. The computer world is finally beginning to embrace the idea that ease of use and reduced configurations do not have to interfere with the power of advanced configurability. Convention and configuration reside together within the Maven world, each providing their own unique perspective of a power tool.

A Brief History of Build Tools

There are literally thousands of books, "collateral damage" if you will, filled with supposed "perfect solutions": perfect programming languages, perfect platforms, perfect IDEs, or perfect processes. But far fewer books cover the build tools that are often every bit as important as the languages themselves. Build tools have a historically underappreciated symbiotic co-history with the programming languages and methodologies that they try to manage. Each has left its own indelible mark on the world of software development; some good (Make), some not so good (shell scripts). But along the way we have inched ever toward the ultimate goals of programming: ease of use, reusability, reliability, portability. Someday Maven, too, will be a footnote in this history adding another step to some unknown end. Until such a time, however, it is currently the best tool we have on our collective belt.

Make / Ant / Maven

In the beginning, there was no standard build tool. If one wanted to compile and link code, cave-dwelling developers were forced to input system commands with their own hairy-knuckled hands; this worked well as long as the whole tribe of developers knew all of the esoteric grunts and commands to make the build work. As their programs increased in complexity, however, it became immediately obvious such methods could be automated using simple tools.

As programming languages themselves have evolved, the focus of incremental improvements have centered on more than just readability and maintainability they were

also developed to be more portable. C was a large improvement over previous languages (as was the operating system) in a large part due to its portability between different types of machines. But despite its undeniable success in this realm, machine architectures were largely diametrical, and system tools were incompatible by today's standards. Eventually, shells became more standard, and the ability to script common commands in a common language became more ubiquitous. This caused a new problem. Although a program was portable between machines with similar setups, computers would often contain different tools for completing the same job, such as a different implementation of the C compiler. Building code across various machines still proved rather painful. Enter: Make. Make was a marked improvement over shell scripting due to its abstraction of specific shell commands into a more general and consistent syntax.

Fast forward a few years to the birth of Java, a popular machine-portable programming language created to relieve the burden of cross-compilation by compiling to machine portable byte code. Now the machine-specific aspects of Make made the build tool even less portable than the code it was meant to build, in addition to other issues the frustrated Java developers (who are you to tell me where to put whitespaces?!), July 19 2000 birthed the first release of Ant as an independent build tool.

Ant is a pure Java build tool defined by XML syntax. Its ease of use and agility at extension made this seemingly simple concept explode to the de facto Java build tool in less than a year's time.

Why not just use Ant?

Ant is a great project. So was Make. However, they both suffer from the inherent weakness of the "Inner-Platform Effect" (*http://thedailywtf.com/forums/69415/ShowPost.aspx*) anti-pattern. The *Inner-Platform Effect* roughly states that by making a system too flexible and configurable, the configuration of the system itself becomes so complex it takes a specialist in order to maintain it - thus re-introducing the problem that the system was attempting to solve. Or in other words, the configuration becomes a platform itself, requiring all of the tools and special knowledge of any programming language... effectively creating a system in a system. For non-trivial projects Ant scripts can become almost as complex than the projects that they are required to build.

Ant is still an excellent tool, and is available as part of the Maven core collection of plugins - enabling you to create custom extensions out of existing Ant scripts, or even embed Ant directly into your project's build lifecycle (more on that later). Maven was created to deal with some of Ant's more egregious limitations. Firstly, the size of an Ant's build file directly correlates to its complexity. In short, the more things you want Ant to do, the larger the build file gets. This is due to the fact that Ant build files are procedural. A user must specify each step that the build system is to take. In Maven, due to its in-built conventions, there is a very good chance that your Maven configuration file will not grow at all (or at least, not grow very much) despite what you use it

for because Maven configuration files are declarative. It is up to individual plug-ins and the Maven core to decide how to take action; you need only direct it with a modicum of information about your project.

Second is the question of portability. Ant is portable only if your users have the same set of tasks that you have. It is up to Ant's users to download and manage their own library of tasks, and it is expected that for any particular task, they know which tasks are required - as well as the correct versions. Oftentimes the most prudent method is to bundle the build tools along with the code to be built, which is hardly pragmatic (though problems of this nature can actually be rectified with Ant extensions like Ivy, which in all fairness, uses the Maven repository writ-large). Maven, on the other hand, deals with this problem natively and automatically, making portability effortless, and keeping even the largest teams in synch.

Summary

This introduction has been kept purposefully short. We have covered a basic outline of what Maven is, and how it stacks up to, and improves upon, other build tools throughout time. The next chapter will dive into a simple project and how Maven can perform phenomenal tasks with the smallest amount of configuration.

CHAPTER 2
A Simple Project

I don't know why we are here, but I'm pretty sure that it is not in order to enjoy ourselves. -- Ludwig Wittgenstein

This chapter follows an example project. You may wish to download killerapp (http://www.sonatype.com/book/examples/book-killerapp.zip) and follow along.

Introduction

Programmers are notoriously gawkish when it comes to words - so are we. With this in mind, the first part of this book is focused on using Maven, explaining with code or graphs rather than words when possible. Readers already familiar with Maven would still do well to read through the first few chapters; Maven evolves quickly and you may learn something new.

Maven is a huge project. Although not as large as something akin to the Eclipse project, it is more complex than Ant. It is conceptually simpler, mind you, and easier to manage, but the sheer number of possibilites that Maven affords the build engineer makes it a chore to learn all of its many facates. As hinted at in the introduction, Maven is more than a core project, but is also an umbrella project for the many plugins necessary to complete the usual build tasks. Because of this inherant complexity there are two levels of understanding

The philosophy of this book is simple. This is a book for complete beginners, to give them a solid understanding of what Maven can do, and the best practices that many of us have learned along the way. It is written in the form of a narrative, following each step in the building, documenting, reporting and deployment of a project called the Killer App.

The Killer App is the next big social networking site! It is broken into several sub-projects that take advantage of Maven's code generation, compilation, test execution, packaging and verification and generating documentation - to higher order use-cases such as profile management, artifact classifiers.

Installation

One of Maven's biggest strengths lies in its ability to manage the components required to build an application. This means that if a user attempts to package a Java project into a JAR artifact then Maven will first reach across the internet to Maven's central repository and download the tools necessary in creating a JAR. In Maven, deployable files are called artifacts; thus, a JAR file is an artifact - so is a WAR file.

Before installing you must have a Java version 1.4 or above installed. Ensure that the JAVA_HOME system variable is set and points to your JDK install path. It is also convenient to have the directory containing your "java" binaries in the system PATH.

Installation of Maven 2 is a snap. Visit the Maven site at *http://maven.apache.org/* to download the most recent version of Maven 2, in the zip format you wish. Unpack the file to a convenient location that you have access to. This is where Maven is installed, no action is required beyond unpacking it. It is, however, wise to add your Maven install directory to the system PATH.

Ex Nihilo

Open up your favorite command line terminal (cmd.exe or command.com in Windows, or some sort of xterm in Unix flavors) and type the following command:

```
$ mvn archetype:create -DgroupId=mavenbook -DartifactId=my-app
```

We have just used Maven to create a project from scratch. The name of the project is "my-app", and the group is "mavenbook" (we will go over "groups" soon, for now note it is required). You will see output similar to the following:

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] ------------------------------------------------------------------------
... <lots of information, lots of downloading artifacts> ...
[INFO] [archetype:create]
[INFO] Defaulting package to group ID: mavenbook
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: checking for updates from central
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating Archetype: maven-archetype-quickstart:RELEASE
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: mavenbook
[INFO] Parameter: packageName, Value: mavenbook
[INFO] Parameter: basedir, Value: ~
[INFO] Parameter: package, Value: mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: my-app
[INFO] ********************* End of debug info from resources from generated POM **********************
[INFO] Archetype created in dir: ~/my-app
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Tue Apr 24 10:45:00 CDT 2007
[INFO] Final Memory: 4M/8M
[INFO] ------------------------------------------------------------------------
```

`mvn` is the Maven 2 command. `archetype:create` is called a Maven "goal". If you are familiar with Ant, you may concieve of this as similar to a task. Finally, the remaining `-Dname=value` pairs are arguments that are passed to the goal and take the form of `-D` properties, similar to the system property options you might pass to the `java` executable via the command line. The purpose of the `archetype:create` goal was to quickly "create" a project from an "archetype". The plugin is the prefix "archetype", and the goal is to "create".

```
$ cd my-app
$ mvn package
```

When you run the `package` command, expect the see the following output:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 1 source file to ~/my-app/target/classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
Compiling 1 source file to ~/my-app/target/test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: ~/my-app/target/surefire-reports

 -------------------------------------------------------
  T E S T S
 -------------------------------------------------------
Running mavenbook.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

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

[INFO] [jar:jar]
[INFO] Building jar: ~/my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Thu Oct 05 21:16:04 CDT 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

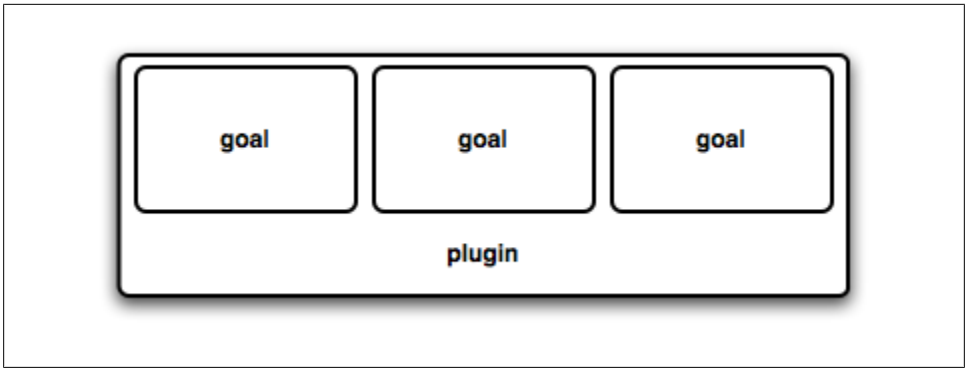


Figure 2-1. A Plugin Contains Goals

It's built! You have now created and run a simple project using Maven. It even works, which you can prove to yourself by running the obligatory hazing program: Hello World.

```
$ java -cp target/my-app-1.0-SNAPSHOT.jar mavenbook.App
Hello World!
```

You may notice something slightly different about the above command from the first one. The argument to the latter `mvn` command was a single word - `package` - not a colon-seperated pair like the initial `archetype:create` goal. The first command was a single goal - an objective - of a plugin. Plugins may contain several goals, each a specific objective that may be executed solo or along with other goals to achieve some end; in this case *create* a project based upon an *archetype*. A goal is a unit of work in Maven, and indeed beyond a few core background and housekeeping functions, goals do all of the work in Maven.

Note that we did not specify what kind of archetype the goal is to create on our command line, merely what to name the project. This is our first brush with "convention over configuration." The convention, or default, for this goal is to create a simple project (called Quickstart, but you don't need to remember that for now). Thanks to the default behavior of the plugin, we saved ourself a lot of typing for doing something we wanted to do anyway! In this case, generate a quick little project to start with. The archetype plugin is far more powerful than this first example lets on, but it is a great way to get new projects started fast (more details on archetypes in a later chapter).

The second command we ran - `mvn package` - was not a goal at all, but rather a Maven *phase*. Phases are a cornerstone of Maven's simplicity in execution, so let us take a little time and go over precisely what Maven folks mean when they speak of phases.

A phase is a step in what Maven calls the "build lifecycle". The build lifecycle is an ordered sequence of phases involved in building a project, beginning with validating the basic integrity of the project to deploying the project to production. These steps are purposefully left vague, as validation, or testing, or deployment may mean different things to different projects. For example, the "package" phase in a jar project means

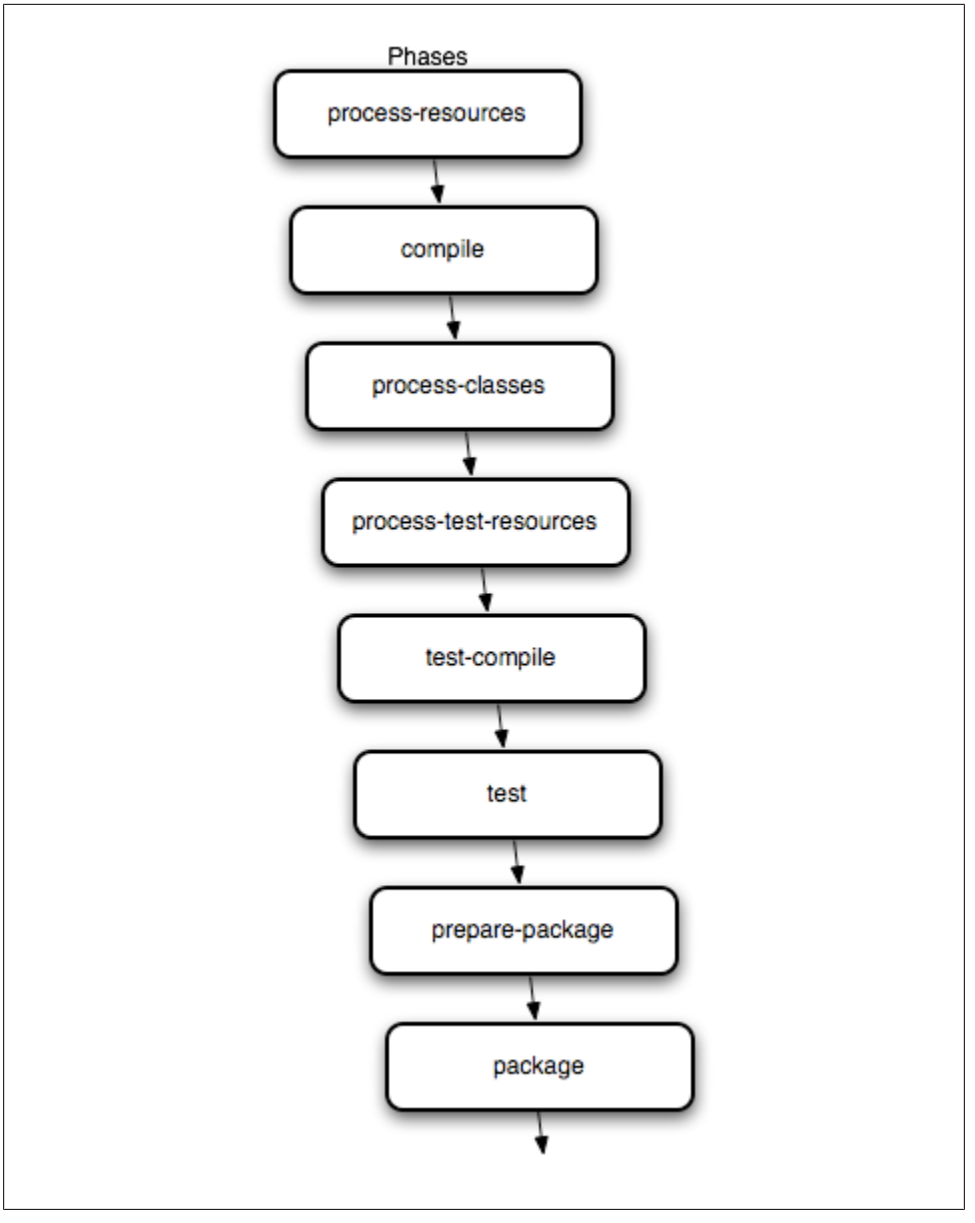


Figure 2-2. A Lifecycle is a Sequence of Phases

"package this project into a jar", where in an ear project it may mean "package this project into an ear along with its dependencies".

You may be thinking "But you said goals do all of the work!" It's true, they do. A phase is actually a key upon which goals piggyback to give an execution set some structure.

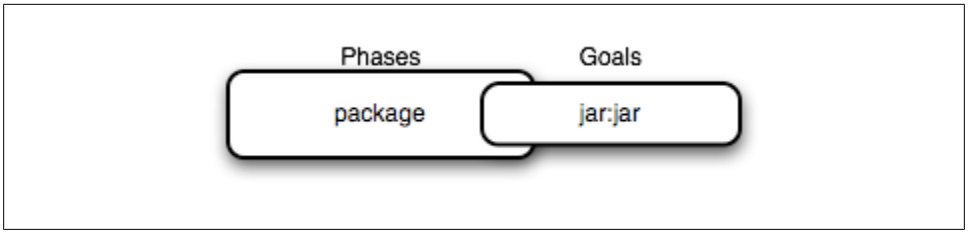


Figure 2-3. A Goal Binds to a Phase

Each phase may have zero or more goals bound to it - but usually zero or one. This enforces that, by default, a set of goals are executed in a specific sequence. If you run the `mvn package` phase again, you will notice that more than one goal was executed. Since our simple quickstart project has (by default) a `jar` packaging type, then the "jar:jar" goal is bound to the `package` phase.

But what of the goals preceding it, like "compiler:compile" and "surefire:test"? These are executed because, unlike executing a specific goal, executing a phase will first execute all proceeding phase, in order, leading up to and including the one specified. Since each phase is actually an abstract correlation to a set of goals, each goal bound to a phase is executed in order. The jar project we are packaging executes the following goals in sequence:

- [resources:resources] goal is bound to the resources phase.
- [compiler:compile] goal is bound to the compile phase
- [resources:testResources] goal is bound to the test-resources phase
- [compiler:testCompile] goal is bound to the test-compile phase
- [surefire:test] goal is bound to the test phase
- [jar:jar] goal is bound to the package phase

Let us revisit the command we executed above.

```
mvn package
```

Since this command executes a phase, it will execute all phases up to that point - well, actually as mentioned before, it is executing all goals bound to each phase up to that point. Since we are working with a `jar` packaging type project, and by looking at the goal sequence above, we may execute the same thing be typing:

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

The package phase is nicer, eh? It is also safer, as it does more than compile and package, but also forces testing to take place since it lay in the sequence. Assuming you have

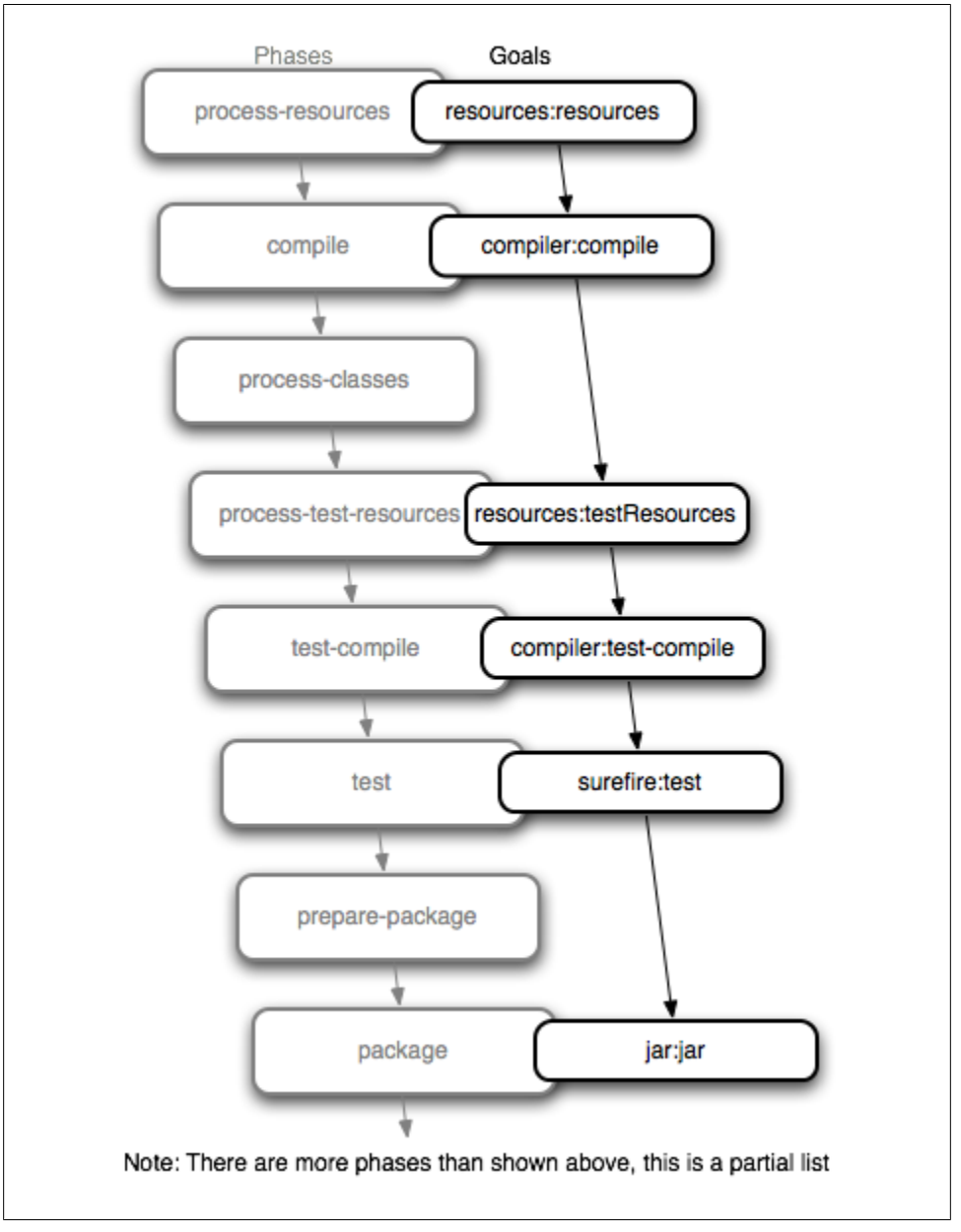


Figure 2-4. Bound Goals are Run when Their Phases Execute

written great tests - which we are certain that you always do - you can be assured that the generated jar artifact will work.


```
<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>
```

Figure 2-5. A POM requires Coordinates

Maven Coordinates

There are two basic components to the Maven system: goals and POMs. We have already covered the first. The second is the POM, which is an acronym for Project Object Model - it is a declarative description of a project defined in a single xml file named pom.xml. They relate together rather simply. Goals are actions we wish to take upon a project, and a project is defined by a POM. Another way of looking at it is: if we consider a build system as a series of statements, goals are the verbs; the POM and other project files, nouns.

Let us take a look at the POM file for the project we have created.

The `groupId`, `artifactId`, `version` and `packaging` type make up a project's coordinate (sometimes a fifth element, `classifier`, exists but more on that in a later chapter). Just like in any other coordinate system, a Maven coordinate is an address for a specific point in space, in increasing detail. Up until now we have ignored how Maven accesses all of the projects it needs - and we will still avoid the details for now. What is currently important is that Maven pinpoints a project via its coordinate when one project relates to another, either as a dependency (this project requires another project for some reason), a transative dependency (a dependency of a dependency), a parent (this project inherits some attributes from another project). In the above pom.xml file for our current project, its coordinate is represented as mavenbook:my-app:jar:1.0-SNAPSHOT.

- groupId: The group, company, team, organization, project, or other group

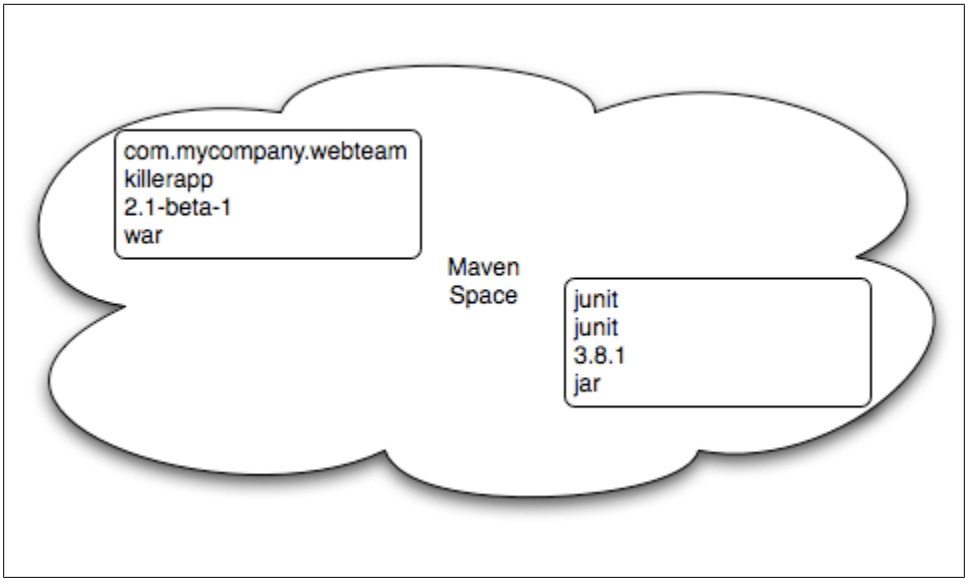


Figure 2-6. Maven Space is a coordinate system of projects

- artifactId: A unique identification under the groupId representing a single project.
- version: A snapshot in time of the project in question.

Moreover, packaging is also an important component, though note that `groupId:artifactId:version` make a project unique - you cannot have a `jar` and a `war` with the same key.

- packaging: The type of project, defaulting to jar, describing the type of build life-cycle this project will encounter. Naturally, this means the type of artifact Maven will generate when packaged.

These four elements become the key to finding and using one particular project in the vast space of other Mavenized projects, locally and out in cyberspace. Each project out in a Maven repository is located and used by these elements that create a form of address represented as `groupId:artifactId:packaging:version`. When this project is installed into the local Maven repository, it immediately becomes locally available to any other project that wish's to use it. All one must do is to add it as a dependency of another project.

Repositories and the POM

A Maven repository is a collection of installed or deployed project artifacts and other metadata information, managed exclusively by Maven. A local Maven repository is a local system directory structure on the building machine containing all of the artifacts that have been downloaded by Maven, either required by plugins used, dependencies

of projects built, or manually installed projects via the install plugin. Maven will not work without a local repository. Even the sample project created above kicked off the creation of a local repository. By default, this local repository is under the current user's directory .m2/repository.

In Windows this is likely *C:\Documents and Settings\USERNAME\.m2\repository*

In *nix systems this is likely *~/.m2/repository*

Wherever the differences in local repository location may be, some things are the same. The layout under the repository is (mostly) always groupId/artifactId/version/artifactId-version.packaging. For example, look in your local repository path under junit/junit/3.8.1/. If you have been following the examples thusfar, there will be a file named junit-3.8.1.jar and a junit-3.8.1.pom file (as well as probably a couple checksum files). The files were automatically downloaded and installed by Maven from the remote Maven central repository - more details on that in later chapters.

If you look under the repository *mavenbook/my-app/1.0-SNAPSHOT/* you should find... nothing. We have not installed the project yet! But now we will.

```
mvn install
```

Just like the package phase executed above, this phase execution is a shorthand notation to execute every goal bound to every phase up to and including install, making this lifecycle equivalent to running:

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

Check the `mavenbook/my-app/1.0-SNAPSHOT/` location again, and you will find `the my-app-1.0-SNAPSHOT.jar` and `my-app-1.0-SNAPSHOT.pom` files. As you can imagine the jar is the artifact generated by the jar:jar goal bound to the package phase. The pom file is an installed and slightly modified version of the project's `pom.xml` file. This pom file gives other projects information about this project, most importantly what dependencies it has. Maven does more than installs a project's dependencies (in our case, `junit:junit:jar:3.8.1`), but also installs a project's dependency's dependencies - its transative dependencies.

The necessary basic piece of information we will cover is the POM itself. The POM is a single unit of work, no smaller unit conceptually exists within Maven. This comes in handy especially when dealing with large scale projects with complicated dependency hierarchies.

The `pom.xml` we are investigating contains a single dependency `junit:junit:jar:3.8.1` (the packaging type defaults to `jar`, remember?). The scope of that dependency is test, meaning that it will not be required to run `compiler:compile`, but it will be

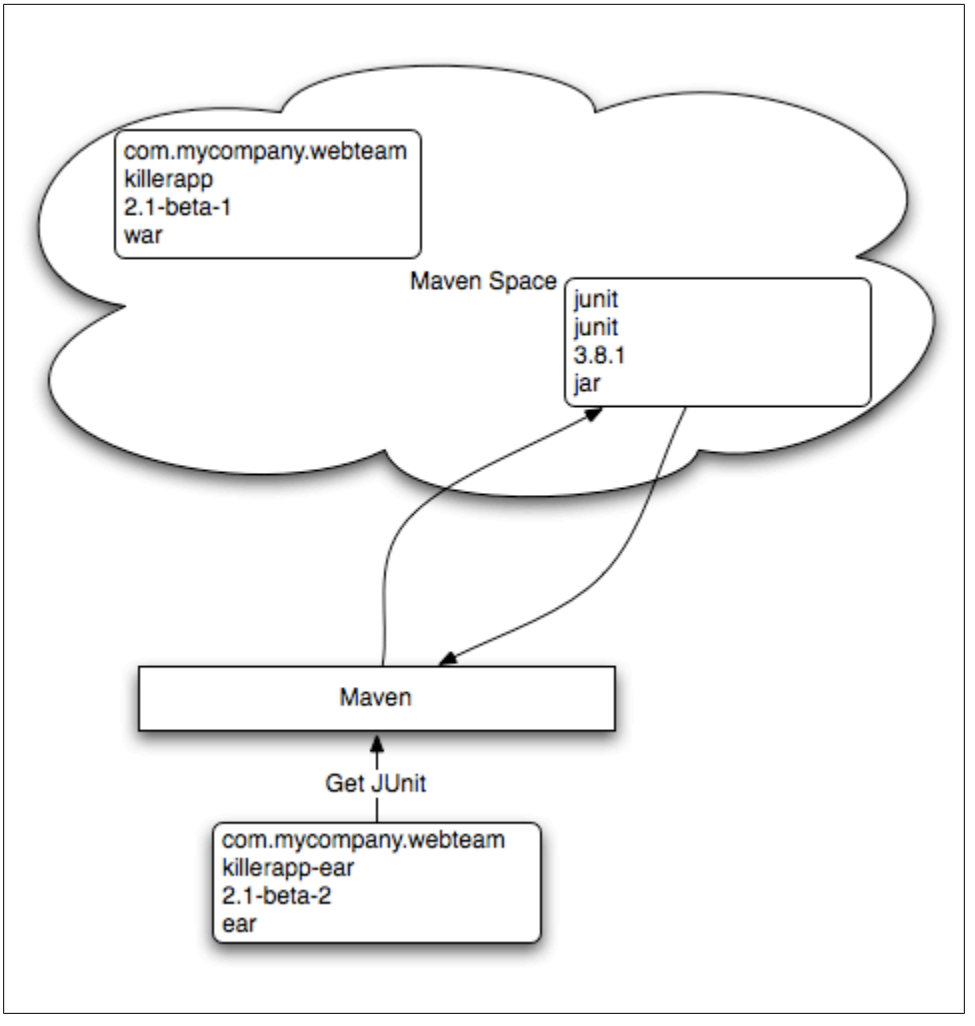


Figure 2-7. Maven connects to this space to download artifacts denoted by the coordinate

required to to successfully execute `compiler:testCompile`. Maven will use dependencies that match a project's coordinates before the first goal execution takes place.

Under normal circumstances dependencies will not be bundled up with the generated artifact (though there are exceptions, such as the ear packaging type) - they are only used for compilation. But as you will find, between the configurability of existing plugins through the POM and the ability to write custom plugins, any structure is possible through Maven.

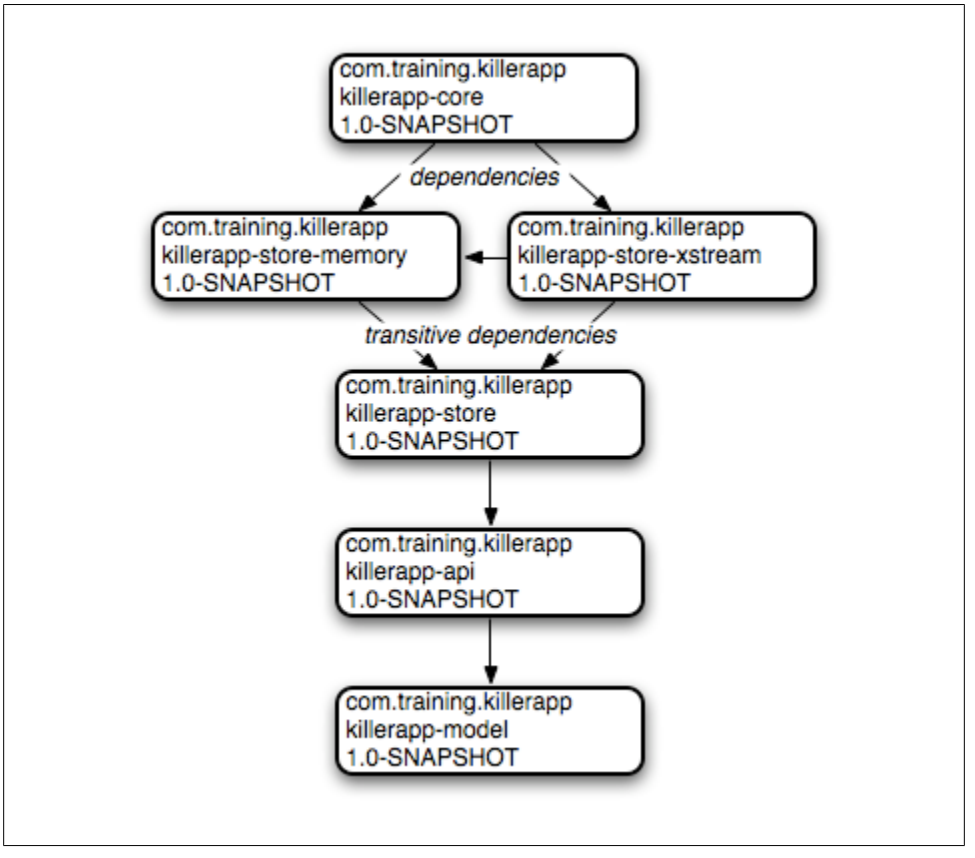


Figure 2-8. You define the dependencies and Maven manages transitive dependencies for you

Site

Lastly, we will quickly cover a rather important piece of Maven's functionality: the site build lifecycle. Unlike the default build lifecycle that manages generation of code, manipulation of resources, compilation, packaging, et cetera, this lifecycle is concerned solely with processing site files under the src/site directories and generating reports.

```
mvn site
```

That's it! Thanks to Maven's convention, this simple command will generate a site with very basic information under the target/site directory. Open (in a web browser) the `index.html` file under that directory and it will display a basic blank Maven site page with a left-hand side menu containing Project Information. This contains normally useful information like a project summary and dependency information - in our case, however, this information is mostly empty, since the POM contains very little information about itself beyond a coordinate, a name, url and single test dependency.

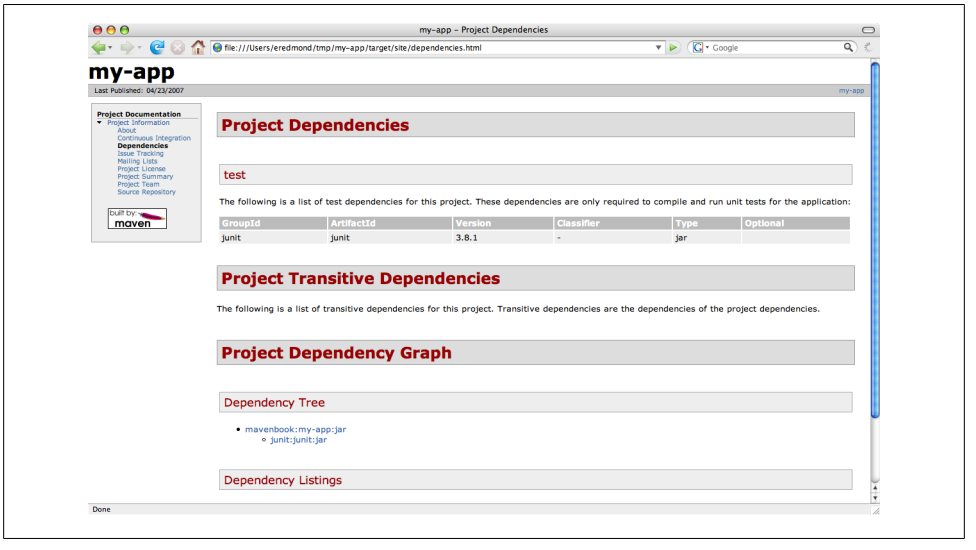


Figure 2-9. Dependency Report is one of Several Reports Generated by Default

First Run

You may wish to download the KillerApp project (http://www.sonatype.com/book/examples/book-killerapp.zip) and follow along.

By this point you should have - at least - a simple understanding of what Maven is. Although this grasp may not yet be an intuitive understanding, you should by now have a high-concept view of what Maven does (project stuff), and what it can achieve (nearly anything). If you are still lost, never fear. We will now begin working with a simple project that will follow us through the rest of this book, as we open up each galaxy of the Maven universe, and then navigate down to the finer systems contained therein.

The project we will work with is called the Killer App - the ultimate, grandiloquently named, application that will change the world! Social networking applications are all the current rage, and this is yet another. This application has a command-line interface for both managing accounts, as well as mining lists of emails for nefarious spamming purposes. The public side is a web application where users can create accounts and personal pages, as well as adding their buddies who also have accounts. The data is held either in main memory or stored in an xml stream. More importantly, the construction of this project and related tools will touch on all of the major points of Maven, as well as giving you a feeling for Maven's best practices.

```
killerapp-api
killerapp-cli
killerapp-core
killerapp-model
killerapp-stores
```

```
    killerapp-stores-memory
    killerapp-stores-xstream
    pom.xml
  killerapp-war
  src/site
  pom.xml
```

Note that the Killer App project is by no means a perfect architecture. In all actuality, it does not even rank as "good" - it's rather overdesigned. What it does do is touch on all of the major points that we will cover in this tutorial concerned with usability. By the end of this book you will have all of the necessary knowledge to tackle any issue you, as a user, may encounter in Maven either through direct examples or by enabling you with the tools necessary to find your own answers.

Under the `killerapp` directory run the following command (if you have not yet downloaded (*http://www.sonatype.com/book/examples/book-killerapp.zip*) and unzipped the project, do so now):

```
mvn install
```

Maven will build all of the projects ordered by the "reactor".

```
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Killer App
[INFO]   Killer App Model
[INFO]   Killer App API
[INFO]   Killer App Stores
[INFO]   Killer App Memory Store
[INFO]   Killer App XStream Store
[INFO]   Killer App Core
[INFO]   Killer App Commandline Interface
[INFO]   Killer App Web WAR
[INFO] ------------------------------------------------------------------------
[INFO] Building Killer App
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing ~/killerapp/pom.xml to ~/.m2/repository/com/training/killerapp/killerapp/1.0-SNAPSHOT/k
[INFO] ------------------------------------------------------------------------
[INFO] Building Killer App Model
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
...
```

It executes the full build lifecycle for each project before proceeding onto the next. This ensures that a project that depended upon by the next is built first. You may note that this kind of build hierarchy will not work for circular dependencies (when project A depends on project B which depends on project A). This is correct. Maven does not allow circular dependencies, and the build will fail (and it will tell you) if this is the case.

```
...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------------
[INFO] Killer App ..................................... SUCCESS [1.530s]
[INFO] Killer App Model ............................... SUCCESS [3.018s]
[INFO] Killer App API ................................. SUCCESS [0.161s]
[INFO] Killer App Stores .............................. SUCCESS [0.011s]
[INFO] Killer App Memory Store ........................ SUCCESS [0.666s]
[INFO] Killer App XStream Store ....................... SUCCESS [0.908s]
[INFO] Killer App Core ................................ SUCCESS [1.802s]
[INFO] Killer App Commandline Interface ............... SUCCESS [1.573s]
[INFO] Killer App Web WAR ............................. SUCCESS [0.644s]
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 24 seconds
[INFO] Finished at: Sun Nov 05 00:58:11 CST 2006
[INFO] Final Memory: 15M/28M
[INFO] ------------------------------------------------------------------------
```

This is the power of Maven's project relationship mechanism. One single base project can handle a whole hierarchy of projects. This project was chosen because it does a good job at representing a reasonable cross-section of issues encountered by a large variety of projects. A rough overview is as follows:

- The *killerapp-model* project houses the User and Page model object beans, generated via the Codehaus Modello plugin.
- The *killerapp-api* is the public API to the Killer App project. It contains the KillerApp and KillerAppDataStore interfaces.
- The *killerapp-stores* project is the parent and multi-module project of the two KillerAppDataStore implementations: killerapp-store-memory - a main memory Map implementation, and killerapp-store-xstream - a file persisted implementation populated via XStream.
- The *killerapp-core* project is the core implementation of the KillerApp. It comes in two "classes" backed by memory or xstream datastore.
- The *killerapp-cli* is the command-line interface. It can flex to be built on either the memory or xstream datastore *killerapp-core* classifier.
- The *killerapp-war* project is the web interface for the KillerApp as a simple WAR.

Finally, you can run the project WAR in memory mode with the help of the Jetty plugin.

```
cd killerapp-war
mvn jetty:run-war
```

That should start the WAR in a Jetty container via Jetty's `run-war` goal. Visit the WAR's URL in a browser (`http://localhost:8080/killerapp/`) and you will see the Killer App in action!

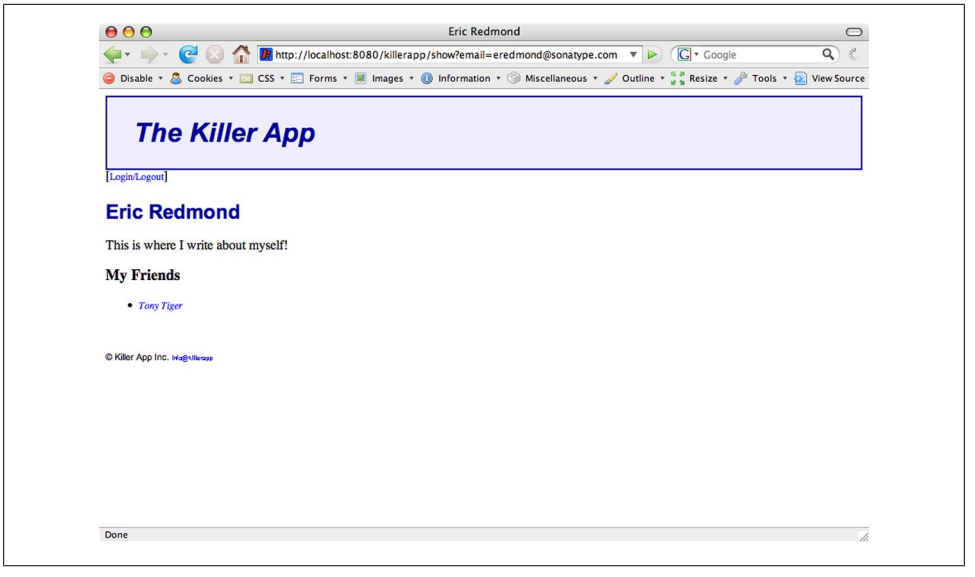


Figure 2-10. The Killer App in action

Look into the `killerapp-war/pom.xml` file and you may notice the following lines:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <version>6.0.1</version>
        <configuration>
          <contextPath>/killerapp</contextPath>
          <scanIntervalSeconds>15</scanIntervalSeconds>
          <connectors>
          <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
              <port>${jettyPort}</port>
            </connector>
          </connectors>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <properties>
    <jettyPort>8080</jettyPort>
  </properties>
  ...
</project>
```

This is all the configuration required to run a Jetty container test with Maven. If you had problems getting the sample to start due to port conflict, just change the value of the `jettyPort` property. Very much flexibility, very little changes. If we could make

some basic assumptions about port numbers, and had no desire to set a specific contextPath or relaunch on code changes every 15 seconds, this configuration would be unnecessary. And thus it is with Maven: stick to the convention and avoid `configuration`!

Getting Help

As comprehensive as a book tries to be, naturally you will encounter problems that we do-not/can-not/will-not cover. In such cases, we suggest searching for answers at the following locations:

- Maven website (*http://maven.apache.org*) - First place to look. A wealth of knowledge, constantly growing, though admittently, sometimes difficult to navigate or find answers if you are not already well on your way to understanding Maven.
- User mailing list - (users@maven.apache.org) very active, very friendly. Tell 'em we sent ya!
- Developer mailing list - (dev@maven.apache.org) only if questions/suggestions are directly related to Maven development, not usability questions.

Tips and Tricks

Most chapters will contain a "tips and tricks" section at the end. We will kick this tradition off with one you will use countless times: the `maven-help-plugin` (*`http://maven.apache.org/plugins/maven-help-plugin/`*).

Helping Yourself

The first three goals are simple. You run them in the base of a project directory, and they show you related information about the project. Remember, when you execute a goal you must first prefix it with the plugin name. For example, if you want to execute the `help` plugin's `active-profiles` goal, you would type `mvn help:active-profiles`.

- `help:active-profiles` lists the profiles which are currently active for the build.
- `help:effective-pom` displays the effective POM for the current build, with the active profiles factored in.
- `help:effective-settings` prints out the calculated settings for the project, given any profile enhancement and the inheritance of the global settings into the user-level settings.

The last goal is slightly more complex, showing you information about a given plugin (or one if it's goals). You must at least give the groupId and artifactId of the plugin you wish to describe.

- `help:describe` describes the attirbutes of a plugin. This need not run under an existing project directory.

With the `plugin` parameter you can specify a plugin you wish to investigate, passing in either the plugin prefix (e.g. maven-help-plugin as `help`) or the `groupId:artifact[:version]`, where version is optional.

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

Gives you

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [help:describe] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [help:describe]
[INFO] Plugin: 'org.apache.maven.plugins:maven-help-plugin:2.0.1'
----------------------------------------------
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:     2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
    the build environment. It includes the ability to view the effective
    POM and settings files, after inheritance and active profiles
   have been applied, as well as a describe a particular plugin goal to give usage information.

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Mar 14 12:03:04 CDT 2007
[INFO] Final Memory: 2M/5M
[INFO] ------------------------------------------------------------------------
```

This is rarely useful for anything other than finding a plugin's version, however. The parameter `full` is worth its (metaphorical) weight in (imaginary) gold.

```
> mvn help:describe -Dplugin=help -Dfull

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [help:describe] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [help:describe]
[INFO] Plugin: 'org.apache.maven.plugins:maven-help-plugin:2.0.1'
----------------------------------------------
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:     2.0.1
```

```
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
    the build environment. It includes the ability to view the effective
    POM and settings files, after inheritance and active profiles
   have been applied, as well as a describe a particular plugin goal to give usage information.

Mojos:

================================================
Goal: 'active-profiles'
================================================
Description:

Lists the profiles which are currently active for this build.

Implementation: org.apache.maven.plugins.help.ActiveProfilesMojo
Language: java

Parameters:
------------------------------------------------

[0] Name: output
Type: java.io.File
Required: false
Directly editable: true
Description:

This is an optional parameter for a file destination for the output of this mojo...the listing of active pr

------------------------------------------------

[1] Name: projects
Type: java.util.List
Required: true
Directly editable: false
Description:

This is the list of projects currently slated to be built by Maven.

------------------------------------------------

This mojo doesn't have any component requirements.
================================================

... <remove the other goals> ...

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Mar 14 12:08:28 CDT 2007
[INFO] Final Memory: 2M/5M
[INFO] ------------------------------------------------------------------------
```

This option is great for discovering all of a plugin's goals as well as their parameters. But sometimes this is far more information than necessary. To get information about a single goal, set the `mojo` parameter instead.

One last tip: All of the help plugins can output to a file via a `output` parameter. Used like this

```
mvn help:effective-pom -Doutput=my-output-file.log
```

Configuring Unit Tests from Maven

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <systemProperties>
            <property>
              <name>my-property</name>
              <value>someValue</value>
            </property>
          </systemProperties>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

```
public class PropertyTest extends junit.framework.TestCase
{
  public void testProperty()
  {
    String propertyValue = System.getProperty( "my-property" );
    assertEquals( "someValue", propertyValue );
  }
}
```

Summary

We have created a simple project, packaged the project into a jar, installed that jar into the Maven repository for use by other projects, and generated a site with documentation... all without writing a single line of code or touching a single configuration file. With a little configuration in the POM the Jetty plugin was able to help us test the WAR of the KillerApp.

CHAPTER 3

The POM and Project Relationships

A place for everything and everything in its place. -- Victorian Proverb

This chapter follows an example project. You may wish to download killerapp (http://www.sonatype.com/book/examples/book-killerapp.zip) and follow along.

The POM

Maven, as a project management system, is based upon two major concepts. The first is the concept that all projects are best represented as objects, or things, or nouns. We speak of a project, or the project as a thing. Speach is a manifestation of existing conceptualizations, so we feel it best to actually represent a project as an "object", since the majority of organizations think of them in that way anyway. This is opposed to Ant which has no such project declaration, since all of its builds are limited strictly to only the second of the two concepts - actions - which it calls "tasks". The second concept is that everything else that is not objectifiable is a goal, or an action, or verb. You compile a project, or deploy a project. Since you already do things to a project, it makes sense that these should remain exclusively as actions. Moreover, those "things" you want done to the project are more than just normal actions, they are goals you wish to achieve. They are not simply tasks to be done, but some sort of transformation using some aspect of the project or build system to achieve. Thus, we call them "goals". This chapter focuses on the first of these two concepts: the object.

The POM is the conceptual project, reified.

Maven lumps all pieces of a project into a single conceptually convenient Project Object Model, or POM. Each project contains one and only one pom.xml file that represents the project in a declarative manner. It defines things like the project's name, the developers who work on the project, the url for the project on the web, the coordinates of the project, and everything else specific to the project. This is important to note, because there is more information in Maven than just the single POM, for example, the settings.xml files which define information for the build system as a whole - such as server authentication data. For now we will stay focused on the POM.

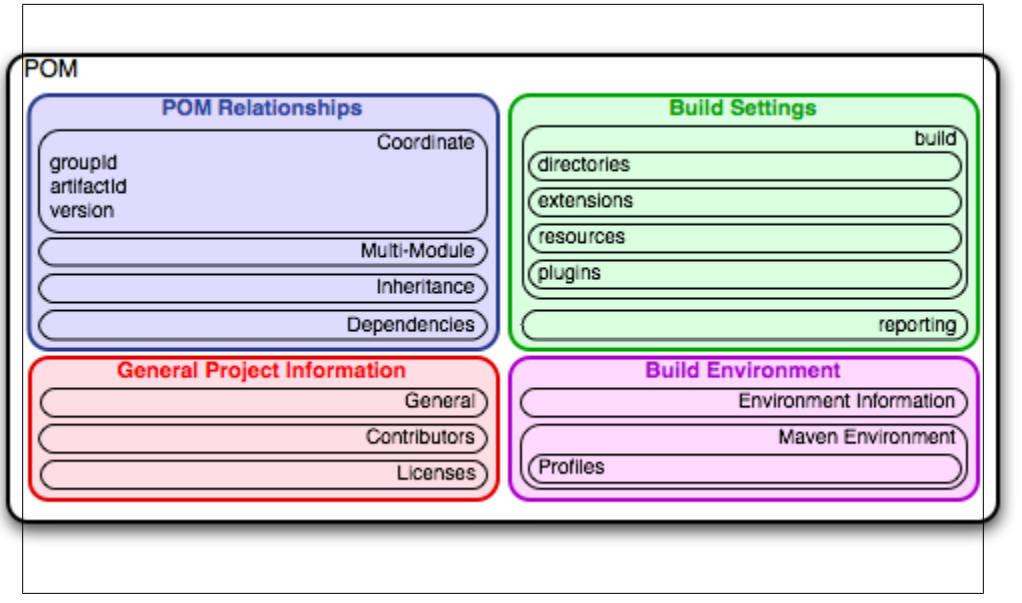


Figure 3-1. A small model of the POM

If you navigate to the killerapp-api file of the Killer App hierarchy you downloaded in the end of the previous chapter, you will notice a pom.xml file. Actually, you will notice many POM files throughout the directory structures. Each of these POMs represent a project. Maven 2.x is represented by POM version 4.0.0. Maven 1.x used version 3.0, which itself was incremented from two previous versions.

The killerapp-stores pom.xml file is as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.training.killerapp</groupId>
    <artifactId>killerapp</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>killerapp-stores</artifactId>
  <name>Killer App Stores</name>
  <packaging>pom</packaging>

  <dependencies>
    <dependency>
      <groupId>com.training.killerapp</groupId>
      <artifactId>killerapp-api</artifactId>
    </dependency>
  </dependencies>

  <modules>
    <module>killerapp-store-memory</module>
    <module>killerapp-store-xstream</module>
```

```
    </modules>

</project>
```

It is a fairly simple pom, only slightly more complex than the one created in the previous chapter via the archetype:create goal. You will notice that there are three different places where "artifactId" is used: under the "parent" element, under the "dependency" element, and the artifactId of this POM itself under the "project" element. There are also elements called "module"s that look suspiciously like artifactIds. This POM represents all of the major POM relationship types in Maven: Dependencies, multi-modules, and Inheritence. The glues that holds these methods of relating together is Maven's coordinate system.

Seperation of Concerns

A good practice to follow for maintaining system integrity, which is the the true role of an architect (don't let anyone tell you differently). Maven helps guide your project design into a good best-practice seperation of concerns (SoC).

More on Coordinates

We mentioned coordinates in Chapter 2 (*simple-project.html*). They are mentioned again here because of their central importance to Maven's ability to manage project relationships. Coordinates are Maven's way of tracking a specific project, like an address, but with a temporal aspect. More than pinpointing a specific project, Maven can pinpoint a specific project in time by its "version" element. Coordinates are used by dependency, plugin and extension elements, as well as for defining inheritence, and anywhere else the project or goal configuration may need to know information about including - or excluding - a specific project.

If we go up to the parent `pom.xml` file of the Killer App project you will see the coordinates peppered throughout the file. Most obvious is the project's own required coordinate definition:

```
<project>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  ...
</project>
```

The `version` element is always required in some capacity. If not specifically specified where you see it, it is likely specified in a parent project, or some sort of management meta-element, or may be inferred through a plugin. The other element you will notice is the packaging element. This tells Maven that the artifact created by this project is a single file: the POM - albeit a possibly annotated version of it. The default packaging type is a jar, and so the default coordinate assumes this is the case.

Whereas the my-app project of the previous chapter had the coordinates `mavenbook:my-app:jar:1.0-SNAPSHOT` this project is represented as `com.training.killerapp:killerapp:pom:1.0-SNAPSHOT`.

How does Maven use these coordinates? In multiple ways. Firstly, it uses them to retrieve the specific projects from remote and local repositories. It also uses coordinates to decifer how project hierarchies fit together, and plugins use them for a wide array of things, such as the maven-war-plugin, that uses coordinates to decide what projects to package into the war.

If you have run `mvn install` on this project, you may wonder what it means to "install" a project. In Maven, *installing* a project means placing a copy of the project's POM and the generated artifact (in `jar` packaging type projects, the artifact is a JAR; in `pom` projects, the artifact is only the POM, etc.) to a local directory structure mirroring the groupId, artifactId and version of the coordinate. The default location for your local repository is `.m2/repository` under your base user directory (retrieved from Java's "user.home" property) represented as `${user.home}`. Each artifact is under its groupId - with dot-notation seperated into directory hierarchy. This is similar to Java's packaging structure. The groupId is followed by the artifactId, followed by version. The POM file can be found under this directory, named as the `artifactId-version.pom`.

In general, the project's packaged artifact lives with its pom, resulting in the following two files (not including optional checksum files, like md5):

```
groupId/artifactId/version/artifactId-version.pom
groupId/artifactId/version/artifactId-version.packaging
```

For our example, the "killerapp" project you have installed above will reside under:

```
${user.home}/.m2/repository/com/training/killerapp/killerapp/1.0-SNAPSHOT/killerapp-1.0-SNAPSHOT.pom
```

Since it is a `pom` type project, the pom.xml file is its own artifact. The killerapp-api project, on the other hand, it a jar project, thus yeilding the following files installed into the local repository:

```
${user.home}/.m2/repository/com/training/killerapp/killerapp-api/1.0-SNAPSHOT/killerapp-api-1.0-SNAPSHOT
${user.home}/.m2/repository/com/training/killerapp/killerapp-api/1.0-SNAPSHOT/killerapp-api-1.0-SNAPSHOT
```

We cover local and remote repositories with more depth in Chapter 13 (*repository.html*). For now simply note the relationship between project coordinates and where they are installed.

Project Relationships

One powerful aspect of Maven is in its handling of project relationships; that includes dependencies (and transitive dependencies), inheritance, and multi-module projects. Dependency management has a long tradition of being a complicated mess for anything but the most trivial of projects. "Jarmageddon" quickly ensues as the dependency tree becomes large and complicated. "Jar Hell" follows, where versions of dependencies on

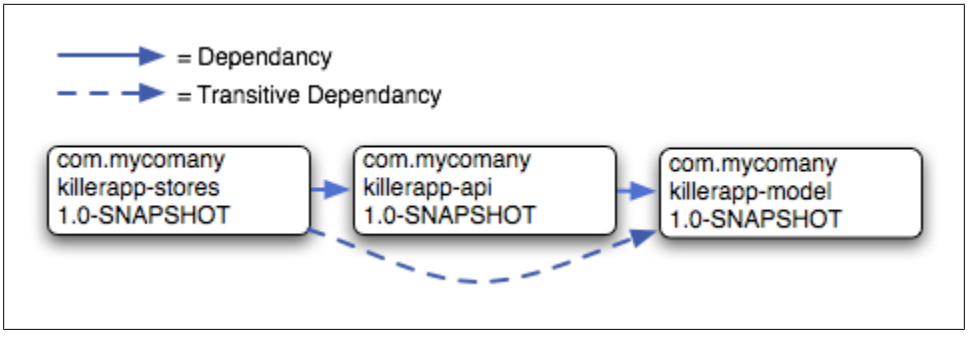


Figure 3-2. A Chain of Project Dependencies

one system are not equivalent to versions as those developed with, either by the wrong version given, or conflicting versions between similarly named jars. Maven solves both problems through a common local repository from which to link projects correctly, versions and all.

Dependencies

The most immediately useful piece of Maven's project relationship mechanism is project dependencies. *A dependency is any project that is required by this project to perform some function.* That function could be compilation, testing or simply runtime.

Before Maven most build systems would require the build user to provide the dependencies of a project on their own. A common method of dealing with this problem was to check in required binaries to version control, which has problems of its own, wasting a significant amount of bandwidth and disk space. Typically you would want the jars to be contained by your tree so that they can be versioned along with it. This may mean that each development branch on a machine has a copy of those jars. It also means they are checked for diffs each time a commit is done. Since they rarely change, another common pattern is to locate the jars in a tree separate from the source. This works around the problems mentioned above, but now you lose reproducibility over time. Another common problem is that the jars aren't versioned in their file name so it's difficult to determine what versions are actually present. These /lib folders tend to become dumping grounds for jars over time that are difficult to keep in sync with the actual dependencies used.

Most of the projects in the killerapp have dependencies. The dependency list of the parent killerapp project is as follows:

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

```
    </dependency>
  </dependencies>
```

Simple! This specifies the coordinates for a project that the killerapp depends upon, as well as the scope of the dependency. Notice that the packaging type is not specified. That is because, like the POM definition itself, the type defaults to "jar" when not specified. Note that junit is required for testing only. That means that for the project compilation or normal runtime, junit is not necessary. However, the project build will only succeed with junit available in the repository, since that would cause the test to fail. The valid scope types are:

- **compile** - this is the default scope, used if none is specified. Compile dependencies are available in all classpaths.
- **provided** - this is much like compile, but indicates you expect the JDK or a container to provide it. It is only available on the compilation classpath (not runtime), is not transitive, nor is it packaged.
- **runtime** - this scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
- **test** - this scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.
- **system** - this scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository. If you declare the scope to be system you must also provide the `systemPath` element. Note that this scope is not recommended, and will likely be removed in future versions of Maven (it is always desired that libraries are in the Maven repository).

Transitive Dependencies

Each of the scope options in the above list affects more than just the scope of the dependency in the declaring project, but also how it acts as a transitive dependency (when this project is itself a dependency of another project). The easiest way to convey this information is through a table. Scopes in the top row represent a dependency-of-a-dependency's scope (transative dependency), while the scopes in the left represent the direct dependency's scope (non-transative) in ths current project. Where they intersect is the scope given to the transative dependency in the current project. Blank on the table means the dependency will be omitted.

	compile	**provided**	**runtime**	**test**
compile	compile	-	runtime	-
provided	provided	provided	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

Conflict Resolution.

If you open the `killerapp-cli` you will notice a more complex dependency definition.

```
<dependency>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp-core</artifactId>
  <version>${project.version}</version>
  <classifier>memory</classifier>
  <exclusions>
    <exclusion>
      <groupId>com.training.killerapp</groupId>
      <artifactId>killerapp-store-xstream</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

This dependency contains an exclusion. An exclusion is a transitive dependency that this project does not wish to require.

By default - unless that project defines its dependencies as "optional" - transative dependencies resolve to the scope following the table above. This element supresses that behavior. In the case of the above definition, the `killerapp-core` project depends upon the `killerapp-store-xstream` project (check the `killerapp-core` POM if you like). The `killerapp-cli` project, however, does not wish to require that dependency, so it is excluded.

Here are a few other reasons you may wish to exclude transitive dependencies.

1. The groupId or artifactId of the artifact has changed, where the current project requires an alternately named version from a dependency's version - resulting in 2 copies of a differently versioned project in the classpath.
2. An artifact that is unused (although the dependency should have declared this optional in it's own POM).
3. An artifact which is provided by your runtime container thus should not be included with your build
4. An API which might have multiple vendors -- ie Sun API which requires click-wrap licensing and manual install into repo vs CDDL/GNU licensed version of the same API which is freely available in the repo.

Versions

Versions are more than just a sequence of numbers, but codifiers of information. They describe an evolving software project through time. In Maven the version number are parsed in a simple manner:

```
major.minor.bug_fix-qualifier-build_number
```

for example:

```
1.3.5-alpha-1
```

When comparing verion numbers: `major` is resolved before `minor`, then `bug_fix`. The `qualifier` is merely compared as a string (luckly for us, `alpha` is before `beta`, which is before `releasecandidate` or `RC`). Finally if all previous versions match, the `build_number` is compared numerically. Finally, if the version cannot be parsed, they are compared entirely as strings.

SNAPSHOT.

`SNAPSHOT` is a special quality that can be appended to the end of a version. It describes to both the user and the Maven system that this project is currently in development, and should be treated as such. You may deploy a specific version of a SNAPSHOT to a remote repository, but users must specifically enable the ability to download snapshots to their projects. This helps keep in development SNAPSHOTs seperate from released versions. Read more about it in The Maven Repository chapter (*repository.html*).

```
1.3.5-alpha-1-SNAPSHOT
```

One last note: when releasing a project you must resolve all -SNAPSHOT versions. The practical reason for this is that you do not wish your users to require on development versions of a project if they are expecting a release-quality item. But functionally, for the reason given above, by default Maven does not download SNAPSHOT versions from repositories - so having a release version depending upon SNAPSHOT versioned dependencies will make those dependencies inaccessible by default.

Ranges.

Versions can be matched by more than a scalar values. They can be given a range of possible values - even with no end. You can do this by using version ranges.

- `(, )` - exclusive quantifiers
- `[, ]` - inclusive quantifiers

For example, if you wished to access any JUnit version greater than or equal to 3.8, but less than 4.0, your dependency would be thus:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[3.8,4.0)</version>
  <scope>test</scope>
</dependency>
```

Note that a version before or after the comma is no required, and means +/- infinity, respectively. `[4.0,)` means any version greater than or equal to `4.0`. `(,2.0)` is any version less than `2.0`. `[1.2]` means only version `1.2`, and nothing else.

Note on Properties

The other thing you will notice is the version is not specifically given, but instead given as a Maven property. Maven properties occur frequently in advanced Maven usage, and are similar to properties in other systems such as Ant. They are simply variables

delimited by `${...}`. Just to break on a short tangent, let us quickly cover the various types of properties available in the Maven POMs:

- env.X: Prefixing a variable with "`env.`" will return the shellâs environment variable. For example, `${env.PATH}` contains the `$PATH` environment variable - or `%PATH%` in Windows.
- project.x: A dot (.) notated path in the POM will contain the corresponding element's value. For example: <project><groupId>org.apache.maven</groupId></project> is accessed as `${project.groupId}`.
- settings.x: A dot (.) notated path in the settings.xml will contain the corresponding element's value. For example: <settings><offline>false</offline></settings> is accessible via `${settings.offline}`. More on the setttings file in a later chapter.
- Java System Properties: All properties accessible via `java.lang.System.getProperties()` are available as POM properties, such as `${java.home}`.
- x: Set within a <properties \/> element or an external files, the value may be used as `${x}`. This is the simplest and most straight-forward property.

You can find more about properties in Appendix 3 (*properties.html*).

Finally we mention the `classifier` element. We will cover how to create artifacts with classifiers later in this book, however, for now just note that those elements which have classifiers may be accessed by specifying this element.

Multi-module

Multi-module projects are pom type projects that contain a list of modules to build. Simple, elegant, and filled with vast potential, multi-module projects are always of packaging type "pom". This is because they do not (normally) produce artifacts of their own, but exist merely to group together other projects - and sometimes other multi-modules. The `killerapp:killerapp:pom:1.0-SNAPSHOT` project is a multi-module project since it contains a list of modules that can be managed as a group. We have already touched on the power of these projects when we ran "mvn install" under the base directory. When the Maven commandline printed out the following list:

```
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Killer App
[INFO]   Killer App Model
[INFO]   Killer App API
[INFO]   Killer App Stores
[INFO]   Killer App Memory Store
[INFO]   Killer App XStream Store
[INFO]   Killer App Core
[INFO]   Killer App Commandline Interface
[INFO]   Killer App Web WAR
```

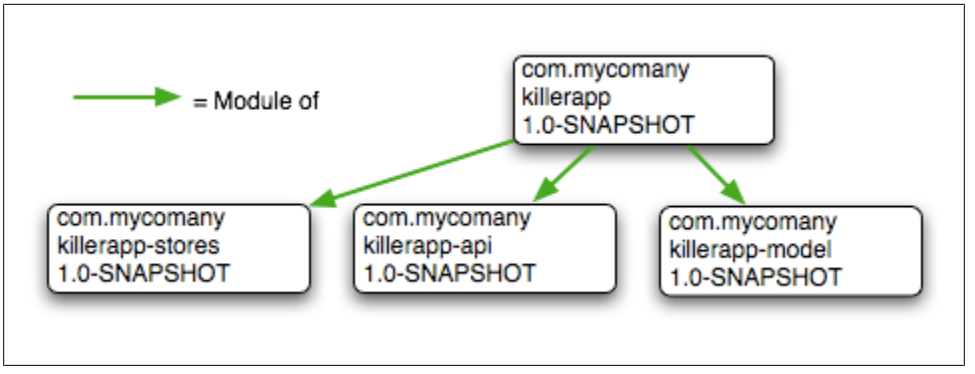


Figure 3-3. A Multi-Module Project and its Modules

It discovered the list because they were added as modules, and those modules were the names of directories immediately under the current POM `${basedir}` (the current project's base directory). The Killer App Stores project is also a multi-module - manages Memory and XStream Stores.

Note that we call the projects under the multi-module projects "modules" and not "children" or "child projects". This is purposeful, so as not to confuse projects grouped by multi-module project with another type of project relationship: inheritence.

Inheritence

Under the killerapp-api project type the following command:

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

Comparing that pom.xml to the contents of the killerapp-api's pom.xml should yield surprising results. Mailing lists? JUnit and `killerapp-model` version in the dependency list? And other information we have not encountered before, such as the build element and child elements. Were did all of this data come from? Peeking at the parent killerapp project should yield some answers. The killerapp-model (and, indeed, all of the projects in the Killer App) inherit from the com.training.killerapp:killerapp project, declared in each POM via the parent element.

```
<parent>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

Remember above we mentioned that groupId/artifactId/versionId are required in some capacity? The killerapp-api POM does not specify a groupId or version. It inherits those values from the parent. Maven assumes that the parent is either installed into the local repository, or available in the parent directory (../pom.xml) of the current project. If neither is true (for example, some organizations prefer to put the parent project in its

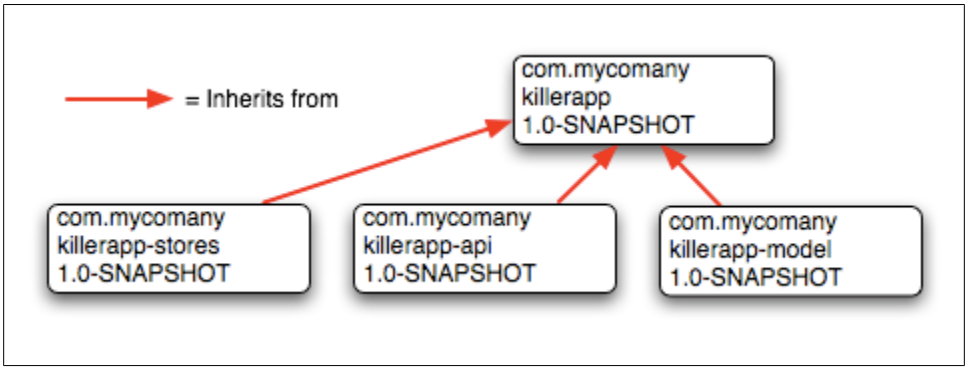


Figure 3-4. A Parent Project that is Inherited From

own directory) this default may be overridden via the relativePath - for example: ../parent-project/pom.xml.

- dependencies
- developers and contributors
- plugin lists
- reports lists
- plugin executions with matching ids
- plugin configuration

Oftentimes a set of projects will require similar values for the set. Just like the inheritence of any other object oriented system (of which Maven is one... it is the Project Object Model afterall) child projects will inherit many of a parent projects values.

The Ultimate Parent

All POMs ultimately inherit from the SuperPOM. The SuperPOM plays a similar role to Java's java.lang.Object class. It is a set of pre-defined values from which all other POMs minimally inherit. This is Convention over Configuration at work. Since there are several values required for proper POM operation, and most of these values are similar for many projects, they are set as the default values of all POMs. The specific values of the SuperPOM can be found in Appendix A (*pom-details.html*). Some important values to note from the SuperPOM are under the "build" element.

The build element defines information directly concerning a project's build settings (as opposed to, say, the project's `name` - like "`Killer App API`"). Up until now we have built the projects without concern for how Maven knows where to look for a project's files, and how it knows where to build and place its artifact. This information is configurable by any Maven project, but inherits certain defaults from the Maven SuperPOM. Any directories which are not absolute (begins with a / in unix environment, a driver letter

like C: in windows) are assumed to be relative to the project's base directory (or `${basedir}`.

```
<project>
  ...
  <build>
    <!-- the base directory where build artifacts are placed -->
    <directory>target</directory>

    <!-- where compiled files are placed -->
    <outputDirectory>target/classes</outputDirectory>

    <!-- The name of the final artifact sans extension (or possible classifiers) -->
    <finalName>${artifactId}-${version}</finalName>

    <!-- where compiled test files are placed -->
    <testOutputDirectory>target/test-classes</testOutputDirectory>

    <!-- Where the Java source files are expected to be -->
    <sourceDirectory>src/main/java</sourceDirectory>

    <!-- Where non-Java script source files are expected to be -->
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>

    <!-- Where the test Java source files are expected to be -->
    <testSourceDirectory>src/test/java</testSourceDirectory>
    <resources>
      <resource>
       <!-- Where non-compiled files reside (such as .properties file). They are put into the outputDirecto
        <directory>src/main/resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
       <!-- Where non-compiled files reside. They are put into the testOutputDirectory -->
        <directory>src/test/resources</directory>
      </testResource>
    </testResources>
  </build>
  ...
</project>
```

There is more to being a parent than just defining defaults for all of its children to inherit. Sometimes the parent needs to be able to manage a child's settings if it requires them at all. Enter the `dependencyManagement` element.

Dependency Management

You may notice in the `com.training.killerapp:killerapp` `pom.xml` file the `dependency Management` definition contains its own `dependencies` element. This is not redundency on the part of the POM schema, but a mechanism for managing child the dependencies of any child project that require them. Like good parents, parent POMs sometimes

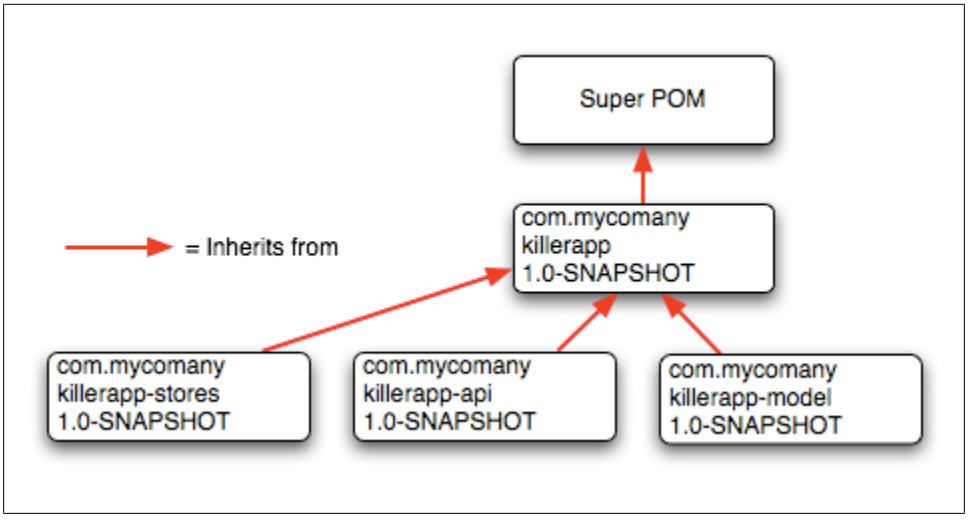


Figure 3-5. The SuperPOM is always the base Parent

needs to manage details for its children. The following is a piece of the `dependencyManagement` element from the `pom.xml` file.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.training.killerapp</groupId>
      <artifactId>killerapp-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    ...
  <dependencies>
</dependencyManagement>
```

Unlike the `junit` dependency defined in the "Dependencies" section in this chapter, this definition does not actually add the `killerapp-model` dependency to the `killerapp` project and its children. Meaning, that any children that inherit from this project will not require the `killerapp-model` project. Instead, this merely "manages" any matching dependencies that this project or its children may have. Look at the `killerapp-api`'s pom file and you will notice that it does not contain a version - although as mentioned above, version is required. Thanks to the `dependencyManagement` definition the version has already been specified for all of its children.

- `dependencyManagement` - is often used by parent POMs to help manage dependency information across all of its children. If the my-parent project uses `dependencyManagement` to define a dependency on junit:junit:4.0, then POMs inheriting from this one can set their dependency giving the groupId=junit and artifactId=junit only, then Maven will fill in the version set by the parent. The benefits of this method are obvious. Dependency details can be set in one central location, which will propagate to all inheriting child POMs.

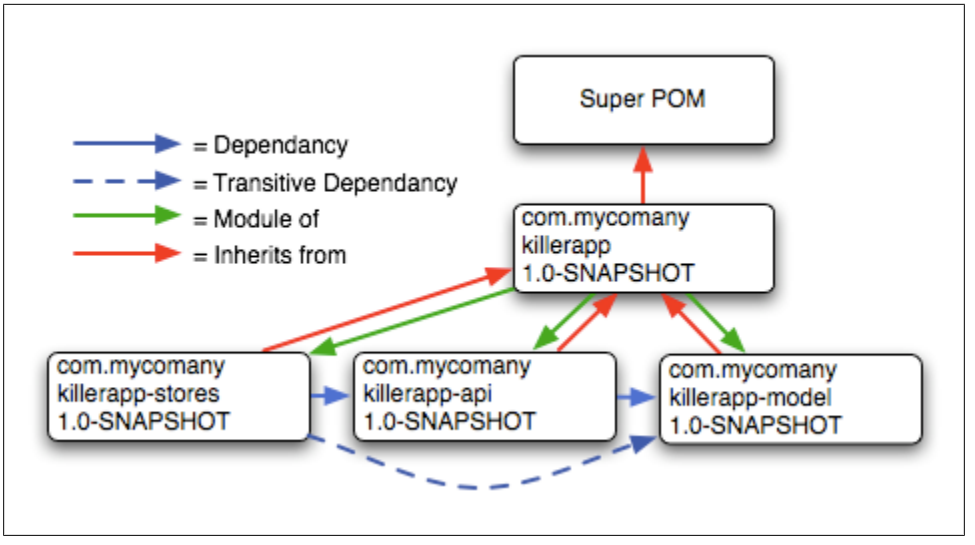


Figure 3-6. Project Relationships Overall

Multi-module versus Inheritance

To re-tread on the previous field, there is a difference between inheriting from a parent project, and being managed by an multi-module project. A parent project is one that passes its values to its children. This is different from a multi-module project, where it merely manages a group of other sub-project or module - however no values are actually passed to the modules it manages. Because of this disconnect - and it is commonly desired that a single project both manage a group of projects as well as set values they they can each utilize - a parent project will commonly contain its children as modules (such as `killerapp` and `killerapp-stores`). This is not to believe that inheritence and multi-modules are related - so do not be confused - they are merely complimentary - yet a best practice to follow for simplicity and cohesion (only one top-level "master" project).

All Together

Maven's goal is to make project management conceptually simpler. When we put the projects above together we get the following.

There are more complex pieces for certain, for example inheriting dependencies, or dependency scope. However, understanding the above graph will take you a long way to understanding Maven's project relationship combinations.

Tips and Tricks

Comprehensive Overview

You can find more details about the POM 4.0.0 structure in Appendix 1 (*pom-details.html*).

Properties

Properties are an excellent way of helping to "future-proof" your POM - especially in combination with a Parent POM. Check out the Tips and Tricks section in Appendix 3: Properties (*properties.html*): *Synchronize versions/groups with an Inherited Property*.

Dependency Groupers

This is a simple trick used to keep a set of dependencies grouped together. For example, the Microsoft SQLServer JDBC implementation which consists of three jars: `msbase.jar`, `msutil.jar`, and `mssqlserver.jar`. The following example assumes that you have a repository containing these jars, grouped under "com.microsoft":

```
<project>
  <groupId>com.mycompany</groupId>
  <artifactId>jdbc</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>com.microsoft</groupId>
      <artifactId>msbase</artifactId>
      <version>${msJdbcVersion}</version>
    </dependency>
    <dependency>
      <groupId>com.microsoft</groupId>
      <artifactId>msutil</artifactId>
      <version>${msJdbcVersion}</version>
    </dependency>
    <dependency>
      <groupId>com.microsoft</groupId>
      <artifactId>mssqlserver</artifactId>
      <version>${msJdbcVersion}</version>
    </dependency>
  </dependencies>
  <properties>
    <msJdbcVersion>(1.0,)</msJdbcVersion>
  </properties>
</project>
```

Install the above project - since it's packaging type is "pom" rather than something like "jar", this file alone is placed into the repository. You can now add this project as a

dependency (*do not forget to add the "pom" type*) and all of it's dependencies will be added to your project - the power of transitive dependencies at work!

```
<project>
  <description>This is a project requiring JDBC</description>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>com.mycompany</groupId>
      <artifactId>jdbc</artifactId>
      <version>1.0</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</project>
```

If you later decide to switch to a different JDBC driver, for example, JTDS, just replace the dependencies in the `com.mycompany:jdbc` project to use `net.sourceforge.jtds:jtds` and update the version. All of your dependant projects will use `jtds` if they decide to update to the newer version. It is an easy way to roll out mass changes without breaking anyone unexpectedly.

```
<project>
  <groupId>com.mycompany</groupId>
  <artifactId>jdbc</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>net.sourceforge.jtds</groupId>
      <artifactId>jtds</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>
</project>
```

Summary

This chapter we covered the different ways in which Maven projects relate to each other: as dependencies, or as hierarchies via inheritence or multi-modules, and followed how the Killer App uses these concepts to create a complex project hierarchy with little configuration. The `com.training.killerapp:killerapp` project pulls double-duty as both a multi-module project and a parent project. So far we have learned that with a single execution (mvn install) we can build an entire hierarchy of projects, test them and install them for future use by other projects into our local repository. Maven's objectification of projects allows it to easily manage project relationships with simple definitions, and propogate actions and relationships to other projects. We have covered the concept of project as object - in the next chapter we explore the concept of object-

base actions, in the form of goals and phases, and how they too have a framework for relating to each other.

CHAPTER 4
The Build Lifecycle

Complexity is a symptom of confusion, not a cause. -- Jeff Hawkins

This chapter follows an example project. You may wish to download maven-zip-plugin (http://www.sonatype.com/book/examples/book-lifecycle.zip) and follow along.

A Structure for Goal Execution

As mentioned in the chapter on Project Relationships (*pom-relationships.html*), Maven builds are based upon two concepts: the idea that projects are best conceptualized and managed as atomic, inter-relating objects - and that eveything else is an action taken upon them. Just as Maven manages the project-as-object concept via a `pom.xml`, and relationships via the `dependency` elements, inheritence and multi-modules, Maven actions are also managed in a well-defined list called the *build lifecycle*.

For the person building a project, this is a great improvement over a system like Ant, where there are innumerable possible names for the tasks that may be performed, varying per project. It is often very difficult for a new person to manage any significantly complex build mechanism without either extensive training as to the nature of certain tasks, or to read the `build.xml` files. A Maven user need only learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

As mentioned in a previous chapter (*simple-project.html*), a build lifecycle is an organized sequence of phases that exist to give order to a set of goals. Those goals are chosen and bound by the packaging type of the project being acted upon. There are three standard lifecycles in Maven: `clean`, `default` (sometimes called `build`) and `site`.

clean

Clean is the simplest lifecycle with only three phases:

- **pre-clean**

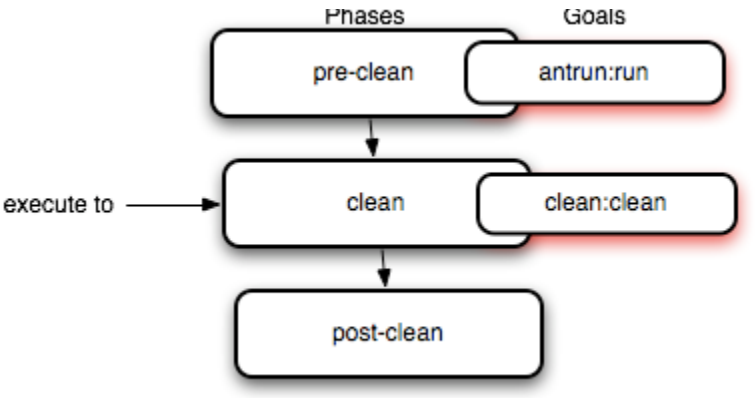
- **clean**
- **post-clean**

Most times you will only use the "clean" phase, which is bound by the `clean:clean` goal. All this plugin does is delete the files under the build directory which is defaulted to "target" by the SuperPOM. However, you should not execute the `clean:clean` goal directory. The reason being that there are three phases to the clean lifecycle for a reason. It is entirely acceptible to bind other goals to the `pre-clean` or `post-clean` phases, which some Maven POMs may take advantage of.

For example, suppose you wanted to trigger an `antrun:run` goal task to echo a notification on `pre-clean`. Simply running the `clean:clean` goal will not execute the lifecycle at all, but running `clean` will first run `pre-clean`. The `killerapp-model` POM binds the `antrun:run` goal to the `pre-clean` phase, which contains tasks alerting the user if the project artifact is about to be deleted or not. *As an aside, the maven-antrun-plugin is a useful mechanism for executing simple tasks. However, a sharable Maven goal is always preferable to an embeded task - less copy-n-pasting in the long-run.*

```
<project>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <id>file-exists</id>
            <phase>pre-clean</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <!-- adds the ant-contrib tasks (if/then/else used below) -->
                <taskdef resource="net/sf/antcontrib/antcontrib.properties"/>

                <available
                   file="${project.build.directory}/${project.build.finalName}.jar"
                     property="file.exists" value="true" />
                <if>
                  <not><isset property="file.exists" /></not>
                <then><echo>No ${project.build.finalName}.${project.packaging} to delete</echo></then>
                <else><echo>Deleting ${project.build.finalName}.${project.packaging}</echo></else>
                  </if>
                </tasks>
            </configuration>
          </execution>
        </executions>
        <dependencies>
          <dependency>
            <groupId>ant-contrib</groupId>
```



If you run the `clean` phase, you will see something similar to the following execute (you may see more information print out if Maven must retrieve the antrun plugin and its dependencies from Maven's central repository before execution):

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Killer App Model
[INFO]    task-segment: [clean]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: file-exists}]
[INFO] Executing tasks
     [echo] Deleting killerapp-model-1.0-SNAPSHOT.jar
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] Deleting directory ~/killerapp/killerapp-model/target
[INFO] Deleting directory ~/killerapp/killerapp-model/target/classes
[INFO] Deleting directory ~/killerapp/killerapp-model/target/test-classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Nov 08 11:46:26 CST 2006
[INFO] Final Memory: 2M/5M
[INFO] ------------------------------------------------------------------------
```

One last note about the `clean` plugin - you are not constrained to the default behavior (delete everything in the build directory). You can configure the plugin to remove specific files in a `fileSet`. The example below configures clean to remove all `.class` files in a directory named `othertarget` using standard Ant file wildcards, * and **.

```
<project>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <configuration>
          <filesets>
            <fileset>
              <directory>othertarget</directory>
              <includes>
                <include>*.class</include>
              </includes>
            </fileset>
          </filesets>
```

```
      </configuration>
    </plugin>
```

You can also create a set of `excludes` files.

default

This lifecycle is really the meat of Maven's action framework. As mentioned in chapter 2 (*simple-project.html*), the default lifecycle is made up of a sequence of phases which, upon execution of a chosen phase, Maven first executes every previous phase in order beginning with the first. The phase sequence is based upon an abstraction of a standard build lifecycle. For example the first phase is validate, where any bound goal is expected to validate that the project conforms to some basic expectations - like project structure or the existence of certain files. From there source files are generated or manipulated then compiled. After that tests are compiled and executed. Barring any failures the compiled files and resources are packaged, integration tested, verified and installed. Finally, the POM and artifact are deployed to a remote repository. Note that this entire lifecycle will not be executed every time. Indeed, the majority of the time - for example during development - you only desire to run up to the `compile` phase, or `test`.

- **validate** - validate the project is correct and all necessary information is available
- **generate-sources** - generate any source code for inclusion in compilation
- **process-sources** - process the source code, for example to filter any values
- **generate-resources** - generate resources for inclusion in the package
- **process-resources** - copy and process the resources into the destination directory, ready for packaging
- **compile** - compile the source code of the project
- **process-classes** - post-process the generated files from compilation, for example to do bytecode enhancement on Java classes
- **generate-test-sources** - generate any test source code for inclusion in compilation
- **process-test-sources** - process the test source code, for example to filter any values
- **generate-test-resources** - create resources for testing
- **process-test-resources** - copy and process the resources into the test destination directory
- **test-compile** - compile the test source code into the test destination directory
- **test** - run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **prepare-package** - perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package (coming in Maven 2.1+)

- **package** - take the compiled code and package it in its distributable format, such as a JAR
- **pre-integration-test** - perform actions required before integration tests are executed. This may involve things such as setting up the required environment
- **integration-test** - process and deploy the package if necessary into an environment where integration tests can be run
- **post-integration-test** - perform actions required after integration tests have been executed. This may including cleaning up the environment
- **verify** - run any checks to verify the package is valid and meets quality criteria
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects

The specific goals bound to each phase default to a set of values declared by the project's packaging type. Although the `killerapp-stores` `pom` packaging will still execute the same set of phases in the same order as the killerapp-api `jar` packaging, the goals that are bound to those phases will be different. Where the `package` phase in a `pom` project will execute the `site:attach-descriptor` goal, a `jar` project will execute the `jar:jar` goal instead.

The following are the built-in packaging types in Maven, and the goals that are bound to their lifecycles by default.

jar

Jar is the default packaging type, the most common, and thus the most commonly encountered lifecycle configuration.

- **process-resources** - resources:resources
- **compile** - compiler:compile
- **process-test-resources** - resources:testResources
- **test-compile** - compiler:testCompile
- **test** - surefire:test
- **package** - jar:jar
- **install** - install:install
- **deploy** - deploy:deploy

pom

POM is the simplest packaging type. The artifact that it generates is itself only, rather than a jar or ejb package.

- **package** - site:attach-descriptor
- **install** - install:install
- **deploy** - deploy:deploy

maven-plugin

You will notice that this packaging type is similar to jar with three additions: `plugin:descriptor`, `plugin:addPluginArtifactMetadata`, and `plugin:updateRegistry`. These goals generate a descriptor file and perform some modifications to the repository data. We cover Maven plugins in more detail in the Using Plugins Chapter. For now just know that every one of the goals bound to the lifecycle phase are themselves packaged in plugins that were built with the maven-plugin lifecycle implementation. How's that for bootstrapping?

- **generate-resources** - plugin:descriptor
- **process-resources** - resources:resources
- **compile** - compiler:compile
- **process-test-resources** - resources:testResources
- **test-compile** - compiler:testCompile
- **test** - surefire:test
- **package** - jar:jar, plugin:addPluginArtifactMetadata
- **install** - install:install, plugin:updateRegistry
- **deploy** - deploy:deploy

ejb

EJBs, or Enterprise Java Beans, are a common data access mechanism for model-driven development in JavaEE, and Maven provides support for EJB 2 and 3. Though you must configure the ejb plugin to specifically package for EJB3, else the plugin defaults to 2.1 and demands certain EJB configuation files.

- **process-resources** - resources:resources
- **compile** - compiler:compile
- **process-test-resources** - resources:testResources
- **test-compile** - compiler:testCompile
- **test** - surefire:test
- **package** - ejb:ejb
- **install** - install:install
- **deploy** - deploy:deploy

war

The war packaging type is similar to the `jar` and `ejb` types (notice a pattern here?). The exception being the package goal of `war:war`. Note that the `war:war` plugin requires a `web.xml` configuration in your `src/main/webapp/WEB-INF/` directory. More on quirks like these later.

- **process-resources** - resources:resources
- **compile** - compiler:compile
- **process-test-resources** - resources:testResources
- **test-compile** - compiler:testCompile
- **test** - surefire:test
- **package** - war:war
- **install** - install:install
- **deploy** - deploy:deploy

ear

EARs are probably the simplest Java EE constructs, consisting primarily of the deployment descriptor (application.xml) file, some resources and some modules. The ear plugin offers a nice addition to the build process in the `ear:generate-application-xml` goal. It generates the `application.xml` based upon the configuration in the EAR project's POM, although this is not required since you may provide your own file if so desired, which the ear:ear goal will use when packaging.

- **generate-resources** - ear:generate-application-xml
- **process-resources** - resources:resources
- **package** - ear:ear
- **install** - install:install
- **deploy** - deploy:deploy

par

The par type may be a little esoteric for some people.

- **process-resources** - resources:resources
- **compile** - compiler:compile
- **process-test-resources** - resources:testResources
- **test-compile** - compiler:testCompile
- **test** - surefire:test
- **package** - par:par
- **install** - install:install

- **deploy** - deploy:deploy

Note that this list is not exhaustive of every packaging type available for Maven. There are more available through external projects, such as the NAR (native archive) packaging type. They require two things, however: access to their remote repository (by default the only repository available is Maven Central, check the SuperPOM), and the packaging plugin must be added as a build extension. The build extension is merely a mechanism to allow you to add a project's artifact into Maven's execution classpath... much simpler and more portable than manually adding "`-cp /path/to/project.jar`" to the mvn execution script.

As an example on how this is done, let us use the Codehaus Mojo project's `jboss-sar` packaging type - to create a deployable JMX Service Archive (SAR) project for JBoss.

```
<project>
  ...
  <package>jboss-sar</package>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jboss-packaging-maven-plugin</artifactId>
        <version>2.0-SNAPSHOT</version>
        <extensions>true</extensions>
      <plugin>
    </plugins>
  </build>
  ...
  <repositories>
    <repository>
      <id>Codehaus Snapshots</id>
      <url>http://snapshots.repository.codehaus.org/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>Codehaus Snapshots</id>
      <url>http://snapshots.repository.codehaus.org/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
  ...
</project>
```

We will cover presicely how to create your own packaging type later on in this chapter, but this example should give you an idea on what is required to use it.

This should give a basic understanding of the default build lifecycle, how to use it, extend it, and how to use the packaging type in the POM for altering the functions of the build phases.

Redundancies

You may notice that many of the packaging type lifecycles were similar in every respect except for the `packaging` phase's bound goal. This makes sense, since moving resource files, or compiling java to class files, are going to be the same reguardless of whether those files are packaged in a JAR, EAR or WAR. In order to get you comfortable operating in the Maven lifecycle requires us to spend some time pointing out some behaviors that, although are still dependant on the underlying goals, are so embedded into the Maven process its easy to forget and treat it as simply a function of that phase.

Resources

The first is the `process-resources` phase, implemented by the `resources:resources` goal. As mentioned above, the `process-resources` phase processes the resources to the destination directory. What does this mean to "process" resources? In the default case, this means simply to copy the files from the `src/main/resources` directory to `target/classes` (the `${project.build.outputDirectory}`) - but there is a more advanced behavior makes this seemingly benign phase interesting: filters. Filters are used by the resources plugin to replace the values in text files with properties in the standard `${...}` form. It has been said (seriously) that people come to Maven for the dependency management, but stay for the filtering.

For example, let us say that you have a project with an xml file `src/main/resources/META-INF/service.xml`. Rather than create one file per distribution, you create a skeleton file, replacing actual values with `\${...}` delimited properties.

```
<service>
  <!-- This URL was set by project version ${project.version} -->
  <url>${jdbc.url}</url>
  <user>${jdbc.username}</user>
  <password>${jdbc.password}</password>
</service>
```

There is also a file `src/main/filters/default.properties`. These types of "filter files" contains `name=value` pairs which will replace `\${name}` strings within filtered resources.

```
jdbc.url=jdbc:hsqldb:mem:mydb
jdbc.username=sa
jdbc.password=
```

In the project's POM we specify two items, a list of files that contain properties to filter by, and a flag to Maven that the resources directory is to be filtered (by default it is not, and will merely be copied to the target directory).

```
<build>
  <filters>
```

```
    <filter>src/main/filters/default.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

Note: The resources directory is not required to be under src/main/resources, this is simply the default that we adhere to for this example.

Filtering resources do not require using filter files since you can filter with any Maven properties (like we did with `\${project.version}`, a default POM property). What good is this? Did we not just move the configuration from the resource file to the POM? Yes, in this limited example case. Let's take a look at the KillerApp. In the `killerapp-cli` project we use a custom resources directory and filters to match the jar's final name.

```
<resources>
  <resource>
    <filtering>true</filtering>
    <directory>${basedir}/src/main/command</directory>
    <includes>
      <include>killerapp.bat</include>
    </includes>
    <targetPath>..</targetPath>
  </resource>
</resources>
```

The command-line bat file generated in turn, converts the `killerapp.bat` file from this:

```
@echo off
java -jar ${project.build.finalName}-jar-with-dependencies.jar %*
```

to this:

```
@echo off
java -jar killerapp-cli-1.0-SNAPSHOT-jar-with-dependencies.jar %*
```

Compile

The compile phase is straightforward enough. All you usually need to know in normal operations is that the compiler:compile goal assumes you are compiling JDK 1.3 compliant code. If you wish to use a different version, you must configure.

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
```

```
            <target>1.5</target>
          </configuration>
        </plugin>
      </plugins>
      ...
    </build>
    ...
  </project>
```

Notice we are configuring the *plugin* `maven-compiler-plugin`, and not the specific *goal* `compile:compile` (in other words, the `configuration` element is under the `plugin` element, and not under `execution` element as we have used it before). When you configure the plugin, the settings are propogated to all of a plugin's executed goals, therefore, the `compile:testCompile` goal bound to the `test-compile` phase will be set with a JDK 1.5 source and target as well.

Test Resources

`process-test-resources` is similar to `process-resources`. The glaring difference being that you define test resources in the POM under the element testResources with a set of children testResource, and they are copied to `target/test-classes` (the `${project.build.testOutputDirectory}`). All other things are equal.

Test

The `test` phase is bound by `surefire:test`. Surefire executes files ending with `*Test.java` by default, though other values may be set (see Appendix B for more information). JUnit tests are executed by default, although others like TestNG are available. You can add TestNG as a dependency to your test project, just like you would with JUnit (remember chapter 2?) but with the extra classifier element.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>4.7</version>
      <scope>test</scope>
      <classifier>jdk14</classifier>
    </dependency>
```

If you wish to use Java 5 annotations rather than JDK 1.4 javadoc annotations, change the classifier to `jdk15`.

Installation and deployment to local and remote repositories will be covered in a later chapter.

site

Up until now we have focused clearly on the concept of building and placing artifacts to the repositories. But Maven can do more than simply generate code, it is also well equiped to generate documentation and reports about the project, or a collection of projects (aggregation). This is such an important task of Maven that "site" generation has its own lifecycle. The site lifecycle contains four phases:

- **pre-site**
- **site** - generate the site and reports
- **post-site**
- **site-deploy** - deploys the site to a remote server

The default goals bound to the site lifecycle is:

- **site** - site:site
- **site-deploy** - site:deploy

Unlike the `default` build lifecycle - but similar to `clean` - the packaging type does not tend to alter this lifecycle since packaging types are concerned primarily with artifact creation, not with the type of site generated. This is actually a good thing as sites of all types will get a similar look-and-feel.

The site plugin is a powerful thing in Maven. It kicks off the execution of Doxia (*http://maven.apache.org/doxia/*) document generation and other report generation plugins.

```
mvn site
```

We will cover site and report generation in depth in Site Generation (*site-generation.html*) and the Reporting (*reporting.html*) chapters, respectively.

Plugins and the Lifecyle

The last item on the agenda is to mention that there is are two more ways for manipulating the build lifecycle. One is to bind a goal to a phase by default using the `@phase` annotation in the Mojo definition. The last is to create your own packaging type and write your own implementation of the phases. So far we have yet to cover plugins in depth, so it seems a good place to start. In the next chapter we will go over plugins in detail, but for now we will write a simple mojo and bind it to the phase.

Mojos are a play-on-words of the Java term POJO (Plain Old Java Object) - but for Maven. A Mojo is an object that implements a goal that can be called. An important aspect of a Mojo is that it is self-contained. Unlike Ant tasks, Mojos cannot contain other Mojos. This is why Maven prefers to call its actions "goals" rather than "tasks". It may require several tasks to complete a purpose you have in mind, but a goal is merely a declaration of a purpose. For example when executing the Maven goal jar:jar, I am declaring that I would like a jar created based upon the class files that are in the pre-

designated location (by default in the SuperPOM, that is target/classes). Now, if you wish to execute two goals, say, as `mvn compile:compile jar:jar` then they work together quite nicely. `compile:compile` generates `*.class` files from your `*.java` files and places them in the `target/classes` directory, then `jar:jar` packages them into a nice `*.jar` artifact. Chances are you will always want to execute both of those goals, in that order. This is why the build lifecycle exists. To bind those goals to this default order, so you, the user, don't have to worry about the details.

But back to mojos. Mojos are the code written that defines what a goal does. When you execute a goal, you are actually asking Maven to execute a mojo for you. The following is our simple mojo definition (if you have yet to download the sample project, get it from here (*examples/book-lifecycle.zip*) to follow along):

```
/**
 * Zips up the output directory.
 * @goal zip
 * @phase package
 */
public class ZipMojo extends AbstractMojo
{
    /**
     * The Zip archiver.
     * @component role="org.codehaus.plexus.archiver.Archiver" role-hint="zip"
     */
    private ZipArchiver zipArchiver;

    /**
     * Directory containing the build files.
     * @parameter expression="${project.build.directory}"
     */
    private File buildDirectory;

    public void execute()
        throws MojoExecutionException
    {
        try {
          zipArchiver.addDirectory( buildDirectory, new String[]{"**/**"}, new String[]{"**/output.zip"}
            zipArchiver.setDestFile( new File( buildDirectory, "ouput.zip" ) );
            zipArchiver.createArchive();
        } catch( Exception e ) {
            throw new MojoExecutionException( "Could not zip", e );
        }
    }
}
```

note: If you are familiar with Java 5 you may wonder, "why not just use the built in annotations?" This is because, by default, Maven executes on JDK 1.3. This allows the widest audience for your plugins. Maven 2.1 will add support for writing plugins for Java 5, but for now, javadoc annotations will have to do.

The above class is part of the `maven-zip-plugin` project which you can install with - you guessed it - `mvn install`. You may then zip up a project's build directory by executing your goal on the command line:

```
mvn com.training.plugins:maven-zip-plugin:zip
```

We will go into more details over the mojo structure in the next chapter. For now draw your attention to the `@phase` annotation in the class javadoc. This is the third method for binding a goal to a phase. Where in the `antrun:run` goal in the beginning of this chapter we manually bound the goal to the `pre-clean` phase, in the above example we may use the "`zip`" goal (name defined by the `@goal` annotation) without specifying the phase, we need only add the `zip:zip` goal to the POM.

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>com.training.plugins</groupId>
        <artifactId>maven-zip-plugin</artifactId>
        <executions>
          <execution>
            <!-- no phase element! -->
            <goals>
              <goal>zip</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

Now you may run `mvn package` and the zip goal will execute at that phase.

But what if the goal is not meant to be part of the standard lifecycle phase - for example as an occasional backup? You would want to run `mvn com.training.plugins:maven-zip-plugin:zip` manually, but the goal will fail without an existing output directory. If you remember in chapter 3 (*pom-relationships.html*) we mentioned that Maven parameters can be accessed via "dot" notation and `\${...}` delimited. So the buildDirectory field is populated with the contents of the `\${project.build.directory}` property - set by default (thanks to the SuperPOM) to the `target` directory. If no goal, such as `compile:compile` has yet to populate the `target` directory, the `zip` plugin has no content to zip up, so we therefore must first force the execution of some goal, phase or lifecycle before the zip goal.

- **@execute goal <goal>** This will execute the given <goal> before execution of this one.
- **@execute phase <phase>** This will fork an alternate build lifecycle up to the specified <phase> before continuing to execute the current one.
- **@execute lifecycle <lifecycle>** This will execute the given alternate lifecycle. A custom lifecycle is defined in a `META-INF/maven/lifecycle.xml` file.

Forking your own Lifecycle

The custom lifecycle must be packaged in the plugin under the `META-INF/maven/lifecycle.xml` file. The straight-forward way to accomplish this is used in the example under `src/main/resources/META-INF/maven/lifecycle.xml`. This configuration declares a lifecycle named "zipcycle" that contains only the zip goal in the package phase.

```
<lifecycles>
  <lifecycle>
    <id>zipcycle</id>
    <phases>
      <phase>
        <id>package</id>
        <executions>
          <execution>
            <goals>
              <goal>zip</goal>
            </goals>
          </execution>
        </executions>
      </phase>
    </phases>
  </lifecycle>
</lifecycles>
```

In the `zip-fork` goal's mojo, the `zipcycle` is the lifecycle executed up to the `package` phase (which is bound by `zip:zip`). The ZipForkMojo doesn't actually do anything itself.

```
/**
 * Forks a zip lifecycle.
 * @goal zip-fork
 * @execute lifecycle="zipcycle" phase="package"
 */
public class ZipForkMojo extends AbstractMojo
{
  public void execute()
    throws MojoExecutionException
  {
    getLog().info( "doing nothing here" );
  }
}
```

Now running the `zip-fork` goal will fork the lifecycle (try it in the `zip-lifecycle-test` under `maven-zip-plugin/src/projects` - you will find other integration tests there as well).

```
mvn zip:zip-fork
```

As you can see, running `zip:zip-fork` will first execute all phases up to the `zipcycle`'s `package` phase - which is only `zip:zip`.

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'zip'.
[INFO] ------------------------------------------------------------------------
```

```
[INFO] Building Maven Zip Forked Lifecycle Test
[INFO]    task-segment: [zip:zip-fork]
[INFO] ----------------------------------------------------------------------------
[INFO] Preparing zip:zip-fork
[INFO] [site:attach-descriptor]
[INFO] [zip:zip]
[INFO] Building zip: ~/maven-zip-plugin/src/projects/zip-lifecycle-test/target/output.zip
[INFO] [zip:zip-fork]
[INFO] doing nothing here
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sun Apr 29 16:10:06 CDT 2007
[INFO] Final Memory: 3M/7M
[INFO] ------------------------------------------------------------------------
```

Overriding the default Lifecycle

Finally, you can creating your own packaging type with its own lifecycle mapping. You do this by overriding Maven's component mechanism implemented by another project called Plexus. Without getting into the details of Plexus, you achieve this by creating a file `META-INF/plexus/components.xml`, and set the name of the packaging type under `role-hint`, and the set of phases containing the versionless coordinates of the goals to bind. Note that not every phase must be bound, and that multipled goals may be executed per phase, defined as a comma-seperated list.

```
<component-set>
  <components>
    <component>
      <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
      <role-hint>zip</role-hint>
     <implementation>org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping</implementation>
      <configuration>
        <phases>
        <process-resources>org.apache.maven.plugins:maven-resources-plugin:resources</process-resources>
          <compile>org.apache.maven.plugins:maven-compiler-plugin:compile</compile>
           <package>com.training.plugins:maven-zip-plugin:zip</package>
        </phases>
      </configuration>
    </component>
  </components>
</component-set>
```

Since the packaging type you have created is not, by default, accessible to Maven's build system, you need to add the plugin that contains the packaging definition with the `extensions` element set to `true`. This lets Maven know that this plugin is not simply executed, but that it is added to Maven's execution classpath.

```
<project>
  ...
  <build>
    ...
```

```
      <plugins>
        <plugin>
          <groupId>com.training.plugins</groupId>
          <artifactId>maven-zip-plugin</artifactId>
          <extensions>true</extensions>
        </plugin>
      </plugins>
    </build>
  </project>
```

Finally, you may add the packaging type to the POM:

```
  <packaging>zip</packaging>
```

If you run `mvn package` on the `maven-zip-plugin-test` project you will see:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Zip Plugin Test
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] No sources to compile
[INFO] [zip:zip]
[INFO] Building zip: ~/maven-zip-plugin/src/projects/zip-packaging-test/target/output.zip
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Thu Nov 09 17:41:21 CST 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

You have just created your own Maven plugin goal as a mojo, manipulated the build lifecycle, and created your own packaging type. We have begun to dig deeper into more advanced levels of Maven usage, a trend that we will continue in the next chapter (*using-plugins.html*) with advanced plugin usage and a few tips and tricks.

If the details of manipulating the build lifecycle has piqued your interest in writing custom plugins, please, read at least the next chapter (*using-plugins.html*) before jumping over to the details of writing them (*writing-plugins.html*).

Tips and Tricks

Clean to the Rescue

It is an unfortunate case that - despite all best and valiant efforts for encapsulating actions - something will go wrong when creating builds mixing non-standard goal and phase executions, or testing configurations. For example, if you run the `dependency:copy` goal into a directory, then configure `war:explode`, to exclude such jars,

it's possible they will end up in your build because they are already there. This is why the `clean` phase exists - and until you become more proficient at using Maven, I would recommend to prefix all executions with `clean`. Rather than run:

```
mvn install
```

Try running:

```
mvn clean install
```

This will remove all output generated by previous executions, giving you a nice clean feeling.

Summary

In this chapter we looked at what the lifecycle is, its structure, and some of its defaults based upon a project's packaging type. We also investigated 5 ways to manipulate build lifecycle:

1. Change project package to use the lifecycle of a default package definition, `jar` or `ear` for example.
2. Manually bind a goal to a phase in the POM.
3. Bind in the goal definition in the mojo, so adding the goal to the POM will default to a phase.
4. Create a forked lifecycle and execute it in a Mojo.
5. Create your own packaging type with a custom default lifecycle.

In the next chapter we will build upon what was covered here by drilling down even further in the action stack, to manipulating the plugins themselves.

CHAPTER 5
Plugins

Never confuse motion with action. -- Benjamin Franklin

Introduction

Plugins are the core of the Maven framework. At its heart, Maven is a plugin execution framework, with goals as the unit of execution. A plugin contains a set of goals with some theme, such as "ear" (with goals that package an ear, and generate-application-xml), or "resources" (with resources, testResources). Plugins may also manipulate the build lifecycle if they introduce a new packaging type, as shown in the previous chapter, but hijacking the lifecycle only scratches the surface of what plugins are capable.

A Quick Look at Plugin Name Resolution

Before we continue let us quickly cover a nagging point you may or may not have noticed. Up until we have simplified our treatment of plugin names for ease of discussion - but its grown-up time now and we need to raise our consciousness a little - to paraphrase Richard Dawkins. When we speak of the clean:clean goal, or the ear plugin, those are truncated versions of the actual plugin coordinates, and the plugin names. For example the clean:clean goal is actually named "org.apache.maven.plugins:maven-clean-plugin:clean", or you may even execute a specific verion of the goal by adding it after the plugin name, such as "org.apache.maven.plugins:maven-clean-plugin:2.1.1:clean".

If you execute "mvn org.apache.maven.plugins:maven-clean-plugin:clean" on the command line you will get the same results as "mvn clean:clean". This is due to two reasons. Reason one is: when no groupId is specified for a plugin Maven defaults to "org.apache.maven.plugins". In the POM plugin definition block, notice the lack of a groupId.

```
<project>
  ...
  <build>
    ...
```

```
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins>
```

This also holds true on the command-line. By default `org.apache.maven.plugins` and `org.codehaus.mojo` are on a list of goals that need not be prefixed in the command-line. You can add more to that list - for example `com.training.plugins` - to your settings.xml `pluginGroups` element.

```
<settings>
  ...
  <pluginGroups>
    <pluginGroup>com.training.plugins</pluginGroup>
  </pluginGroups>
</settings>
```

You may now execute `mvn maven-zip-plugin:zip`

Note: *The above information is correct as of Maven version 2.0.7, earlier versions would not honor plugin names that supercede found plugins in the two default plugin groups*

from website: [The second reason the command-line is shortened is because <<<maven-${name}-plugin>>> and `${name}-maven-plugin` artifactId patterns are treated in a special way. If no plugin matches the artifactId specified on the command line, Maven will try expanding the artifactId to these patterns in that order. So in the case of our example, you can use `mvn sample.plugin:hello:sayhi` to run your plugin.]> Because of this the zip plugin execution can be even shorter: `mvn zip:zip`. All core Maven plugins and other official Maven project follow one of the two conventions.

Configuring Plugins

Using plugins in Maven seems trivial - you add them to the build element's plugins list and configure them. What can make this simple process seem complex is that each plugin has its own set of configurations (sometimes nested configurations). Before you get scared off, however, remember a core Maven principle is convention over configuration. This leads Maven's mojos to contain acceptable defaults. But there will be times where a goal actually requires a configuration to execute properly, for example the antrun plugin. Without an element configuring ant tasks or at least a file to execute, the run goal does nothing.

Plugins are configured under the "build" element. We know that plugin goals have default configurations, and that certain goals are bound by default bound certain phases of the build lifecycle, dependent upon the type of project being built. However, you

will occasionally want to make certain changes to this default, some we have mentioned, and some not:

- Configure a plugin or specific goal that is already part of the lifecycle.
- Configure a plugin or specific goal not in the lifecycle. - only when goal does not have a default phase, else this is similar to the next one
- Configure and add a specific goal to be part of the lifecycle.
- Alter the plugins dependency list
- Add this plugin's extensions to the build classloader (for example, in the previous chapter when we added the zip packaging type)

The first three items in the list all use the `configuration` element of the plugin, but to different ends. Configuring a plugin or goal that already exists in the lifecycle is the simplest. In the parent killerapp project you will notice the maven-compiler-plugin is configured to compile at JDK 1.4 (which is actually the default JDK).

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.4</source>
          <target>1.4</target>
        </configuration>
      </plugin>
```

You may wonder why the plugin is configured, rather than the goal. Do goals not do all of the actual work? Yes, they do. This is merely a shortcut for configuring all goals in the plugin with the same configuration. If we were to configure all of the compiler goals, the configuration would instead look like this:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
            <configuration>
              <source>1.4</source>
              <target>1.4</target>
            </configuration>
          </execution>
        </executions>
      </plugin>
```

Notice that executions contain a set of goals? This is because, as shown above, a configuration may be shared between multiple goals, or because several goals may be bound to a single phase. Execution ids need not be unique, but executions with matching ids are merged - executing once on multiple phases if need be.

The second type of configuration is a plugin or specific goal not in the lifecycle. This method of configuration is the same as the other two, but the use-case is different. A good example is the antrun plugin, since the run goal does not default to a specific lifecycle. When you configure the antrun plugin

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <configuration>
          <tasks>
            <echo>The JAVA_HOME var is ${env.JAVA_HOME}</echo>
          </tasks>
        </configuration>
      </plugin>
```

It will not be executed by running any phase. To run, you must execute the goal `mvn antrun:run`

If it is bound to phase, for example verify, then it becomes the third type of plugin configuration.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <configuration>
          <tasks>
            <echo>The JAVA_HOME var is ${env.JAVA_HOME}</echo>
          </tasks>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>run</goal>
            </goals>
            <phase>verify</phase>
          </execution>
        </executions>
      </plugin>
```

The fourth type of plugin configuration is useful when the plugin itself may wish to use a dependency that is not required by default. Sticking with the maven-antrun-plugin in the `killerapp-model` project:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <id>file-exists</id>
            <phase>pre-clean</phase>
            <goals>
                              <goal>run</goal>
                    </goals>
            <configuration>
              <tasks>
                <!-- adds the ant-contrib tasks (if/then/else used below) -->
                <taskdef resource="net/sf/antcontrib/antcontrib.properties"/>

                <available
                 file="${project.build.directory}/${project.build.finalName}.jar"
                   property="file.exists" value="true" />
                       <if>
                             <not><isset property="file.exists" /></not>
                    <then><echo>No ${project.build.finalName}.${project.packaging} to delete</echo></th
                    <else><echo>Deleting ${project.build.finalName}.${project.packaging}</echo></else>
                       </if>
              </tasks>
            </configuration>
          </execution>
        </executions>
                <dependencies>
          <dependency>
            <groupId>ant-contrib</groupId>
            <artifactId>ant-contrib</artifactId>
            <version>1.0b2</version>
          </dependency>
                </dependencies>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

The last reason for plugin configuration was first explored in Chapter 4 when we created the `zip` packaging type. To understand precisely what it means to add a plugin `extension` requires us to understand the basics of Inversion of Control and how Maven uses Plexus to manage its dependency injection defined in the configuration.xml files. We will explore what this means later in this chapter, but for now suffice it to say that this element hints to Maven that it should add this plugin to its running classloader.

Gotcha: *When executing a plugin from the command line, you must put the configuration tag outside of the <<<executions>>>* element, else you may get an error similar to:>

```
One or more required plugin parameters are invalid/missing for 'plugin:goal'
```

This make sense when you consider that the `plugin` *element is the domain of direct interaction (such as, the command-line), while the* `execution` *is for a very specifically defined execution.*

The examples above show the "why" of the configuration syntax. The following will show how to perform common configuration for commonly used plugins and goals.

Default Lifecycle Plugins

maven-resources-plugin - process-resource, process-test-resource

We have mentioned earlier that resources are non-source-code files used by your project. We have also relayed the notion of filtering resource text files. But text files come in several flavors, or encodings. Maven's resources plugin supports the types supported by Java, defaulting to the installed JRE's default, but changable to: US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, and UTF-16.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <configuration>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

The management of resources is so integral to Maven that they are largely configured by more POM elements than just the configuration block. The resources and testResources have their own elements - which we have encountered before; resource and testResources, respectively. Beyond the simple configuration of adding new resource directories, the resources element can also set a specific output directory with the targetPath, and include/exclude specific files for processing by the resources goals.

```
<project>
  ...
  <build>
    <resources>
      <resource>
        <directory>src/my-properties</directory>
        <targetPath>META-INF</targetPath>
        <filtering>true</filtering>
        <excludes>
          <exclude>**/*test*.properties</exclude>
        </excludes>
      </resource>
    </resources>
```

```
    <testResources>
      <testResource>
        <directory>src/my-test-properties</directory>
        <targetPath>META-INF</targetPath>
        <filtering>true</filtering>
        <includes>
          <include>**/*.properties</include>
        </includes>
      </testResource>
    </testResources>
  </build>
```

maven-compiler-plugin - compile, test-compile

We have already seen how to configure the compiler for a specific version of JDK. Note that you must be running Maven in at least that version. You cannot compile JDK 1.5 code if you are running Maven off of 1.4. However, you can configure the compiler plugin to use a javac other than the version Maven is currently running on (the default).

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <executable>/usr/bin/javac</executable>
          <compilerVersion>1.5</compilerVersion>
          <fork>true</fork>
        </configuration>
      </plugin>
    </plugins>
```

`executable` is the location of the javac version you wish to use. You must remember to always set fork to true to define your own compiler, otherwise it will not fork compile into a seperate process, thus using the default compiler inline.

The other commonly configured compiler property are arguments to the compiler, such as classpath. This setting only has relevence if the compiler is forked.

```
        <configuration>
          <compilerArguments>
            <classpath>${java.home}/lib/tools.jar</classpath>
          </compilerArguments>
        </configuration>
```

maven-surefire-plugin - test

Maven created the build lifecycle for a purpose... to give a well-defined sequence to every build. If you want to compile the source, you first need to verify that the project is kosher and generate and process sources and resources first.

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>
```

Sometimes it is necessary to turn off testing - almost every case is for development purposes. You can skip tests by setting the `skip` element to true, or by setting the `maven.test.skip` property. This property is useful through the command line via -Dmaven.test.skip=true. You should not normally turn off testing in a POM, so the command-line is the prefered method for development use.

`surefire:test` also has the ability to filter certain files for testing with includes and excludes elements, to add test scope system properties,

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
    <include>**/*Test.java</include> <!-- Only run tests ending with Test.java -->
    </includes>
    <systemProperties>
      <property>
        <name>sys.path</name>
        <value>${env.PATH}</value>
      </property>
    </systemProperties>
    <forkMode>pertest</forkMode>
    <argLine>-enableassertions</argLine>
  </configuration>
</plugin>
```

Note that in order to use Surefire you must use one of its supported frameworks. By default this is junit, so in order to use it you must add it as a dependency:

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

If you wish to use TestNG instead, add that as a dependency. The classifier must match the jdk version you wish to use, jdk14 or jdk15 (for Java 5 annotations).

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>4.7</version>
  <scope>test</scope>
  <classifier>jdk15</classifier>
</dependency>
```

maven-install-plugin - install

Installs files to the local repository. There is not much to configure in the install goal, but in the interest of completeness we will mention them here.

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <createChecksum>true</createChecksum>
    <updateReleaseInfo>true</updateReleaseInfo>
  </configuration>
</plugin>
```

Set the createChecksum to true to generate some cryptographic hash checksum used to verify the integrity of all associated files (the pom file and artifacts). The updateReleaseInfo element is true if you want to update the local repository metadata file to set this as release. Both elements are false by default, and used mostly for development purposes.

These elements are almost never set permanently within the POM, but rather passed through to the command-line on occasion, by way of the -D flag (-DcreateChecksum=true).

maven-deploy-plugin - deploy

Deploy files using wagon, etc. Cover this in the repository section.

Packaging Types

maven-jar-plugin - jar, test-jar

The jar packaging type is the default for the POM, and is undoubtedly the most common type of Java project artifact. Simply running `mvn package` for a simple project will run this goal.

```
<project>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp-applet</artifactId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <classifier>unsigned</classifier>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

The `classifier` element is used to add an extra dimension to the jar coordinate. In the example above we create a jar with the classifier `unsigned`. This will simultaniously generate an artifact of the name `${project.build.finalName}-${classifier}.jar`, as well as set the artifact's coordinate to be `groupId:artifactId:packaging:classifier:version`. This is useful when we have multiple similar artifacts available, generated from the same project but in slightly different ways. Perhaps we generate two jars for a project, one signed and one unsigned. When they are installed in the local repository, we may then use them by adding a classifier element to the dependency definition:

```
<dependency>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp-applet</artifactId>
  <version>1.0-SNAPSHOT</version>
  <classifier>signed</classifier>
</dependency>
```

The jar plugin goals generates a manifest file (`META-INF/MANIFEST.MF`) by default. There are several archive configuration elements for setting the desired values in the manifest file, such as the following example to give the JAR and entry point (making the jar executable).

```
<configuration>
  <archive>
    <manifest>
      <mainClass>fully.qualified.MainClass</mainClass>
      <addClasspath>true</addClasspath>
    </manifest>
  </archive>
</configuration>
```

Or use an existing one from the given location in your project structure:

```
<configuration>
  <archive>
    <manifestFile>src/meta/MANIFEST.MF</manifestFile>
  </archive>
</configuration>
```

As with all of the plugins we are going over, a more comprehensive list is in the Appendix B.

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <executions>
     <execution>
        <goals>
           <goal>sign</goal>
        </goals>
     </execution>
  </executions>
  <configuration>
     <keystore>/path/to/keystore</keystore>
     <alias>youralias</alias>
```

```
      <storepass>yourstorepassword</storepass>
      <signedjar>${project.build.finalName}-signed.jar</signedjar>
      <verify>true</verify>
    </configuration>
  </plugin>
```

If signedjar is not specified, then the plugin will generate the signedjar as your sole artifact. Setting "verify" to true automatically verifies the signed jar after generation. If this is set to false, you may still verify the jar at a later time via the `jar:sign-verify` goal.

maven-plugin-plugin - maven-plugin

Besides being just plain fun to say, this plugin defines the packaging type maven-plugin, and contains goals used for generating documents required for creating Maven plugins. Although it contains many goals, and it performs a fair amount of work, this plugin is not very configurable. About the only configuration you would wish to do it to specify the goal prefix.

```
<plugin>
  <artifactId>maven-plugin-plugin</artifactId>
  <configuration>
    <goalPrefix>plugin</goalPrefix>
  </configuration>
</plugin>
```

This specifies the string preceeding the goal name, before the colon (':') delimiter.

maven-ejb-plugin - ejb

Enterprise Java Beans (EJB) are commonly used Java EE data access framework which may split a single project into two parts, a client and server jar.

The most important (and required) configuration element is ejbVersion, which defaults to EJB version 2.1. If the version is less than 3.x (where x is any digit) the ejb-jar.xml file is required for packaging. The valid versions are 2.x and 3.x.

```
<plugin>
  <artifactId>maven-ejb-plugin</artifactId>
  <configuration>
    <ejbVersion>3.0</ejbVersion>
  </configuration>
</plugin>
```

Beyond that, the EJB configuration is fairly light. If you are running versions less than 3.0 you can generate a seperate client jar, but the default is false.

```
<plugin>
  <artifactId>maven-ejb-plugin</artifactId>
  <configuration>
    <generateClient>true</generateClient>
    <clientIncludes>
      <clientInclude>**/**</clientInclude>
    </clientIncludes>
```

```
  </configuration>
</plugin>
```

If you choose to generate a client, then you can set the clientInclude and clientExcludes elements to filter those specific files you wish included into the generated client EJB JAR. The defaults are:

- clientIncludes: **/**
- clientExcludes: **/*Bean.class, **/*CMP.class, **/*Session.class, **/package.html

Beyond the client generation information, the `archive` configuration element is available and the same as the `archive` element in the `maven-jar-plugin` goals.

maven-war-plugin - war

The most common Java web artifact is the Web Archive, or WAR.

The war packaging type looks to src/main/webapp for the web application files such as JSPs, but still looks for non-code files in the standard src/main/resources directory. Where you place files such as *.properties files is of personal choice, however, if you think you may filter the resources it is suggested that you place them in resources, not webapp.

Like the jar and ear plugins goals, the archive configuration element is available via the same MavenArchiveConfiguration settings. One important item of note is that many web containers use the classpath defined in the manifest in classloading. By default this is false, but you can set the value with the following:

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

The manifest classpath is constructed by the war project's compile and runtime scope dependencies and transitive dependencies which will appear in WEB-INF/lib.

```
  <configuration>
    <dependentWarIncludes>**/images</dependentWarIncludes>
    <dependentWarExcludes>WEB-INF/web.xml,index.*</dependentWarExcludes>
    <workDirectory>target/war/work</workDirectory>
  </configuration>
```

...Creating Skinny Wars, from the maven-war-plugin doc...

```
<plugin>
   <artifactId>maven-war-plugin</artifactId>
   <configuration>
     <warSourceExcludes>WEB-INF/lib/*.jar</warSourceExcludes>
```

```
      <archive>
        <manifest>
          <addClasspath>true</addClasspath>
          <classpathPrefix>lib/</classpathPrefix>
        </manifest>
      </archive>
    </configuration>
  </plugin>
```

Of course you will have to include these jars in the containing EAR, which leads us to the next plugin.

maven-ear-plugin - ear

The Enterprise Archive (EAR) is the last common packaging type configuration that we will cover. EARs are packages of other packaging types along with configuration files. The application.xml file is required, and can either be provided or generated via configuration of the generate-application-xml goal. For JBoss, the jboss-app.xml can also optionally be generated.

When generating application.xml, the most important piece is inclusion of modules.

```
<plugin>
  <artifactId>maven-ear-plugin</artifactId>
  <configuration>
    <modules>
      <jarModule>
        <groupId>com.training.killerapp</groupId>
        <artifactId>killerapp-ejb</artifactId>
      </jarModule>
      <warModule>
        <groupId>com.training.killerapp</groupId>
        <artifactId>killerapp-war</artifactId>
        <contextRoot>/killerapp</contextRoot>
      </warModule>
      <artifactTypeMappings>
        <artifactTypeMapping type="zip" mapping="zip" />
      </artifactTypeMappings>
    </modules>
  </configuration>
</plugin>
```

Above we show the jarModule (generated as javaModule in the descriptor file) and warModule with its context root (the access-point to the WAR model).

- ejbClientModule
- ejbModule
- jarModule
- parModule
- rarModule
- sarModule

- webModule
- wsrModule
- harModule

Notice the artifactTypeMappings element, allowing the addition of new mapping types not defined by default.

...Using Skinny Wars, from the maven-war-plugin doc...

```
<plugin>
  <artifactId>maven-ear-plugin</artifactId>
  <configuration>
    <defaultJavaBundleDir>lib</defaultJavaBundleDir>
  </configuration>
</plugin>
```

From the maven-ear-plugin doc...

JBoss Support.
The EAR plugin can generate the jboss-app.xml automatically. To do so, the 'jboss' element must be configured and takes the following child elements:

- version: the targeted JBoss version to use (3.2 or 4 which is the default).
- security-domain: the JNDI name of the security manager (JBoss 4 only)
- unauthenticated-principal: the unauthenticated principal (JBoss 4 only)
- jmx-name: the object name of the ear mbean.

Hibernate archives (HAR) and Service archives (SAR) will be recognized automatically and added the the jboss-app.xml file.

You can take a look at the examples for more information on the JBoss support.

```
<plugin>
  <configuration>
    <jboss>
      <version>4</version>
      <unauthenticated-principal>guest</unauthenticated-principal>
    </jboss>
  </configuration>
</plugin>
```

Non-Lifecycle Tools (Stand-Alone Goals)

The following goals are not meant to be attached to any phase, so it goes without saying that they are not bound by default.

maven-install-plugin:install-file

This is a command-line only goal to manually install a project to the local repository. You may wish to install a jar manually for several reasons, but the most common is for

licensing issues, which make the artifact unable to be deployed to a remote repository. Or simply because the artifact is not a "Mavenized" project.

```
mvn install:install-file  -Dfile=/path/to/commercial.jar \
                          -DgroupId=com.training.killerapp \
                          -DartifactId=killerapp-commercial \
                          -Dversion=1.0 \
                          -Dpackaging=jar \
                          -DgeneratePom=true
```

Rather than have the pom file be generated by the goal, you may instead set the path to a custom pom file, which sets the coordinate elements.

```
mvn install:install-file  -Dfile=/path/to/commercial.jar \
                          -DpomFile=/path/to/pom.xml
```

maven-deploy-plugin:deploy-file

The install-file goal has a corrosponding goal in deploy-file for deployment. The difference being that a URL must be given so the goal knows where and how to deploy to the remote repository.

```
mvn deploy:deploy-file -Durl=file:///path/to/repo \
                          -DrepositoryId=my-local-repo \
                          -Dfile=/path/to/commercial.jar \
                          -DgroupId=com.training.killerapp \
                          -DartifactId=killerapp-commercial \
                          -Dversion=1.0 \
                          -Dpackaging=jar \
                          -DgeneratePom=true
```

There is actually much more involved with the creation and deployment of a remote repository, and will be covered in the next chapter.

maven-ant-plugin:ant

Ant does not need to be seen as a competing tool to Maven. In fact, Maven can handle Ant build scripts quite well through the maven-antrun-plugin. The converse of this plugin is the maven-ant-plugin that actually generates an Ant maven-build.xml file which may be imported into any build.xml file. This goal is a good example of a goal that is not configurable (in any interesting way). This goal places the generated file in the `${basedir}` directory.

maven-assembly-plugin:assembly

When no packaging type exists that creates the type of zipped artifact bundle you desire, the assembly plugin is a decent solution. Although it is almost always preferable to use an existing plugin, sometimes it is not feasable or desired to create an entire plugin simply to create a zip file (which is why there is no zip packaging type by default - if you were wondering).

The `assembly:assembly` goal, unlike the assembly:attached goal shown below, will fork its own lifecycle, so it is necessary to run `assembly:assembly`, and never bind the goal to a phase.

Note: *There is plenty of information on the assembly plugin in Chapter 10: Assemblies (assemblies.html).*

Lifecycle-Bound Tools

maven-jar-plugin:test-jar

There are occasions where you may wish to re-use test cases in multiple places. This is the purpose of the test-jar goal which packages the test artifacts into a test-jar type. This goal is meant to be bound to be bound to the package phase, which it does by default.

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Like any other type any project may use the project's test-jar artifact by adding it to its dependency list.

```
<dependency>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp-api</artifactId>
  <version>1.0-SNAPSHOT</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
```

Isn't Maven swell?

maven-source-plugin:jar, test-jar

The sources plugin packages the sourcecode (as well as generated sourcecode from the generate-sources phase, or generate-test-sources) into a jar suffixed with the -sources flag, or -test-source by default for the goals jar and test-jar, respectively. They default to being bound to the package phase. By default the source artifacts attach themselves to the project, but that can be surpressed via the attach flag (though the use-cases for such behavior are rare).

```
<plugin>
  <artifactId>maven-source-plugin</artifactId>
  <executions>
```

```
      <execution>
        <id>source-jar</id>
        <goals>
          <goal>jar</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <finalName>${project.build.finalName}-my-sources.jar</finalName>
      <attach>false</attach>
    </configuration>
  </plugin>
```

maven-antrun-plugin:run

The maven-antrun-plugin is diametric to maven-ant-plugin. Where the ant:ant goal creates a build file from an existing project, the antrun:run goal executes either tasks inline the POM configuration, or external to a build.xml file.

maven-assembly-plugin:attached

The definition of this goal is the same as the assembly:assembly goal shown above, with an important cavet. Where assembly spawns a seperate build lifecycle when executed, the attached goal is meant to be attached to a phase manually.

```
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
      <execution>
        <goals>
          <goal>attached</goal>
        </goals>
        <phase>package</phase>
      </execution>
      <configuration>
      </configuration>
    </executions>
  </plugin>
```

If you were to attempt to bind the assembly:assembly goal to a lifecycle phase, the results could be surprising. Once Maven reached the specified phase bound to assembly, the assembly goal would then fork a new build lifecycle, effectively executing all of the goals twice. This is rarely what is desired.

Look to configurations. Since plugins are released on their own schedules and not bound to the Maven core, changes can and do happen often. Although the Appendix B of this book is a good reference for goal configuration values, the Maven website plugin documentation should be considered the definitive documentation on the subject.

Summary

Plugins are the core of Maven's ability to perform actions. Plugins are, at their heart, just like any other Maven artifacts complete with dependencies and parents. The big difference is that, with a packaging type of `maven-plugin` and some annotated Mojos - they are recognized within Maven as special artifacts that can be executed to perform whatever action the Mojo was written to perform. We call the action that a Mojo performs a "goal" - these goals form the backbone of all executions that Maven performs. The ability to manipulate and manage goals forms the backbone of Maven - turning it from a simple project mangement system, to a full-blown convention-over-configuration-based build system.

CHAPTER 6
Archetypes

Using

Archetypes are a simple and useful way to bootstrap new development across your organization, and urge your developers to follow a similar project pattern. Archetypes are a template of a Maven project used to generate skeleton layout for projects of any desired type in a consistent way.

The default archetype is called *quickstart*, and generates a simple project with some "Hello World" Java code and a unit test. Running the `archetype:create` goal as so:

```
mvn archetype:create -DgroupId=com.mycompany -DartifactId=my-proj
```

This will yield a project with the following project structure:

```
my-proj
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- AppTest.java
```

The archetype that generates this simple project is outlined by two mechanisms: the `META-INF/maven/archetype.xml` resource definition file, and the archetype resources under the `src/main/resources/archetype-resources` directory.

```
maven-quickstart-archetype
|-- pom.xml
`-- src
    `-- main
        `-- resources
            |-- META-INF
            |   `-- maven
```

```
|       `-- archetype.xml
`-- archetype-resources
    |-- pom.xml
    `-- src
        |-- main
        |   `-- java
        |       `-- App.java
        `-- test
            `-- java
                `-- AppTest.java
```

There are other archetypes available by default from Maven Central Repository. Check out the list at *http://repo1.maven.org/maven2/org/apache/maven/archetypes*. At the time of the writing of this book, the list is:

- maven-archetype-archetype
- maven-archetype-bundles
- maven-archetype-j2ee-simple
- maven-archetype-marmalade-mojo
- maven-archetype-mojo
- maven-archetype-plugin-site
- maven-archetype-plugin
- maven-archetype-portlet
- maven-archetype-profiles
- maven-archetype-quickstart
- maven-archetype-simple
- maven-archetype-site-simple
- maven-archetype-site
- maven-archetype-webapp

Creating Your Own Archetypes

The easiest way to start creating archetypes is with the `org.apache.maven.archetypes:maven-archetype-archetype`.

```
mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes \
                     -DarchetypeArtifactId=maven-archetype-archetype \
                     -DarchetypeVersion=1.0 \
                     -DgroupId=com.mycompany \
                     -DartifactId=my-archetype
```

It will generate a simple archetype that is built to generate a simple project - of the same vein as the `maven-quickstart-archetype` shown in the beginning of this article, under the directory of the artifactId defined.

By default an archetype cannot overwrite a project. A useful construct for converting your existing non-Maven projects to Maven is to create a simple archetype with a pom construct of your design. Let's create a simple archetype that will be run over non-Maven projects to give them a `pom.xml` file with a custom MANIFEST.MF file.

Let us begin by removing the extraneous files under the `src/main/resources/archetype-resources/src` directory, leaving us just with a `pom.xml` and create a file `src/main/resources/archetype-resources/src/main/resources/META-INF/MANIFEST.MF`. This will leave the following project structure:

```
my-archetype
|-- pom.xml
`-- src
    `-- main
        `-- resources
            |-- META-INF
            |   `-- maven
            |       `-- archetype.xml
            `-- archetype-resources
                |-- pom.xml
                `-- src
                                `-- main
                        `-- resources
                            `-- META-INF
                                `-- MANIFEST.MF
```

Alter the `src/main/resources/archetype-resources/pom.xml` to be a project which contains a base MANIFEST.MF file to be packaged into a jar with extra entries. Since most - if not every - non-Maven project places its source code in a directory other than Maven's default `src/main/java`, we are also setting the `sourceDirectory` build element to another directory, `src`. Set this directory to whatever your legacy project structure requires.

```
<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>${groupId}</groupId>
  <artifactId>${artifactId}</artifactId>
  <version>${version}</version>
  <name>Project - ${artifactId}</name>
  <url>http://mycompany.com</url>

  <build>
    <sourceDirectory>src</sourceDirectory>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <excludes>
          <exclude>**/MANIFEST.MF</exclude>
        </excludes>
      </resource>
    </resources>
    <plugins>
```

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
            <manifestEntries>
              <Built-By>${user.name}</Built-By>
              <Project-Name>${project.name}</Project-Name>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>

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

Fill the `src/main/resources/archetype-resources/src/main/resources/META-INF/MANIFEST.MF` file with whatever valid manifest values you wish. Mine contains:

```
Manifest-Version: 1.1
Created-By: Apache Maven 2
Company-Name: My Company
```

Now we need to set the `src/main/resources/META-INF/maven/archetype.xml` descritor to bundle up our MANIFEST.MF file as a resource, and to allow us to run our archetype over the top of an existing one via the `allowPartial` element. By default the `archetype:create` goal will not allow the creation of an archetype when a project with the same artifactId already exists in the current directory.

```
<archetype>
  <id>my-archetype</id>
  <allowPartial>true</allowPartial>
  <resources>
    <resource>src/main/resources/META-INF/MANIFEST.MF</resource>
  </resources>
</archetype>
```

Like any other Maven project, you can install it by running in the base directory:

```
mvn install
```

Which builds and installs the archetype to your local repository. To test our new archetype, run the following command, which will generate a new project with the pom.xml and MANIFEST.MF files. If you run the same command again, it will work - only because we set `allowPartial` to `true`.

```
mvn archetype:create -DarchetypeGroupId=com.mycompany \
                     -DarchetypeArtifactId=my-archetype \
                     -DarchetypeVersion=1.0-SNAPSHOT
                     -DgroupId=com.mycompany \
                     -DartifactId=my-project \
```

Viola! You can now outfit your legacy projects with a shiny-new Maven 2 complient version.

Summary

Archetypes are an excellent way to get started on a project quickly - as well as keep many developers in sync regarding project layout or required files such as licenses or READMEs.

CHAPTER 7

Profiles

What Are They For?

Profiles are the next item on the agenda. The main purpose behind profiles in Maven is portability between different build environments. Environment examples are: development, testing, performance, staging, production, clients, or any other type you can dream up. Profiles allow you to create a single project that can be ported to all environments, making minor tweaks to the final build along the way.

Types of Portability

There are different levels of portability, from narrow to wide, listed below.

Non-Portability

The lack of portability is exactly what all build tools are made to prevent - however, any tool can be configured to be non-portable. What makes Maven unique is the relative difficulty in making this happen (Ant is relatively easier to create non-portable builds - Make even moreso). A non-portable project is buildable only under a specific set of circumstances and criteria (e.g., your local machine). Unless you have no plans on porting your project to any other machinesever, it is best to avoid non-portability entirely. Non-portable is similar to environmental portability with a single definition; the project may be built in only one environment setting.

Environmental Portability

Maven created profiles for the environmental portability level, which largely concerns code and resource generation. As an example, consider a test environment that points to a separate database from that of production. When built on a test box, therefore, at least one resource file (or codehopefully not) must be manipulated. If a file must be altered to successfully build and run on a specific environment, then the project is at best environmentally portable. A project that contains a reference to a test database in a test environment, for example, and a production database in a production environ-

ment, is environmentally portable. When you move to a different environment, one that is not defined and has no profile created for it, the project will not work. Hence, it is only portable between defined environments.

In-House Portability

For most non-trivial software efforts, in-house portability is the best you can hope for. The majority of software projects fall under this listing, either for an open source development team or a closed-source production company. The center of this level of portability is your project's requirement that only a select few may access an internal (in-house) remote repository. In-house does not necessarily mean a specific company. A widely dispersed open-source project may require specific tools or connection access; this would be classified as in-house.

Another in-house portability example is a common database to which all in-house members may connect. If you attempt to build an in-house project from scratch outside of the in-house network (for example, outside of a corporate firewall), the build will fail. It may fail because certain required custom plugins are unavailable, or project dependencies cannot be found. Hence, your project is portable only in-house.

Nothing is wrong with being in-house portable, *per se*, so do not twist your project out of shape attempting to avoid it. If it makes sense to split a project into multiple interdependent parts, then do it. If it makes sense to use a dependency version that is not available in the Maven Central repository, then use it. You can always make your own public repository and reference dependencies in your projects POM via the "repository" element, giving your project the coveted "widely portable" status.

Wide Portability

In the Maven world, anyone may download a wide portability project's source, and then compile and install it (sans modification) to the POM or requirements beyond standard Maven. This is the highest level of portability; anything less requires extra work for those who wish to build your project. This level of portability is especially important for open source projects, which thrive on the ability for would-be contributors to easily download and install.

As you may imagine, being the highest level of portability makes it generally the most difficult to attain. It restricts your dependencies to those projects and tools that may be widely distributed according to their licenses (unlike many commercial software packages - which may not be made available before accepting a certain license). It also restricts dependencies to those pieces of software that may be distributed as Maven artifacts. For example, if you depend upon MySQL, your users will have to download and install it; this is not widely portable (only MySQL users can use your project without changing their systems). Using HSQLDB (HyperSonic Database), on the other hand, is available from Maven Central repository, and thus is widely portable.

Where Do Profiles Come In?

The goal is to make your project as widely portable as possible. The wider the portability the lighter the work on your builders. "Builders" constitute anyone (or any*thing*) attempting to build the project. A profile in Maven is an alternative set of configurations which set or override the default values (and sometimes each other) under certain circumstances. This simple gesture lends Maven the power to completely alter its build by a simple activation of a profile - and in this way adds a portability for a certain environment. In other words - profiles turn a non-portable project into an environmentally portable one.

POM Profiles

The Maven POM contains an element under `project` called `profiles` containing a project's alternate configurations. The elements below have the same definitions as the project-level elements - see the Appendix on the POM for details.

```
<project>
  ...
  <profiles>
    <profile>
      ...
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
  </profiles>
</project>
```

Though a subset of the higher-level POM elements, it can often compose the bulk of a POM.

Activation

The `activation` element is the only strictly `profile` element. Activations contain different types of selectors - if the chosen selector is true of the environment being run in then the profile becomes active. If the profile is active then all elements override the corrosponding project-level elements (profile modules override project modules).

```
<project>
  ...
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
```

```
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.5</value>
        </property>
        <file>
          <exists>file2.properties</exists>
          <missing>file1.properties</missing>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
</project>
```

Build

The Maven POM proper contains a specification for tweaking build parameters - the artifact's output name, configurations for plugins and modifications to the build lifecycle. The `profile`'s `build` element contains a subset of the POM's main `build` element.

```
<project>
  ...
  <profiles>
    <profile>
      <build>
        <defaultGoal>install</defaultGoal>
        <directory>${basedir}/target</directory>
        <finalName>${artifactId}-${version}</finalName>
        <filters>
          <filter>filters/filter1.properties</filter>
        </filters>
        <resources>
          <resource>
            <targetPath>META-INF/plexus</targetPath>
            <filtering>false</filtering>
            <directory>${basedir}/src/main/plexus</directory>
            <includes>
              <include>configuration.xml</include>
            </includes>
            <excludes>
              <exclude>**/*.properties</exclude>
            </excludes>
          </resource>
        </resources>
        <testResources>
          <testResource>...</testResource>
```

```
        </testResources>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.1</version>
            <extensions>false</extensions>
            <inherited>true</inherited>
            <configuration>
              <sourceRoot>${project.build.directory}/generated-sources</sourceRoot>
            </configuration>
            <dependencies>
              <dependency>
                <groupId>ant</groupId>
                <artifactId>ant-optional</artifactId>
                <version>1.5.2</version>
              </dependency>
            </dependencies>
            <executions>
              <execution>
                <id>echodir</id>
                <goals>
                  <goal>run</goal>
                </goals>
                <phase>verify</phase>
                <inherited>false</inherited>
                <configuration>
                  <tasks>
                    <echo>Build Dir: ${project.build.directory}</echo>
                  </tasks>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
        <pluginManagement>
          <plugins>...</plugins>
        </pluginManagement>
      </build>
    </profile>
  </profiles>
</project>
```

In other words, `profile`'s `build` contains the POM's standard `build` elements, sans `extensions` and the directory definition elements {sourceDirectory, scriptSourceDirectory, testSourceDirectory, outputDirectory, testOutputDirectory}.

External Profiles

Before wrapping up POM profiles, one last note. There may be occasions when you wish to externalize the profiles from the POM. For these scenarios, `profiles.xml` was created. Just place all `profiles` elements into the file in the `${basedir}` and run as normal. If you wish to see the source of the active profiles, run the `maven-help-plugin`.

```
mvn help:active-profiles
```

The output will be something like this:

```
Active Profiles for Project 'My Project':

The following profiles are active:

 - my-settings-profile (source: settings.xml)
 - my-external-profile (source: profiles.xml)
 - my-internal-profile (source: pom.xml)
```

Note two things:

1. Having an external `profiles.xml` profile does not preclude you from declaring profiles within the POM.
2. There is a third source for profiles - the system's or user's `settings.xml` file. We will cover that next.

Settings Profiles

Although somewhat similar in ideology (they both offer alternative values) project and settings profiles are different in form and function. Whereas project profiles concern themselves with overriding the values of a specific set of system criteria, settings profiles are concerned with the system as a whole - because of this, they are a subset of the project's profiles.

The `activation` element is the same as the project's `activation` above. One extra level of activation in settings is the `activeProfiles` list element. Any profile ID set will automatically be activated for all builds using that settings.xml.

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>dev</activeProfile>
  </activeProfiles>
</settings>
```

Note that these will activate settings profiles only, not project profiles of matching names. However, there are tricks you can do to activate environment-specific profiles easily.

The `id` and `activation` are just an identifier and profile selector. The only real system-wide use of profiles is to set the repositories to use and runtime properties - exemplified by `repositories` and `pluginRepositories`, and `properties`, respectively.

Tips and Tricks

Once you understand the elements and concept, there is not much to `profiles`. All of the interesting things are around the tricks you can do with them.

Environment

The core motivation for project profiles were to enact environment-specific behaviors for project - and the simplest way to mark a specific environment is by creating a common property for activation. The example below shows how to set an `environment.type` property for the `dev` environment:

```
<settings>
  <profiles>
    <profile>
      <activeByDefault>true</activeByDefault>
      <properties>
        <environment.type>dev</environment.type>
      </properties>
    </profile>
  </profiles>
</settings>
```

Then all with profiles with a matching POM activations will be triggered when built on that specific machine.

```
<project>
  ...
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <property>
          <name>environment.type</name>
          <value>dev</value>
        </property>
      </activation>

      <!-- Development Environment Values -->

    </profile>
  </profiles>
</project>
```

Moreover, you can activate a profile by its absence as well. For example:

```
<project>
  ...
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <property>
          <name>!environment.type</name>
        </property>
      </activation>
    </profile>
  </profiles>
</project>
```

Note the bang (!) - or *not* - character prefixing the property name. This profile is activated when no `${environment.type}` property is set.

Classifiers

Although profiles are excellent for altering builds for seperate environments, how can you differentiate between those builds? Classifiers are a powerful answer to this question. There are several ways to create an artifact with a classifier - the most common are via the `maven-assembly-plugin` or the `maven-jar-plugin`. There are two methods for setting a classifier in this way: configuring the plugin within the profile, or under the project plugin configuration specified by a property. which you choose depends heavily upon how much duplication is required for each profile. For example, the following will build a jar classified for Windows or Linux, depending on the building environment:

```
<project>
  ...
  <profiles>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>windows</family>
        </os>
      </activation>
      <build>
        <plugins>
          <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>
              <classifier>win</classifier>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
    <profile>
      <id>linux</id>
      <activation>
        <os>
          <family>unix</family>
        </os>
      </activation>
      <build>
        <plugins>
          <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>
              <classifier>linux</classifier>
            </configuration>
          </plugin>
        </plugins>
```

```
      </build>
    </profile>
  </profiles>
</project>
```

This works, but is a lot of redundancy for each additional environment. The easier way is to set a common property:

```
<project>
  ...
  <build>
        <plugins>
          <plugin>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                  <classifier>${envClassifier}</classifier>
                </configuration>
          </plugin>
        </plugins>
  </build>
  ...
  <profiles>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>windows</family>
        </os>
      </activation>
      <properties>
        <envClassifier>win</envClassifier>
      </properties>
    </profile>
    <profile>
      <id>linux</id>
      <activation>
        <os>
          <family>unix</family>
        </os>
      </activation>
      <properties>
        <envClassifier>linux</envClassifier>
      </properties>
    </profile>
  </profiles>
</project>
```

In either case - the JAR artifact will be named `${finalName}-${envClassifier}.jar` and usable like:

```
    <dependency>
      <groupId>com.mycompany</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0</version>
      <classifier>linux</classifier>
    </dependency>
```

Summary

Profiles are a useful way for increasing the portability of a Maven project across environments with different setups - activated by a defined set of criteria.

CHAPTER 8

Site Generation

No one wants advice, only collaboration. -- John Steinbeck

This chapter follows an example project. You may wish to download the site-generation examples (http://www.sonatype.com/book/examples/book-site-generation.zip) and follow along.

Introduction

Most software projects today are developed by more than one author. If a project is successful, it will have more than one user. These may seem like trivial observations, but they both point to a single, critical need for every software project, that enables its development and user communities to thrive: a project website.

Project-oriented websites often contain all sorts of material, from FAQs and new-user guides to design documents, code reports, issue tracking, and more. However, keeping this much information up-to-date and relevant to the community can be a never-ending job. Perhaps this is why the websites of so many open-source projects fall hopelessly out of date.

Fortunately, Maven provides a much easier approach for maintaining your project's web content. Using Maven, you can:

- generate project status reports and other code-related content
- maintain consistent website navigational elements, including menu items that unfold
- render consistent website content from a mixture of of source-document formats
- publish content in multiple formats, including XHTML and PDF
- create *portable* documentation bundles, for embedding in the project's binary release archive

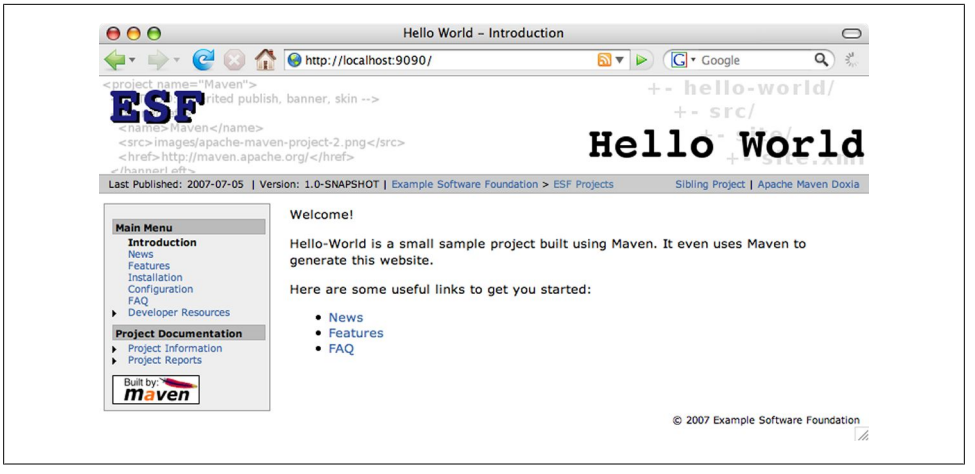


Figure 8-1. This is our basic hello-world website

Best of all, Maven can do all of this for you -- and then deploy the generated website -- with a single command. So, you've heard enough; where do we start? The same place we always start in the programming world: with a simple "Hello, World" example.

Hello World: Building A Simple Project Website

Before we can start building a project website, we need a project. For this, we turn to the trusty archetype plugin:

```
$ mvn archetype:create -DgroupId=book.sitegen -DartifactId=hello-world
```

This should give us a nice, basic project structure to work with, including a sample Java class and its accompanying JUnit test class. It also gives us a very bare-bones POM, which will need a lot of TLC if we're going to give our community a useful website.

For starters, let's see what sort of website Maven will generate from the plain-vanilla archetype result:

```
$ cd hello-world
$ mvn site:run
```

Once this build completes, fire up your browser and direct it to *http://localhost:8080*. You should see something similar to the following:

Publishing Project Documentation

The default project website is woefully inadequate as a source of documentation for the community. Users - and developers - look to your project's website for a wide range of information and resources:

- feature lists
- screenshots of the application in action
- project news and release changelogs
- downloads for released distributions
- installation instructions
- usage and configuration how-to's
- architectural diagrams
- frequently asked questions
- forums

Not to worry; creating such project documentation is a fairly straightforward process when you use Maven. It consists of three basic steps:

1. Define your website's menu by writing a *site descriptor*.
2. Write your project documentation, using one or more of the document formats supported by Maven.
3. Deploy the website.

For the purposes of the following examples, we'll define a relatively simple site structure. We'll start with the assumption that the main site content should be user documentation, with developer-oriented documentation relegated to a sub-section called `Developer Resources`.

Though it's possible to suppress the reports we generated in the examples above, we will leave them in at the default location within this website. In more advanced use cases, you may choose to suppress the project reports when you have a project that is dedicated solely to producing a website with user-oriented content; project reports then become nested in the generated, developer-oriented project website.

For now, let's keep things simple.

Writing a Basic Site Descriptor

First, let's get a rough outline going for the site. Start by pasting the following XML into a file called `site.xml` in the `src/site` directory of our hello-world project:

```
<project name="Hello World">
  <body>
    <menu ref="reports"/>
  </body>
</project>
```

At this point, if you regenerate the project website (you may have to `clean` first - always a good tip in maven when experimenting)

```
mvn clean site
```

and refresh your browser, you should see exactly the same thing as before, with the exception of the "Hello World" link in the navigation bar at the top-right of the page. From here, we can add a project banner that links back to the main page, along with a secondary logo:

```
<project name="Hello World">

  <bannerLeft>
    <name>Hello, World</name>
    <src>images/banner-left.png</src>
    <href>http://dev.example.com/sites/hello-world</href>
  </bannerLeft>

  <bannerRight>
    <src>images/banner-right.png</src>
  </bannerRight>
  ...
</project>
```

This time when you regenerate and refresh, you'll immediately notice that the page headers are all gone! This is because we haven't yet put the logo images in the site directory structure, so they aren't included in the generated website. To fix this, copy the banner-left.png and banner-right.png from the `examples/hello-world-resources` directory into the `src/site/resources/images directory` of the hello-world project, and regenerate once more. This time, the banner images should appear normally.

GOTCHA! Notice that the image `src` tags in the site.xml use the `images` directory, not the `resources/images` directory. This is because Maven uses the `resources` for website resources (including pre-existing HTML files) that don't need any processing. When the website is generated, Maven simply copies the contents of this directory, if it exists, directly into the generated website. At that time, what was in `resources/images` winds up in `images` in the new website.

Now that we've mastered the large-scale look-and-feel given by the `site.xml`, we need to create a custom navigational menu to separate user content from developer content. Typically, user documentation will contain things like features, screenshots, installation and configuration guides, and frequently asked questions. As an example of user content, then, let's create a few menu items reflecting this sort of content:

```
<project name="Hello World">
  ...
  <body>

    <menu name="Main Menu">
      <item name="Introduction" href="index.html"/>
      <item name="News" href="news.html"/>
      <item name="Features" href="features.html"/>
      <item name="Installation" href="installation.html"/>
      <item name="Configuration" href="configuration.html"/>
      <item name="FAQ" href="faq.html"/>
    </menu>
    ...
```

```
  </body>
</project>
```

Regenerate, and this time two things should pop out at you about the new `Main Menu` section. First, both the `Introduction` and `Project Information -> About` menu items are highlighted. This happens because the default index.html file for a Maven project website is the `About` report from the project-info reports plugin. Don't worry about this; we'll fix it soon.

The second thing you'll probably notice about the website is that none of the new menu items - with the exception of the aforementioned `Introduction` link - work! Remember, we haven't yet created any documentation beyond the default reports; providing content for these (and other) links comes next, after we complete the `Developer Resources` menu.

So, without further ado, let's create a menu for developer content. This sort of content usually includes architectural diagrams, integration guides, and similar items. Let's create a small developers' menu that reflects some of these elements:

```
<project name="Hello World">
  ...
  <body>
    ...
    <menu name="Main Menu">
      ...
     <item name="Developer Resources" href="/developer/index.html" collapse="true">
        <item name="System Architecture" href="/developer/architecture.html"/>
        <item name="Embedder's Guide" href="/developer/embedding.html"/>
      </item>
    </menu>
    ...
  </body>
</project>
```

TIP: The `collapse="true"` attribute in the `Developer Resources` menu item is a hint to Maven that this is a sub-menu which should be collapsed into a single menu item until it is clicked. Once clicked, its contents should be displayed in a sub-list of the menu navigation section.

Regenerate again, and you should see the new menu items (with broken links) under a new `Developer Resources` sub-menu. Now, we're ready to write the content that will fix these broken links. But first, let's take a brief look at some of the documentation features supported by Maven.

Writing Your Project Documentation

Project documentation often exists (or is created) in multiple formats, reflecting the different processes or tools used in the different stages of the project's development lifecycle. Blending these different formats together into a seamless, well-tailored website can be a daunting task, often involving exporting the documents to HTML using

some native tool or other, and massaging the HTML from these disparate exports into a common look and feel. This can be a very hands-on process, consuming hours everytime an update occurs in the source documents.

To address this diversity, Maven uses a documentation-processing engine called Doxia, which reads multiple source formats into a common document model, applies any embedded Doxia macros, then renders documents from that model into an output format, such as XHTML. Using Maven (with Doxia) allows you to delegate the process of creating or updating your project website to an automated process, which can then be run at any time using a simple task scheduler such as cron.

Documentation Formats

Doxia's support for multiple formats means you have the freedom to mix-and-match source formats in order to use the best format for each document in your site. Currently, Doxia has support for Almost Plain Text (APT), XDoc (a Maven 1.x documentation format), XHTML, and FML (useful for FAQ documents) formats. It also has experimental support for Twiki and Confluence wiki syntaxes. Beyond these - though the topic is beyond the scope of this chapter - Doxia is relatively simple to extend. Supporting new source-document formats is just a matter of parsing the source document and generating a series of Doxia events.

For more information about some of the different source formats supported by Doxia, see the following:

1. APT Reference: *http://maven.apache.org/doxia/format.html*
2. XDoc Reference: *http://jakarta.apache.org/site/jakarta-site2.html*

Directory Structure for Website Sources

In order to make a more clean separation of the different document-source formats, Maven enforces a fairly strict one-format-per-subdirectory rule on the `src/site` directory structure. The only exception to this rule are those source files which are meant to be passed directly through as-is; these reside in the `src/site/resources` directory (a good example of this is the PNG files used in the site banners for the hello-world project, discussed previously). Aside from this special directory, in general source documents should reside in a structure like this:

```
src/site/<lower-case-file-extension>
```

As an example, consider the following directory structure which contains a variety of APT, XHTML, PNG, XDoc, and FML files:

```
hello-world
+- src/
   +- site/
      +- apt/
      |  +- index.apt
      |  +- about.apt
```

```
|  |
|  +- developer/
|     +- embedding.apt
|
+- fml/
|  +- faq.fml
|
+- resources/
|  +- images/
|  |  +- banner-left.png
|  |  +- banner-right.png
|  |
|  +- architecture.html
|  +- jira-roadmap-export-2007-03-26.html
|
+- xdoc/
|  +- xml-example.xml
|
+- site.xml
```

GOTCHA! The XDoc files in the above example actually have a file extension of `.xml`. This represents a deviation from the aforementioned rule of directory structures; the main reason for this deviation is that a directory called `xml` is not descriptive enough to make it clear that the documents within are meant to adhere to the XDoc format.

One other thing to notice about the directory structure above is the `developer` directories under `apt` and `resources`. These, along with the `images` directory, will be mapped into the root of the resulting website. You can think of the format directories as being a root-directory fragment; when the website is generated, their contents are rendered to XHTML and merged into a single directory structure, with subdirectories intact. The only change is in the file extension: from `apt` (or `xml`, etc.) to `html`.

Macros

In addition to its advanced document rendering features, Doxia also provides a macro engine that allows each input format to trigger injection of dynamic content. An excellent example of this is the `snippet` macro, which allows a document to pull a code snipped out of a source file that's available via HTTP (using a web-enabled version control system, for instance). Using this macro, a small fragment of APT can be rendered into XHTML:

Usage of the snippet macro in APT syntax:

```
%{snippet|id=modello-model|url=http://svn.apache.org/repos/asf/maven/archetype/trunk/maven-archetype/ma
```

Output of the snippet macro in XHTML:

```
<div class="source"><pre>

<model>
  <id>archetype</id>
  <name>Archetype</name>
 <description><![CDATA[Maven's model for the archetype descriptor.]]></description>
```

```
  <defaults>
    <default>
      <key>package</key>
      <value>org.apache.maven.archetype.model</value>
    </default>
  </defaults>
  <classes>
    <class rootElement="true" xml.tagName="archetype">
      <name>ArchetypeModel</name>
      <description>Describes the assembly layout and packaging.</description>
      <version>1.0.0</version>
      <fields>
        <field>
          <name>id</name>
          <version>1.0.0</version>
          <required>true</required>
          <type>String</type>
        </field>
        ...
      </fields>
    </class>
  </classes>
</model>

</pre></div>
```

GOTCHA! Doxia macros **MUST NOT** be indented in APT source documents. Doing so will result in the APT parser skipping the macro altogether.

For more information about defining snippets in your code for reference by the snippet macro, see the *Guide to the Snippet Macro* on the Maven website, at *http://maven.apache.org/guides/mini/guide-snippet-macro.html*.

Currently, the only useful macro supported out-of-the-box by Doxia is the `snippet` macro. Again, though it's beyond the scope of this chapter, building your own Doxia macros is a relatively simple process. See the *Guide to Doxia Macros*, at *http://maven.apache.org/doxia/guide-doxia-macros.html* for more information.

Examples

Taking what we now know about site documentation and Maven, we can construct a small end-to-end example containing some of the content we hinted at previously in our site descriptor `src/site/apt/index.apt`, *Introduction*:

```
---
Introduction
---
Andrea Penn
---
26-Mar-2007
---

Welcome!
```

```
Hello-World is a small sample project built using Maven.
It even uses Maven to generate this website.

Here are some useful links to get you started:

 * {{{news.html}News}}

 * {{{features.html}Features}}

 * {{{faq.html}FAQ}}

 []
```

src/site/apt/developer/index.apt, *Developer Resources*:

```
---
Developer Resources
---
Andrea Penn
---
04-Apr-2007
---

This section is meant to guide hello-world developers and integrators,
by providing information on the development process, system architecture,
and APIs available.

Quick Links

* {{{architecture.html}System Architecture}}

* {{{embedding.html}Embedding Guide}}

[]
```

src/site/fml/faq.fml, *FAQ:*

```
<?xml version="1.0" encoding="UTF-8"?>
<faqs title="Frequently Asked Questions">
  <part id="General">
    <faq id="dead-doo-dad">
      <question>My doo-dad is dead. How can I regenerate it?</question>
      <answer>
        <p>
          Doo-dad generation happens at system boot. Therefore, the simplest
          answer is to restart the hello-world-doodad service.
        </p>
        <p>
          If your doodad service is widget-enabled, you can also regenerate
          dead doo-dads using the /widgets/reincarnate.doodad address and the
          following message data:
        </p>
        <source>
          <reincarnation>
            <id>1</id>
            <password>ThisShouldBeEncrypted</password>
          </reincarnation>
```

```
        </source>
      </answer>
    </faq>
    <faq id="what-is-a-doo-dad">
      <question>What the h@!! is a doo-dad anyway?</question>
      <answer>
        <p>
          Doo-dads are components of the hello-world system used to obfuscate
          the misdirection server. It is critical to system health that the
          doo-dad pool have an appropriate threshold and that it be culled
          regularly.
        </p>
      </answer>
    </faq>
  </part>
</faqs>
```

NOTE: The content of the `answer` section is not pure XHTML. In fact, this section is formatted using XDoc syntax, which is *nearly* XHTML, with some added goodies like the `source` element.

This time, if you regenerate and refresh, you should see a new front page to the project website, one that reflects the content of the new APT document we just wrote. Likewise, if you click on the FAQ link (either in the menu at left, or in the main page), you should see the two entries we just added to the faq.fml page, nicely rendered into XHTML.

Notice that the newly generated website doesn't contain two menu references for the main page. This happens when we create a file called `index.apt` in the root `apt` format directory. Since there is concrete main page for the site, Maven assumes that the `About` report is no longer needed, so it's no longer generated. Also, this doesn't *have to* be an `apt` file; any file in any of the format directories (or the `resources` directory) called `index` with the appropriate extension for the format will suppress the `About` report and take its place on the main page.

Next, turn your attention to the `Developer Resources` link. If you click this link, you should see that our landing page for developers is displayed, and that the `Developer Resources` menu item is expanded to show the contents underneath it. This expansion is a result of the initial site-descriptor hint of `collapse="true"` on the `Developer Resources` menu item.

Finally, notice that the source files are all named with an extension that is appropriate to identify it as a file of a particular format (with the exception of XDoc sources). It's critical to remember that, regardless of the source file's extension, all references to these pages inside the content must reference the rendered page, not the source page. That is, any link to the index page must reference `index.html`, not `index.apt`. You can see this rule at work in the site descriptor, above.

This should get you started writing content for the hello-world project. Filling in the remainder of the links shown in the site menu is left as an exercise for the reader.

Deploying Your Project Website

NOTE: *It should go without saying, but we'll say it anyway: you'll have to supply your own DAV server and deployment URL for this example to work properly. The following is provided for illustrative purposes.*

After you iterate on the content of your project's documentation source a few times, and you feel it's ready to go live, you still need a mechanism for publishing out to your webserver. To address this, Maven's site plugin has a deployment feature that can handle several network protocols, including FTP, SCP, and DAV. For the purposes of our example, we'll stick to DAV. To use this deployment mechanism, you must first configure the `site` entry of the `distributionManagement` section in the POM, like this:

```
<project>
  ...
  <distributionManagement>
    <site>
      <id>hello-world.website</id>

      <!-- Set to the deployment URL on your own DAV server. -->
      <url>dav:https://dav.dev.example.com/sites/hello-world</url>
    </site>
  </distributionManagement>
  ...
</project>
```

NOTE: In much the same way as the SCM URLs above (in the `Source Repository` report discussion), the URL in this snippet has two protocols; the first one is trimmed from the rest of the URL - which is used as the connection URL for the deployment - and tells Maven what sort of connection to use (in our case, DAV).

Once we've added the `distributionManagement` section to our hello-world POM, we can try deploying the site:

```
$ mvn clean site-deploy
```

Configuring Server Options for Deployment

In some cases, a simple deployment like the one above will not work because you haven't supplied the correct username or password. At other times, such a deployment may cause problems for others because your deployment used the wrong file or directory mode. Whether you need to configure a username or a directory mode, the file you should edit is the `settings.xml`, usually located in the `.m2` subdirectory within your home directory. All configuration that pertains to our site deployment will happen inside the `servers/server` section with an `id` of 'hello-world.website'.

Configuring Server Authentication.

To configure a username/password combination for use during the site deployment, we'll include the following in `$HOME/.m2/settings.xml`:

```
<settings>
  ...
  <servers>
    <server>
      <id>hello-world.website</id>
      <username>jsmith</username>
      <password>BadPassword</password>
    </server>
    ...
  </servers>
  ...
</settings>
```

TIP: In the event you're using SCP for deployment, you may wish to use public-key authentication. To do this, specify the `publicKey` and `passphrase` elements, instead of the `password` element. You may still want to configure the `username` element, depending on your server's configuration.

Configuring File and Directory Modes.

To configure specific file and directory modes for use during the site deployment (modes control file and directory access among file owners, their groups, and the rest of the world), we'll include the following in `$HOME/.m2/settings.xml`:

```
<settings>
  ...
  <servers>
    ...
    <server>
      <id>hello-world.website</id>
      ...
      <directoryPermissions>0775</directoryPermissions>
      <filePermissions>0664</filePermissions>
    </server>
  </servers>
  ...
</settings>
```

The above settings will make any directories readable, writable, and listable by either the owner or members of the owner's primary group; the rest of the world will only have access to read and list the directory. Similarly, the owner or members of the owner's primary group will have access to read and write any files, with the rest of the world restricted to read-only access.

Tuning Your Project Website

Sometimes, the default look and feel of a Maven project website isn't enough. You may wish to customize your project's website beyond simply adding content, navigational elements, and custom logos. Maven offers several mechanisms for customizing your website that offer successively deeper access to content decoration and website structure. For small, per-project tweaks, providing a custom `site.css` is often enough. However, if you want your customizations to be reusable across multiple projects, or

if your customizations involve changing the XHTML that Maven generates, you should consider creating your own Maven website skin.

In order to demonstrate the types of customization available in both scenarios, we'll explore the `site.css` and experiment with building our own site skin below.

Create a Custom `site.css`

The simplest way to customize a Maven-generated website a single project is to provide a `site.css` file. Just like any images or XHTML content you provide for the website, the `site.css` file goes in the `src/site/resources` directory. More precisely, Maven expects this file to be in the `src/site/resources/css` subdirectory. This file allows you to override any theme-specific or global website style for your own project website only.

For example, if we decided that to change our hello-world website so that our menu headings stand out a little more, we might try the following:

`src/site/resources/css/site.css`

```
#navcolumn h5 {
  font-size: smaller;
  border: 1px solid #aaaaaa;
  background-color: #bbb;
  margin-top: 7px;
  margin-bottom: 2px;
  padding-top: 2px;
  padding-left: 2px;
  color: #000;
}
```

When you regenerate the website, you should notice that the menu headers are framed by a gray background, and separated from the rest of the menu content a little more. Using this file, any structure in the Maven-generated website can be decorated with custom CSS. The only drawback is that these styles are specific to the current project, which could mean that the project's look and feel doesn't integrate smoothly into a larger web presence. To make your custom site styles reusable, you'll have to build a custom skin.

TIP: Maven doesn't publish an annotated reference of the site structure or CSS styles used by the the site plugin. Although this site structure can vary if you're using a custom page template (we'll talk more about these below), not even the default page structure is documented. This can make discovering which id's and classes to override in your website a serious chore - unless you use a special CSS editing tool. I highly recommend the Web Developer and Firebug extensions for FireFox for this sort of work. Using these tools, you can view attributes such as HTML element hierarchy and CSS attributes that are in effect on any element from a loaded web page. In addition, you can even experiment with new CSS and see your changes reflected on the page instantaneously. Best of all, these are open-source tools! For more information, see the *Resources* section at the end of this chapter.

Create a Custom Site Template

At times, it's not just a simple matter of wanting a slightly different look ant feel for your project website. At times, you may want to add entire new features to the structure of the XHTML that is rendered. One such example is Javascript-enabled menus. Fortunately, Maven allows you to customize this sort of page-rendering through the incorporation of a custom page template. To illustrate, let's create a custom page template to make the hello-world website's menus come alive with the power of Javascript.

The `maven-site-plugin` uses a rendering engine called Doxia, which in turn uses a Velocity template to render the XHTML for each page. To change the page structure that is rendered by default, we can configure the site plugin in our POM to use a custom page template. We'll start by copying the default template from Doxia's Subversion repository, at http://svn.apache.org/viewvc/maven/doxia/doxia-sitetools/trunk/ doxia-site-renderer/src/main/resources/ org/apache/maven/doxia/siterenderer/resources/ default-site.vm?view=log (*http://svn.apache.org/viewvc/maven/doxia/doxia-sitetools/trunk/ doxia-site-renderer/src/main/resources/ org/apache/maven/doxia/siterenderer/resources/default-site.vm?view=log*) to the `src/site/site.vm` file in the `hello-world-site-skin` project directory. As you'll notice, this template is fairly involved, so we'll follow the copy-and-tweak methodology that has made open source so successful. Now that we have a basic page template to work with, all we need to do is make a few relatively small changes.

First, locate the `menuItem` macro. It resides in a section that looks like this:

```
#macro ( menuItem $item )

  ...

#end
```

TIP: Velocity works a lot like Java, in that it uses nested pairs of start and end markers. Since the end marker (quite literally `#end` in Velocity) is always the same, indentation is important to keep the code readable. In the example above, look for an `#end` line that matches the `#macro` line in indentation.

Now, we'll *replace* this macro definition with a new one, that injects Javascript references:

```
#macro ( menuItem $item $listCount )
  #set ( $collapse = "none" )
  #set ( $currentItemHref = $PathTool.calculateLink( $item.href, $relativePath ) )
  #set ( $currentItemHref = $currentItemHref.replaceAll( "\\", "/" ) )

  #if ( $item && $item.items && $item.items.size() > 0 )
    #if ( $item.collapse == false )
      #set ( $collapse = "collapsed" )
    #else
      ## By default collapsed
      #set ( $collapse = "collapsed" )
    #end
```

```
    #set ( $display = false )
    #displayTree( $display $item )

    #if ( $alignedFileName == $currentItemHref || $display )
      #set ( $collapse = "expanded" )
    #end
  #end
  <li class="$collapse">
    #if ( $item.img )
     #if ( ! ( $item.img.toLowerCase().startsWith("http") || $item.img.toLowerCase().startsWith("https")
        #set ( $src = $PathTool.calculateLink( $item.img, $relativePath ) )
        #set ( $src = $item.img.replaceAll( "\\", "/" ) )
        <img src="$src"/>
      #else
        <img src="$item.img" align="absbottom" style="border-width: 0"/>
      #end
    #end
    #if ( $alignedFileName == $currentItemHref )
      <strong>$item.name</strong>
    #else
      #if ( $item && $item.items && $item.items.size() > 0 )
      <a onclick="expand('list$listCount')" style="cursor:pointer">$item.name</a>
          #else
      <a href="$currentItemHref">$item.name</a>
          #end
    #end
  #if ( $item && $item.items && $item.items.size() > 0 )
    #if ( $collapse == "expanded" )
    <ul id="list$listCount" style="display:block">
        #else
    <ul id="list$listCount" style="display:none">
        #end
      #foreach( $subitem in $item.items )
            #set ( $listCounter = $listCounter + 1 )
        #menuItem( $subitem $listCounter )
      #end
    </ul>
  #end
  </li>
#end
```

Notice that we've added a new parameter to the `menuItem` macro. We'll also have to change references to this macro, or the resulting template may produce unwanted or internally inconsistent XHTML. To finish changing these references, we have to make a similar replacement in the `mainMenu` macro. Find this macro by looking for something similar to the following:

```
#macro ( mainMenu $menus )

  ...

#end
```

And replacing it with this:

```
#macro ( mainMenu $menus )
  #set ( $counter = 0 )
  #set ( $listCounter = 0 )
  #foreach( $menu in $menus )
    #if ( $menu.name )
    <h5 onclick="expand('menu$counter')">$menu.name</h5>
    #end
    <ul id="menu$counter" style="display:block">
      #foreach( $item in $menu.items )
        #menuItem( $item $listCounter )
                #set ( $listCounter = $listCounter + 1 )
      #end
    </ul>
    #set ( $counter = $counter + 1 )
  #end
#end
```

This new `mainMenu` macro is compatible with the new `menuItem` macro above, and also provides support for a Javascript-enabled top-level menu.

Next, we must also provide a small modification to the main XHTML template at the bottom of this template file, to introduce the Javascript method referenced in the new `menuItem` macro. Simply find the section that looks similar to the following:

```
<head>
  ...
 <meta http-equiv="Content-Type" content="text/html; charset=${outputEncoding}" />
  ...
</head>
```

and replace it with this:

```
<head>
  ...
 <meta http-equiv="Content-Type" content="text/html; charset=${outputEncoding}" />
  <script type="text/javascript">
      function expand( item ) {
              var expandIt = document.getElementById( item );
              if( expandIt.style.display == "block" ) {
                      expandIt.style.display = "none";
                      expandIt.parentNode.className = "collapsed";
              } else {
                      expandIt.style.display = "block";
                      expandIt.parentNode.className = "expanded";
              }
      }
  </script>
  #if ( $decoration.body.head )
    #foreach( $item in $decoration.body.head.getChildren() )
      #if ( $item.name == "script" )
        $item.toUnescapedString()
      #else
        $item.toString()
      #end
    #end
```

```
    #end
  </head>
```

Finally, we need to make a minor modification to the hello-world POM, which will configure the site plugin to use our new page template:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <configuration>
          <templateDirectory>src/site</templateDirectory>
          <template>site.vm</template>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Now, you should be able to regenerate your project website.

GOTCHA! Apparently, in cases where the website - or a custom site skin - provides its own Velocity template, Doxia's site-renderer component (version 1.0-alpha-8, brought in by the maven-site-plugin, version 2.0-beta-5) won't use the default `maven-base.css` file provided by Doxia. To get your website looking right again, you'll need to copy all of the resources *except the* `maven-theme.css` *file* into the hello-world-site-skin project. Perhaps the simplest way to do this is to checkout the Doxia site-renderer project, remove the `maven-theme.css` file, and copy the rest of the files over, like this:

```
hello-world$ cd ..
workdir$ svn co \
   http://svn.apache.org/repos/asf/maven/doxia/doxia-sitetools/trunk/doxia-site-renderer
workdir$ rm \
   doxia-site-renderer/src/main/resources/org/apache/maven/doxia/siterenderer/resources/css/maven-theme
workdir$ cp -rf \
   doxia-site-renderer/src/main/resources/org/apache/maven/doxia/siterenderer/resources/* \
    hello-world/src/site/resources
```

Then, rebuild the skin and regenerate the site, and your website should look normal again. This issue has documented in Doxia's JIRA under DOXIA-106, found at *http://jira.codehaus.org/browse/DOXIA-106*.

When you regenerate the site, you'll notice that the `Developer Resources` menu item now looks just like regular text. This is caused by a quirky interaction between the site's CSS and our new custom page template. It can be fixed by modifying our `site.css` to restore the proper link color for these menus. Simply add this:

```
li.collapsed, li.expanded, a:link {
  color:#36a;
}
```

Once again, regenerate the website. This time, the menu's link color should be corrected. When you click on the `Developer Resources` link, it will expand the submenu and display the `Architecture` and `Embedding` menu-items. Finally, since we've turned the `Developer Resources` menu-item into a dynamically-folding submenu, we've lost the ability to reach the `developer/index.apt` page, which is our landing page for this subsection of the website. To restore this, let's modify our site descriptor once more, to add a new menu item for the page:

```
<project name="Hello World">
  ...
  <menu name="Main Menu">
    ...
    <item name="Developer Resources" collapse="true">
      <item name="Overview" href="/developer/index.html"/>
      <item name="System Architecture" href="/developer/architecture.html"/>
      <item name="Embedder's Guide" href="/developer/embedding.html"/>
    </item>
  </menu>
  ...
</project>
```

Regenerate the website once more, then click on the `Developer Resources` menu, and you'll see that order has been restored. Of course, we could be a little more clever, and automatically load the first sub-menu item when this menu is triggered. Then, there are endless small tweaks we could make to this basic template modification. Perfecting this template is a subjective exercise best left to you, the reader.

Reusable Website Skins

Quite often, projects are part of a larger organization. Whether that be an open-source software foundation, some sort of affiliation program, or even the modularization of a larger project, these project's websites often work better when they share a common navigation and over-arching style. Just as it is difficult to preserve such common elements between individual HTML pages, coordinating a series of `site.css` files from multiple projects represents a serious maintenance burden.

This is where Maven's support for website skinning can save you a lot of pain and suffering. In many cases, one of Maven's alternative website skins can do the job nicely. You can choose from several skins:

- **Maven Classic Skin** - `org.apache.maven.skins:maven-classic-skin:1.0`
- **Maven Default Skin** - `org.apache.maven.skins:maven-default-skin:1.0`
- **Maven Stylus Skin** - `org.apache.maven.skins:maven-stylus-skin:1.0.1`

You can find an up-to-date and comprehensive listing in the Maven repository: `http://repo1.maven.org/maven2/org/apache/maven/skins/`. These each provide their own layout for navigation, content, logos, etc. However, if none of these fit the bill, you also have the option of creating your own.

Creating a custom skin is a simple matter of wrapping your customized `maven-theme.css` in a Maven project, so that it can be referenced by groupId, artifactId, and version. It can also include resources such as images, and a replacement website template (written in Velocity) that can generate a completely different XHTML page structure. In most cases, custom CSS can manage the changes you desire. To demonstrate, let's create a designer skin for the hello-world project, starting with a custom `maven-theme.css`.

Before we can start writing our custom CSS, we need to create a separate Maven project to allow the hello-world site descriptor (not to mention those from other projects) to reference it. First, use Maven's archetype plugin to create a basic project. Issue the following command from the directory *above* the hello-world project's root directory:

```
$ mvn archetype:create -DartifactId=hello-world-site-skin -DgroupId=book.sitegen
```

Just as before, this will create a project (and a directory) called `hello-world-site-skin`.

NOTE: Since we're working on the custom skin, you should issue all of the following commands from the new `hello-world-site-skin` directory, unless otherwise directed.

Also as before, Maven will create a skeletal source file and test case for this new project. We don't need these files, but we do need to create a resources directory under `src/main` to house our skin content:

```
$ rm -rf src/main/java src/test
$ mkdir src/main/resources
```

In addition, you may want to remove the `dependencies` section from the new `pom.xml` file; we don't have any dependencies for this site skin (since we've deleted the sample source code).

Create a Custom Theme CSS

Now, we're ready to write some CSS for our custom skin. To build on the lessons learned with the `site.css` file, let's generalize those custom styles here, for use in other projects. Unlike the `site.css` file, which goes in the site-specific source directory for a project, the `maven-theme.css` file lives in the `src/main/resources/css` directory. The reason for the change is that we're building a jar artifact here. Therefore, in order to have the `maven-theme.css` file included in that jar, it must reside in the main project-resources directory, `src/main/resources`.

Also, since we're simply attempting to customize the default Maven website theme - called `maven-classic-skin` - we should begin with the theme file from this skin, then customize it to include our new styles. To get a copy of this theme file, save the contents of *http://svn.apache.org/viewvc/maven/skins/trunk/maven-default-skin/src/main/resources/css/maven-theme.css?view=co* to `src/main/resources/css/maven-theme.css` in our new skin project.

Now that we have the base theme file in place, let's customize it using the CSS from our old `site.css` file. **NOTE:** You'll need to **replace** the corresponding CSS in the original theme file with the following:

```
#navcolumn h5 {
  font-size: smaller;
  border: 1px solid #aaaaaa;
  background-color: #bbb;
  margin-top: 7px;
  margin-bottom: 2px;
  padding-top: 2px;
  padding-left: 2px;
  color: #000;
}
```

Now, we must install this skin into the local Maven repository before we can reference it:

```
$ mvn clean install
```

Once the installation is complete, switch back to the hello-world project directory, and backup the `site.css` file, to get it out of the way:

```
$ mv src/site/resources/css/site.css src/site/resources/css/site.css.bak
```

Then, modify the site descriptor (`site.xml`, if you remember) to use the new skin:

```
<project name="Hello World">
  ...
  <skin>
    <groupId>book.sitegen</groupId>
    <artifactId>hello-world-site-skin</artifactId>
  </skin>
  ...
</project>
```

Regenerate the project website for hello-world, and you should see the same gray-panelled menu headers that we setup in the preceding section. Now, we can go a step further, and incorporate the custom page template that we created previously into the new hello-world-site-skin.

Incorporate a Custom Site Template

Previously, we customized the hello-world project site directly using our own Velocity page template and a minor configuration of the site plugin in the POM. In much the same way, we can customize any number project websites with the same custom template by incorporating it in our new hello-world-site-skin project.

As I mentioned previously, Maven's site plugin makes use of the Doxia rendering engine to combine source documents with a page template, and output XHTML. When reading the page template from a custom skin, Doxia's site-rendering tools will expect to find a file called `META-INF/maven/site.vm` inside the skin jarfile. So, incorporating the

custom page template we developed previously is a simple matter of copying the template file into the correct location within the hello-world-site-skin:

```
$ cp hello-world/src/site/site.vm \
    hello-world-site-skin/src/main/resources/META-INF/maven
```

Also, since we're not using the per-project page template any more, we should remove the reference to it in the hello-world POM. Remove or comment out the following section:

```
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <configuration>
    <templateDirectory>src/site</templateDirectory>
    <template>site.vm</template>
  </configuration>
</plugin>
```

The `src/site/site.vm` file in the hello-world project isn't used any longer. It will not hurt to leave it where it is, but neither will it hurt to remove it. The same goes for the `src/site/resources/css/site.css.bak` file.

Finally, remember the workaround for **DOXIA-106**. We'll need to move these files (which are now polluting our hello-world project unnecessarily) into the new hello-world-site-skin's `src/main/resources` directory:

```
$ cd ..
$ mkdir -p hello-world-site-skin/src/main/resources/css
$ mv hello-world/src/site/resources/css/maven-base.css \
    hello-world-site-skin/src/main/resources/css
$ mkdir -p hello-world-site-skin/src/main/resources/images
$ mv hello-world/src/site/resources/images/logos \
    hello-world-site-skin/src/main/resources/images
$ mv hello-world/src/site/resources/images/expanded.gif \
    hello-world-site-skin/src/main/resources/images
$ mv hello-world/src/site/resources/images/collapsed.gif \
    hello-world-site-skin/src/main/resources/images
```

Now, we can rebuild the site skin, and regenerate the hello-world website. If you refresh your browse, and find that you're wondering what happened to the color of the `Developer Resources` menu, remember that we modified the `site.css` file to accommodate the new page template. To carry this behavior over to the new skin, we simply perform the same modification on the `maven-theme.css` file. Change this:

```
a:link {
  ...
}
```

to this:

```
li.collapsed, li.expanded, a:link {
  ...
}
```

Rebuild the skin, then regenerate the website, and you'll see that the `Developer Resources` menu has returned to normal.

Tips and Tricks

In addition to the techniques used to create a basic project website and fill it in with content, there are some useful tips that you should know. Each of the following items can be used independently of the rest. For this reason, it's not necessary that you read this section straight through; think of it as a reference for those obscure-but-useful site-plugin details.

Inject HTML into <HEAD/> of Each Page

Occasionally, it would be nice to have the ability to inject a small amount of XHTML into the head of each page on the generated website. This could be useful if we wanted to add an RSS link to the news page of our hello-world project, for instance. To do this, Maven supports the following section in the site descriptor:

```
<project name="Hello World">
  ...
  <body>
    <head>
      <link href="http://dev.example.com/sites/hello-world/feeds/news"
            type="application/atom+xml"
            id="auto-discovery"
            rel="alternate"
            title="Hello-World News" />
    </head>
    ...
  </body>
</project>
```

GOTCHA! Don't be confused by the fact that the `head` section goes *inside* the `body` section in the site descriptor. Apparently, the site descriptor is not meant to represent an abstracted page template, despite the common terminology.

Add Links under Your Site Logo

If your project has an affiliation with one or more other projects, you may want to display persistent links to those projects prominently on your project website. A good example of this is Codehaus Mojo website, at *http://mojo.codehaus.org*. This site hosts Maven plugins that, for one reason or another, cannot be hosted on Maven's main plugin website. Intuitively, it makes sense that users looking at these plugins might want to know more about Maven itself. To make their lives easier, the Mojo website contains a link to the Maven project on the right side of the bar under the site logo. This link is always in the same place, regardless of where you are in the Mojo website.

It gives the user instant access to other websites that have a close, integral relationship with the current one.

In the nomenclature of the site descriptor, these are simply called `links`. To add a links section to your project website, simply modify your site descriptor to include something similar to this:

```
<project name="Hello World">
  ...
  <body>
    ...
    <links>
      <item name="Sibling Project" href="http://dev.example.com/sites/sibling"/>
      <item name="Apache Maven Doxia" href="http://maven.apache.org/doxia"/>
    </links>
    ...
  </body>
</project>
```

Add Breadcrumbs to Your Site

When your project's website makes up only a subsection of another, encompassing website, it often helps users to know how to navigate back up to the outer site. This can often come about when the project is hosted as part of a larger community, where the over-arching main page may provide classifications, searchability, and other cross-project features.

Such breadcrumb links are easy to configure for your project website. Simply add a new section to the `body` of your site descriptor that looks similar to this:

```
<project name="Hello World">
  ...
  <body>
    ...
    <breadcrumbs>
      <item name="Example Software Foundation" href="http://www.example.com"/>
      <item name="ESF Projects" href="http://dev.example.com"/>
    </breadcrumbs>
    ...
  </body>
</project>
```

NOTE: The `item` entries in the `breadcrumbs` section are listed in parent-to-child order, to symbolize that each successive link is a sub-site of the last.

Add the Project Version

Often, it's helpful to give your users a clear indication of which version of your project is documented by a particular website. This is particularly useful when used in combination with the *Maintain Documentation for Multiple Project Versions* tip, where

documentation for multiple project versions is maintained side-by-side. To display your project's version on the website, simply add the following to your site descriptor:

```
<project name="Hello World">
  ...
  <version position="left"/>
  ...
</project>
```

This will position the version (in the case of the hello-world project, it will say "Version: 1.0-SNAPSHOT") in the upper left-hand corner of the site, right next to the default "Last Published" date. Valid positions for the project version are:

- `left` Left side of the bar just below the site logo
- `right` Right side of the bar just below the site logo
- `navigation-top` Top of the menu
- `navigation-bottom` Bottom of the menu
- `none` Suppress the version entirely

Modify the Publication Date Format and Location

In some cases, you may wish to reformat or reposition the "Last Published" date for your project website. Just like the project version tip above, you can specify the position of the publication date by using one of the following:

- `left` Left side of the bar just below the site logo
- `right` Right side of the bar just below the site logo
- `navigation-top` Top of the menu
- `navigation-bottom` Bottom of the menu
- `none` Suppress the publication date entirely

Now, add the publication date element, using the position you selected above, like this:

```
<project name="Hello World">
  ...
  <publishDate position="navigation-bottom"/>
  ...
</project>
```

By default, the publication date will be formatted using **MM/dd/yyyy**. You can change this format by using the standard notation found in the JavaDocs for `java.text.SimpleDateFormat` (see *http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html* for more information). To reformat the date using **yyyy-MM-dd**, use the following:

```
<project name="Hello World">
  ...
  <publishDate position="navigation-bottom" format="yyyy-MM-dd"/>
```

```
  ...
</project>
```

Summary

The instructions in this chapter have guided you through the process of creating a customized project website, using the `maven-site-plugin`. We wrote documentation for a sample project in various source formats, and watched as the Doxia rendering engine transformed our source documents into XHTML. In addition, we investigated how to tailor the many options supported by the site plugin. Among these were navigational menus, breadcrumbs and other special links, and even reusable website skins that are capable of applying a custom look-and-feel - including logos, javascript, CSS, and more - to a set of websites. Although Maven's documentation-rendering features make maintaining such documentation much simpler, this represents only one aspect of the website-generation capabilities present in Maven's site plugin.

Resources

- Firefox: *http://www.firefox.com*
- Web Developer extension for Firefox: *https://addons.mozilla.org/en-US/firefox/addon/60*
- Firebug extension for Firefox: *https://addons.mozilla.org/en-US/firefox/addon/1843*

CHAPTER 9

Assemblies

Introduction

Maven is more than a simple build tool. But this is not to say that Maven is not very good a doing simple builds. In fact that is a core feature of Maven's 95% principle (the Maven team's goals are to make available tools for 95% of the most common use-cases). With this in mind, the `maven-assembly-plugin` was created to handle the most common artifact packaging concerns - such as bundling source code, or zipping up an entire project. I suggest you make close friends with this plugin, since - you may find - it is capable of most packaging features you may require.

Using Assemblies

The `maven-assembly-plugin` contains seven goals. The most common goals are `assembly` - meant to be used as a stand-alone goal (since it forks a lifecycle to the `package` phase) - and `attached` - which does not fork a lifecycle, and is used to attach an assembly to a build lifecycle (defaulting to `package`). Those two goals have corresponding "directory" goals - `directory` and `directory-inline`, respectively) to run assemblies that assemble projects into a build directory, rather than a single zip-type archive artifact (jar, tar.gz, etc.). Finally, there are two more corresponding goals for projects under multi-module projects - `single` and `directory-single`, respectively.

The seventh goal is `unpack` which is rarely necessary to use - since one should write a descriptor which executes the same behavior: to unpack dependencies inside the archive.

```
mvn assembly:assembly
mvn assembly:directory
```

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
```

```
        <executions>
          <execution>
            <id>assemble</id>
            <goals>
              <goal>attached</goal>
              <goal>directory-inline</goal>
              <goal>single</goal>
              <goal>directory-single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Default Assemblies

The simplest assemblies to use are the pre-defined ones. At the time of this writing, there are four types (note that the first three assemblies each create three file typed artifacts - `zip`, `tar.gz` and `tar.bz2`):

- **bin** - Assembles artifacts packaged with `README*`, `LICENSE*`, and `NOTICE*` files in the project's base directory.
- **src** - Assembles artifacts packaged with `README*`, `LICENSE*`, `NOTICE*` and the `pom.xml`, along with all files under the project's `src` directory.
- **project**: this is used to create artifacts of the entire source project, including the Maven POM and other files outside of your source directory structure, but excluding all SCM metadata directories - such as `CVS>> or <<<.svn` - and the `$basedir/target` directory (specifically - if you output to a directory other than `target`, you may pack up binaries).
- **jar-with-dependencies** - Explodes all dependencies of this project and packages the exploded forms into a `jar` along with the project's `outputDirectory`. Note that this default assembly descriptor only creates a jar, not the three zip types.

There are two ways to use the above assemblies. As usual - the command line, or under the plugin's `configuration` element in the target project's pom. If you use the command-line, only one assembly may be used, via the `descriptorId` property.

```
mvn assembly:assembly -DdescriptorId=jar-with-dependencies
```

Otherwise, you may define one or more `descriptorRefs` assemblies.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
```

```
            <descriptorRef>jar-with-dependencies</descriptorRef>
            <descriptorRef>src</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

When an assembly is utilized, the name of the assembly is used as the assembly artifact's classifier. This means that the name of the descriptor `id` is appended to the end of the artifact file's prefix. The usage of an assembly does not affect the standard artifact created by a project's packaging type. For example, if I have a project with coordinates of `com.mycompany:my-app:1.0:jar` and configured `descriptorRef` of `src`, then four artifacts are created: `my-app-1.0.jar`, and the three `src` assembly artifacts: `my-app-1.0-src.zip`, `my-app-1.0-src.tar.gz`, and `my-app-1.0-src.tar.bz2`.

Custom Assemblies

Although the default assemblies above will cover the majority of use-cases, the ability to use custom assemblies is equally important. If your group has an assembly, simple place the descriptor xml file under your project structure and link to it within the POM. You may wish to bundle the assembly execution to a phase as well.

Beyond the POM, this is the largest configuration file in the standard Maven plugin set.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptor>src/main/assembly/my-src.xml</descriptor>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>attached</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

Creating Assemblies

It is a relatively simple task to create a descriptor that specifies the type and scope of files that are to be packaged as an assembly artifact. As mentioned above, assemblies

are defined in xml files and imported via the maven-assembly-plugin configuration. A simple assembly descriptor that zips up a project's source code would look like this:

```
<assembly>
  <id>my-src</id>
  <formats>
    <format>zip</format>
  </formats>
  <fileSets>
    <fileSet>
      <includes>
        <include>LICENSE*</include>
        <include>pom.xml</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>src</directory>
    </fileSet>
  </fileSets>
</assembly>
```

We will look at the different types of components available for assembly, but first look at the formats that an assembly may package as.

NOTE: *To avoid duplication of descriptor elements, and as an easy way to conceptualize element sets that appear in multiple locations - look for named BEGIN: / END: comment blocks. Where the comment names appear again, those same elements appear serving the same function.*

Archive types

When the package phase is run it will still create the artifact of the packaging type, for example the `target/artifactId-version.jar` file, but in addition will bundle up the source code into a `target/artifactId-version-src.zip` file. This assumbly will generate all the formats defined above. The possible archive types are limited to the Plexus implementations of the `org.codehaus.plexus.archiver.Archiver` component, in the `plexus-archiver` project. The list at the time of this writing is:

- bzip2
- dir
- ear
- gzip
- jar
- tar
- tar.gz
- tar.bz2
- tbz2

- war
- zip

One or more of these formats may be defined in the descriptor.

```
<assembly>
  <id>release-bin</id>
  <formats>
    <format>tar.gz</format>
    <format>tbz2</format>
    <format>zip</format>
  </formats>
  <baseDirectory>release/bin</baseDirectory>
  <includeBaseDirectory>true</includeBaseDirectory>
  <includeSiteDirectory>false</includeSiteDirectory>
  ...
</assembly>
```

Other than the required `id` and `formats` elements, there are three more simple top-level elements.

- **baseDirectory** - The name of the base directory to use if `includeBaseDirectory` is set to `true`. Defaults to the project's artifactId. Everything within the assembly-generated archive will not be in the *root* of the archive, but under the `baseDirectory` instead.
- **includeBaseDirectory** - Includes the base directory in the artifact if set to `true` - the default - otherwise the output will be put in the archive's root.
- **includeSiteDirectory** - Set to `true` if you wish to assemble the project's site into the artifact, otherwise `false` - the default.

Files

The simplest and most common assemblies package collections of denoted project files into an artifact. It does this in two ways, by specifying single files, or sets of files (directories).

Single files are moved into the assembly in a very straightforward manner, and can probably be understood by the example below, without much description.

```
<assembly>
  ...
  <files>
    <file>
      <source>src/main/exec/linux-run.sh</source>
      <outputDirectory>bin</outputDirectory>
      <destName>run.sh</destName>
      <filtered>true</filtered>
      <lineEnding>unix</lineEnding>
      <fileMode>0554</fileMode>
    </file>
  </files>
```

```
  ...
</assembly>
```

- **source** - The module's relative path - or absolute path - to the file to include in the assembly.
- **outputDirectory** - The directory within the assembly to output the file to.
- **destName** - The destination filename in the `outputDirectory` - if not set, the default is source filename.
- **filtered** - `true` or `false`, depending upon whether the file should be filtered by Maven (if you recall, filtering in Maven replaces a text file's `${property.name}` with the value of `property.name` - where this is a POM property).
- **lineEnding** - The style of the archived file's line-endings; there are five legal values.
  - — `keep` - Keep all line endings as they are - default
  - — `crlf` - Carraige-return/line-feed
  - — `lf` - Line-feed
  - — `unix` - Unix-style line endings
  - — `dos` - DOS-style line endings
- **fileMode** - The archived file's mode encoded in Unix's octal permission style - (user)(group)(other) where each component is a sum of read = 4, write = 2, or execute = 1. Defaults to 0644. From the value 0554 above: the user and group can read/execute, while other can read only.

```
<assembly>
  ...
  <!-- BEGIN: File Sets -->
  <fileSets>
    <fileSet>
      <directory>src/main/java</directory>
      <filtered>false</filtered>
      <lineEnding>keep</lineEnding>

      <!-- BEGIN: File Set Includes/Excludes -->
      <includes>
        <include>**/*.java</include>
      </includes>
      <excludes>
        <exclude>**/*Test.java</exclude>
      </excludes>
      <!-- END: File Set Includes/Excludes -->

      <!-- BEGIN: Directory Output Mask Elements -->
      <outputDirectory>src</outputDirectory>
      <useStrictFiltering>false</useStrictFiltering>
      <useDefaultExcludes>true</useDefaultExcludes>
      <fileMode>0444</fileMode>
      <directoryMode>0755</directoryMode>
      <!-- END: Directory Output Mask Elements -->
```

```
    </fileSet>
  </fileSets>
  <!-- END: File Sets -->
  ...
</assembly>
```

- **directory** - The absolute or relative location from the module's directory. For example, "src/main/java" would select this subdirectory of the project in which this dependency is defined.
- **filtered**, **lineEnding** - *Same as above, but for all files in the fileSet*

File Set Includes/Excludes

- **includes** - Contains a list of `include` elements which are an ant-like filename mask, denoting files to include in this fileSet. Default is **/** (all files).
- **excludes** - Contains a list of `exclude` elements which are an ant-like filename mask, denoting files to exclude from this fileSet. Default is no exclusions. Note that excludes always take priority over includes - this is true for all `include`/`exclude` pairs.

Directory Output Mask Elements

The remainder of the `fileSet` description contains a set of elements used to determine several things about the output directory, including naming, file filters and final file and directory states.

- **outputDirectory** - Sets the output directory relative to the root of the root directory of the assembly. For example, "log" will put the specified files in the log directory.
- **useStrictFiltering** - Any include/exclude patterns which are unused in filtering any files will cause the build to fail with an error. For the example above, if the project contains no `.java` files, this assembly will fail.
- **useDefaultExcludes** - Denotes whether the `maven-assembly-plugin`'s standard exclusion set should apply. Defaults to `true`; set to `false` if you wish this assembly to package `CVS` and `.svn` metadata directories.
- **fileMode** - *Same as above, but for all files in the fileSet*
- **directoryMode** - *Same as above, but for the directory of this fileSet*. Defaults to 0755.

Dependencies

Beyond files, project dependencies are also common to package within an archive. These can be defines under the `dependencySets` element list. The "Directory Output Mask Elements" comment is a marker for the same set of elements defined above.

```
<assembly>
  ...
```

```
  <!-- BEGIN: Dependency Sets -->
  <dependencySets>
    <dependencySet>
      <scope>compile</scope>

      <!-- BEGIN: Artifact Matching Includes/Excludes -->
      <includes>
        <include>com.mycompany:module*</include>
      </includes>
      <excludes>
        <exclude>com.mycompany:module3</exclude>
      </excludes>
      <!-- END: Artifact Matching Includes/Excludes -->

      <!-- BEGIN: File Output and Unpacking -->
      <outputFileNameMapping>${artifactId}</outputFileNameMapping>
      <unpack>true</unpack>
      <unpackOptions>
        <!-- File Set Includes/Excludes -->
        <filtered>true</filtered>
      </unpackOptions>
      <!-- END: File Output and Unpacking -->

      <!-- Directory Output Mask Elements -->

    </dependencySet>
  </dependencySets>
  <!-- END: Dependency Sets -->
  ...
</assembly>
```

- **scope** - The scope for this dependency set; the are five legal values are the same as the POM's dependency scopes.
  - —**runtime** - this scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath. This is the default.
  - —**compile** - this is the default scope, used if none is specified. Compile dependencies are available in all classpaths.
  - —**test** - this scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.
  - —**provided** - this is much like compile, but indicates you expect the JDK or a container to provide it. It is only available on the compilation classpath, and is not transitive.
  - —**system** - this scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

Artifact Matching Includes/Excludes

- **includes** - Contains a list of `include` elements, each containing an artifact-matching pattern - modules matching these elements will be included in this set. If none is present, then `includes` represents all valid values.
- **excludes** - Similar to `includes`, but modules matching these elements will be excluded from the set. `excludes` takes priority over includes

File Output and Unpacking

- **outputFileNameMapping** - The pattern for all dependencies included in the assembly. Default is `${artifactId}-${version}.${extension}` where the properties are the dependency values.
- **unpack** - Unpack all dependencies into the specified output directory if `true`, else dependencies will be includes as archives (such as jars), which is the default.
- **unpackOptions** - Allows the specification of includes, excludes and filtering options for unpacked items from dependency artifacts.

  —**includes**, **excludes**, **filtered** - *Same descriptions as fileSet definitions above*

A little more information about the artifact-matching patterns - from the assembly-plugin document:

The following easy rules should be applied when specifying artifact-matching patterns:

1. Artifacts are matched by a set of identifier strings. In the following strings, type is 'jar' by default, and classifier is omitted if null.
   - groupId:artifactId
   - groupId:artifactId:type:classifier
   - groupId:artifactId:version:type:classifier
2. Any '*' character in an include/exclude pattern will result in the pattern being split, and the sub-patterns being matched within the three artifact identifiers mentioned above, using String.indexOf(..).
3. When no '*' is present in an include/exclude pattern, the pattern will only match if the entire pattern equals one of the three artifact identifiers above, using the String.equals(..) method.
4. In case you missed it above, artifact-identification fields are separated by colons (':') in the matching strings. So, a wildcard pattern that matches any artifact of type 'war' might be specified as *:war:*.

Modules

A `moduleSet` represent one or more `module`s present inside a pom.xml. This allows you to include sources or binaries belonging to a multi-module project's individual `modules`. It contains the ability to extract specific modules, sources, and even binaries

```
<assembly>
  ...
  <moduleSets>
    <moduleSet>
      <includeSubModules>true</includeSubModules>

      <!-- Artifact Matching Includes/Excludes -->

      ...
    <moduleSet>
  <moduleSets>
  ...
</assembly>
```

- **includeSubModules** - Process all sub-modules under this module-set if true - the default; else the sub-modules will be excluded from processing.
- **includes**, **excludes** - *Same descriptions as dependencySet definitions above*

The `sources` element represents non-output files from specified module projects. The element specifies a set of files (`fileSets`) and contains the "Directory Output Mask Elements" to specify how these files will be output into the assembly archive. Moreover, sources contains three other elements.

```
<assembly>
  ...
  <moduleSets>
    <moduleSet>
      ...
      <sources>
        <includeModuleDirectory>true</includeModuleDirectory>
        <outputDirectoryMapping>${artifactId}</outputDirectoryMapping>
        <excludeSubModuleDirectories></excludeSubModuleDirectories>

        <!-- File Sets -->

        <!-- File Set Includes/Excludes -->

        <!-- Directory Output Mask Elements -->

      </sources>
      ...
    </moduleSet>
  </moduleSets>
  ...
</assembly>
```

- **includeModuleDirectory** - The `outputDirectoryMapping` value is prepended to the outputDirectory values of any fileSets applied to it if `true` - the default.
- **outputDirectoryMapping** - The base directory pattern for all modules included in this assembly. Default is the module's `${finalName}`.
- **excludeSubModuleDirectories** - Specifies whether sub-module directories below the current module should be excluded from fileSets applied to that module.

This might be useful if you only mean to copy the sources for the exact module list matched by this ModuleSet, ignoring (or processing separately) the modules which exist in directories below the current one. Default value is true.

The `binaries` element represents the output files of matching module projects. This element specifies a set of dependencies (`dependencySets`), the set of "File Output" elements, and also contains the "Directory Output Mask Elements" to specify how these files will be output into the assembly archive. Moreover, sources contains two other elements.

```
<assembly>
  ...
  <moduleSets>
    <moduleSet>
      ...
      <binaries>
        <attachmentClassifier></attachmentClassifier>
        <includeDependencies>true</includeDependencies>

        <!-- Dependency Sets -->

        <!-- File Output and Unpacking -->

        <!-- File Set Includes/Excludes -->

        <!-- Directory Output Mask Elements -->

      </binaries>
    </moduleSet>
  </moduleSets>
  ...
</assembly>
```

- **attachmentClassifier** -
- **includeDependencies** - Include the direct and transitive dependencies of the project modules if true - the default; else, only include the module packages.

Assembling Maven Repositories

```
<assembly>
  ...
  <repositories>
    <repository>
      <includeMetadata>true</includeMetadata>
      <scope>runtime</scope>
      <groupVersionAlignments>
        <groupVersionAlignment>
          <id>com.mycompany</id>
          <version>2.1</version>
          <excludes>
            <exclude>my-artifact1</exclude>
          </excludes>
```

```
        </groupVersionAlignment>
      </groupVersionAlignments>

      <!-- File Set Includes/Excludes -->

      <!-- Directory Output Mask Elements -->

    </repository>
  </repositories>
  ...
</assembly>
```

- **includeMetadata** - If set to true, this property will trigger the creation of repository metadata which will allow the repository to be used as a functional remote repository. Default value is false.
- **scope** - Specifies the scope for artifacts included in this repository. Default scope value is "runtime".
- **groupVersionAlignments** - Align a group of artifacts to a specified version. For example, all artifacts with groupId of "com.mycompany" can be aligned to one version "2.1", excluding specific artifacts if desired.
  - —**id** - The groupId of the artifacts for which you want to align the versions.
  - —**version** - The version you want to align this group to.
  - —**excludes** - When *exclude* subelements are present, they define the artifactIds of the artifacts to exclude. If none is present, then *excludes* represents no exclusions. An exclude is specified by providing one or more of *exclude* subelements.

Assembly Tricks and Tips

The first half of this chapter went over basic useage and a look at the assembly descriptor - what the elements are and logical groupings. Much of that information can be gathered from the standard documentation but was added here for completeness (online references are linked at the end of this chapter). The rest of this section will describe some tricks/tips to make the most out of your `maven-assembly-plugin` experience.

Externalizing Component Descriptors

There is one last element in the assembly descriptor that we have skipped - `componentDescriptors`. This allows one to reuse common component descriptions across several assemblies. A `component` element is placed in a seperate file and contains only four elements from the assembly descriptor: `fileSets`, `files`, `dependencySets`, `repositories`.

For example: suppose we have a project with two assemblies - one for assembling all xml files in a project, and another for assembling all xml files in a project and it's dependencies. The components "all xml files in a project" overlap in the two assemblies

- so we externalize them into a file named `src/main/assembly/component/xml-files.xml`.

```
<component>
  <fileSets>
    <fileSet>
      <directory>src</directory>
      <outputDirectory>xml</outputDirectory>
      <includes>
        <include>**/*.xml</include>
      </includes>
    </fileSet>
  </fileSets>
</component>
```

This component descriptor can then be imported as a path from the project's `${base dir}`, creating the two assemblies below.

```
<assembly>
  <id>local-xml</id>
  <componentDescriptors>
   <componentDescriptor>src/main/assembly/component/xml-files.xml</componentDescriptor>
  </componentDescriptors>
</assembly>
```

```
<assembly>
  <id>all-xml</id>
  <dependencySets>
    <dependencySet>
      <scope>compile</scope>
      <unpack>true</unpack>
      <unpackOptions>
        <includes>
          <include>**/*.xml</include>
        </includes>
        <filtered>true</filtered>
      </unpackOptions>
      <outputDirectory>xml</outputDirectory>
      <useStrictFiltering>true</useStrictFiltering>
    </dependencySet>
  </dependencySets>
  <componentDescriptors>
   <componentDescriptor>src/main/assembly/component/xml-files.xml</componentDescriptor>
  </componentDescriptors>
</assembly>
```

Example Module Assembly

Modules sets are complex beasts to wrestle, as you might imagine from the descriptor above. The `maven-assembly-plugin` has some excellent documentation concerning the `moduleSet` element and it's usage. For the sake of completeness, we'll cover a quick example here.

Suppose you have a multi-module project named `com.mycompany:my-app`, with two modules `com.mycompany:module1` and `com.mycompany:module2`. The directory structure is as follows:

```
my-app
|-- module1
|   `-- ... some projecty stuff in here ...
|-- module2
|       `-- runscript
|               `-- run.sh
|-- pom.xml
`-- src
        `-- main
                `-- assembly
                    `-- bin-release.xml
```

We want to create a binary release bundle, that builds the module1, and then zip the jar up in a lib directory, with a script that runs the module1 jar in a bin directory that lives in module2. Finally, when a user unzips the bundle it will extract to a directory of the project name - `my-app`.

```
<project>
  <groupId>com.mycompany</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>module1</module>
    <module>module2</module>
  </modules>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorSourceDirectory>src/main/assembly</descriptorSourceDirectory>
        </configuration>
        <executions>
          <execution>
            <id>bin-release</id>
            <phase>package</phase>
            <goals>
              <goal>attached</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

The `bin-release.xml` descriptor then includes the scripts of module2, while packaging the `binaries` of all other modules (and submodules) and their dependencies into a `lib` directory. These files are then placed into a

```
<assembly>
  <id>bin-release</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>true</includeBaseDirectory>
  <baseDirectory>${artifactId}</baseDirectory>
  <moduleSets>
    <moduleSet>
      <excludes>
        <exclude>com.mycompany:module2</exclude>
      </excludes>
      <includeSubModules>true</includeSubModules>
      <binaries>
        <outputDirectory>lib</outputDirectory>
        <unpack>false</unpack>
        <includeDependencies>true</includeDependencies>
        <dependencySets>
          <dependencySet>
            <scope>runtime</scope>
            <outputDirectory>lib</outputDirectory>
          </dependencySet>
        </dependencySets>
      </binaries>
    </moduleSet>

    <moduleSet>
      <includes>
        <include>com.mycompany:module2</include>
      </includes>
      <sources>
        <fileSets>
          <fileSet>
            <directory>src/main/runscript</directory>
            <filtered>true</filtered>
            <useStrictFiltering>true</useStrictFiltering>
            <outputDirectory>bin</outputDirectory>
            <includes>
              <include>*.sh</include>
            </includes>
            <fileMode>0544</fileMode>
          </fileSet>
        </fileSets>
      </sources>
    </moduleSet>
  </moduleSets>
</assembly>
```

All one need now do to execute `bin-release` is run the `package` phase for the `my-app` project (in it's `${basedir}`).

```
mvn package
```

Does that seem like a lot of configuration? Try writing your own plugin... or doing it in Ant! :)

OS-specific distributions with Profiles

Similar to the previous task, a common usage for assemblies is to generate multiple distributions with little work. Combined with POM profiles, assemblies are suddenly composed by a new dimension.

```
<project>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptor>src/main/assembly/${assembly.osname}</descriptor>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>attached</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <profiles>
    <profile>
      <id>linux</id>
      <activation>
        <os>
          <family>linux</family>
        </os>
      </activation>
      <properties>
        <assembly.osname>linux-assembly.xml</assembly.osname>
      </properties>
    </profile>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>windows</family>
        </os>
      </activation>
      <properties>
        <assembly.osname>windows-assembly.xml</assembly.osname>
      </properties>
    </profile>
```

```
  </profiles>
</project>
```

Creating Test Jars

Although not usually condoned (we prefer you use the `jar:test-jar` goal), it is possible to use `maven-assembly-plugin` to package a project's test code usable by other projects.

```
<assembly>
  <id>test-jar</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${testOutputDirectory}</directory>
    </fileSet>
  </fileSets>
</assembly>
```

The plus-side of using assembly for creating test-jars is the increased level of control. The downside is that test-jar is a legitimate dependency `type`, whereas creating test-jars via assembly would instead be `classifiers`. In short, use `jar:test-jar`, until you have a good reason not to.

Removing SCM Metadata

It is sometimes necessary to remove content management metadata throughout a project - for example remove all `.svn` directories. This is not really a condoned usage of the assembly plugin, but a useful one-line hack:

```
mvn assembly:directory -DdescriptorId=project
```

This will copy all project files sans SCM metadata to a target directory.

Descriptors

Official Assembly Descriptor

http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html

Official Assembly Components Descriptor

http://maven.apache.org/plugins/maven-assembly-plugin/component.html

Summary

Assemblies are useful when the standard packaging constructs cannot serve your purposes. Moreover, they have a level of control over packaging external items that more specific plugins such as maven-source-plugin and maven maven-dependencies-plugin may not. Finally, assemblies can be most often useful for packaging a finished project into several different packaging formats at once for serving a wide-range of users who may prefer bz2 to zip.

CHAPTER 10

Writing Plugins

If you've heard this story before, don't stop me, because I'd like to hear it again. -- Groucho Marx

This chapter follows an example project. You may wish to download the example (http://www.sonatype.com/book/examples/book-writing-plugins.zip) and follow along.

Introduction

In the previous chapter we touched upon the subject of writing plugins as they relate to the build lifecycle, but skimmed over a lot of the details that makes the Maven plugin model so powerful.

Inversion of Control and Dependency Injection

Maven is run on the powerful Plexus inversion of control (IoC) framework. The idea of IoC is most often implemented in Java with a mechanism called dependency injection. The basic idea of IoC is that the control of creating objects is removed from the code itself and placed into the hands of an IoC framework... a framework whose onus is management of object creation. The control is now no longer in the code, but has been (you guessed it) inverted to be framework controled. The most famous example of an IoC framework in Java is Spring, but there are many implementations of this basic idea, such as Pico or Nano. The [idea] is that you can create code that is not bound to specific implementations of a class or interface, but manage the actual implementing class (as well as values to properly populate that new object) externally, often with an XML file. In the case of Plexus, components (objects managed by the Plexus IoC container) are defined with files findable in the classpath under META-INF/plexus/components.xml.

Dependency injection is the specific mechanism implementing most Java IoC solutions. There are several methods for injecting dependant values into a component object: constructor, setter, or field injections. Plexus can handle all three of them.

Constructor injection is populating an object's values through its constructor when it is created. For example, if I used java.lang.Integer as a component, a new instance would be constructed by the IoC container and populated with some value (specified by the external definition), lets say 3, as "new Integer(3)" (although actually implemented using Java reflection, you get the basic idea).

Setter injection is using the setter method of a Java bean to populate the object - but accessible as a property. This is the method used by Spring. For example if we wanted to construct a java.util.Date object and set the hours through the setHours method, your external definition would likely be something akin to `<hours>3</hours>` or hours=3, where the container would create `"Date d = new Date(); d.setHours(3);"` by appending the property name "hours" to "set" to construct the method name.

Field injection is a way to directory populate a field based upon its name. It is similar than the method above, without the name construction... the name of the field is the name of the external property. Using the java.util.Date, we can set the time in milliseconds directly to the null date (Jan 1, 1970) internal field fastTime, even though it is a private field as "Date d = new Date(); d.fastTime = 0;".

Plugin Execution and The Plugin Descriptor

Although Plexus is capable of all three dependency injection techniques, Maven only uses two types: field and setter injection. A mojo is a Maven container - with parts of the underlying implementation managed by Plexus. Maven maps execution details by the use of annotations and implementation of the Mojo interface. Maven's plugin configuration definition is `META-INF/maven/plugin.xml`. The annotations generate the configuration, and the interface provides the hook. This is how mojos attach themselves into the Mavan runtime.

If you look back to the previous chapter when displaying the lifecycle goals bound to the "maven-plugin" packaging type, the plugin:descriptor goal was bound to the generate-resources phase. This goal is in charge of generating the plugin descriptor (the plugin.xml file). This is configuration that Maven uses to manage a particular plugin and set of mojos, as well as contains some configuration information used by Maven to inject values.

```
<plugin>
  <description></description>
  <groupId>com.training.plugins</groupId>
  <artifactId>maven-zip-plugin</artifactId>
  <version>1-SNAPSHOT</version>
  <goalPrefix>zip</goalPrefix>
  <isolatedRealm>false</isolatedRealm>
  <inheritedByDefault>true</inheritedByDefault>
  <mojos>
    <mojo>
      <goal>zip</goal>
      <description>Zips up the output directory.</description>
      <requiresDirectInvocation>false</requiresDirectInvocation>
```

```
      <requiresProject>true</requiresProject>
      <requiresReports>false</requiresReports>
      <aggregator>false</aggregator>
      <requiresOnline>false</requiresOnline>
      <inheritedByDefault>true</inheritedByDefault>
      <phase>package</phase>
      <implementation>com.training.plugins.ZipMojo</implementation>
      <language>java</language>
      <instantiationStrategy>per-lookup</instantiationStrategy>
      <executionStrategy>once-per-session</executionStrategy>
      <parameters>
        <parameter>
          <name>baseDirectory</name>
          <type>java.io.File</type>
          <required>false</required>
          <editable>true</editable>
          <description>Base directory of the project.</description>
        </parameter>
        <parameter>
          <name>buildDirectory</name>
          <type>java.io.File</type>
          <required>false</required>
          <editable>true</editable>
          <description>Directory containing the build files.</description>
        </parameter>
      </parameters>
      <configuration>
       <buildDirectory implementation="java.io.File">${project.build.directory}</buildDirectory>
        <baseDirectory implementation="java.io.File">${basedir}</baseDirectory>
      </configuration>
      <requirements>
        <requirement>
          <role>org.codehaus.plexus.archiver.Archiver</role>
          <role-hint>zip</role-hint>
          <field-name>zipArchiver</field-name>
        </requirement>
      </requirements>
    </mojo>
  </mojos>
  <dependencies/>
</plugin>
```

Notice that the configuration elements are named the same as the fields in the ZipMojo class. The property values should look familiar - much like properties and filters. They will be replaced with values set by the given property's value, assuming the implementation can match.

This is a good time for a sidebar concerning more advanced properties. Properties in Maven are more than simply name=value string pairs. The values can be any sort of Java object. In the example above the given properties are java.io.File objects, but they could just as easily be java.lang.Integer (popualted by a number) or org.apache.maven.project.MavenProject (populated by `${project}`).

There were three elements in the ZipMojo plugin. The third is under the "requirement" element. This is how the plugin definition injects Plexus components into the mojo. It detects Plexus components through the "component." property. Plexus components are not set by properties like normal values, but rather required, which is good. If you are adding them, chances are you want to use them.

The Mojo

It was mentioned earlier that Maven maps execution details by the use of annotations and implementation of the Mojo interface. The annotations are used to generate the plugin.xml file above - this file tells Maven how to execute the goal properly. The mojo does the actual execution. It was no exageration to say that Maven is largely a plugin execution framework. If you think back to the build lifecycle, all of the work is done by the goals bound to the phases - the rest of the work is merely managerial, and is the same reguardless of what is actually being executed. This gives Maven an extreme flexibility as a build framework. If you wish to create a goal that emails the developer list on execution it is entirely possible (please don't do this... its very annoying, especially once you start automating your builds with some continuous integration tools).

Sticking with the zip plugin example from the previous chapter, let us take a closer look at how to use the plugin/goal/lifecycle/configuration/property dynamic to your advantage.

Java Mojos

Since Maven is a Java project, it lends to reason that plugins may only be written in Java. Though only Java bytecode may ultimately be executed by Maven, there are several ways to get that bytecode a-churnin'. Plexus is capable of loading several JVM runnable programming languages such as Java, Ant, Groovy, Jython and JRuby. But our focus will be strictly on Java and Ant based mojos.

Stepping back from the mojo of the previous chapter, let us begin again more slowly, with a toy plugin.

```
/**
 * Echos an object string to the output screen.
 * @goal echo
 */
public class EchoMojo extends AbstractMojo
{
    /**
     * Any Object to print out.
     * @parameter expression="${echo.message}" default-value="ECHO Echo echo..."
     */
    private Object message;

    public void execute()
        throws MojoExecutionException, MojoFailureException
```

```
    {
        getLog().info( message.toString() );
    }
}
```

This mojo implements a simple goal that echos the message to the logger (the Maven logger defaults to the command-line) inherited from the AbstractMojo. The logger can output in different levels through the methods: debug, info, warn, and error. You should only communicate output via this logger in a mojo, and avoid other methods such as System.out.print(...).

- @goal *goalName* - This is the only required annotation which gives a name to this goal unique to this plugin.
- @requiresDependencyResolution *requiredScope* - Flags this mojo as requiring the dependencies in the specified scope (or an implied scope) to be resolved before it can execute. Supports compile, runtime, and test.
- @requiresProject - Marks that this goal must be run inside of a project, default is true. This is opposed to plugins like archetypes, which do not.
- @requiresReports ? false
- @aggregator ? false
- @requiresOnline ? false
- @requiresDirectInvocation ? false
- @phase *phaseName* - We saw this annotation in chapter 4, where we used it to set the default phase this goal will bind to if configured as a plugin execution in the POM.
- @execute [goal=*goalName*|phase=*phaseName* [lifecycle=*lifecycleId*]] - This annotation was also touched upon in the previous chapter. This annotation notes to the Maven runtime that some alternate execution must first take place before this goal will execute. There are three main examples of this in use:

  —@execute phase="package" lifecycle="zip"

  —@execute phase="compile"

  —@execute goal="zip:zip"

If lifecycle is not specified, it defaults to "default".

Note that annotations are used solely to generate the plugin.xml descriptor. If you reeeeally want to go to all of the trouble writing the plugin.xml file by hand, you won't need these annotations, but its not recommended by sane people!

Now is a good time to run the goal in any project's basedir: "`mvn com.training.plugins:echo-maven-plugin:echo -Decho.message=$project.version`". It will print out something like:

```
[INFO] [echo:echo]
[INFO] 1.0-SNAPSHOT
```

Cool, eh? The ${project.version} property sets the ${echo.message} property, which in turn populates the message field via Maven's dependency injection mechanism. How does Maven know what field to populate, and what property to relate to it? What if no value is set at all to echo.message? These are answered by the parameter annotations.

```
/**
 * Any Object to print out.
 * @parameter
 *       expression="${echo.message}"
 *       default-value="ECHO Echo echo..."
 */
private Object message;
```

There are two methods for setting the above field. One is to set the property ${echo.message} in some way. This can be through the command line as shown above, or in the POM as a property, or in the system's settings.xml file.

```
<project>
  ...
  <properties>
    <echo.message>Print Me!</echo.message>
  </properties>
</project>
```

Whatever way implemented note that the priority is, from highest to lowest: command line (via the -D flag), profiles, system settings, POM. If none of those methods have set a property then Maven will inject the default-value into the field (but note, not the expression).

- @parameter [alias="someAlias"] [expression="$someExpression"] [default-value="value"] - Required to mark a private field as a paramter. (Note parameters may also be setter methods, but their utility is rarely necessary)
- @required - Whether this parameter is required for the Mojo to function. This is used to validate the configuration for a Mojo before it is injected, and before the Mojo is executed from some half-state. NOTE: Specification of this annotation flags the parameter as required; there is no true/false value.
- @readonly - Specifies that this parameter cannot be configured directly by the user (as in the case of POM-specified configuration). This is useful when you want to force the user to use common POM elements rather than plugin configurations, as in the case where you want to use the artifact's final name as a parameter. In this case, you want the user to modify <build><finalName/></build> rather than specifying a value for finalName directly in the plugin configuration section. It is also useful to ensure that - for example - a List-typed parameter which expects items of type Artifact doesn't get a List full of Strings. NOTE: Specification of this annotation flags the parameter as non-editable; there is no true/false value.
- @component - Indicates to lookup and populate the field with an implementation with a Plexus Component. You can enact the same behavior through the @parameter expression="${component.yourpackage.YourComponentClass}", however, @com

ponent is the prefered method. Component injection via expression may be deprecated in the future.

- @deprecated - Marks a parameter as deprecated. The rules on deprecation are the same as normal Java with language elements. This will trigger a warning when a user tries to configure a parameter marked as deprecated.

```
/**
 * Zips up the output directory.
 * @goal zip
 * @phase package
 */
public class ZipMojo extends AbstractMojo
{
    /**
     * The Zip archiver.
    * @parameter expression="${component.org.codehaus.plexus.archiver.Archiver#zip}"
     */
    private ZipArchiver zipArchiver;

    /**
     * Directory containing the build files.
     * @parameter expression="${project.build.directory}"
     */
    private File buildDirectory;

    /**
     * Base directory of the project.
     * @parameter expression="${basedir}"
     */
    private File baseDirectory;

    /**
     * A set of file patterns to include in the zip.
     * @parameter property="includes"
     */
    private String[] mIncludes;

    /**
     * A set of file patterns to exclude from the zip.
     * @parameter property="excludes"
     */
    private String[] mExcludes;

    public void setExcludes( String[] excludes ) { mExcludes = excludes; }

    public void setIncludes( String[] includes ) { mIncludes = includes; }

    public void execute()
        throws MojoExecutionException
    {
        try {
            zipArchiver.addDirectory( buildDirectory, includes, excludes );
            zipArchiver.setDestFile( new File( baseDirectory, "ouput.zip" ) );
            zipArchiver.createArchive();
```

```
            } catch( Exception e ) {
                throw new MojoExecutionException( "Could not zip", e );
            }
        }
    }
```

When you install the maven-zip-plugin containing the above goal (notice the addition of the includes and excludes fields), you can run the zip goal in the maven-zip-plugin-test project as before

```
mvn zip:zip
```

The output.zip file generated will contain the README.txt from from `src/main/resources`. However, if you add the following configuration to the POM

```
<configuration>
  <excludes>
    <exclude>**/*.txt</exclude>
  </excludes>
</configuration>
```

and rerun "mvn zip:zip" the zip file will be empty.

The corrolation should be obvious. The default behavior of a field annotated with `@parameter` is to be configurable in the POM.

Common Tools

Since Maven exists to make your life easier, it is only right that it provides tools to make writing mojos easier as well with tools. Up until now we have yet to mention Plexus, but Plexus is really the lifeblood of Maven. It is a dependency injection framework, an IoC container like Spring or Nano.

Mojos in Other Languages

BeanShell

BeanShell plugins will be easily readable to anyone who knows their Java kin. BeanShell is a clean way to write Java-like plugins without a lot of the more verbose syntax. You can use beanshell for mojo-writing with the following structure (rather than the source residing under `src/main/java`, place it under `src/main/scripts`):

```
beanshell-maven-plugin
|-- pom.xml
`-- src
    `-- main
        `-- scripts
            `-- echo.bsh
```

You then create a BeanShell script, annotated in a similar method to a Java plugin using Javadoc.

```
/**
 * @goal echo
 */
import org.apache.maven.plugin.Mojo;
import org.apache.maven.script.beanshell.BeanshellMojoAdapter;

execute()
{
    logger.info( "Echo: " + message );
}

/**
 * Outputs the message.toString()
 *
 * @parameter expression="${message}" type="java.lang.Object" default-value="ECHO Echo echo..."
 */
setMessage( msg )
{
    message = msg;
}

return new BeanshellMojoAdapter( (Mojo) this, this.interpreter );
```

Since BeanShell is built into the Maven core, installing and running this goal is similar to any Java plugin.

```
mvn install
mvn com.mycompany:beanshell-maven-plugin:echo -Dmessage=Hi!
```

You can learn more about BeanShell from the website: *http://www.beanshell.org/*

Ant

Unlike BeanShell, Ant plugins are not built into the core of Maven, although it is easy to use them. Before creating an Ant plugin, let's take a quick dive into two important areas of the Maven plugin framework: descriptor extraction and component runtime.

As mentioned above, plugin descriptors describe to the Maven runtime how to execute a mojo in a plugin. Just because you create a project of type "maven-plugin" that contains some Java files named something akin to EchoMojo.java - does not mean that Maven knows what to do with it. The plugin descriptor (`META-INF/maven/plugin.xml` file) describes runtime details such as:

1. What goal named `X` maps to mojo `Y` (eg. "echo" goal maps to "EchoMojo.class")
2. What fields are injected with what values (eg. let `$message` populate the `private Object message;` field)
3. and more...

"Descriptor extraction" is what happens when Maven (the `maven-plugin-plugin`) scans through your plugin project and tries to build a plugin descripor based off of available data. This can take many forms. In the case of Java and BeanShell, it is a Javadoc-like markup. In the case of Ant we will see it's own XML markup. Descriptor markup will always be in a form comfortable to the Mojo language - for example JRuby uses RDoc as it's descriptor form.

The other important area of mojo execution in the Maven plugin framework is the component runtime. In cases like Java based Mojos execution is no problem: just run the execute method in the Mojo object. In cases like BeanShell this is slightly different, since the script must be adapted to run as a normal Java Mojo (hence, the `Beanshell MojoAdapter`). Luckly, it is built in.

The reason we mention these two items now is because for non-built-in language extensions, they must be injected into the Maven runtime. Luckly for us, this is just a slight change to the plugin's POM. Your plugin consumers will never know the difference.

Back to writing an Ant plugin. Let us start with the POM:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.training.plugins</groupId>
  <artifactId>maven-antbased-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>
  <name>Ant-Based Plugin</name>

  <dependencies>
    <!--  -->
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-script-ant</artifactId>
      <version>2.0.6</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>

        <!-- Add the Ant plugin tools to the plugin -->
        <dependencies>
          <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-tools-ant</artifactId>
            <version>2.0.5</version>
          </dependency>
        </dependencies>

        <configuration>
          <goalPrefix>antbased</goalPrefix>
```

```
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

This makes the `maven-script-ant` artifact available at runtime (so it can execute the ant script goal for you). It also adds the `maven-plugin-tools-ant` as a dependency of the `maven-plugin-plugin`, so the `plugin` plugin can create the necessary plugin descriptor - the thing that makes a plugin a plugin.

With all that out of the way, let us get back to writing plugins. Unlike other mojo language extensions, Ant plugins do not use markup within the `build.xml` (Ant's version of a set of Mojos), but rather an external `mojos.xml` file. This was done to simplify turning existing Ant build.xml files into full-fledged Maven mojos without much internal alteration. The directory structure of our new `antbased-maven-plugin` will be as such:

```
antbased-maven-plugin
|-- pom.xml
`-- src
    `-- main
        `-- scripts
            |-- echo.build.xml
            `-- echo.mojos.xml
```

The file's prefix does can be any name, as long as they match. Since the `*.mojos.xml` file stores the data extracted to create a plugin descriptor, it must exist alongside a *.build.xml file, or else the `maven-plugin-plugin` will pass it by. As you might have guessed, the `echo.build.xml` is just a normal Ant build.xml, and contains the logic of our mojo(s). `echo.build.xml` contains:

```
<project>
  <target name="echotarget">
    <echo>${message}</echo>
  </target>
</project>
```

There the `echo.mojos.xml` file contains the following:

```
<pluginMetadata>
  <mojos>
    <mojo>
      <goal>echo</goal>
      <call>echotarget</call>
      <description>Echos a Message</description>

      <parameters>
        <parameter>
          <name>message</name>
          <property>message</property>
          <required>false</required>
          <expression>${message}</expression>
          <type>java.lang.Object</type>
```

```
          <description>Outputs the messag</description>
        </parameter>
        </parameter>
      </parameters>
    </mojo>
  </mojos>
</pluginMetadata>
```

The build file is fairly strightforward - it echoes the value of the `${message}` property. The corrosponding mojos.xml gives Maven some information about how to interact with the build file. `goal` is the name of the goal, in our case `echo`. `call` specifies the name of the target to execute when the goal is called. The name of the target is `echotarget>>. Finally, the <<<parameters` element contains a list of properties that Maven should populate for Ant prior to executing the build script. Note the similarity of values to our previous "echo" incantations.

You can find more about Ant-based plugins at the Maven homepage: *http://maven.apache.org/guides/plugin/guide-ant-plugin-development.html*

JRuby

The JRuby plugin exists under the Codehaus "Mojo" project repository. Since the newest versions are not necessarily deployed to central repository at any given time if you wish to utilize the latest-and-greatest you must point Maven at the remote repository locations by adding the following to your pom.xml (*pom-details.html*) (or settings.xml (*settings-details.html*)) file. Learn more about this in their respective Appendices.

```
  <repositories>
    <repository>
      <id>codehaus.org</id>
      <url>http://repository.codehaus.org/</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>codehaus.org</id>
      <url>http://repository.codehaus.org/</url>
    </pluginRepository>
  </pluginRepositories>
</project>
```

JRuby is yet another plugin language - but is covered here for a simple purpose: it is trivial to manipulate files in Ruby, and that is a large component of build systems, making it a convenient mechanism for writing Maven plugins.

Since JRuby can reflect on the JVM runtime, you can invoke and access Java class instances available in the classpath through JRuby plugins. You can find details at the JRuby site: *http://jruby.codehaus.org/*.

To create a JRuby plugin, first create a simple project. Rather than the source residing under `src/main/java`, place it under `src/main/scripts`.

```
jrubybased-maven-plugin
|-- pom.xml
`-- src
    `-- main
        `-- scripts
            `-- my_mojo.rb
```

The Mojo class must extend Mojo (which you then extend "execute", or the mojo won't run). At the end of the file add "run_mojo" followed by the mojo's class name (not an instance of the class).

```
# This is a mojo description
# @goal "my-mojo"
# @phase "validate"
# @requiresDependencyResolution "compile"
class MyMojo < Mojo

  # @parameter type="java.lang.String" default-value="nothing" alias="a_string"
  def prop;;end

  # @parameter type="org.apache.maven.project.MavenProject" expression="${project}"
  # @required true
  def project;;end

  def execute
    info "The following String was passed to prop: '#{$prop}'"
    info "My project artifact is: #{$project.artifactId}"
  end
end

run_mojo MyMojo
```

There are some important values above to note:

1. The annotations are similar to the Java versions here.
2. type must be provided as the fully-qualified name of the Java class, for example "org.apache.maven.project.MavenProject". The following are comma-seperated.
3. Parameter values are accessed as global variables of the given name - i.e., the parameter named project can be used as `${project}`. Java objects can be used as normal JRuby objects.
4. You can output using the normal ruby methods (`puts`, `print`) or use the corrosponding mojo methods (`info`, `error`).

Finally, the POM resembles the following:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany</groupId>
```

```
  <artifactId>jrubybased-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>

  <dependencies>
    <dependency>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jruby-maven-plugin</artifactId>
      <version>1.0-beta-4</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.1</version>
        <dependencies>
          <dependency>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>jruby-maven-plugin</artifactId>
            <version>1.0-beta-4</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

You can find more about the `jruby-maven-plugin` at it's homepage: *http://mojo.codehaus.org/jruby-maven-plugin/*.

Summary

The ability to *use* and configure plugins is a powerful tool in the Maven world - but can only take you so far. If you ever require an action that the community has not yet considered, or with to add your own features to the ever-growing Maven toolset, the ability to *write* plugins is a necessity. Luckly for all of us, it is not a difficult feat to achive - made simpler still by Maven's support of alternate programming languages - for those uncomfortable with Java.

CHAPTER 11
Reporting

This chapter follows an example project. You may wish to download the reporting examples (http://www.sonatype.com/book/examples/book-reporting.zip) and follow along.

Introduction

As we saw in the **Site Generation** chapter, Maven can do much more than simply build and manage your project binaries; it can also help you manage your project's web presence. While it's critical to provide documentation to your community in the form of how-to's, feature lists, and other user-friendly formats, it's no less important to provide the community with some mechanism for assessing the health of the project itself. These mechanisms may include static code-analysis results, statistics on tests run against the project source code (unit, integration, and other types), along with any number of other reports. While it is possible to provide these reports to your community manually, this is an extremely time-consuming and tedious process that involves manually compiling statistics from the various output created by the build process, such as unit-test reports. These reports are critical for the user community, but they represent an unreasonable burden on developers; at least, as long as they are maintained manually.

Much like the build process itself, the process of creating project reports cries out for automation. It is precisely this sort of automation that Maven can provide.

A key over-arching goal of Maven is to make it easy for developers to do the right thing for their project and community. To that end, Maven provides a wealth of options for rendering these project-status reports into XHTML during the website-generation process, thereby automating an otherwise arduous and difficult process.

To make the best use of Maven for project reporting, you'll need to understand two key configurations:

1. configuration for Maven's project-info reports (in the `maven-project-info-reports-plugin` plugin)

2. configuration of other, additional project reports

We'll begin by exploring the configuration necessary to flesh out the project-info reports that are run by default when Maven generates a project website. Then, we'll discuss the basics of configuring additional project reports. Finally, we'll take a look at some of the most common and useful reports available at the Apache Maven and Codehaus Mojo projects, among other places.

A Quick Refresher: Building and Viewing a Project Website

Before we get into the meat of this chapter, it's important to review the steps necessary to create a simple example project, then build and view its generated website. If you already know this part (we covered this in the **Site Generation** chapter), feel free to skip ahead.

Bootstrap Revisited: Hello-World

Before we can configure project reports, we'll need a project. For this, we'll turn back to the handy hello-world example used in the **Site Generation** chapter. If you haven't read this chapter, or didn't follow along by creating your own hello-world project, you can get started with a quick Maven command:

```
workdir$ mvn archetype:create -DgroupId=book.reporting -DartifactId=hello-world
```

NOTE: We're not creating the hello-world project from the `maven-archetype-site` or `maven-archetype-site-simple` archetypes. Although this would give us a good start at a project website, we really need some source code from which to generate things like unit-test reports. Maven will generate a basic website for any project, so we'll use the default archetype instead.

Build it, View it

To generate the project website for our hello-world project, simply issue the following Maven command:

```
workdir$ cd hello-world
hello-world$ mvn clean site:run
```

Initially, the `clean` step won't do anything. However, cleaning a project before you issue a build (or site) command is a good habit to get into, to avoid unpredictable results. Once you run the command above, you should be able to load the project website into your browser by going to `http://localhost:8080`. To stop the server, you'll need to press `CTL-C` in the console window. In the text below, when I say "regenerate and refresh" this `site:run` command -- coupled with a refresh of the browser window -- is what I'm referring to.

Also, note that in some cases it may not be possible to preview the project website using the `site:run` mojo, due to a bug in its implementation. This is a known problem, described in the JIRA issue MSITE-220 (*http://jira.codehaus.org/browse/MSITE-220*), and it leaves us with little alternative but to generate the site and load the HTML page straight off the filesystem. In these cases, you will need to direct your browser to open the corresponding path within the `target/site/` sub-directory, under your `hello-world` project directory. I will give a clear indication when this is necessary.

NOTE: This bug only affects the `site:run` mojo, not website generation in general. Rest assured that when you deploy your project's website, all configured reports will be included.

Configuring Project-Info Reports

By default, Maven generats a project website that contains several links, all under the heading of Project Information. However, if we don't add the right information to your project's POM, these pages will remain basically empty. These reports are the defaults because the Maven development team believes they are critical for giving relevant feedback about a project, and thus sustaining a healthy community around that project. So, to avoid offering false hope to users by way of these links, we should make sure they're up and running. For simplicity, we'll explore the project-info reports in the order that they appear in the default project website.

Now, it's time to start customizing our example a little bit. The first thing any self-respecting project website needs is a declaration of the project's name and what it does, in simple terms (we'll leave the details for later). When the project website doesn't supply its own, custom index page, this role is filled by the `About` report. We'll start by configuring this report, then move on to provide some other standard information that every project should supply for its community.

About (Hello-World)

We can configure the `About` report to provide the user with some basic information about our hello-world project by setting the `name` and `description` elements in the POM, as follows:

```
<project>
  ...
  <name>Hello, World</name>
  ...
  <description>
   This project is a sample to help explain how project-website generation can be handled by Maven.
  </description>
  ...
</project>
```

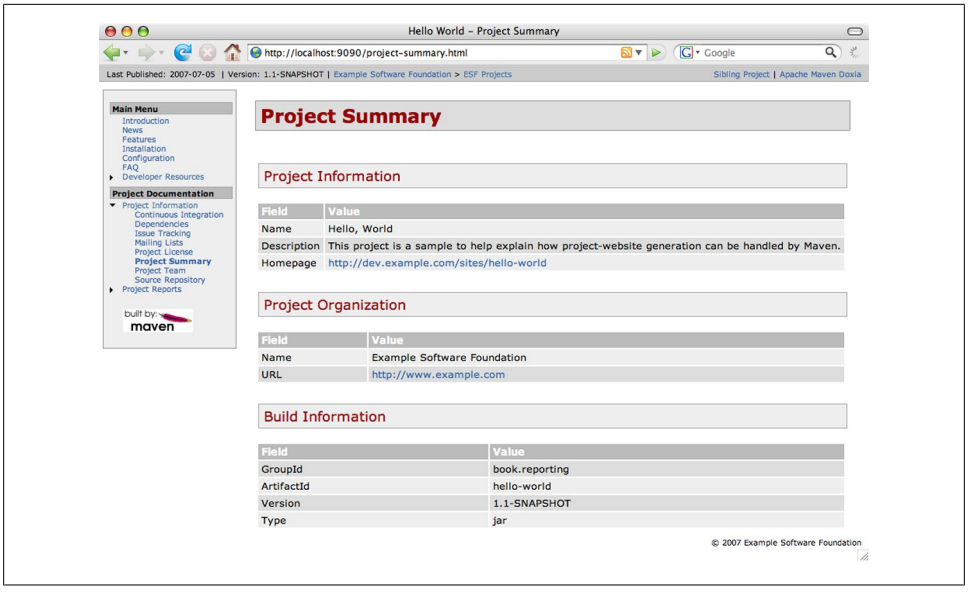


Figure 11-1. Regenerate and refresh the project website and you will see this

GOTCHA! It seems that Maven's `project-info-reports` plugin currently doesn't handle HTML embedded in the POM `description` element very elegantly. In fact, putting HTML into the description haphazardly will cause the build to fail, with Maven complaining that it cannot parse the POM. If you take a more wily approach, and wrap the description content in a `CDATA` section, it will simply escape the HTML, and pass it on through...resulting in HTML markup that's visible on your project's front page. If you find yourself in this predicament, consider writing your own front page for the project, as described in the **Site Generation** chapter.

Now that we have a simple project description composed, we can direct attention to the other project reports (the project description is actually part of the *About* report). As you scan through the other project reports listed in the site navigation, it should become apparent just how little project information is presented here. We'll fix that next by adding some more information to the POM.

Continuous Integration

Your projects are all built in a continuous integration (CI) environment, so you can know immediately when someone breaks the codebase...right?

By supplying the information for the project's CI environment, we give our developers (and users) an easy way to view up-to-date CI results. The key part of the POM for this is the `ciManagement` section. At a minimum, we should supply the type of our CI system and its URL, so users can find it:

```
<project>
  ...
  <ciManagement>
    <system>continuum</system>
    <url>http://dev.example.com/continuum</url>
  </ciManagement>
  ...
</project>
```

This time, when you regenerate the project website and click on the Continuous Integration link, you should see two key items:

1. The project uses Continuum as its CI system.
2. The CI system for this project can be found at http://dev.example.com/continuum (and there's a link to this address on the page).

So now, users and developers will be able to go to our project website and follow a link from there to the continuous integration system that has up-to-date build results for our project. But...is there more we could do? For example, how does our CI server let developers know of build failures? If the project gets a new developer, how would this developer subscribe for CI notifications? Clearly, we need to provide a little more information in the `ciManagement` section:

```
<project>
  ...
  <ciManagement>
    ...
    <notifiers>
      <notifier>
        <type>mail</type>
        <configuration>
          <address>hello-world-notifications@dev.example.com</address>
        </configuration>
      </notifier>
    </notifiers>
  </ciManagement>
  ...
</project>
```

When you regenerate and refresh, you should see the notifications email address show up under the Notifiers section, giving users the final piece of information necessary to track CI builds for our project.

Dependencies

Most projects have dependencies on other libraries or applications that are developed externally. The `Dependencies` report simply gives our community a view into what libraries the project depends on, and for what. As you know, Maven has five dependency scopes - `compile`, `test`, `runtime`, `provided`, and the strange (and hopefully, soon-to-be deprecated) `system`. This report will calculate the complete dependency set (including

dependencies of our dependencies, and so on) for each of these scopes, and maps out where in the dependency hierarchy each one was found.

For our hello-world project, this report is quite simplistic, containing only the single `junit` dependency. The JUnit jar has no dependencies of its own, and JUnit itself is only a test-scope dependency for our project - which means it won't be needed by users declaring a dependency on our project. This means our dependency report is pretty short and uninteresting. To illustrate the power of the `Dependencies` report, let's add another dependency to the POM:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-project</artifactId>
      <version>2.0.4</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Regenerate the site, refresh your browser, and take another look. This time, you should see a fair bit more information on the page, including a new direct compile-scope dependency on `maven-project` at the top of the page, about ten dependencies which were included transitively by this new dependency under "Project Transitive Dependencies", and a much larger dependency graph just below that, indicating how the transitive dependencies are related to our direct dependency set. At the bottom, you'll find specific information about each dependency, including the project's name, description, and website URL (if one was provided).

This information can help users track down potential conflicts with their own project dependencies, not to mention giving valuable information about which libraries are needed for this project to function.

Issue Tracking

All software has issues, ranging from functional bugs to tasks and ideas for new features. Likewise, all projects should have an issue tracker, so community members can file and track those work items they feel need to be addressed.

However, it's not enough to simply have an issue-tracking system; your project also needs to publicize its existence, so new community members can find it more easily. Obviously, the project website is an ideal place for this sort of information - which brings us to the `Issue Tracking` report. Like the name suggests, this report simply publishes the information needed to access your project's issue-tracking system.

Returning once again to our hello-world example, we can see that this project doesn't publish any information about its issue tracker. We can remedy this by adding a small section to the POM, which simply lists the issue-tracking system type and its web URL:

```
<project>
  ...
  <issueManagement>
    <system>bugzilla</system>
    <url>http://dev.example.com/issues/hello-world</url>
  </issueManagement>
  ...
</project>
```

After adding this and regenerating the site, you should see a page with two links when you click on the `Issue Tracking` link under `Project Information`: The first one will link to the Bugzilla project website, and the other should point to our own project's issue tracker at http://dev.example.com/issues/hello-world. This should give users enough information to find the issue-tracking system used by our project, along with more information about the system itself (courtesy of that application's project website).

Mailing Lists

Mailing lists are one of the most ubiquitous modes of collaboration in the open source world. They provide a persistent, open communication channel that fosters a high level of participation. They also run on infrastructure that is well-understood and well-supported by a diverse array of applications, on all platforms. As such, they form the backbone of most collaborative development efforts.

There are five basic elements of information that a user needs to know about each of your project's mailing lists (projects often have an array of mailing lists, to host separate types of discussion):

- the name of the mailing list, to help when referring to it
- the email address to use when subscribing to the list
- the email address to use when unsubscribing from the list
- the email address to use when posting messages to the list
- at least one list-archive URL, for historical reference

The first is obvious; having a name for each mailing list makes referring to those lists in an unambiguous way simpler. The second and third elements (subscribe/unsubscribe addresses) are necessary to account for the variability between mailing-list management applications, which all handle such requests differently. Next, users will obviously need to know the address to which they should send new messages bound for the list. And finally, users need the ability to search a permanent, historical record of each mailing list. This will help them research different aspects of the project's behavior, and understand the historical context of discussions that are taking place on the list at the present moment.

So, in order to publish a reference to this critical collaboration infrastructure on our hello-world project's website, we need to configure the `Mailing Lists` report. Assuming our project has a somewhat minimal set of mailing lists that handle user, development, and system-notification traffic, we can add the following section to the POM to configure this report:

```
<project>
  ...
  <mailingLists>
    <mailingList>
      <name>Hello-World Users</name>
      <post>hello-world-users@dev.example.com</post>
      <subscribe>hello-world-users-subscribe@dev.example.com</subscribe>
      <unsubscribe>hello-world-users-unsubscribe@dev.example.com</unsubscribe>
      <archive>http://dev.example.com/mail-archives/hello-world/users</archive>
      <otherArchives>
       <otherArchive>http://www.nabble.com/Hello---World---Users-f000.html</otherArchive>
      </otherArchives>
    </mailingList>
    <mailingList>
      <name>Hello-World Developers</name>
      <post>hello-world-dev@dev.example.com</post>
      <subscribe>hello-world-dev-subscribe@dev.example.com</subscribe>
      <unsubscribe>hello-world-dev-unsubscribe@dev.example.com</unsubscribe>
     <archive>http://dev.example.com/mail-archives/hello-world/developers</archive>
      <otherArchives>
       <otherArchive>http://www.nabble.com/Hello---World---Dev-f000.html</otherArchive>
      </otherArchives>
    </mailingList>
    <mailingList>
      <name>Hello-World Notifications</name>
      <post>hello-world-notifications@dev.example.com</post>
      <subscribe>hello-world-notifications-subscribe@dev.example.com</subscribe>
     <unsubscribe>hello-world-notifications-unsubscribe@dev.example.com</unsubscribe>
    </mailingList>
  </mailingLists>
  ...
</project>
```

When you regenerate the site and refresh the `Mailing Lists` report, you'll notice that all three of our mailing lists are present and accounted for. You may also notice that the notifications list doesn't have archives of any sort; this is by design, since notifications are trantient by nature. Users hitting this page will now have all the information they need to join the community in a very real and vocal way.

Project License

Perhaps one of the most important pieces of information to be published about your project is its license. This document specifies whether and how others are authorized to make use of your project. In the open-source world, the project license can determine whether two libraries are able to be used together or not, since some open-source li-

censes are incompatible with one another. In the commercial world, the license document has legal ramifications, not only because it specifies what users are allowed to do with the project, but also since it defines what sort of warranty or guarantees your company is placing on this project and its resulting binary.

Needless to say, no project website can be complete without a clear indication of the project's licensing. To that end, we should configure the `Project License` report in our own hello-world project. For simplicity, let's assume this is an open-source project. Because we're also sympathetic to the business users of the project, let's make use of the Apache Public License (version 2.0) to define our terms:

```
<project>
  ...
  <licenses>
    <license>
      <name>The Apache Software License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
  ...
</project>
```

Most of the elements in the above section have obvious meaning; however, the `distribution` element bears some explanation. This element specifies what sort of distribution the stated license allows. The two supported types are currently:

- `repo` - meaning the project binary can be distributed via a Maven repository
- `manual` - meaning the project binary must be downloaded manually and installed before it can be used.

The `manual` mode is usually indicated for projects that have a click-through license agreement on their website, or else simply want to maintain strict control over the integrity of the binaries they publish (much the way the Eclipse Foundation, with its Eclipse Public License - or EPL - does).

Returning once more to our own project, if you regenerate the site and refresh the `Project Licenses` report, you'll see on prominent display the title of the license and its full text. This information will provide the user with very clear licensing information for our project.

Project Summary

It may seem obvious, but every project should have a very clear identity. This not only means having an unambiguous identity, but also stating that identity clearly on the project website. An identity may take the form of a simple project name, used to locate the project on a site like SourceForge or Freshmeat; or, it may explain the exact coordinates of that project's binary and other artifacts in a Maven repository. Whatever the project's internal culture, it's important to provide the simplest route to consumption

for users. This means being proactive and telling them where to find the project under various organizational systems.

The `Project Summary` report provides this identity - along with other basic information about the project and its organization - to the user. Of particular interest here is the Maven repository coordinates for the project, which allow users to refer to the project as a dependency within their own POMs.

Our hello-world project already provides some of the information necessary to complete this report: we have the artifact identity in the Maven repository, along with the project's name and description. By adding two small sections to the POM, we can give our users a complete picture of who develops this project, and where to find it in the Maven repository:

```
<project>
  ...
  <url>http://dev.example.com/sites/hello-world</url>

  <organization>
    <name>Example Software Foundation</name>
    <url>http://www.example.com</url>
  </organization>
  ...
</project>
```

Regenerate and refresh once again, and you should now see a section containing our organization's name and main website URL, along with the URL pointing to the hello-world project's main page.

TIP: Usually teams developing standalone applications will want to separate the main project website from the one containing these project reports, which tend to be most useful to developers and integrators. In these cases, the project `url` above (not to be confused with the organization url) should probably point at the main website, with this developer-specific site content hosted as a subsection.

Project Team

The people who work so hard to make your project successful deserve recognition; particularly when they take the time to contribute improvements from the outside, without commit access to the code. These contributors are critical to the health of a project, and they're also some of the easiest people to overlook. The project's website should make a special effort to recognize those who keep the lights on, and keep things humming along.

Aside from giving credit, providing a list of developers on the project website has its practical benefits. With a developer list, users can identify the people in the community who are most likely to have the answers to tough questions, and organizations know who to ask for tips on finding commercial support the project. Perhaps more impor-

tantly, these are the people you should turn to if you want to get involved in the project, and need some guidance.

Fortunately, Maven's `Project Team` report can fill this need. Let's flesh out this report for the hello-world project, using some imaginary developers/contributors. Configuring this report is a simple process of adding two new sections to the POM, like this:

```
<project>
  ...
  <developers>
    <developer>
      <id>jsmith</id>
      <name>Joan Smith</name>
      <email>jsmith@example.com</email>
      <url>http://blogs.example.com/jsmith</url>
      <organization>ESF</organization>
      <organizationUrl>http://www.example.com</organizationUrl>
      <roles>
        <role>Architect</role>
        <role>Developer</role>
      </roles>
      <timezone>-5</timezone>
      <properties>
        <gpg-key>jsmith@personal.com</gpg-key>
      </properties>
    </developer>
    <developer>
      <id>pbunyan</id>
      <name>Paul Bunyan</name>
      <email>pbunyan@elsewhere.com</email>
      <url>http://www.blogs.com/paul</url>
      <organization>Elsewhere, Inc.</organization>
      <organizationUrl>http://www.elsewhere.com</organizationUrl>
      <roles>
        <role>Developer</role>
      </roles>
      <timezone>-8</timezone>
      <properties>
        <gpg-key>pbunyan@elsewhere.com</gpg-key>
      </properties>
    </developer>
  </developers>

  <contributors>
    <contributor>
      <name>George W. Newman</name>
      <email>gwnewman@example.com</email>
      <roles>
        <role>Mascot</role>
      </roles>
      <timezone>-5</timezone>
    </contributor>
    <contributor>
      <name>Andrea Penn</name>
      <email>apenn@writers.org</email>
```

```
      <url>http://www.blogs.com/apenn</url>
      <organization>Writers Foundation</organization>
      <organizationUrl>http://www.writers.org</organizationUrl>
      <roles>
        <role>Documentation</role>
      </roles>
      <timezone>0</timezone>
    </contributor>
  </contributors>
  ...
</project>
```

When you refresh, you should see that the community members who are active in our project are listed in one of two sections in the `Project Team` report, along with any identity or contact information they have supplied. From looking at this page, it should be fairly clear who you might want to ask about the design of our project (Ms. Smith), or about a typo you found in the documentation (Ms. Penn, though she can't actually commit the corrections).

Source Repository

The final step in laying a good foundation for a project website is particularly critical for open-source projects. If your project is developed under an open-source license, you absolutely have to give people access to your source code. This enables outsiders to review your code, and suggest improvement. It also gives them an opportunity to use the project's source code to aid in debugging. If your project is commercial or operates under some other brand of closed-source licensing, it's still a good idea to provide information for accessing the project sources; in the closed-source case, it just becomes a question of where you publish this project website (which is, almost by definition, a developer-oriented site). Maven's `Source Repository` report gives you the ability to provide this sort of documentation.

Since we've chosen an open-source license for our example hello-world project, it is imperative that we document the location of our project's source repository. Like all the other reports we've covered here, this just involves a small addition to the POM:

```
<project>
  ...
  <scm>
   <developerConnection>scm:svn:https://dev.example.com/svn/repo/hello-world/trunk</developerConnection>
    <connection>scm:svn:http://dev.example.com/scm/hello-world/trunk</connection>
    <url>http://dev.example.com/svn/viewvc/hello-world/trunk</url>
  </scm>
  ...
</project>
```

When you refresh this report, you should see three important pieces of information:

1. The type of source-control system in use, including a link to instructions for using that system, if available.

2. A link to the web-enabled view on our project's SCM system (linked to the URL given in the `url` element above).
3. Instructions for retrieving a local copy of our project's source code, using the appropriate source-control client (such as `svn` for Subversion). These instructions will include details on accessing the source as both a developer with commit rights, and as a user with read-only access. Additionally, they may provide techniques you should use if you're behind a firewall, or using a web proxy.

TIP: The formatting of URLs within the `scm` section of the POM is not the same throughout. The `developerConnection` and `connection` elements actually have URLs with multiple nested protocols, such as `scm:svn:https`, while the `url` element is meant to be a simple `http` or `https` URL. The extra protocol information is used by Maven's Source Control APIs to determine what type of system our project uses. In the case of this report, it also tells Maven what type of instructions to generate in order to guide the user to the project source code effectively. For more information on the URL format for source repositories, see the *URL Format* page on the Maven website, at *http://maven.apache.org/scm/scm-url-format.html* page. For more information on which SCM systems are supported by Maven, see the *Supported SCMs* page, at *http://maven.apache.org/scm/scms-overview.html*.

With this report configured, we've given our community a great touchstone for reference information on the hello-world project. These reports will enable easy participation in the hello-world community, and should give users the tools they need to get involved in the project more deeply. The next step is to publish the documentation specific to our project, which may include installation and usage instructions, design and architectural documentation, or even screenshots and other introductory material.

Configuring Additional Reports

Looking back at the website we've created so far, it's still lacking some important features. Modern project websites - particularly those provided by open-source projects - absolutely must provide some statistics about the project's source code. For example, an open-source project won't get much attention without a decent test suite; to reassure prospective users, wouldn't it be a good idea to publish the test-result and coverage statistics? Also, when we're talking about Java projects, how many successful open-source projects *don't* publish their JavaDoc API documentation to their website?

We now know from experience that the project-info reports don't provide any of this information. How, then, do we supply this information to the user? As I mentioned above, Maven provides a rich set of project reports that can help you publish everything from basic project information -- which we've pretty well covered in the preceding section -- to API documentation, test statistics, static code analysis, and much more. Since the project-info reports only provide a small fraction of this capability, we must

configure some additional reports. But before we do, it would be best to cover the basics of configuring these reports.

Generally speaking, report configuration in Maven is done in a very similar way to plugin configuration. Instead of the `build/plugins` section of the POM, report configuration happens in the `reporting/plugins` section. This section uses nearly the same `plugin` syntax as the `build/plugins` section, except for two important differences:

1. Report plugins specified here do not need to be explicitly bound to a lifecycle phase
2. Instead of `execution` entries in an `executions` subsection, the `reporting` section uses `report-sets` and `report-set` to separate invocations and associated configuration from one another.

One other thing to take note of is that additional project reports (i.e. non-project-info reports) will appear under a new menu called `Project Reports` in the generated website by default.

Now that we have a basic idea of *how* to configure project reports in the POM, let's explore some of the reports that are available.

A Quick Note on Additional Reports in the Project Website

When configuring the reports of the `maven-project-info-reports-plugin`, we noticed that each new report showed up in the `Project Information` menu of our generated website. In this way, the project-info reports are unique; all other reports appear under the `Project Reports` menu by default. So, as we talk about the additional reports you can configure for your projects, be sure to note the change and look for updates in the right spot. And don't worry; I'll remind you!

Other Reports Available from the Maven Project (at ASF)

Aside from the `maven-project-info-reports-plugin`, the Apache Maven project maintains several additional reports that can provide considerable value to a project's website. Each of these reports shares some common naming elements with the project-info-reports plugin, namely the following:

- **GroupId:** *org.apache.maven.plugins*
- **ArtifactId:** *maven-<something>-plugin*

Let's take a look at some of the things that these reports can provide for your website.

`maven-surefire-report-plugin`

Your project has unit tests, right? Nevermind, I don't want to know...

If you are running unit tests as part of your build, wouldn't it be nice to publish those results to the project website, to keep all of your fellow developers (and users) in the loop? Using the Surefire report, visitors to your website can see how many tests are succeeding, and how fast they're running.

NOTE: For those of you who missed it, Surefire is the name of Maven's unit-testing framework, used to execute tests written with JUnit, TestNG, and other testing APIs. Correspondingly, the `surefire-report` plugin contains the report that formats unit test results for the project website.

To illustrate the usage of the Surefire report, let's put our hello-world project's test results on its website. To add the Surefire report, we add the plugin's identity to the `reporting` section of the POM, like this:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

When you regenerate the website, you should see the `Maven Surefire Report` link appear under the `Project Reports` menu. If you click on this link, you will see a page that with unit-test statistics that allow you to see results by summary, package, and individual class.

maven-javadoc-plugin

Besides unit tests, all projects should have some form of documentation within the code itself. In Java, this usually takes the form of JavaDocs, which has the nice side effect of allowing that documentation to be extracted and published to a website. But, I'm not telling you anything new; you've been publishing your JavaDocs for years, right?

To publish the JavaDocs for our hello-world project, we can start by adding a reference to the `javadoc` plugin to our POM's `reporting` section, as follows:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
```

```
  ...
</project>
```

NOTE: Currently, this report does not work with the `site:run` mojo. To view the output of this report, use the file-based previewing strategy discussed at the beginning of this chapter. The file you're looking for is `target/site/apidocs/index.html` in your hello-world project directory.

Since our source code uses classes from external libraries (admittedly, only `java.lang.String` so far in this example), we should provide links to any external JavaDocs that we need. Configuring these links is simple:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>

        <configuration>
          <links>
            <link>http://java.sun.com/j2se/1.5.0/docs/api</link>
          </links>
        </configuration>

      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

This time, when we regenerate the site, we should see that the `java.lang.String` reference in App.java is hyperlinked to the Sun JDK JavaDocs.

maven-changelog-plugin

The changelog plugin reads your project's SCM, and produces three different types of reports from the information it finds there:

- The `changelog` report renders a page called `Change Log` which lists the SCM commits that have taken place in the recent past, providing the developer, a list of files changed, and the commit message for each.
- The `dev-activity` report renders a page called `Developer Activity` which provides summary statistics, organized by developer, which include the number of commits and number of files changed on your project in the recent past. Results are listed in alphabetical order.
- The `file-activity` report is similar to the `dev-activity` report, except it organizes its statistics by file name. It produces a page called `File Activity` with statistics

that show how many times each file was changed in the recent past. Results are listed in descending order, according to the number of changes.

To enable these reports, you have to have a few basic pieces of information in your project's POM. The most important -- for obvious reasons -- is the `scm` section. This gives the changelog reports the connection information it needs to retrieve the SCM logs. If you want to use the `dev-activity` report, you'll also need to provide a `developers` section, since the report pulls the list of available developer names from that section. If a developer is not in the `developers` list, her recent contributions will not show up on the `dev-activity` report. For more information on configuring these sections of the POM, see the `Project Team` and `Source Repository` configuration summaries, in the project-info section above.

Once you've configured those two sections, you'll also have to add the `maven-changelog-plugin` to your `reporting` section, like this:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-changelog-plugin</artifactId>
      </plugin>
      ...
    </plugins>
  </reporting>
  ...
</project>
```

Regenerate and refresh, and you'll see these new reports pop up under the `Project Reports` menu. If you've configured your `scm` and `developers` sections correctly, you should notice that the `dev-activity` report links to the `Project Team` report from the project-info plugin. Likewise, each file mentioned in these three reports is linked back to the SCM system, so you can browse its contents.

Sample Configuration

If you don't have a project in SCM to play with yet, you can try modifying your hello-world project with the following information, to see a sample report:

```
<project>
  ...
  <developers>
    <developer>
      <id>jdcasey</id>
      <name>John</name>
    </developer>
    <developer>
      <id>jvanzyl</id>
      <name>Jason</name>
```

```
    </developer>
  </developers>

  ...

  <scm>
   <developerConnection>scm:svn:https://svn.apache.org/repos/asf/maven/components/trunk</developerConne
   <connection>scm:svn:http://svn.apache.org/repos/asf/maven/components/trunk</connection>
    <url>http://svn.apache.org/viewvc/maven/components/trunk</url>
  </scm>
  ...
</project>
```

Customizing Your File Links

Sometimes, you may wish to have the file links in these reports point to a web-based SCM tool, like ViewVC or Fisheye. To enable this, the changelog plugin provides the following configuration point (the example uses the ViewVC system at svn.apache.org, for use with the tip above):

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-changelog-plugin</artifactId>
        <configuration>
        <displayFileDetailUrl>http://svn.apache.org/viewvc%FILE%?view=markup</displayFileDetailUrl>
        </configuration>
      </plugin>
      ...
    </plugins>
  </reporting>
  ...
</project>
```

Unfortunately, when you make this customization, the `changelog` report no longer provides links to the specific revisions in each entry, only the latest revision.

maven-changes-plugin

In contrast to the `maven-changelog-plugin`, the changes plugin extracts a list of changes in the project from either its issue-tracking system or an XML-formatted changelog file maintained in the project's directory structure. They may seem redundant at first blush, but these two reports are actually complementary, each filling in a little more information about the project's current release or snapshot (depending on what the published website is based on).

This plugin offers two different reports, depending on what method you employ to track the issues you've resolved. If your project uses JIRA from Atlassian, the changes plugin will include the `jira-report`, which links to the `Jira Report` page. If you use a

changes.xml file (which could be maintained manually or generated from some other issue-tracking system), then it will include the changes-report, which links to the Changes Report page.

If your project uses JIRA, the changes-report is probably not of interest to you. Instead, you can simply verify that the information in your POM's issueManagement section resembles this:

```
<project>
  ...
  <issueManagement>
    <system>jira</system>
    <url>http://jira.codehaus.org/browse/MNG</url>
  </issueManagement>
  ...
</project>
```

Regenerate and refresh, and you'll see the Jira Report link appear under the Project Reports menu. By default, this report will contain the top 100 issues, sorted in order of creation date and priority - regardless of whether they're open or closed.

If your project doesn't use JIRA, you will have to get by with some level of manual maintenance of the changes report. In some cases, this may mean maintaining the file (found under src/changes/changes.xml by default) and keeping in SCM along with other project files. Below is a sample changes.xml:

```
<document>
  <properties>
    <title>Hello-World Issues</title>
    <author email="pbunyan@elsewhere.com">Paul Bunyan</author>
  </properties>
  <body>
   <release version="1.1" date="2007-03-01" description="Version 1.1 Release Notes">
      <action dev="pbunyan" type="add">
        Added changes.xml
      </action>
      <action dev="jsmith" type="fix" issue="01011">
        Fix the Widget Control Panel.
      </action>
     <action dev="jsmith" type="remove" due-to="Aaron Penn" due-to-email="apenn@writers.org">
        Fixed typos in How-To document.
      </action>
    </release>

    <release version="1.0" date="2007-01-01" description="Version 1.0 Release">
      <action dev="pbunyan" type="update">
        Uploaded documentation.
      </action>
    </release>
  </body>
</document>
```

If you include this sample report under src/changes/changes.xml, then regenerate and refresh, you should see the Changes Report link appear under the Project Reports

menu. The resulting report will list each issue in the changes document, in the order it appears. `action` sections with the `issue` attribute will have that issue number linked to the issue-tracking system (if you've configured the `issueManagement` section). The `due-to` and `due-to-email` attributes of the `action` element will be rendered as "Thanks to \ *$due-to*", with the due-to name rendered as a link to the email given by *\$due-to-email*. Finally, the `dev` attribute of the `action` element is linked back to the appropriate team member entry in the `Project Team` report, if it has been configured correctly.

Limit the `Jira Report` to Closed Items

If you'd like to limit the `Jira Report` to only display closed issues (which will approximate a release notes-style of report), simply include the following configuration for the changes plugin:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-changes-report</artifactId>
        <configuration>
          <statusIds>Closed</statusIds>
        </configuration>
      </plugin>
      ...
    </plugins>
  </reporting>
  ...
</project>
```

When you regenerate and refresh, you should notice that none of the items in the `Jira Report` have a status other than **Closed**.

TIP: The `statusIds` configuration accepts a comma-separated list of statuses, so you can tailor the exact set of issue-status types to include in the report.

maven-checkstyle-plugin

The Checkstyle plugin performs static code analysis, and generates a report of how well it adheres with predetermined code-style policies -- using Checkstyle (*http://checkstyle.sourceforge.net*), of course. The single report that this plugin provides -- `checkstyle` -- links to `Checkstyle` in the `Project Reports` menu, and provides summary statistics about which rules were violated, how many times, and by which file(s). At the bottom of the report, it gives the line number for each rule violation. By default, it uses the Sun code-style guidelines.

Many teams use some form of modified code style, other than that suggested by Sun. Therefore, it's important to that we can specify our own policy file used to evaluate the

project's source code. To accommodate common cases, the plugin has four built-in policy files:

- `config/sun_checks.xml` applies the Sun code-style guidelines.
- `config/turbine_checks.xml` applies those from the Turbine project.
- `config/avalon_checks.xml` applies the Avalon code style.
- `config/maven_checks.xml` applies the style used by Maven projects.

Beyond these built-ins, it's also possible to specify your own custom rules, using the `configLocation` parameter. All policy files should conform to the Checkstyle Checker module XML format (*http://checkstyle.sourceforge.net/config.html#Modules*). To specify your own file, add a configuration similar to the following:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>

        <configuration>
        <configLocation>src/checkstyle/checkstyle-configuration.xml</configLocation>
        </configuration>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

NOTE: The checkstyle plugin uses three strategies to resolve the value of the `configLocation`. They are, in order: classpath-resource, URL, and file. That is, Checkstyle will look for a resource in your classpath with a patch matching the `configLocation`. If it cannot find one, it will attempt to construct a new `java.net.URL` instance from the `configLocation`, open a stream, and download the configuration. Failing that, it will try to find a file that matches the given path, and load it from there.

The checkstyle plugin offers all the same options that would be available if you were to run Checkstyle directly, with the addition of some nice features for referencing configurations outside of your local project directory. Here are some examples of useful features provided by the checkstyle plugin:

Reviewing Your Checkstyle Configuration File

When the `configLocation` is successfully resolved, it's contents will be copied to the `${project.build.directory}/checkstyle-checker.xml` file (usually, `${project.build.directory} == ${basedir}/target`). This gives you the opportunity to review the actual policies against which the source code has been evaluated, which can

be handy when the configuration comes from a URL or a resource embedded in a dependency jar.

Verify Your Source Code License Header

One very useful feature of Checkstyle is that it allows you to verify that each source file has the correct license header. License headers are usually block comments (or blocks of single-line comments) that appear at the top of each source file, to give a clear indication of the license terms applied to that file. They are very common among open-source projects (usually, they're mandatory) but they could also be quite useful in proprietary code.

To verify that all of your project's source files have the appropriate license header, paste that header into its own file. For the purposes of this example, we'll copy the following into `src/checkstyle/checkstyle-header.txt`:

```
^package
^\s*$
^/\*\s*$
^ \*\s* Copyright 2007, Example Software Foundation$
^\s*$
^ \*\s+ This is the project license. See http://dev.example.com/LICENSE.txt for more information.$
^ \*/\s*$
^\s*$
```

You'll notice that the header used by Maven consists of a series of regular expressions. This is the format used by the `RegexpHeader` module; it gives us the flexibility to adapt to minor whitespace deviations introduced by code formatters and other tools, without sacrificing the content of the license header.

Next, we must configure the `configLocation` parameter to refer to the Maven code rules. To accommodate for a hard-coded license header that made it out in the latest release of the checkstyle plugin, we can't use the `config/maven_checks.xml` built-in rules. Instead, we can reference the updated configuration in SVN as follows. For convenience, we're also going to tell the checkstyle plugin which header file to use for the license check:

```
<configuration>
  <configLocation>http://svn.apache.org/repos/asf/maven/plugins/trunk/maven-checkstyle-plugin/src/main/r
  <headerLocation>src/checkstyle/checkstyle-header.txt</headerLocation>
</configuration>
```

If we regenerate and refresh, then click on the `Checkstyle` link under `Project Reports`, we should see that the App.java file has a non-compliant license header. We can fix that by adding the following comment to `src/main/java/book/reporting/App.java`:

```
package book.reporting;

/*
 * This is the project license. See LICENSE.txt for more information.
 */
...
```

This time when we regenerate, we'll see that the previous Checkstyle error has gone away.

maven-jxr-plugin

JXR is a source-code cross-referencing API maintained by the Maven project. The JXR plugin renders your project's source code to XHTML, with links for each line number to allow direct referencing of your source code, and links from source file to source file, wherever a cross-reference exists. In addition, if you've configured your POM to include JavaDoc API documentation, each browsable source file rendered by JXR will contain a link the corresponding JavaDoc page.

When you add the JXR plugin to your POM, it will create up to two new links under the `Project Reports` menu. One, called `Source Xref`, contains the cross-referenced pages for your main project sources. This link will always be included by JXR. The other link is called `Test Source Xref`, and contains a similar set of cross-reference pages for the unit-test source code for your project. Whether the `Test Source Xref` pages are produced or not is up to you; they are provided by default, but you can exclude them by disabling a single boolean parameter in the plugin's configuration.

As you might guess, the JXR plugin is extremely simple to use; to illustrate just how easy, let's try an example with the hello-world project. Begin by adding the JXR plugin to the `reporting` section, like this:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jxr-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

NOTE: Like the javadoc plugin, the jxr plugin is not compatible with the current version of the `site:run` mojo. Therefore, to preview these reports, you'll have to use the alternate command discussed at the beginning of the chapter:

```
hello-world$ mvn clean site
```

Then, load the rendered website from the `target/site` directory structure within your hello-world project directory.

When you regenerate and preview this time, you should see both the `Source Xref` and `Test Source Xref` links under the `Project Reports` menu. If you look at these new reports, you'll quickly notice that the sample code -- one class and one test case -- are

far too simplistic to show any sort of cross-referencing. You *should* notice that each line in the reference page has a unique anchor, allowing you to reference any line directly from an external page. Also, the link to the corresponding JavaDoc page at the top of the the `App` reference page (JavaDocs are not normally produced for unit tests, so you may not see a corresponding link in the `AppTest` page).

But let's turn our attention back to the cross-referencing feature. We can make two relatively easy changes to make cross-reference links show up. To create a cross-reference in the main project source, let's create a second class called `Configuration` for our application, and allow each `App` instance to accept a `Configuration` instance via a `setConfiguration` method. First, the new class:

```
package book.reporting;

public class Configuration
{
    ...
}
```

Now, the change to the `App` class:

```
...

public class App
{
    ...
    private Configuration configuration;

    public void setConfiguration( Configuration configuration )
    {
        this.configuration = configuration;
    }
}
```

This time, when we regenerate (remember, you cannot use the `site:run` mojo!) and preview, you'll notice that the cross-reference pages are a little more interesting. Specifically, the source listing for the `App` class now contains a link to the `Configuration` class. Alright, it's not *a lot* more interesting, but this is just a simple example!

GOTCHA! One thing the JXR plugin will *not* currently handle is cross-references that exist between unit-test code and main-project code. While this is clearly an oversight in this plugin's implementation, browsing from test code to main code is not generally considered the strongest use case for this plugin. Far more useful is the ability to browse and link to a project's source code, without getting into the arcane paths and strange interfaces of many SCM web front-ends.

maven-clover-plugin

Clover is a tool from Cenqua (*http://www.cenqua.com/clover*) which analyzes code coverage. For those who somehow don't know, code coverage is a form of analysis that measures how much of your project's source code is actually tested by a suite of tests.

In most cases, code coverage refers to a measurement of how well your *unit* tests exercise the main-project source code. For each line in each of your main-project source files, Clover will give you a number that corresponds to how many separate unit tests traversed that line of code. While traversal of a particular line doesn't necessarily mean that code is adequately tested, it's quite obvious what a count of zero means.

Maven's Clover plugin binds the Clover tool into your build process, and allows it to do one of several things. First, it can fail a conventional build (that is, one aimed at producing a project binary by compiling, archiving, etc.) if the average code-coverage percentage is too low. This has obvious quality-control attraction. However, this chapter is really more concerned with the second possible action: reporting those code-coverage statistics to the world, through the `Clover` report (under the `Project Reports` menu...I'm sure you see a pattern here) that this plugin can generate.

Before you can get started with the Clover plugin, you should know that Clover is not free software; you must obtain a valid license file if you want to generate this report. The good news is that Cenqua offers free, 30-day trial licenses on their website (again, that's *http://www.cenqua.com*). Once you've downloaded your evaluation license, we can proceed with an example.

First of all, the `clover` report requires a Clover database for the project, which can be generated with the `instrument` mojo. The `instrument` mojo is not a report, so it has to be configured in the `build` section of the POM:

```
<project>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-clover-plugin</artifactId>

        <configuration>
          <licenseLocation>clover.license</licenseLocation>
        </configuration>

        <executions>
          <execution>
            <id>mk-clover-db</id>
            <phase>pre-site</phase>
            <goals>
              <goal>instrument</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

NOTE: We bind the `instrument` mojo to the `pre-site` lifecycle phase, which is present in the `site` lifecycle (the one that gets executed whenever you issue the command `mvn site`). This allows Clover to run automatically as part of the site build. Also, remember that you had to download that evaluation license; its location is configured above for Clover to use. This single configuration will make the license file available to both the `instrument` and `clover` mojos (reports are mojos, too, in the strictest sense).

Next, since we want to include the Clover report in our project's website, we must also include an entry in the `reporting` section for Clover:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-clover-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

NOTE: Again, just like the JavaDoc plugin, the `clover` report is not compatible with the `site:run` mojo. To preview this report, use the alternate method discussed at the beginning of the chapter. This involves issuing the following command:

```
hello-world$ mvn clean site
```

Then, load the generated website from the `target/site` directory structure within your hello-world project directory.

When you regenerate and preview, you should see the `Clover` report link present under `Project Reports`. When you click on the Clover report, you'll notice that the `App` class has 50% code coverage (owing to the methods we added in this and the `AppTest` test case in the `maven-jxr-plugin` section above). Again, the important thing to take away from this report is **not** the idea that the `setApplicationName` method is adequately tested; rather, the value of this coverage report is to show that the `setConfiguration` method is *not* tested.

Generate PDF and XML Output from Your Clover Results

A nice, XHTML code-coverage report that's embedded directly in the project website is really convenient for many users. However, there are times when you may want to include PDF or even XML versions of the same report. PDFs are nice because they can be downloaded for offline perusal, and XML is great as a source for further machine processing of the data produced by Clover.

To render our Clover report in PDF, XML, and XHTML simulatneously, all we have to do is add a small configuration snippet to the clover plugin:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-clover-plugin</artifactId>

        <configuation>
          <generateXml>true</generateXml>
          <generatePdf>true</generatePdf>
          <!-- generateHtml is an option, but it's already true by default. -->
        </configuration>
      </plugin>
    </plugins>
  </reporting>
</project>
```

Now, regenerate and refresh. Since we're working off the filesystem for previewing purposes, it's possible to go to the `target/site/clover` directory without having a web-server redirect you to `target/site/clover/index.html`. You can take advantage of this fact to view the regenerated clover output directory. Since we've enabled PDF and XML output, in addition to XHTML, we can now see two new files in the `target/site/clover` directory: `clover.pdf` and `clover.xml`. To expose these alternative report formats to users, simply provide links in your website content or navigation menu.

maven-pmd-plugin

Similar to Checkstyle (discussed above), PMD (*http://pmd.sourceforge.net*) is another tool that performs static analysis on your source code. However, where Checkstyle looks for violations in the accepted code-style guidelines, PMD is more concerned with finding potential problems in the code that could eventually express themselves as bugs in the running product. In addition, the PMD plugin also provides a cut-and-paste detector (CPD) to detect duplicate code that could cause inconsistencies in your project (as one copy is modified but the other is not).

By default, the PMD plugin generates two reports, called `PMD Report` and `CPD Report`, under the `Project Reports` menu. To include these reports, simply add the following to your `reporting` section:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
      </plugin>
    </plugins>
```

```
  </reporting>
  ...
</project>
```

After inserting the above lines into the hello-world project POM, regenerate and refresh the project website (note that `site:run` works fine here). If you click on the `PMD Report` link, you'll see that we already have one violation (from the code we wrote for the JXR plugin, above). Basically, PMD is saying that it's bad practice to assign and store an instance variable that is never used. If you didn't follow the example given in the JXR plugin section above, then you probably won't see any errors at all.

To force the issue, and make sure we have some content for this report, let's modify the `App` code to contain a couple of violations of basic PMD rules:

```
...

public class App
{
    private String applicationName;

    public String getApplicationName()
    {
        if ( applicationName == null )
        {
            // empty if statement, to trigger a PMD reporting flag!
        }

        return applicationName;
    }

    public void setApplicationName( String applicationName )
        throws Exception
    {
        this.applicationName = applicationName;
    }
    ...
}
```

The `Basic` PMD rule-set doesn't allow empty `if` statements. Sure enough, when we regenerate the website and refresh the `PMD Report`, we can see a new error message, indicating that the code we just added contains an empty `if` statement. It's also important to notice that if you still have the JXR report in the POM from the examples earlier, the PMD plugin will notice this and link each problem listing directly to the corresponding line in the source reference. Remember, though, that the JXR report doesn't work with the `site:run` goal, so you'll have to use the filesystem-style preview to actually follow these links.

Before we dig into the `PMD Report` any further, let's turn our attention to the `CPD Report`. As I mentioned earlier, this page lists cases of possible cut-and-paste programming; areas where the code is the same, but where modification of one (and not the other) could create inconsistent behavior in the running binary. Cut-and-paste practi-

ces cause a enough of a maintenance headache that it pays to take steps early to try to minimize the endless (and, in some cases, mindless) duplication they embody.

If you click on the `CPD Report` generated against the hello-world code, you'll see that there are no detected cut-and-paste problems. To make things interesting, let's re-create the `App` class (`src/main/java/book/reporting/App.java`) under a different package -- say, `book/reporting/services`. As the report implies, you can trigger an error by simply copying the App.java file to the `src/main/java/book/reporting/services` directory (though for correctness, I also change the `package` in the new file):

```
package book.reporting.services;
...
public class App
{
    ...
}
```

Now, we have to make one more important modification to the POM. Since the CPD report's default configuration is to let any duplications under 100 lines in length slip through, we need to dial this down so our little example shows up:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>

        <configuration>
          <!-- CPD configuration, for small examples. -->
          <minimumTokens>10</minimumTokens>
        </configuration>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Regenerate the website and refresh again, and this time you should see the full text of the duplicated code for the two `App` classes, along with JXR links for each (remember, though: JXR doesn't work with `site:run`).

Customizing Your PMD Rule-Sets

By default, the `PMD Report` checks your source code the Basic, Imports, and Unused Code rule-sets. However, at times you may want to change this; you may want to use a different mix of built-in rule-sets, or maybe you want to include a custom rule-set that you created. The PMD plugin supports this, of course, through the `rulesets` configuration parameter. To illustrate, let's take advantage of a new problem we introduced earlier in our discussion of this plugin:

```
public void setApplicationName( String applicationName )
    throws Exception
```

Remember that one? If you winced at the sight of that `throws Exception` clause, you were right to do so. Any lower-level exceptions -- even derivatives of RuntimeException, which are normally unchecked -- will be channeled through this clause. Even worse, anyone calling this method will have to catch `java.lang.Exception`, then try to make sense of the myriad errors it could indicate. PMD has a rule to guard against this sort of coding mistake, so let's enable it:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>

        <configuration>
          ...
          <!-- we've already customized this plugin a little for CPD -->

          <rulesets>
            <!-- include all of the default rule-sets: -->
            <ruleset>/rulesets/basic.xml</ruleset>
            <ruleset>/rulesets/imports.xml</ruleset>
            <ruleset>/rulesets/unusedcode.xml</ruleset>

            <!-- add the rule-set for exception-handling practices -->
            <ruleset>/rulesets/strictexception.xml</ruleset>
          </rulesets>
        </configuration>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

NOTE: We must respecify the default rule-sets here, since PMD cannot assume that we still want those when we specify our own rule-sets.

When you regenerate and refresh the `PMD Report`, you should see a new error listed, referring to the `throws Exception` clause above. It's important to know that the PMD plugin can accept classpath resources (like those in the configuration above), URLs, or filesystem paths for `ruleset` configurations.

Additional Reports Available from the Codehaus Mojo Project

While the reports provided by the Apache Maven project contains many valuable additions to the project website, they don't constitute the full complement of what's available. Most notably, since Apache projects cannot host code that depends on li-

braries with licenses that are incompatible with the Apache Public License (APL), the reports hosted under the Apache Maven umbrella cannot contain reporting or metric-gathering tools that have certain licenses (GPL and EPL, to name just a couple). For this reason, Maven has a sister project called Mojo, hosted at Codehaus, that provides a home for these reports.

Just like those hosted in the Apache Maven project, each of the reports hosted at the Codehaus Mojo project shares some certain naming elements, namely the following:

- **GroupId:** *org.codehaus.mojo*
- **ArtifactId:** <*something*>*-maven-plugin*

Let's take a look at some of the things that the Mojo reports can provide for your website.

clirr-maven-plugin

Clirr (*http://clirr.sourceforge.net*) is a nice little utility to check for binary and source compatibility between two releases of the same project. This sort of check is critical to help ensure backward compatibility between minor-version releases of a project. To bind this into the build process, Maven provides two mojos in the `clirr-maven-plugin`: the `check` mojo is used to cause the build to fail when a compatibility violation occurs, and the `clirr` mojo is a report that displays the project's compatibility with previous releases.

Since we're primarily concerned with the project website, the `clirr` report is of particular interest here. It generates a link called `Clirr` under the `Project Reports` menu (where else?). However, before we can investigate how to include this report for our hello-world sample project, we need a previous release to check against.

Creating the example environment for this report turns out to be a slightly more involved process. First, we must deploy a version of the hello-world project to a dummy remote Maven repository, so we'll have something to check against. We can do this by adding a `distributionManagement` section to the POM, pointing at our dummy repository, and adjusting the POM `version` so it does't look like a SNAPSHOT:

```
<project>
  ...
  <version>1.0</version>
  ...
  <distributionManagement>
    <repository>
      <id>dummy</id>
      <url>file://${java.io.tmpdir}/book-repository</url>
    </repository>
  </distributionManagement>
</project>
```

Having created our new "release" POM, we can deploy it to the new dummy repository:

```
mvn clean deploy
```

Next, we have to add a new `repository` entry to the hello-world POM that points to this dummy repository, so the Clirr plugin can locate the previous "release", and update the version to the next SNAPSHOT (as if we were ready to develop on that new version):

```
<project>
  ...
  <version>1.1-SNAPSHOT</version>
  ...
  <repositories>
    <repository>
      <id>dummy</id>
      <url>file://${java.io.tmpdir}/book-repository</url>
    </repository>
  </repositories>
  ...
</project>
```

Finally, once all of this is setup, we can generate the `clirr` report by adding a simple entry to the `reporting` section:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>clirr-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

NOTE: The Clirr plugin does not seem to be compatible with the `site:run` mojo, so you'll need to use the filesystem-based approach to previewing this report. For more information, see the instructions near the beginning of this chapter.

Now, regenerate and refresh the project website, and you should see the `Clirr` report link appear. If you click on it, you'll notice that there are no differences between the two versions! (I know, I was surprised too.)

So, we have a working `Clirr` report; now, let's make it interesting. Modify the `src/main/java/book/reporting/App.java` file as follows.

Change this:

```
    public String getApplicationName()
    {
        if ( configuration == null )
        {
              // empty if statement, to trigger a PMD reporting flag!
        }
```

```
        return applicationName;
    }

    public void setApplicationName( String applicationName )
        throws Exception
    {
        this.applicationName = applicationName;
    }
```

To this:

```
    public String getName()
    {
        if ( configuration == null )
        {
            // empty if statement, to trigger a PMD reporting flag!
        }

        return applicationName;
    }

    public void setName( String applicationName )
        throws Exception
    {
        this.applicationName = applicationName;
    }
```

This time, when you refresh, you should see two errors pop up on the report. Just like many of the other reports we've discussed, if you use the JXR plugin in concert with this one, each error will be linked back to the line of source code that caused it.

Show Only Errors, Not Warnings

So, you don't care about those pesky compatibility warnings? You just want them gone from your beautiful report?? Well, say no more. A small tweak to the Clirr report configuration, and you're good to go:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>clirr-maven-plugin</artifactId>

        <configuration>
          <minSeverity>error</minSeverity>
        </configuration>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

TIP: The `minSeverity` parameter accepts the values: `info`, `warning`, and `error`. The default value is `warning`.

cobertura-maven-plugin

Similar to Clover (discussed above), Cobertura (*http://cobertura.sourceforge.net*) is a code-coverage tool. Unlike Clover, Cobertura is released under an open-source license (GPL), so you don't have to worry about evaluation licenses or an eventual software purchase in order to use Cobertura. Both tools generate statistics on the number of unit (or other) tests that trigger any particular line of code in your project's sources; however, Cobertura also throws in some metrics analyzing your code's complexity.

Like the Clover plugin, the Maven `cobertura-maven-plugin` provides several mojos. Many of these are concerned with managing the statistics database used by Cobertura, and with instrumenting the project's classes for statistics-gathering. In addition, the `check` mojo gives you the ability to fail the build if your project's code coverage isn't up to snuff. However, the mojo we're really after for this chapter is the lone report, called simply `cobertura`. This report generates a page linked under `Cobertura Test Coverage` in the now-familiar `Project Reports` menu. To add this report to our sample project, simply add the following to the hello-world POM:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugins>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

NOTE: The Cobertura plugin is incompatible with the `site:run` mojo, so be sure to use the filesystem-based approach to previewing your project website (discussed near the beginning of the chapter) when working with these examples.

When you regenerate the site, and click on the `Cobertura Test Coverage` link, you'll see a report very similar to the Clover report. However, notice the extra column on the right, called `Complexity`. This refers to the McCabe cyclomatic code complexity metric for the package or class in question, which measures the number of distinct "paths" that can traverse that piece of code (to find more information on cyclomatic complexity, see the *Resources* section at the end of this chapter).

While the `check` mojo offers many options for fine-tuning the criteria that a project must pass before the build can succeed, the `cobertura` report offers relatively few. Simple options for generating XML in addition to XHTML, for suppressing informational-

level messages, and for adjusting the maximum memory for testing runs are among the most notable. It is worth mentioning that the `instrument` mojo (which is used by the `cobertura` report) also supports several options for including or excluding certain packages and file-sets. However, for most common use cases this report just works -- cleanly and simply -- without much intervention or configuration.

findbugs-maven-plugin

Much like the PMD plugin (described above), the FindBugs (*http://findbugs.sourceforge.net*) report produces a list of potential problem errors in your project's source code using static analysis techniques. The FindBugs plugin will create a page linked to `FindBugs Report` under `Project Reports`. To enable this report for out hello-world project, add the plugin to the POM as follows:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>findbugs-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Regenerate and refresh your project website (`site:run` is fine here), and you'll see the report link appear. When you click on the link, you'll see a short list of potential problems that FindBugs noticed in our source code. Specifically, you'll notice the empty `if` statement listed as a *Bug*.

The FindBugs plugin supports a few configuration options that look interesting, specifically `effort` and `threshold`. The `effort` parameter allows you to tell FindBugs how much memory and time to spend in pinning down more subtle bugs. Valid values for this parameter are `Min`, `Default`, and `Max` -- can you guess which one it uses by default? The `threshold` parameter tells FindBugs what priority of messages you want to see in the output, and supports the values `Low`, `Medium`, `High`, and `Exp` (experimental). Unfortunately, even the configuration for highest output did not reveal our ugly little `throws Exception` clause.

javancss-maven-plugin

The JavaNCSS (*http://www.kclee.de/clemens/java/javancss*) plugin compiles some simple statistics about your project's source code, including non-comment source statements (NCSS), number of methods, number of classes, and number of javadocs. At the per-method level, it also computes cyclomatic complexity statistics. The idea of this

report is to give you some insight into the distribution of code throughout the project, along with the proportion of code that contains formal documentation (JavaDocs), so that you can tell identify any hot-spots (those in dire need of documentation of refactoring).

The report it linked to the `Project Reports` menu (where else?) as `JavaNCSS Report`, and you can enable it by adding the following to the hello-world POM:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>javancss-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

When you regenerate and refresh (`site:run` is fine here), you'll find that this report gives a nice code summary of our (very small) hello-world project. If any of the table-column headings are confusing, just click on the `explanation` link at the top of each section.

jdepend-maven-plugin

JDepend (*http://clarkware.com/software/JDepend.html*) is a tool that produces design-quality metrics for your project. These metrics can help isolate potential architectural "smells" in your code, such as classes with cyclic dependencies, and give an indication of how well the project rates in terms of reusability, extensibility, maintainability, and other -abilities. Like the JavaNCSS plugin, the JDepend plugin for Maven produces a report that can isolate any hot-spots in your code which may need refactoring. This report links to the `Project Reports` menu as simply `JDepend`, and can be included in your project's website with the following snippet:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jdepend-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

If you regenerate and refresh (using `site:run` works fine here), then click on `JDepend`, you'll see the various metrics for the project, collected in an overall summary and by package. Toward the bottom, the report will list any package cycles (package 'a' has a class 'A' that uses class 'B' in package 'b', which uses another class from package 'a') that it detects.

JDepend is a very simple report; in fact, the only configuration options have to do with controlling which directories to use for output, classes, and your project (it's not entirely clear what this one does, but it's probably used for relative-path calculations when the classes directory is not specified).

Some of the metrics and explanations provided on this report can be pretty dense, and a little abstract. For a more in-depth explanation of how best to use this data, I encourage you to visit the JDepend website.

taglist-maven-plugin

Have you ever stopped to think about all those TODO or FIXME tags that are lost in your source code? Those tags are created for a purpose, as reminders to fix something later, after the current task at hand is completed. Many IDEs provide a view that collects these tasks from the source code, to make them easier to track down and fix. The point is, these tags represent known shortcomings of the existing source code. As such, they should be monitored closely, and available to anyone who wants to know the health of the project. If the reports section of a project website is all about assessing health, then knowing the locations of these known warts is a critical element in that section.

Fortunately, the `taglist-maven-plugin` provides exactly this sort of report for your project website. The taglist plugin is one of the gems in the Codehaus Mojo project, fitting well with other reports that produce critical quality metrics, such as the Surefire (unit test) report and the Cobertura report. The taglist report lists all occurrences of a set of tags (think @todo or //TODO) in your project source code, and lists them on a single page, with links back to the browsable source code rendered by JXR (if you've included JXR, that is). When generated, it normally appears as `Tag List` in the `Project Reports` menu.

When configuring the taglist report, it's important to remember that by default, the report will only track `@todo` and `TODO` tags. This means it will miss `FIXME` -- a commonly used marker for critical shortcomings in the code -- completely. So, when we add the taglist report to the hello-world project, we'll go ahead and configure the tags at the same time:

```
<project>
  ...
  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
```

```
        <artifactId>taglist-maven-plugin</artifactId>

        <configuration>
          <tags>
            <!-- Add the definitions for FIXME javadoc and code comments... -->
            <tag>@fixme</tag>
            <tag>FIXME</tag>
            <tag>@todo</tag>
            <tag>TODO</tag>
          </tags>
        </configuration>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Now, regenerate and refresh the project (again, `site:run` is perfectly fine here). When you click on the `Tag List` report, you should see sections for each of the tags, and notice that no tags were found.

Let's see if we can make this report a little more interesting. We'll start by adding some comments for future improvement to our `App` class:

```
/**
 * Hello world!
 *
 * @todo Provide JavaDocs for the methods in this class!
 */
public class App
{
    ...
    // TODO: How can we initialize this when the application starts??
    private String applicationName;

    // FIXME: Re-add getApplicationName() for backward compatibility
    public String getName()
    {
        // FIXME: This is a NullPointerException waiting to happen...
        if ( configuration.toString() == null )
        ...

    // FIXME: Re-add setApplicationName() for backward compatibility
    public void setName( String applicationName )
        throws Exception
    ...
        }
}
```

Now, if you simply refresh the report in your browser, you'll see that there are several tag occurrences listed, each with a link back to the JXR source page. This provides the user with an easy way to browse the locations where there are known issues that have yet to be resolved.

The taglist report is deceptively simple, yet powerful. There are very few configuration options for this report; it simply scans through your source code, picking up on the lines that contain one of the tags listed. However, aggregating this sort of information into a single-page report on the project website can make it very easy to track down the areas of the project where the source code is less mature, and could fail.

Assembling a Killer Report Suite for Your Project

As we've mentioned several times in this chapter, certain reports provide critical metrics for assessing the health of your project. Therefore, by publishing the right kinds of reports, your project's website can become an indispensible tool for guiding the priorities and tasks of the development team. Likewise, it can help users make a decision about whether the project is a good fit for their environments, rather than risking a wrong guess.

However, you'll also notice that many of the reports we discussed overlap in functionality, or else provide total functional duplication. Obviously, providing all of these reports can actually muddy the waters, making it harder to make good decisions about the project. So, what reports should we use to assemble a killer report suite?

The key is to get the right mix of metrics and other project information. In doing this, there is no absolute right answer as to which particular reports you should use; otherwise, it really wouldn't make sense to cover alternative reports that provide such similar functionality. Your project's environment may determine some of the answers (for instance, does your company have a Cenqua site license that would make Clover a natural choice?), and your preferences for data presentation may also provide some guidance. Whatever choices you make, you should try to include a complete snapshot of the metrics and other information that can help people make informed decisions. Some important metrics include:

- test coverage and status (successes vs. failures)
- NCSS and other basic measurements of class size
- metrics of class/package design quality
- browsable source
- API documentation
- changelog for the project
- backward compatibility information
- potential problem areas in the source code

Also, to give your users and developers convenient access to critical project infrastructure, you should also consider publishing the following project information:

- mailing list addresses
- issue tracker URL

- continuous integration URL
- development team listing
- contributor listing
- project dependencies
- source repository (especially if it's an open-source project)
- project license (again, especially if it's an open-source project)

While there's not much point in having reports that overlap one another, it is important to include as much of these report types as you can. Publishing a comprehensive project website that includes the right mix of reports will give your user and developer communities a common rallying point, which helps to focus and enrich these communities.

Tips and Tricks

Control the Locations of Report Links in the Site Menu

If you want to change the location of the `Project Reports` menu in your website navigation, you can refer to the entire menu with the following:

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

Simply add this line to your site descriptor, in the desired position. When you regenerate the website, you should see the `Project Reports` menu move accordingly.

If, on the other hand, you want to remove the `Project Reports` menu altogether, and move the reports it contains into other menus, you'll have to reference each report separately, according to their page location on the site. For example, if you wanted to move your project's JavaDocs into the `Developers` menu, you might have a menu entry like this:

```
<project name="Hello World">
  ...
  <body>
    ...
    <menu name="Developers">
      ...
      <item name="JavaDocs" href="/apidocs/index.html"/>
    </menu>
    ...
  </body>
</project>
```

TIP: If you're not sure what `href` to use, generate the website with the `Project Reports` menu in place, and take notice of the link `href` generated for the report in question.

Summary

Project websites serve as a focal point for all activity surrounding a project. They house documentation, providing links to other project infrastructure, attract new community members, and help the project team to organize their plans. In open-source projects -- and increasingly, in the field of commercial software development as well -- this focal point is particularly important, since the development team can be widely dispersed in terms of both geography and time (time-zone differences can be much more difficult to overcome than geographical ones).

In this chapter, we've talked about many different reports that Maven offers for the project website, in addition to discussing the ways that Maven reports are configured. We've discussed how reporting can offer a valuable counter-balance to the documentation commonly found on project websites, and deliver the information your users need to make decisions about integrating your project into their own environments. In addition, we've talked about how the right reports can help developers identify targets for improving existing project code, and plan for future development.

This chapter is meant to provide the second half of the project-website story; for the first part, which shows you how to publish project documentation and configure the website using Maven, see the **Site Generation** chapter.

Resources

Cyclomatic Code Complexity

- On Wikipedia: *http://en.wikipedia.org/wiki/Cyclomatic_complexity*
- OnJava Article, *Code Improvement Through Cyclomatic Complexity* [Measurement]: *http://www.onjava.com/pub/a/onjava/2004/06/16/ccunittest.html*

CHAPTER 12

The Maven Repository

Discovering the Magic

As you are hopefully beginning to understand, Maven leaps beyond the previous concepts of simple build tools into a realm of its own - as some fancy amalgam of an object-oriented, Ant-like toolset (but convention-based) and a Java CPAN or RubyGems repository of both tools and dependencies. Since half of the magic of Maven takes place in the Maven repository, let's take a quick look at the pieces that make it.

Wagon

Wagon is a Maven sub-project which manages Maven's transport mechanisms - tasked with uploading-to and downloading-from the remote repositories. By default Wagon can transport from HTTP (http://...) and to/from local filesystems (file://...). Other types of repositories - like WebDAV uploading - require you to add the provider extension to the POM.

Repositories

Remote repositories are where all Maven artifacts are stores - whereas local repositories are more like local caches of

Creating an In-House Repository

The most straightforward and common method of creating repositories is with a web server that points to a base directory of deployed Maven artifacts - a directory of uploaded files deployed via wagon. If it sounds complex, it's not, as we are about to see.

Repository with Apache HTTPD ("Getting")

1. Download and install Apache HTTPD 2 server.

2. Alter httpd.conf to point to your deployment repository. Mine looks like this (complete with fancy headers and footers, note the importance of NameWidth=* to get complete file names... else the webserver might cut them off).

```
Alias /maven-snap "/mnt/maven2/snapshot_repo"
<Directory "/mnt/maven2/snapshot_repo">
  Options +Indexes
  HeaderName /HEADER.html
  ReadmeName /FOOTER.html
  IndexOptions FancyIndexing HTMLTable NameWidth=* SuppressHTMLPreamble VersionSort
  Order allow,deny
  Allow from all
</Directory>
```

   Just replace the directory path above with your own repository's base path.

   The HEAD and FOOTER used for this example are available for download here (http://www.sonatype.com/book/examples/repository/apachehttpd.zip).

3. Finally, add the repository to either your project poms (parent is best... remember inheritance!) and/or add the repository to your build system's `settings.xml` file.

```
<settings>
  <!-- repos -->
  <profiles>
    <profile>
      <id>my-snap-repo-profile</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>my-snap-repo</id>
          <name>My In-House SNAPSHOT Repository</name>
          <url>http://www.my-server.com/maven-snap</url>
          <releases>
            <enabled>false</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>my-snap-plugin-repo</id>
          <name>My In-House SNAPSHOT Plugin Repository</name>
          <url>http://www.my-server.com/maven-snap</url>
          <releases>
            <enabled>false</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>interval:15</updatePolicy>
```

```
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>

  <!-- mirrors -->
</settings>
```

One small note while we are on the repository topic: you can alter your local repository's location in your settings.xml file. This is usually unnecessary, however, is quite useful for keeping build machine local repos in synch reguardless of who may logged in at the time.

```
<settings>
  <localRepository>~/.m2/repository</localRepository>
</settings>
```

Deploying to the Repository with Wagon ("Setting")

The other half of the repository equation - pushing artifacts to be available to everyone. Like the get repository settings above, distribution management must be set under a profile in settings, or within a project's POM. The three types of projects to distribute are regular (`repository`) deployments, `snapshotRepository` deployments, and `site` file deployments.

```
<settings>
  <profiles>
    <profile>
      <distributionManagement>
        <repository>
          <id>wicket-repo</id>
        <url>scpexe://shell.sourceforge.net/home/groups/w/wi/wicket/htdocs/maven2</url>
        </repository>
        <snapshotRepository>
          <uniqueVersion>false</uniqueVersion>
          <id>maven.sateh.com</id>
        <url>scpexe://maven.sateh.com/sateh/services/lighttpd/maven.sateh.com/www/wicket</url>
        </snapshotRepository>
        <site>
          <id>wicket-site</id>
          <url>scpexe://shell.sourceforge.net/home/groups/w/wi/wicket/htdocs</url>
        </site>
      </distributionManagement>
    </profile>
  </profiles>
</settings>
```

Note: The deploy function is not currently transaction-based. This shall be fixed in a later version of Wagon - so take care when deploying, and keep an eye out for deployment failures (don't be scared - they are not common)

File

This is built-in and the simplest method of deployment, especially if you have a closed-network. In pure MS Windows shops this can easily be enacted by creating a common shared directory and prefix it with "file://".

```
file://\\someserver\mavenrepo1\
```

Or if you can - map it to a drive letter.

```
file://M:\
```

In non-Windows shops, it's as easy as mounting a network drive (if not deploying locally, which is usually a good idea to seperate your build machines from the repository machines).

```
file:///mtn/mvn-repo/
```

FTP

The file-transfer protocol is the first in our list of providers that usually require password. More on that later. For now note the `distributionManagement` url of FTP, which should look familiar.

```
ftp://someserver/mavenrepo1/
```

FTP is not available by default, and must be added as a Maven extension. Adding an extension to Maven is as simple as adding a dependency, yet in the `extensions` element under project `build`.

```
<project>
  ...
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ftp</artifactId>
         <version>1.0-beta-2</version>
      </extension>
    </extensions>
  </build>
</project>
```

The ability to leave out oft unused extensions keeps Maven small and fast, and the ability to add extensions helps shield it from obsolescence. This is the most common type of extension you will use from the Maven core project set.

Quick note on servers.

It is important to note that you would never want to add a password to a POM - so Maven does not support that. Instead, each user/system must set his/her/its own credentials in the `settings.xml` file. The server `id` must be the same as the `id` used in the `distributionManagement` used on `repository`, `snapshotRepository` or `site` deployment.

```
<settings>
  <servers>
```

```
    <server>
      <id>my-ftp-repo</id>
      <username>joe</username>
      <password>pa$$wOrd</password>
    </server>
  </servers>
</settings>
```

Using the ftp URL and matching ID, the deployment repository may be defined in the POM as the following:

```
<project>
  ...
  <distributionManagement>
    <repository>
      <id>my-ftp-repo</id>
      <url>ftp://someserver/mavenrepo1/</url>
    </repository>
  </distributionManagement>
</project>
```

HTTP(s)

The http and https deployment servers are built into Maven by default. You do not need to add an http extension, you just need to set up some sort of REST server that will accept uploads.

```
https://someserver/mavenrepo1/
```

WebDAV

A WebDAV server can be easily set up using Plexus.

```
dav:http://someserver/mavenrepo1/
```

SSH

Secure Shell is the *de facto* method of deploying for the security-minded.

SCP.

```
scp://someserver/mavenrepo1/
```

```
scpexe://someserver/mavenrepo1/
```

To use scpexe, you must add the `wagon-ssh-external` extension.

```
<project>
  ...
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ssh-external</artifactId>
         <version>1.0-beta-2</version>
      </extension>
    </extensions>
```

```
  </build>
</project>
```

SCM

There are cases where, for some reason or another, you need to put artifacts into a repository. The `wagon-scm` extension is created for just such occasions. Let me take this opportunity to beg, to plead, not to ever use this. Unless you are required by some laws or processes to have a version control repository to hold your artifacts, ignore this - it is slow, harder to scan, and redundant (versions are part of the artifact name anyway, remember?).

With that said, use it like this.

Crazy SCM url.

```
scm://
```

Deploy

Once the repository is available for download and upload (or "getting" and "putting", in Wagon parlance) all you must do is type:

```
mvn deploy
```

Here is where the power of Maven truely becomes apparent. Maven will - due to the build lifecycle - execute every phase up to deployment. If you follow good testing conventions, your code must successfully pass all unit tests as well as integration tests before Maven allows your project to live in the public realm. The benefits are hopefully obvious.

Repository Managers

Although creating a local repository with Apache HTTPd server is fine for serving local repositories, it is not optimal for managing several or very large remote repositories.

Proximity

Proximity lies outside of the Maven Apache projects family - however is the oldest repository manager in active development.

Tips and Tricks

A Custom Artifact Handler

When creating a new packaging type you sometimes wish to name the packaging name differently from the extension it generates.

The default artifact handler maps the packaging, type, and file extension to the same value. For example, if you create a packaging of `plexus-application`, then the file in the repository will be `myartifactId-1.0.plexus-application`. The same applies if you request a particular type from a dependency. If you want to change that extension, or change how a dependency's type field is mapped to the other fields, you must include a custom artifact handler in your plugin.

Like giving a packaging, this is currently achieved by adding a Plexus descriptor to your plugin (or modifying it if it already exists). This file is META-INF/plexus/components.xml in the plugin JAR.

A complete artifact handler for the test-jar type looks like this:

```
<component-set>
  <components>
    <component>
      <role>org.apache.maven.artifact.handler.ArtifactHandler</role>
      <role-hint>test-jar</role-hint>
     <implementation>org.apache.maven.artifact.handler.DefaultArtifactHandler</implementation>
      <configuration>
        <classifier>tests</classifier>
        <extension>jar</extension>
        <type>test-jar</type>
        <packaging>jar</packaging>
        <language>java</language>
        <addedToClasspath>true</addedToClasspath>
      </configuration>
    </component>
  </components>
</component-set>
```

The fields are configured as follows:

Required

- **role-hint** - The type being defined.
- **type** - Must match the role-hint.

Not Required

- **extension** - The extension to give the artifact in the repository.
- **packaging** - The packaging of the artifact to look for.
- **classifier** - The classifier to append to the artifact name (after version and before extension) when using this type.

- **language** - The language the artifact is written in, it defaults to `java`.
- **addedToClasspath** - Confirms whether the artifact should be included in a classpath or library path when used, defaults to `true`.
- **includesDependencies** - If the artifact already includes all of its dependencies, this setting ensures they are not propogated transitively, which by default they are not (`false`).

As above, the extensions flag will need to be provided whenever the plugin is declared and the type is used.

Going Offline

You're not always connected to the internet, or even your network. The `maven-dependency-plugin` makes going offline simple with the `go-offline` goal. Just run in the base of the project you will work on offline:

```
mvn dependency:go-offline
```

A great thing about this plugin is for organizations who wish to block the outside world and use only "authorized" projects. Just run the above goal, disconnect from the network.

Some of the following settings are useful for limiting the scope of this plugin (check the online documentation for an exhaustive list). *Note: all excludes are ignored if includes are set, and default to include all of each respective field*

- **silent** - If the plugin should be silent. Default value is false.
- **includeGroupIds**, **excludeGroupIds** - Comma Seperated list of `groupId`s to include or exclude.
- **includeArtifactIds**, **excludeArtifactIds** - Comma Seperated list of Artifact names to include or exclude.
- **includeScope**, **excludeScope** - Maximum scope to include or exclude. For example, `test` will also include `compile` scopes.
- **overWriteIfNewer** - Overwrite artifacts that don't exist or are older than the source, it defaults to `true`.
- **overWriteReleases** - Overwrite release artifacts, it defaults to `false`.
- **overWriteSnapshots** - Overwrite snapshot artifacts, it defaults to `false`.

Summary

Maven repositories are the hub of the Maven's artifact management philosophy - without them a large part of Maven power is gone. Because of this, it is important that repositories be populated correctly and utilized as well, since one missing piece makes the whole puzzle pointless. Due to Maven's Wagon project which abstracts commu-

nication between a user and remote repository, there are many different ways of both deploying artifacts and accessing them from a repository - though the latter tends to stick to http downloads, since it is trivial to set up a good webserver that can handle lots of download traffic.

CHAPTER 13
Java EE

This chapter follows an example project. You may wish to download the calculator-stateless-pojo example (http://www.sonatype.com/book/examples/book-j2ee.zip) and follow along.

The Culmination

Although this book is not slated to be a cookbook of project recipies, but rather a guide to the pieces that make up the Maven puzzle, we would be remiss in out duties were we to ignore one of the most common project structures: a Java Enterprise Edition (Java EE - formerly J2EE) project. Although it is unnecessary to read the entire book to make sense of this chapter, nor can you proceed green. To this end, we suggest you begin with the first couple of chapters, get a good feel for Maven, before proceeding here.

The Project

Our Java EE project will have the following structure: an EJB project consisting of a simple EJB 3 Session bean, a WAR containing presentation files as JSPs, and an EAR containing the modules and dependencies. We will then go over the task of using Maven to help develop, test and finally deploy this project. The sample project is based upon the Apache Geronimo Stateless Calculator POJO sample project, version 2.

Note: *The examples in this chapter will download and install two entire application servers in the course of testing - one Jetty and one JBoss. You mut have 200 Megs of diskspace available to install and run each safely. If diskspace is a consideration, then do not run the demos.*

Note: *You must be running Maven version 2.0.5 or above to run these examples properly.*

Using Archetypes to Generate Project

Maven Central comes with several archetypes available to generate the basic project structures. First, create a base directory and change to it and generate the EJB directo-

ries. Since we wil make use of EJB3, this project requires Java 5 (jdk 1.5) to run. Why? Because EJB 2 is *so* 2004.

Alert: We avoid using the maven-archetype-j2ee-simple archetype here because it generates a sample J2EE project assuming EJB2. If you use EJB2, then by all means, go ahead and use it!

```
mkdir calculator-stateless-pojo
cd calculator-stateless-pojo
```

Create the base project and site.

```
mvn archetype:create \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-site \
  -DarchetypeVersion=1.0 \
  -DgroupId=org.apache.geronimo.samples \
  -DartifactId=calculator-stateless
  -Dversion=2.0-SNAPSHOT
```

We use the site creation archetype for a dual purpose. It generates a starting calculator-stateless project POM - and the project's site structure with some example docs. We will alter the generated `pom.xml` later which will server as a parent and multi-module POM for the entire project.

Create the project which will house the EJB layer.

```
mvn archetype:create \
  -DgroupId=org.apache.geronimo.samples \
  -DartifactId=calculator-stateless-ejb
  -Dversion=2.0-SNAPSHOT
```

And next create the WAR project which will be the presentation layer.

```
mvn archetype:create \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DarchetypeVersion=1.0 \
  -DgroupId=org.apache.geronimo.samples \
  -DartifactId=calculator-stateless-war
  -Dversion=2.0-SNAPSHOT
```

Finally, create the simple EAR project, which will house these modules when packaged.

```
mkdir calculator-stateless-ear
touch calculator-stateless-ear/pom.xml
```

With these few short commands, you now have the following structure. Although unbuildable in its current form, the basic skeleton is complete.

Filling Out the Project

The previous section displayed how one may begin a Maven-based Java EE project in under a minute. Now, we will make the project work. You can either blow away the project structure we created above and copy the sample project (it follows the same

strcuture), or you can copy the sample project files into the structure you created manually. In either case, your project stucture should resemble this.

Note the addition of the `src/test` directory to the EAR project, and the removal of the EJB's generated `App.java` and `test` files.

```
calculator-stateless-pojo
|-- calculator-stateless-ear
|   |-- pom.xml
|   `-- src
|     |-- main
|     |   `-- resources
|     |     `-- META-INF
|     |       `-- geronimo-application.xml
|     `-- test
|       |-- java
|       |   `-- org
|       |     `-- apache
|       |     `-- geronimo
|       |       `-- samples
|       |       `-- calculator
|       |         `-- SeleniumTest.java
|       `-- resources
|         `-- my-selenium-test.properties
|-- calculator-stateless-ejb
|   |-- pom.xml
|   `-- src
|     `-- main
|       `-- java
|         `-- org
|           `-- apache
|           `-- geronimo
|             `-- samples
|             `-- slsb
|               `-- calculator
|                 |-- Calculator.java
|                 |-- CalculatorLocal.java
|                 `-- CalculatorRemote.java
|-- calculator-stateless-war
|   |-- pom.xml
|   `-- src
|     `-- main
|       |-- java
|       |   `-- org
|       |     `-- apache
|       |     `-- geronimo
|       |       `-- samples
|       |       `-- calculator
|       |         `-- CalculatorServlet.java
|       `-- webapp
|         |-- index.html
|         |-- sample-docu.jsp
|         |-- images
|         |   |-- menu_javadoc_off_116x19.gif
|         |   `-- menu_src_off_116x19.gif
```

```
|         `-- WEB-INF
|          `-- web.xml
`-- calculator-stateless
  |-- pom.xml
  `-- src
    `-- site
      |-- site.xml
      |-- site_fr.xml
      |-- fr
      |   |-- apt
      |   |   |-- format.apt
      |   |   `-- index.apt
      |   |-- fml
      |   |   `-- faq.fml
      |   `-- xdoc
      |     `-- xdoc.xml
      |-- apt
      |   |-- format.apt
      |   `-- index.apt
      |-- fml
      |   `-- faq.fml
      `-- xdoc
        `-- xdoc.xml
```

- `calculator-stateless` - This is the parent project and the multi-module project for the EJB, WAR and EAR modules. This also contains the project site. In some project layouts this pom and site data can be in the base directory. I placed it in a parallel directory to illustrate two things: 1) how the `module` elements are relative paths to project base directories, not artifactIds, and 2) the Parent `relativePath` has a very useful purpose.
- `calculator-stateless-ejb` - A simple EJB 3 project containing a Calculator Stateless Session bean. This performs two actions: add and multiply two integers.
- `calculator-stateless-war` - A WAR that contains a CalculatorServlet that uses the EJB. It is populated by the Java EE container via dependency injection. Beyond that, it also contains a JSP and other static web files. When built, it also houses the project site (moved from the calculator-stateless project when built).
- `calculator-stateless-ear` - Generates an application descriptor and packages the EJB and WAR project along with dependencies as a deployable EAR. The EAR POM also contains three profiles: one to run on Geronimo, one to run on JBoss, and a third that runs integration tests on JBoss.

Before digging into the details of the POM structure, let's first generate the project site under the `calculator-stateless` base project and install it.

```
cd calculator-stateless
mvn site install
```

Since it is a multi-module project, it will run the site lifecycle phases, then the build lifecycle phases (up to install) on each of the project modules. Your build should begin with

```
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Geronimo Calculator
[INFO]   Geronimo Calculator EJB
[INFO]   Geronimo Calculator WAR
[INFO]   Geronimo Calculator EAR
[INFO] ------------------------------------------------------------------------
[INFO] Building Geronimo Calculator
[INFO]  task-segment: [site, install]
[INFO] ------------------------------------------------------------------------
```

and end with

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------------
[INFO] Geronimo Calculator ................................ SUCCESS [12.767s]
[INFO] Geronimo Calculator EJB ............................ SUCCESS [1.082s]
[INFO] Geronimo Calculator WAR ............................ SUCCESS [1.482s]
[INFO] Geronimo Calculator EAR ............................ SUCCESS [0.560s]
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Sat Apr 14 20:54:30 CDT 2007
[INFO] Final Memory: 28M/51M
[INFO] ------------------------------------------------------------------------
```

Now that we know the projects are all installed, it's time to run the Geronimo profile install phase inside of the EAR project. It will likely take a few minutes, as the Geronimo plugin will download a Jetty 6 container to your project's target. This should only happen once, for then it lives in your local repository. *Note: certain environments may require special access to open ports, for example, by executing the <<<startup.sh>>* command via "sudo" in Mac OSX or Ubuntu Linux.>

```
cd ../calculator-stateless-ear
mvn install -P geronimo
./target/geronimo-jetty6-jee5-2.0-SNAPSHOT/bin/startup.sh
```

You can visit the project in a web browser at: http://localhost:8080/calculator-stateless/

```
./target/geronimo-jetty6-jee5-2.0-SNAPSHOT/bin/shutdown.sh
```

You may be asked for a Username and Password to complete shut down. They are `system` and `manager`, respectively.

```
Using GERONIMO_BASE:   ~/calculator-stateless-pojo/calculator-stateless-ear/target/geronimo-jetty6-jee5-
Using GERONIMO_HOME:   ~/calculator-stateless-pojo/calculator-stateless-ear/target/geronimo-jetty6-jee5-
Using GERONIMO_TMPDIR: ~/calculator-stateless-pojo/calculator-stateless-ear/target/geronimo-jetty6-jee5-
Using JRE_HOME:   /System/Library/Frameworks/JavaVM.framework/Versions/CurrentJDK/Home
Username: system
Password: *******
Locating server on port 1099... Server found.
```

```
Server shutdown begun
Server shutdown completed
```

The Details

Assuming all went to plan you should have had a fully functioning build. Considering that you did not have to download anything manually from the internet, it was relatively painless operation: a few configurations, install the project and start the server. If only all things were so easy! But, although it was simple from the user perspective, a fair amount of configuration lay underneath. Let's take a look at the simplest POM, for the EJB project.

Note: These POMs will be slightly different from the sample projects available for download to be simpler to read. See if you can spot the differences.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
  <groupId>org.apache.geronimo.samples</groupId>
  <artifactId>calculator-stateless</artifactId>
  <version>2.0-SNAPSHOT</version>
  <relativePath>../calculator-stateless/pom.xml</relativePath>
  </parent>
  <artifactId>calculator-stateless-ejb</artifactId>
  <name>Geronimo Calculator EJB</name>
  <packaging>ejb</packaging>

  <build>
  <plugins>
    <plugin>
    <artifactId>maven-ejb-plugin</artifactId>
    <configuration>
      <ejbVersion>3.0</ejbVersion>
    </configuration>
    </plugin>
  </plugins>
  </build>
</project>
```

This project couldn't be more straightforward. The EJB project inherits from the `org.apache.geronimo.samples:calculator-stateless` project which is relative to the same depth as this project, is of packaging type EJB, and we configured the `maven-ejb-plugin` to treat this project as ejbVersion 3.0. If not set, `ejbVersion` defaults to 2.1.

The WAR is slightly more complex, but not by much.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples</groupId>
    <artifactId>calculator-stateless</artifactId>
    <version>2.0-SNAPSHOT</version>
    <relativePath>../calculator-stateless/pom.xml</relativePath>
```

```
  </parent>
  <artifactId>calculator-stateless-war</artifactId>
  <name>Geronimo Calculator WAR</name>
  <packaging>war</packaging>

  <dependencies>
    <dependency>
      <groupId>${groupId}</groupId>
      <artifactId>calculator-stateless-ejb</artifactId>
      <version>${version}</version>
     <type>ejb</type>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-parent-site</id>
            <phase>process-resources</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>Copying site directory from parent</echo>
                <copy
                    todir="${project.build.directory}/${artifactId}-${version}"
                    failonerror="false"
                    overwrite="true">
                  <fileset dir="${basedir}/../calculator-stateless/target/site"/>
                </copy>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <configuration>
          <warSourceExcludes>WEB-INF/lib/*.jar</warSourceExcludes>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <classpathPrefix>lib/</classpathPrefix>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Like the EJB, the WAR inherits from `calculator-stateless`. This project is packaged as a WAR, meaning that - besides generating an artifact with a *.war extension - it also has different goals bound to the build lifecycle phases as well as different requirements. A WAR requires a web.xml file to succeed, else a build error.

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] Error assembling WAR: Deployment descriptor: ~/calculator-stateless-pojo/calculator-stateless-war/
```

By default it looks for `src/main/webapp/WEB-INF/web.xml` - see the `maven-war-plugin` for details on configuring the `war` package goal. The WAR lifecycle also packages all files under `src/main/webapp` into the root of the WAR, similar to `src/main/resources` - in fact you can use both, for example to filter an .html file but leave a .jsp file alone.

The two plugins that are configured here are the `maven-antrun-plugin` and the `maven-war-plugin`. The antrun plugin is being used as a convenient way to copy all the `calculator-stateless` site to the `${project.build.directory}/${artifactId}-${version}` directory (which, when the properties are realized, becomes `target/calculator-stateless-war-2.0-SNAPSHOT`). This is the directory that the `war:war` goal uses as a temporary location to hold files as it builds the final WAR artifact (the parameter is `webappDirectory`, which can naturally be reconfigured - were you inclined to do so).

The `war` plugin is configured to strip out all JAR files from being packaged. This practice is known as creating a "skinny war". By default the WAR plugin packages up it's dependencies (a "fat war"). However, we want the EAR artifact to bundle the WAR's dependencies in it's root instead. Were we to allow the WAR to bundle it's own dependencies, then the generated EAR would contain two sets of the same dependencies - one set in it's own root, and one set in the bundled WAR.

GOTCHA: The downside of this "skinny war" process is that you must add the WAR's dependencies into the EAR, since the `maven-ear-plugin` does not currently download a WAR's transative dependencies. This should be fixed in later versions of the plugin.

The EAR is what will bundle the WAR and EJB into a single Enterprise Archive.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples</groupId>
    <artifactId>calculator-stateless</artifactId>
    <version>2.0-SNAPSHOT</version>
    <relativePath>../calculator-stateless/pom.xml</relativePath>
  </parent>
  <artifactId>calculator-stateless-ear</artifactId>
  <name>Geronimo Calculator EAR</name>
  <packaging>ear</packaging>

  <dependencies>
    <dependency>
      <groupId>${groupId}</groupId>
      <artifactId>calculator-stateless-ejb</artifactId>
```

```
      <version>${version}</version>
      <type>ejb</type>
    </dependency>
    <dependency>
      <groupId>${groupId}</groupId>
      <artifactId>calculator-stateless-war</artifactId>
      <version>${version}</version>
      <type>war</type>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-ear-plugin</artifactId>
        <configuration>
          <displayName>Geronimo Sample EAR for Stateless Session</displayName>
          <description>Geronimo Sample EAR for Stateless Session</description>
          <version>5</version>
          <modules>
            <webModule>
              <groupId>${groupId}</groupId>
              <artifactId>calculator-stateless-war</artifactId>
              <contextRoot>/calculator-stateless</contextRoot>
           <bundleFileName>calculator-stateless-war-${version}.war</bundleFileName>
            </webModule>
            <ejbModule>
              <groupId>${groupId}</groupId>
              <artifactId>calculator-stateless-ejb</artifactId>
           <bundleFileName>calculator-stateless-ejb-${version}.jar</bundleFileName>
            </ejbModule>
          </modules>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <profiles>
    ...
  </profiles>
</project>
```

The profiles have been removed for now. The things to note are the two dependencies which will be packaged into the EAR, and the EAR plugin configuration, which is used by the `ear:generate-application-xml` goal to generate the EAR's application descriptor - which is a required component of an EAR, much like the web.xml for the WAR.

The important piece to note on the EAR plugin is the `modules` configuration. There are ten supported types at the time of writing.

- `ejb3Module` - ejb3 (Enterprise Java Bean version 3)
- `ejbClientModule` - ejb-client (Enterprise Java Bean Client)
- `ejbModule` - ejb (Enterprise Java Bean)

- `jarModule` - jar (Java Archive)
- `harModule` - har (Hibernate Archive)
- `parModule` - par (Plexus Archive)
- `rarModule` - rar (Resource Adapter Archive)
- `sarModule` - sar (JBoss Service Archive)
- `webModule` - war (Web Application Archive)
- `wsrModule` - wsr (JBoss Web Service Archive)

Each of the above modules accepts the following configuration elements.

- `groupId` - the groupId of the configured artifact
- `artifactId` - the artifactId of the configured artifact
- `classifier` - the classifier of the configured artifact, if needed
- `bundleDir` - the location of this artifact inside the ear archive - defaults to the root of the archive
- `bundleFileName` - the new name of this artifact inside the ear archive - defaults to the artifact's `finalname`
- `excluded` - exclude this artifact from being packaged into the ear archive - defaults to false
- `uri` - sets the uri path of this artifact within the ear archive. Automatically determined when not set
- `unpack` - unpack this artifact into the ear archive according to its uri - defaults to false

The webModule then has a special element, `contextRoot`:

- `contextRoot` - sets the context root of this web artifact

The jarModule has another element, since by default JARs are not added to the application descriptor.

- `includeInApplicationXml` - generate an entry of this module in application.xml - defaults to false

Finally, if you have a requirement for a non-supported artifact mapping (for example a Native Archive, or a NAR), add the following to the `maven-ear-plugin`'s configuration to map the artifact type to an existing module type.

```
<artifactTypeMappings>
  <artifactTypeMapping type="nar" mapping="jar"/>
</artifactTypeMappings>
```

Like most plugins in this book, there are more configurations available for the `maven-ear-plugin`. Consult the online documentation for a more comprehensive and up to date list *http://maven.apache.org/plugins/maven-ear-plugin/*.

The last pom that we will go over is the base pom. It it both a parent and a multi-module POM, meaning that the sub-projects inherit this project's POM settings and this project coallates the sub-projects into a single build.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.geronimo.samples</groupId>
  <artifactId>calculator-stateless</artifactId>
  <version>2.0-SNAPSHOT</version>
  <name>Geronimo Calculator</name>
  <packaging>pom</packaging>

  <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>

  <!-- note that the modules are relative paths -->
  <modules>
    <module>../calculator-stateless-ejb</module>
    <module>../calculator-stateless-war</module>
    <module>../calculator-stateless-ear</module>
  </modules>

  <!-- these dependencies will be inherited by all children -->
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jsp_2.1_spec</artifactId>
      <version>1.0-M1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-ejb_3.0_spec</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.5_spec</artifactId>
      <version>1.1-M1</version>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <!-- all children src/main/resources directories will now be filtered -->
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
```

```
  <plugins>
    <!-- set this project compiler to jdk version 1.5 -->
    <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
    </plugin>
  </plugins>
  </build>

  <reporting>
    <!-- run the Javadoc and JXR reports -->
    <plugins>
      <plugin>
        <artifactId>maven-javadoc-plugin</artifactId>
        <configuration>
          <aggregate>true</aggregate>
          <minmemory>128m</minmemory>
          <maxmemory>512</maxmemory>
          <breakiterator>true</breakiterator>
          <quiet>true</quiet>
          <verbose>false</verbose>
          <source>1.5</source>
          <linksource>true</linksource>
          <links>
            <link>http://java.sun.com/j2se/1.5.0/docs/api/</link>
            <link>http://java.sun.com/j2ee/1.4/docs/api/</link>
            <link>http://jakarta.apache.org/commons/collections/apidocs</link>
            <link>http://www.junit.org/junit/javadoc/</link>
          </links>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-jxr-plugin</artifactId>
        <configuration>
          <aggregate>true</aggregate>
        </configuration>
      </plugin>
    </plugins>
  </reporting>

  <profiles>
  ...
  </profiles>
</project>
```

Since we configure the licenses, reporting, resources, set the compiler to 1.5 and added some required dependencies in the parent POM, we have no need to repeat this information to the child projects - and have a single point from which to make changes to all inheriting project's defaults (which the children can override, see more on inheritence in the chapter on The POM and Project Relationships (*pom-relationships.html*).

Geronimo Configuration

The POMs shown above were examples of each projects in the simplest sense. Although running `mvn site` from the `calculator-stateless` directory will generate a Maven site - complete with Javadoc and a Java Cross Reference (JXR) source report; and although running `mvn install` will indeed generate descriptor files, compile, unit test, package and install all of the projects to the local repository - they will not install and run our Geronimo server, which we used to test the application on the local machine.

For this alternate use-case we create a profile.

```
<project>
  ...
  <artifactId>calculator-stateless</artifactId>
  ...
  <profiles>
    <profile>
      <id>geronimo</id>
      <build>
        <pluginManagement>
          <plugins>
            <plugin>
              <groupId>org.apache.geronimo.plugins</groupId>
              <artifactId>geronimo-maven-plugin</artifactId>
              <configuration>
                <assemblies>
                <!-- hint: see the geronimo-maven-plugin documentation for more assemblies -->
                    <assembly>
                      <id>jetty</id>
                      <groupId>org.apache.geronimo.assemblies</groupId>
                      <artifactId>geronimo-jetty6-jee5</artifactId>
                      <version>${version}</version>
                      <classifier>bin</classifier>
                      <type>zip</type>
                    </assembly>
                  </assemblies>
                  <defaultAssemblyId>jetty</defaultAssemblyId>
                  <background>true</background>
               <moduleArchive>${project.build.directory}/${artifactId}-${version}.ear</moduleArchive>
              </configuration>
              <executions>
                <execution>
                  <id>deploy-and-start</id>
                  <phase>install</phase>
                  <goals>
                    <goal>deploy-module</goal>
                    <goal>start</goal>
                  </goals>
                </execution>
              </executions>
            </plugin>
          </plugins>
        </pluginManagement>
      </build>
```

```
    </profile>
  </profiles>
</project>
```

A quick glance will tell us a few things about this configuration

1. It will only be executed if a plugin actively adds the plugin to it's configuration (vis-a-vis the `pluginManagement` element). We had two options: place the profile in the EAR or configure using the `pluginManagement` element in the base project. We chose to do the former for posterity - EAR developers down the road may with to use the same configuration without all that typing... DRY.
2. Geronimo will use Jetty in some capacity. Which is correct - the plugin will download an assembly artifact containing this server and install it locally before use.
3. The server will attempt to launch a project's default-named EAR artifact (`moduleArchive`).
4. the `deploy-module` and `start` goals will be bound to the `install` phase.

   Despite the scarey-looking nature of the configuration, it does quite a lot. Our EAR POM can now use the plugin by simply asking to (and, pointing to it's own geronimo descriptor).

```
<project>
  ...
  <artifactId>calculator-stateless-ear</artifactId>
  ...
  <profiles>
    <profile>
      <id>geronimo</id>
      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.geronimo.plugins</groupId>
            <artifactId>geronimo-maven-plugin</artifactId>
            <configuration>
             <modulePlan>${project.build.outputDirectory}/META-INF/geronimo-application.xml</modulePl
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
</project>
```

   We put it under a matching profile `id`. Now when we run the command `mvn install -P geronimo` from the EAR project, it will inherit the configuration from the matching active profile in the parent POM - and run the EAR with Geronimo using Jetty.

   GOTCHA: Since the plugin is set in the EAR POM under a profile, the parent project's geronimo plugin must be configured under a profile that will be active - otherwise, Maven will not be able to find the parent configuration (since it is

inactive). The easiest way to do this is by making the profile `id`s or `activation`s match.

Running with Free Cargo

Although the above example using the `org.apache.geronimo.plugins:geronimo-maven-plugin` plugin is sufficient for Geronimo, what if you need to depoy to another application server, such as JBoss? Enter Cargo - an abstract way to `start`, `stop` and `deploy` to various types of Java EE contrainers. You can find all sorts of detailed information about Cargo at the project website: *http://cargo.codehaus.org*.

The Cargo Plugin

The `maven-cargo-plugin` has several goals:

- **start** - Start a container and optionally deploy deployables (WAR, EAR, etc)
- **stop** - Stop a container
- **deployer-deploy** - (aliased to cargo:deploy) Deploy a deployable to a running container
- **deployer-undeploy** - (aliased to cargo:undeploy) Undeploy a deployable from a running container
- **deployer-start** - Start a deployable already installed in a running container
- **deployer-stop** - Stop a deployed deployable without undeploying it
- **deployer-redeploy** - Undeploy and deploy again a deployable
- **uberwar** - Merge several WAR files into one
- **install** - Installs a container distribution on the file system. Note that this is performed automatically by the cargo:start goal

At the time of this writing, Cargo supported 9 containers and 16 versions.

- Geronimo 1.x - `geronimo1x`
- JBoss 3.x, 4.x - `jboss3x`, `jboss4x`
- Jetty 4.x, 5.x, 6.x - `jetty4x`, `jetty5x`, `jetty6x`
- jo! 1.x - `jo1x`
- OC4J 9.x - `oc4j9x`
- Orion 1.x, 2.x - `orion1x`, `orion2x`
- Resin 2.x, 3.x - `resin2x`, `resin3x`
- Tomcat 3.x, 4.x, 5.x - `tomcat3x`, `tomcat4x`, `tomcat5x`
- WebLogic 8.x - `weblogic8x`

JBoss Example

With Cargo, deploying to JBoss is as easy as the Geronimo example. Just like the Geronimo plugin downloaded the Jetty 6 container, so too have we configured the Cargo plugin to download a JBoss zip - so be prepared to wait a while if you have a slow connection - JBoss is approximately 80+ Megs. Once it is loaded and started, visit http://localhost:8080/ to see that the JBoss base page is running. *If it does not, check to ensure that the ZIP was downloaded successfully*. Hit Control-C to stop the server from running.

```
<project>
  ...
  <artifactId>calculator-stateless</artifactId>
  ...
  <profiles>
    <profile>
      <id>jboss</id>
      <build>
        <pluginManagement>
          <plugins>
            <plugin>
              <groupId>org.codehaus.cargo</groupId>
              <artifactId>cargo-maven2-plugin</artifactId>
              <version>0.3.1</version>
              <configuration>
                <container>
                  <containerId>jboss4x</containerId>
                  <zipUrlInstaller>
                   <url>http://downloads.sourceforge.net/jboss/jboss-4.0.5.GA.zip</url>
                       <installDir>${user.home}/jboss</installDir>
                    </zipUrlInstaller>
                </container>
                <configuration>
                  <home>${project.build.directory}/jboss/server</home>
                </configuration>
              </configuration>
              <executions>
                <execution>
                  <id>start-container</id>
                  <phase>install</phase>
                  <goals>
                    <goal>start</goal>
                  </goals>
                </execution>
              </executions>
            </plugin>
          </plugins>
        </pluginManagement>
      </build>
    </profile>
  </profiles>
</project>
```

Like the geronimo-maven-plugin the bulk of the configuration is done in the parent project under pluginManagement - so as not to attach the plugin to all inheriting POMs. We can see a few facts about this configuration:

1. The container used is JBoss 4 and can be downloaded in zip from from the given URL. Cargo will download and install JBoss for you in your user directory/jboss. Where as the Geronimo plugin tackles this issue by packaging up a container server as a special assembly, Cargo accepts the project's existing layout.
2. The configuration for the server will be put up under the build directory (probably target/jboss/server).
3. The cargo:start goal is to be bound to the install phase.

   Just like the Geronimo plugin, the EAR project need only put the org.codehaus.cargo:cargo-maven2-plugin plugin under the jboss profile. Note that there is no real necessity that these configurations live under profiles, we do it here out of convenience. In fact, the next section, integration testing, may well be desired default behavior.

   More interesting is the next profile, configurated to run as part an integration test.

Integration Testing

There exist a few ways to run integration tests in a Java EE application, but they all have a one step in common: before they begin any application servers must be started. So far we have seen how to start a server using Geronimo and Cargo. Now we will bind the Cargo start>> goal to the <<<pre-integration-test phase, and stop the server on post-integration-test.

The jboss-test profile is similar to the jboss profile with a couple changes. First: the install phase biding execution is replaced with the following two executions:

```
<executions>
  <execution>
    <id>start-container</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>start</goal>
    </goals>
  </execution>
  <execution>
    <id>stop-container</id>
    <phase>post-integration-test</phase>
    <goals>
      <goal>stop</goal>
    </goals>
  </execution>
</executions>
```

Second: we need to set a configuration element wait to false, otherwise the cargo-maven2-plugin will run (wait) until you manually shutdown with "Ctrl-C".

```
        <wait>false</wait>
```

And finally, the `pluginManagement` configures an execution to start a Selenium server before integration testing.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>selenium-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start-server</goal>
      </goals>
      <configuration>
        <background>true</background>
      </configuration>
    </execution>
  </executions>
</plugin>
```

You can learn more about Selenium as an integration test tool for browser client-side testing from the Selenium website (*http://openqa.org/selenium/*). We embed the test into a JUnit class file for this example. In order to get the integration test to run, we need to make a few changes to the default EAR build lifecycle. We achieve this by binding the `compiler:testCompile` goal to the `test-compile` phase, and finally binding `surefire:test` (unit test execution framework) to run at the `integration-test` phase.

The `cargo` and `selenium` plugins are, again, added here so that they will run inheriting the parent `configuration`.

```
<project>
  ...
  <artifactId>calculator-stateless-ear</artifactId>
  ...
  <profiles>
    <profile>
      <id>jboss-test</id>
      <build>
        <testResources>
          <testResource>
            <directory>src/test/resources</directory>
            <filtering>true</filtering>
          </testResource>
        </testResources>
        <plugins>
          <plugin>
            <groupId>org.codehaus.cargo</groupId>
            <artifactId>cargo-maven2-plugin</artifactId>
          </plugin>
          <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>selenium-maven-plugin</artifactId>
          </plugin>
          <plugin>
```

```
            <artifactId>maven-compiler-plugin</artifactId>
            <executions>
              <execution>
                <phase>test-compile</phase>
                <goals>
                  <goal>testCompile</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
          <plugin>
            <artifactId>maven-resources-plugin</artifactId>
            <executions>
              <execution>
                <phase>process-test-resources</phase>
                <goals>
                  <goal>testResources</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
          <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <executions>
              <execution>
                <phase>integration-test</phase>
                <goals>
                  <goal>test</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
      <properties>
        <selenium.browser.type>*firefox</selenium.browser.type>
      </properties>
    </profile>
  </profiles>
</project>
```

Our test also contains a resource file that we wish to filter, containing the type of browser we wish `selenium` to test our selenium script on. This is not required, but merely a good practice, for now we can change the browser tested upon by editing a property in our POM, leaving this configuration up to the builder to decide. So we add `src/test/resources` resource directory and `filter`, and bind `resources:testResources` to the `process-test-resources` phase. The `my-selenium-test.properties` file is this:

```
browser=${selenium.browser.type}
```

But after the `process-test-resources` phase will be used by the `test` as this:

```
browser=*firefox
```

Our unit, test in question (`org.apache.geronimo.samples.calculator.SeleniumTest`) contains a single test:

```
public void testRunningCalculator()
  throws Exception
{
 // getBrowserType reads the my-selenium-test.properties file for the browser to run on
  DefaultSelenium selenium =
   new DefaultSelenium( "localhost", 4444, getBrowserType(), "http://localhost:8080/calculator-stateles

  selenium.start();

  selenium.open( "http://localhost:8080/calculator-stateless/sample-docu.jsp" );

  selenium.waitForPageToLoad( "5000" );

  assertEquals( "A Stateless Session Sample - Calculator", selenium.getTitle() );

  assertTrue( selenium.isTextPresent( "Calculator" ) );

  selenium.stop();
}
```

Our install execution will now run as follows when running `mvn install -P jboss-test` in the EAR project:

```
[INFO] [ear:generate-application-xml]
[INFO] Generating application.xml
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [resources:testResources {execution: default}]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile {execution: default}]
[INFO] Compiling 1 source file to ~/calculator-stateless-pojo/calculator-stateless-ear/target/test-classe
[INFO] [ear:ear]
...
```

Once the `ear` is packaged, cargo is started at the `pre-integration-test` phase.

```
[INFO] [cargo:start {execution: start-container}]
[INFO] [neLocalConfiguration] Configuring JBoss using the [default] server configuration
[INFO] [talledLocalContainer] Parsed JBoss version = [4.0.5]
[INFO] [stalledLocalDeployer] Deploying [~/calculator-stateless-pojo/calculator-stateless-ear/target/cal
[INFO] [talledLocalContainer] JBoss 4.0.5 starting...
[INFO] [talledLocalContainer] 23:56:14,100 INFO  [Server] Starting JBoss (MX MicroKernel)...
...
[INFO] [talledLocalContainer] JBoss 4.0.5 started on port [8080]
```

The selenium server is also started at the `pre-integration-test` phase. This plugin has no shutdown phase, and so will shutdown with the Maven lifecycle is complete. *C'est la vie*.

```
[INFO] [selenium:start-server {execution: default}]
[INFO] Starting Selenium server...
[INFO] User extensions: ~/calculator-stateless-pojo/calculator-stateless-ear/target/selenium/user-extens
[INFO] 00:38:15,897 INFO  [org.mortbay.http.HttpServer] Version Jetty/0.8.1
```

```
...
[INFO] 00:38:16,655 INFO  [org.mortbay.util.Credential] Checking Resource aliases
[INFO] Selenium server started
```

The SeneliumTest case executes as part of the `intergration-test`. We used the `surefire:test` goal for that. This is a good example of the power of `goals` - since good goals are atomic, we are free to move them anywhere in the lifecycle - even re-purpose them to other uses (`surefire:test` traditionally only runs in the unit `test` phase.

```
[INFO] [surefire:test {execution: default}]
[INFO] Surefire report directory: ~/calculator-stateless-pojo/calculator-stateless-ear/target/surefire-r

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.apache.geronimo.samples.calculator.SeleniumTest
[INFO] queryString = cmd=getNewBrowserSession&1=*firefox&2=http%3A%2F%2Flocalhost%3A8080%2Fcalculator-st
[INFO] Preparing Firefox profile...
[INFO] Launching Firefox...
[INFO] 01:23:54,994 INFO  [org.mortbay.util.Container] Started HttpContext[/,/]
[INFO] Got result: OK,1176963830527
[INFO] queryString = cmd=open&1=http%3A%2F%2Flocalhost%3A8080%2Fcalculator-stateless%2Fsample-docu.jsp&s
[INFO] Got result: OK
[INFO] queryString = cmd=waitForPageToLoad&1=5000&sessionId=1176963830527
[INFO] Got result: OK
[INFO] queryString = cmd=getTitle&sessionId=1176963830527
[INFO] Got result: OK,A Stateless Session Sample - Calculator
[INFO] queryString = cmd=isTextPresent&1=Calculator&sessionId=1176963830527
[INFO] Got result: OK,true
[INFO] queryString = cmd=testComplete&sessionId=1176963830527
[INFO] Killing Firefox...
[INFO] Got result: OK
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.052 sec
...
```

Finally, we can stop the application server on the `post-integration-test` phase.

```
[INFO] [cargo:stop {execution: stop-container}]
[INFO] [talledLocalContainer] JBoss 4.0.5 is stopping...
[INFO] [talledLocalContainer] Shutdown message has been posted to the server.
[INFO] [talledLocalContainer] Server shutdown may take a while - check logfiles for completion
[INFO] [talledLocalContainer] Shutdown complete
[INFO] [talledLocalContainer] Halting VM
[INFO] [talledLocalContainer] JBoss 4.0.5 is stopped
```

Passing all tests, the EAR is installed as normal. Had it failed, well, then it would be a good thing we did not install a broken artifact!

Tips & Tricks

Use Profiles for Dev, Test, Prod

A useful tactic for enterprise environments is to create a profile per server type. There are a few ways to address this. The most straightforward approach is to create a profile in the POM with tweaks for the various environments. For example, an EJB project may have a `persistance.xml` file which needs to flex for different environments - for example, with a different JDBC url. Like we did with the `my-selenium-test.properties` file above, filter by resource location - replace the JDBC string in the XMl file with a descriptive property, for example `\${ejb.jdbc.url}`. Each profile can contain a different string.

Moreover, it might behoov you to deploy these projects under different names to tell them apart. Here is where classifiers come in handy. You can now generate and deploy different EJB projects for different servers.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-ejb-plugin</artifactId>
        <configuration>
          <ejbVersion>3.0</ejbVersion>
          <classifier>${server.type}</classifier>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <profiles>
    <profile>
      <id>prod</id>
      <properties>
        <ejb.jdbc.url>jdbc:odbc:prod;UID=prod;PWD=p4sswOrd</ejb.jdbc.url>
        <server.type>prod</server.type>
      </properties>
    </profile>
    <profile>
      <id>test</id>
      <properties>
        <ejb.jdbc.url>jdbc:odbc:test;UID=test;PWD=p4sswOrd</ejb.jdbc.url>
        <server.type>test</server.type>
      </properties>
    </profile>
    <profile>
      <id>dev</id>
      <properties>
        <ejb.jdbc.url>jdbc:derby:dev</ejb.jdbc.url>
        <server.type>dev</server.type>
```

```
        </properties>
      </profile>
    </profiles>
  <project>
```

This works for small or finite numbers of server. If your demands require more flexibility, then externalize the profiles into a `profiles.xml` file. This is a file that sits in the basedir, that has a top `profiles` element on cotaining any number of `profile` child elements.

Summary

An important peice not to overlook is the easy at which a Java EE project managed by Maven can switch entire execution environments by the mere configuration of a few plugin. In the examples in this chapter we creates an deployed a Geronimo project effortlessly on the Jetty 6 runtime - and with a few deft configurations, created a profile capable of deploying the project to JBoss. Maven cannot take complete credit, of course, since it is mostly a testement to the Java EE framework. But never before has swapping framework implementations and servers been so conceptually simple.

There are several ways to do the same thing in Maven, as the Geronimo deployment example illustrates: either use the Apache project' plugin, or utilize Cargo instead. The mechanism is up to you but do not forget the important point: from your user's point of view, the trigger is the same... "mvn install".

CHAPTER 14

Appendix: POM Details

Introduction

What is the POM?

POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named `pom.xml`. When in the presence of Maven folks, speaking of a project is speaking in the philosophical sense, beyond a mere collection of files containing code. A project contains configuration files, as well as the developers involved and the roles they play, the defect tracking system, the organization and licenses, the URL of where the project lives, the project's dependencies, and all of the other little pieces that come into play to give code life. It is a one-stop-shop for all things concerning the project. In fact, in the Maven world, a project need not contain any code at all, merely a `pom.xml`.

Quick Overview

This is a listing of the elements directly under the POMs project element. Notice that `modelVersion` contains 4.0.0. That is currently the only supported POM version for Maven 2, and is always required.

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
```

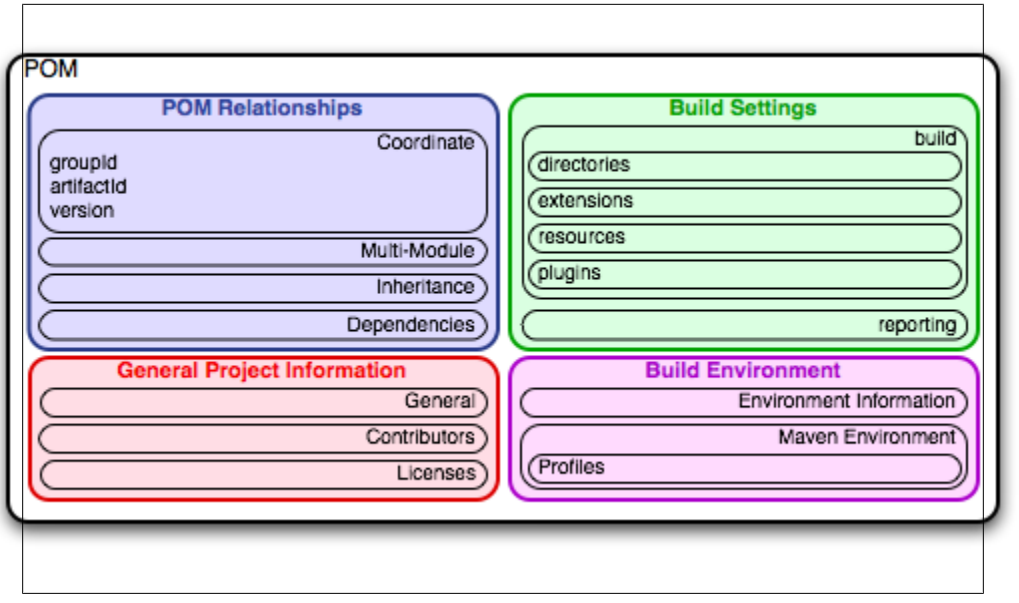


Figure 14-1. A small model of the POM

```
    <build>...</build>
    <reporting>...</reporting>

    <!-- More Project Information -->
    <name>...</name>
    <description>...</description>
    <url>...</url>
    <inceptionYear>...</inceptionYear>
    <licenses>...</licenses>
    <organization>...</organization>
    <developers>...</developers>
    <contributors>...</contributors>

    <!-- Environment Settings -->
    <issueManagement>...</issueManagement>
    <ciManagement>...</ciManagement>
    <mailingLists>...</mailingLists>
    <scm>...</scm>
    <prerequisites>...</prerequisites>
    <repositories>...</repositories>
    <pluginRepositories>...</pluginRepositories>
    <distributionManagement>...</distributionManagement>
    <profiles>...</profiles>
</project>
```

The XSD Schemas

The POM XSD is available at *http://maven.apache.org/maven-v4_0_0.xsd* and the Settings XSD is available at *http://maven.apache.org/xsd/settings-1.0.0.xsd*. They can be referenced in the POM by setting the `<project>` element like so:

```
<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">

  ...
</project>
```

And settings as such:

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
</settings>
```

Finally, this cheat sheet (*http://sonatype.com/book/images/pom-details/pom-cheat-sheet.png*) may help you visualize the layout.

The Basics

The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process. It is, effectively, the declarative manifestation of the "who", "what", and "where", while the build lifecycle is the "when" and "how". That is not to say that the POM cannot affect the flow of the lifecycle it can. For example, by configuring the `maven-antrun-plugin`, one can effectively embed ant tasks inside of the POM. It is ultimately a declaration, however. Where as a `build.xml` tells ant precisely what to do when it is run (procedural), a POM states its configuration (declarative). If some external force causes the lifecycle to skip the ant plugin execution, it will not stop the plugins that are executed from doing their magic. This is unlike a `build.xml` file, where tasks are almost always dependant on the lines executed before it.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>
```

Maven Coordinates

The POM defined above is the minimum that Maven 2 will allow. `groupId:artifactId:version` are all required fields (although, groupId and version need not be explicitly defined if they are inherited from a parent more on inheritance later). The three fields act much like an address and timestamp in one. This marks a specific place in a repository, acting like a coordinate system for Maven projects.

- **groupId**: This is generally unique amongst an organization or a project. For example, all core Maven artifacts do (well, should) live under the groupId org.apache.maven. Group ID's do not necessarily use the dot notation, for example, the junit project. Note that the dot-notated groupId does not have to correspond to the package structure that the project contains. It is, however, a good practice to follow. When stored within a repository, the group acts much like the Java packaging structure does in an operating system. The dots are replaced by OS specific directory separators (such as / in Unix) which becomes a relative directory structure from the base repository. In the example given, the `org.codehaus.mojo` group lives within the directory `$M2_REPO/org/codehaus/mojo`.
- **artifactId**: The artifactId is generally the name that the project is known by. Although the groupId is important, people within the group will rarely mention the groupId in discussion (they are often all be the same ID, such as the Codehaus Mojo (*http://mojo.codehaus.org/*) project groupId: `org.codehaus.mojo`). It, along with the groupId, create a key that separates this project from every other project in the world (at least, it should :)). Along with the groupId, the artifactId fully defines the artifacts living quarters within the repository. In the case of the above project, `my-project` lives in `$M2_REPO/org/codehaus/mojo/my-project`.
- **version**: This is the last piece of the naming puzzle. `groupId:artifactId` denote a single project but they cannot delineate which incarnation of that project we are talking about. Do we want the `junit:junit` of today (version 4), or of four years ago (version 2)? In short: code changes, those changes should be versioned, and this element keeps those versions in line. It is also used within an artifacts repository to separate versions from each other. `my-project` version 1.0 files live in the directory structure `$M2_REPO/org/codehaus/mojo/my-project/1.0`.

  The three elements given above point to a specific version of a project letting Maven knows *who* we are dealing with, and *when* in its software lifecycle we want them.
- **packaging**: Now that we have our address structure of `groupId:artifactId:version`, there is one more standard label to give us a really complete address. That is the project's artifact type. In our case, the example POM for `org.codehaus.mojo:my-project:1.0` defined above will be packaged as a `jar`. We could make it into a `war` by declaring a different packaging:

```
<project>
  ...
  <packaging>war</packaging>
```

```
  ...
</project>
```

When no packaging is declared, Maven assumes the artifact is the default: `jar`. The valid types are Plexus role-hints (read more on Plexus for a explanation of roles and role-hints) of the component role `org.apache.maven.lifecycle.mapping.LifecycleMapping`. The current core packaging values are: `pom`, `jar`, `maven-plugin`, `ejb`, `war`, `ear`, `rar`, `par`. These define the default list of goals which execute to each corresponding build lifecycle stage for a particular package structure.

You will sometimes see Maven print out a project coordinate as `groupId:artifactId:packaging:version`.

- **classifier**: You may occasionally find a fifth element on the coordinate, and that is the `classifier`. We will visit the classifier later, but for now it suffices to know that those kinds of projects are displayed as `groupId:artifactId:packaging:classifier:version`.

POM Relationships

One powerful aspect of Maven is in its handling of project relationships; that includes dependencies (and transitive dependencies), inheritance, and aggregation (multi-module projects). Dependency management has a long tradition of being a complicated mess for anything but the most trivial of projects. *"Jarmageddon"* quickly ensues as the dependency tree becomes large and complicated. *"Jar Hell"* follows, where versions of dependencies on one system are not equivalent to versions as those developed with, either by the wrong version given, or conflicting versions between similarly named jars. Maven solves both problems through a common local repository from which to link projects correctly, versions and all.

Dependencies

The cornerstone of the POM is its dependency list. Most every project depends upon others to build and run correctly, and if all Maven does for you is manage this list for you, you have gained a lot. Maven downloads and links the dependencies for you on compilation and other goals that require them. As an added bonus, Maven brings in the dependencies of those dependencies (transitive dependencies), allowing your list to focus solely on the dependencies your project requires.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <type>jar</type>
      <scope>test</scope>
      <classifier>memory</classifier>
```

```
      <optional>true</optional>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

- **groupId**, **artifactId**, **version**: These elements are self-explanatory, and you will see them often. This trinity represents the coordinate of a specific project in time, demarcating it as a dependency of this project. You may be thinking: "This means that my project can only depend upon Maven artifacts!" The answer is, "Of course, but that's a good thing." This forces you to depend solely on dependencies that Maven can manage. There are times, unfortunately, when a project cannot be downloaded from the central Maven repository. For example, a project may depend upon a jar that has a closed-source license which prevents it from being in a central repository. There are three methods for dealing with this scenario.

  1. Install the dependency locally using the install plugin. The method is the simplest recommended method. For example:

     ```
     mvn install:install-file Dfile=non-maven-proj.jar DgroupId=some.group DartifactId=non-maven-p
     ```

     Notice that an address is still required, only this time you use the command line and the install plugin will create a POM for you with the given address.

  2. Create your own repository and deploy it there. This is a favorite method for companies with an intranet and need to be able to keep everyone in synch. There is a Maven goal called `deploy:deploy-file` which is similar to the `install:install-file` goal (read the plugin's goal page for more information).

  3. Set the dependency scope to `system` and define a `systemPath`. This is not recommended, however, but leads us to explaining the following elements:

- **type**: Corresponds to the dependant artifact's `packaging` type. This defaults to `jar`. While it usually represents the extension on the filename of the dependency, that is not always the case. A type can be mapped to a different extension. Some examples are `ejb-client` and `test-jar`, whose extension is actually `jar`.

- **scope**: This element refers to the classpath of the task at hand (compiling and runtime, testing, etc.) as well as how to limit the transitivity of a depedency. There are five scopes available:

  — **compile** - this is the default scope, used if none is specified. Compile dependencies are available in all classpaths.

  — **provided** - this is much like compile, but indicates you expect the JDK or a container to provide it. It is only available on the compilation classpath, and is not transitive.

  — **runtime** - this scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.

— **test** - this scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.

— **system** - this scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

- **systemPath**: is used *only* if the the dependency `scope` is `system`. Otherwise, the build will fail if this element is set. The path must be absolute, so it is recommended to use a property to specify the machine-specific path (more on `properties` below), such as `$java.home/lib`. Since it is assumed that system scope dependencies are installed *a priori*, Maven will not check the repositories for the project, but instead checks to ensure that the file exists. If not, Maven will fail the build and suggest that you download and install it manually.
- **classifier**: This is an optional fifth element to define the coordinate of a dependency that contains a classifier. Classifiers were briefly mentioned above, and will be covered in more detail below.
- **optional**: Marks optional a dependency when this project itself is a dependency. Confused? For example, imagine a project `A` that depends upon project `B` to compile a portion of code that may not be used at runtime, then we may have no need for project `B` for all project. So if project `X` adds project `A` as its own dependency, then Maven will not need to install project `B` at all. Symbolically, if `=>` represents a required dependency, and `-->` represents optional, although `A=>B` may be the case when building A `X=>A-->B` would be the case when building `X`.

  In the shortest terms, `optional` lets other projects know that, when you use this project, you do not require this dependency in order to work correctly.

Exclusions.

Exclusions explicitly tell Maven that you don't want to include the specified project that is a dependency of this dependency (in other words, its transitive dependency). For example, the `maven-embedder` requires `maven-core`, and we do not wish to use it or its dependencies, then we would add it as an `exclusion`.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-embedder</artifactId>
      <version>2.0</version>
      <exclusions>
        <exclusion>
          <groupId>org.apache.maven</groupId>
          <artifactId>maven-core</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
```

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

- **exclusions**: Exclusions contain one or more `exclusion` elements, each containing a `groupId` and `artifactId` denoting a dependency to exclude. Unlike `optional`, which may or may not be installed and used, `exclusions` actively remove themselves from the dependency tree.

Inheritance

One powerful addition that Maven brings to build management is the concept of project inheritance. Although in build systems such as Ant, inheritance can certainly be simulated, Maven has gone the extra step in making project inheritance explicit to the project object model.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <packaging>pom</packaging>
</project>
```

The `packaging` type required to be `pom` for *parent* and *aggregation* (multi-module) projects. These types define the goals bound to a set of lifecycle stages. For example, if packaging is `jar`, then the `package` phase will execute the `jar:jar` goal. If the packaging is `pom`, the goal executed will be `site:attach-descriptor`. Now we may add values to the parent POM, which will be inherited by its children. The elements in the parent POM that are inherited by its children are:

- dependencies
- developers and contributors
- plugin lists
- reports lists
- plugin executions with matching ids
- plugin configuration

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>my-parent</artifactId>
    <version>2.0</version>
    <relativePath>../my-parent</relativePath>
  </parent>
  <artifactId>my-project</artifactId>
</project>
```

Notice the `relativePath` element. It is not required, but may be used as a signifier to Maven to first search the path given for this project's parent, before searching the local and then remote repositories.

The Super POM.

Similar to the inheritance of objects in object oriented programming, POMs that extend a parent POM inherit certain values from that parent. Moreover, just as Java objects ultimately inherit from `java.lang.Object`, all Project Object Models inherit from a base Super POM.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>

  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      <layout>default</layout>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
    </pluginRepository>
  </pluginRepositories>

  <build>
    <directory>target</directory>
    <outputDirectory>target/classes</outputDirectory>
    <finalName>${artifactId}-${version}</finalName>
    <testOutputDirectory>target/test-classes</testOutputDirectory>
    <sourceDirectory>src/main/java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>src/test/java</testSourceDirectory>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
    </resources>
```

```
  <testResources>
    <testResource>
      <directory>src/test/resources</directory>
    </testResource>
  </testResources>
</build>

<reporting>
  <outputDirectory>target/site</outputDirectory>
</reporting>

<profiles>
  <profile>
    <id>release-profile</id>
    <activation>
      <property>
        <name>performRelease</name>
        <value>true</value>
      </property>
    </activation>

    <build>
      <plugins>
        <plugin>
          <inherited>true</inherited>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-source-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-sources</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>

        <plugin>
          <inherited>true</inherited>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-javadoc-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-javadocs</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>

        <plugin>
          <inherited>true</inherited>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-deploy-plugin</artifactId>
```

```
            <configuration>
              <updateReleaseInfo>true</updateReleaseInfo>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>

</project>
```

You can take a look at how the Super POM affects your Project Object Model by creating a minimal `pom.xml` and executing on the command line: `mvn help:effective-pom`

Dependency Management.

Besides inheriting certain top-level elements, parents have elements to configure values for child POMs that do not affect the parents own build lifecycle. One of those elements is `dependencyManagement`.

- **dependencyManagement**: is used only by parent POMs to help manage dependency information across all of its children. If the `my-parent` project uses `dependencyManagement` to define a dependency on `junit:junit:4.0`, then POMs inheriting from this one can set their dependency giving the `groupId=junit` and `artifactId=junit` only, then Maven will fill in the `version` set by the parent. The benefits of this method are obvious. Dependency details can be set in one central location, which will propagate to all inheriting POMs.

Aggregation (or Multi-Module)

A project with modules is known as a multimodule, or aggregator project. Modules are projects that this POM lists, and are executed as a group. An `pom` packaged project may aggregate the build of a set of projects by listing them as modules, which are relative directories to those projects.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <modules>
    <module>my-project<module>
  </modules>
</project>
```

A final note on Inheritance v. Aggregation.

Inheritance and aggregation create a nice dynamic to control builds through a single, high-level POM. You will often see projects that are both parents and aggregators. For example, the entire maven core runs through a single base POM `org.apache.maven:maven`, so building the Maven project can be executed by a single command: `mvn compile`. However, although both POM projects, an aggregator project and a parent project are not one in the same and should not be confused. A POM project

may be inherited from - but does not necessarily have - any modules that it aggregates. Conversely, a POM project may aggregate projects that do not inherit from it.

Properties

Properties are the last required piece in understanding POM basics. Maven properties are value placeholder, like properties in Ant. Their values are accessible anywhere within a POM by using the notation `${X}`, where `X` is the property. They come in five different styles:

1. `env.X`: Prefixing a variable with "env." will return the shells environment variable. For example, `${env.PATH}` contains the $path environment variable (`%PATH%` in Windows).
2. `project.x`: A dot (.) notated path in the POM will contain the corresponding elements value. For example: `<project><version>1.0</version></project>` is accessible via `${project.version}`.
3. `settings.x`: A dot (.) notated path in the `settings.xml` will contain the corresponding elements value. For example: `<settings><offline>false</offline></settings>` is accessible via `${settings.offline}`.
4. Java System Properties: All properties accessible via `java.lang.System.getProperties()` are available as POM properties, such as `${java.home}`.
5. `x`: Set within a <properties /> element or an external files, the value may be used as `${someVar}`.

Build Settings

Beyond the basics of the POM given above, there are two more elements that must be understood before claiming basic competency of the POM. They are the `build` element, that handles things like declaring your project's directory structure and managing plugins; and the `reporting` element, that largely mirrors the build element for reporting purposes.

Build

According to the POM 4.0.0 XSD, the `build` element is conceptually divided into two parts: there is a `BaseBuild` type which contains the set of elements common to both `build` elements (the top-level build element under `project` and the build element under `profiles`, covered below); and there is the `Build` type, which contains the `BaseBuild` set as well as more elements for the top level definition. Let us begin with an analysis of the common elements between the two.

Note: These different `build` *elements may be denoted "project build" and "profile build".*

```
<project>
  <!-- "Project Build" contains more elements than just the BaseBuild set -->
  <build>...</build>

  <profiles>
    <profile>
      <!-- "Profile Build" contains a subset of "Project Build"s elements -->
      <build>...</build>
    </profile>
  </profiles>
</project>
```

The BaseBuild Element Set

`BaseBuild` is exactly as it sounds: the base set of elements between the two `build` elements in the POM.

```
<build>
  <defaultGoal>install</defaultGoal>
  <directory>${basedir}/target</directory>
  <finalName>${artifactId}-${version}</finalName>
  <filters>
    <filter>filters/filter1.properties</filter>
  </filters>
  ...
</build>
```

- **defaultGoal**: the default goal or phase to execute if none is given. If a goal is given, it should be defined as it is in the command line (such as `jar:jar`). The same goes for if a phase is defined (such as install).
- **directory**: This is the directory where the build will dump its files or, in Maven parlance, the build's target. It aptly defaults to `${basedir}/target`.
- **finalName**: This is the name of the bundled project when it is finally built (sans the file extension, for example: `my-project-1.0.jar`). It defaults to `${artifactId}-${version}`. The term "finalName" is kind of a misnomer, however, as plugins that build the bundled project have every right to ignore/modify this name (but they usually do not). For example, if the `maven-jar-plugin` is configured to give a jar a `classifier` of `test`, then the actual jar defined above will be built as `my-project-1.0-test.jar`.
- **filter**: Defines `*.properties` files that contain a list of properties that apply to resources which accept their settings (covered below). In other words, the `"name=value"` pairs defined within the filter files replace `${name}` strings within resources on build. The example above defines the `filter1.properties` file under the `filter/` directory. Mavens default filter directory is `${basedir}/src/main/filters/`.

  For a more comprehensive look at what filters are and what they can do, take a look at the quick start guide (*/guides/getting-started*).

Resources.

Another feature of `build` elements is specifying where resources exist within your project. Resources are not (usually) code. They are not compiled, but are items meant to be bundled within your project or used for various other reasons, such as code generation.

For example, a Plexus project requires a `configuration.xml` file (which specifies component configurations to the container) to live within the `META-INF/plexus` directory. Although we could just as easily place this file within `src/main/resource/META-INF/plexus`, we want instead to give Plexus its own directory of `src/main/plexus`. In order for the JAR plugin to bundle the resource correctly, you would specify resources similar to the following:

```
<project>
  <build>
    ...
    <resources>
      <resource>
        <targetPath>META-INF/plexus</targetPath>
        <filtering>false</filtering>
        <directory>${basedir}/src/main/plexus</directory>
        <includes>
          <include>configuration.xml</include>
        </includes>
        <excludes>
          <exclude>**/*.properties</exclude>
        </excludes>
      </resource>
    </resources>
    <testResources>
      ...
    </testResources>
    ...
  </build>
</project>
```

- **resources**: is a list of resource elements that each describe what and where to include files associated with this project.
- **targetPath**: Specifies the directory structure to place the set of resources from a build. Target path defaults to the base directory. A commonly specified target path for resources that will be packaged in a JAR is META-INF.
- **filtering**: is `true` or `false`, denoting if filtering is to be enabled for this resource. Note, that filter `*.properties` files do not have to be defined for filtering to occur resources can also use properties that are by default defined in the POM (such as `${project.version}`), passed into the command line using the "-D" flag (for example, `"-Dname=value"`) or are explicitly defined by the properties element. Filter files were covered above.
- **directory**: This element's value defines where the resources are to be found. The default directory for a build is `${basedir}/src/main/resources`.

- **includes**: A set of files patterns which specify the files to include as resources under that specified directory, using * as a wildcard.
- **excludes**: The same structure as `includes`, but specifies which files to ignore. In conflicts between `include` and `exclude`, `exclude` wins.
- **testResources**: The `testResources` element block contains `testResource` elements. Their definitions are similar to `resource` elements, but are naturally used during test phases. The one difference is that the default (Super POM defined) test resource directory for a project is `${basedir}/src/test/resources`. Test resources are not deployed.

Plugins.

```
<project>
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.0</version>
        <extensions>false</extensions>
        <inherited>true</inherited>
        <configuration>
          <classifier>test</classifier>
        </configuration>
        <dependencies>...</dependencies>
        <executions>...</executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Beyond the standard coordinate of `groupId:artifactId:version`, there are elements which configure the plugin or this builds interaction with it.

- **extensions**: `true` or `false`, whether or not to load extensions of this plugin. It is by default false. Extensions are covered later in this document.
- **inherited**: `true` or `false`, whether or not this plugin configuration should apply to POMs which inherit from this one.
- **configuration**: This is specific to the individual plugin. Without going too in depth into the mechanics of how plugins work, suffice it to say that whatever properties that the plugin Mojo may expect (these are getters and setters in the Java Mojo bean) can be specified here. In the above example, we are setting the classifier property to test in the `maven-jar-plugin`'s Mojo. It may be good to note that all configuration elements, wherever they are within the POM, are intended to pass values to another underlying system, such as a plugin. In other words: values within a `configuration` element are never explicitly required by the POM schema, but a plugin goal has every right to require configuration values.

- **dependencies**: Dependencies are seen a lot within the POM, and are an element under all plugins element blocks. The dependencies have the same structure and function as under that base build. The major difference in this case is that instead of applying as dependencies of the project, they now apply as dependencies of the plugin that they are under. The power of this is to alter the dependency list of a plugin, perhaps by removing an unused runtime dependency via `exclusions`, or by altering the version of a required dpendency. See above under **Dependencies** for more information.
- **executions**: It is important to keep in mind that a plugin may have multiple goals. Each goal may have a separate configuration, possibly even binding a plugin's goal to a different phase altogether. `executions` configure the `execution` of a plugins goals.

  For example, suppose you wanted to bind the `anrun:run` goal to the `verify` phase. We want the task to echo the build directory, as well as avoid passing on this configuration to its children (assuming it is a parent) by setting `inherited` to `false`. You would get an `execution` like this:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>

        <executions>
          <execution>
            <id>echodir</id>
            <goals>
              <goal>run</goal>
            </goals>
            <phase>verify</phase>
            <inherited>false</inherited>
            <configuration>
              <tasks>
                <echo>Build Dir: ${project.build.directory}</echo>
              </tasks>
            </configuration>
          </execution>
        </executions>

      </plugin>
    </plugins>
  </build>
</project>
```

- **id**: Self explanatory. It specifies this execution block between all of the others. When the phase is run, it will be shown in the form: `[plugin:goal {execution: id}]`. In the case of this example: `[antrun:run {execution: echodir}]`
- **goals**: Like all pluralized POM elements, this contains a list of singular elements. In this case, a list of plugin `goals` which are being specified by this `execution` block.

- **phase**: This is the phase that the list of goals will execute in. This is a very powerful option, allowing one to bind any goal to any phase in the build lifecycle, altering the default behavior of Maven.
- **inherited**: Like the `inherited` element above, setting this false will supress Maven from passing this execution onto its children. This element is only meaningful to parent POMs.
- **configuration**: Same as above, but confines the configuration to this specific list of goals, rather than all goals under the plugin.

Plugin Management.

- **pluginManagement**: is an element that is seen along side plugins. Plugin Management contains plugin elements in much the same way, except that rather than configuring plugin information for this particular project build, it is intended to configure project builds that inherit from this one. However, this only configures plugins that are actually referenced within the plugins element in the children. The children have every right to override `pluginManagement` definitions.

```
<project>
  <build>
    ...
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <version>2.0</version>
          <executions>
            <execution>
              <id>pre-process-classes</id>
              <phase>compile</phase>
              <goals>
                <goal>jar</goal>
              </goals>
              <configuration>
                <classifier>pre-process</classifier>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </pluginManagement>
    ...
  </build>
</project>
```

If we added these specifications to the plugins element, they would apply only to a single POM. However, if we apply them under the `pluginManagement` element, then this POM *and all inheriting POMs* that add the `maven-jar-plugin` to the build will get the `pre-process-classes` execution as well. So rather than the above mess included in every child `pom.xml`, only the following is required:

```
<project>
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
      </plugin>
    </plugins>
    ...
  </build>
</project>
```

The Build Element Set

The `Build` type in the XSD denotes those elements that are available only for the "project build". Despite the number of extra elements (six), there are really only two groups of elements that project build contains that are missing from the profile build: directories and extensions.

Directories.

The set of directory elements live in the parent build element, which set various directory structures for the POM as a whole. Since they do not exist in profile builds, these cannot be altered by profiles.

```
<project>
  <build>
    <sourceDirectory>${basedir}/src/main/java</sourceDirectory>
    <scriptSourceDirectory>${basedir}/src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>${basedir}/src/test/java</testSourceDirectory>
    <outputDirectory>${basedir}/target/classes</outputDirectory>
    <testOutputDirectory>${basedir}/target/test-classes</testOutputDirectory>
    ...
  </build>
</project>
```

If the values of a `*Directory` element above is set as an absolute path (when their properties are expanded) then that directory is used. Otherwise, it is relative to the base build directory: `${basedir}`.

Extensions.

Extensions are a list of artifacts that are to be used in this build. They will be included in the running build's classpath. They can enable extensions to the build process (such as add an ftp provider for the Wagon transport mechanism), as well as make plugins active which make changes to the build lifecycle. In short, extensions are artifacts that activated during build. The extensions do not have to actually do anything nor contain a Mojo. For this reason, extensions are excellent for specifying one out of multiple implementations of a common plugin interface.

```
<project>
  <build>
    ...
```

```
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
        <artifactId>wagon-ftp</artifactId>
        <version>1.0-alpha-3</version>
      </extension>
    </extensions>
    ...
  </build>
</project>
```

Reporting

Reporting contains the elements that correspond specifically for the `site` generation phase. Certain Maven plugins can generate reports defined and configured under the reporting element, for example: generating Javadoc reports. Much like the build element's ability to configure plugins, reporting commands the same ability. The glaring difference is that rather than fine-grained control of plug-in goals within the executions block, reporting configures goals within `reportSet` elements. And the subtler difference is that a plugin `configuration` under the `reporting` element works as `build` plugin `configuration`, although the opposite is not true (a `build` plugin `configuration` does not affect a `reporting` plugin).

Possibly the only item under the `reporting` element that would not be familiar to someone who understood the `build` element is the Boolean `excludeDefaults` element. This element signifies to the site generator to exclude reports normally generated by default. When a site is generated via the `site` build cycle, a *Project Info* section is placed in the left-hand menu, chock full of reports, such as the **Project Team** report or **Dependencies** list report. These report goals are generated by `maven-project-info-reports-plugin`. Being a plugin like any other, it may also be suppressed in the following, more verbose, way, which effectively turns off project-info reports.

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <outputDirectory>${basedir}/target/site</outputDirectory>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <reportSets>
          <reportSet></reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

The other difference is the `outputDirectory` element under `plugin`. In the case of reporting, the output directory is `${basedir}/target/site` by default.

Report Sets

It is important to keep in mind that an individual plugin may have multiple goals. Each goal may have a separate configuration. Report sets configure execution of a report plugins goals. Does this sound familiar deja-vu? The same thing was said about builds `execution` element with one difference: you cannot bind a report to another phase. Sorry.

For example, suppose you wanted to configure the `javadoc:javadoc` goal to link to "*http://java.sun.com/j2se/1.5.0/docs/api/*", but only the `javadoc` goal (not the goal `maven-javadoc-plugin:jar`). We would also like this configuration passed to its children, and set `inherited` to true. The `reportSet` would resemble the following:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        ...
        <reportSets>
          <reportSet>
            <id>sunlink</id>
            <reports>
              <report>javadoc</report>
            </reports>
            <inherited>true</inherited>
            <configuration>
              <links>
                <link>http://java.sun.com/j2se/1.5.0/docs/api/</link>
              </links>
            </configuration>
          </reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Between build `executions` and reporting `reportSets`, it should be clear now as to why they exist. In the simplest sense, they drill down in configuration. The POM must have a way not only to configure plugins, but they also must configure individual goals of those plugins. That is where these elements come in, giving the POM ultimate granularity in control of its build destiny.

More Project Information

Although the above information is enough to get a firm grasp on POM authoring, there are far more elements to make developer's live easier. Many of these elements are related to site generation, but like all POM declarations, they may be used for anything, depending upon how certain plugins use it. The following are the simplest elements:

- **name**: Projects tend to have conversational names, beyond the `artifactId`. The Sun engineers did not refer to their project as "java-1.5", but rather just called it "Tiger". Here is where to set that value.
- **description**: Description of a project is always good. Although this should not replace formal documentation, a quick comment to any readers of the POM is always helpful.
- **url**: The URL, like the name, is not required. This is a nice gesture for projects users, however, so that they know where the project lives.
- **inceptionYear**: This is another good documentation point. It will at least help you remember where you have spent the last few years of your life.

Licenses

```
<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>
```

Licenses are legal documents defining how and when a project (or parts of a project) may be used. Note that a project should list only licenses that may apply directly to this project, and not list licenses that apply to this project's dependencies. Maven currently does little with these documents other than displays them on generated sites. However, there is talk of flexing for different types of licenses, forcing users to accept license agreements for certain types of (non open source) projects.

- **name**, **url** and **comments**: are self explanatory, and have been encountered before in other capacities. The fourth license element is:
- **distribution**: This describes how the project may be legally distributed. The two stated methods are repo (they may be downloaded from a Maven repository) or manual (they must be manually installed).

Organization

Most projects are run by some sort of organization (business, private group, etc.). Here is where the most basic information is set.

```
<project>
  ...
  <organization>
    <name>Codehaus Mojo</name>
    <url>http://mojo.codehaus.org</url>
  </organization>
</project>
```

Developers

All projects consist of files that were created, at some time, by a person. Like the other systems that surround a project, so to do the people involved with a project have a stake in the project. Developers are presumably members of the project's core development. Note that, although an organization may have many developers (programmers) as members, it is not good form to list them all as developers, but only those who are immediately responsible for the code. A good rule of thumb is, if the person should not be contacted about the project, they need not be listed here.

```
<project>
  ...
  <developers>
    <developer>
      <id>eric</id>
      <name>Eric</name>
      <email>eredmond@codehaus.org</email>
      <url>http://eric.propellors.net</url>
      <organization>Codehaus</organization>
      <organizationUrl>http://mojo.codehaus.org</organizationUrl>
      <roles>
        <role>architect</role>
        <role>developer</role>
      </roles>
      <timezone>-6</timezone>
      <properties>
        <picUrl>http://tinyurl.com/prv4t</picUrl>
      </properties>
    </developer>
  </developers>
  ...
</project>
```

- **id**, **name**, **email**: These corrospond to the developer's ID (presumably some unique ID across an organization), the developer's name and email address.
- **organization**, **organizationUrl**: As you probably guessed, these are the developer's organization name and it's URL, respectively.
- **roles**: A `role` should specify the standard actions that the person is responsible for. Like a single person can wear many hats, a single person can take on multiple `roles`.
- **timezone**: A numerical offset in hours from GMT where the developer lives.
- **properties**: This element is where any other properties about the person goes. For example, a link to a personal image or an instant messenger handle. Different plugins may use these properties, or they may simply be for other developers who read the POM.

Contributors

Contributors are like developers yet play an ancillary role in a project's lifecycle. Perhaps the contributor sent in a bug fix, or added some important documentation. A healthy open source project will likely have more contributors than developers.

```
<project>
  ...
  <contributors>
    <contributor>
      <name>Noelle</name>
      <email>some.name@gmail.com</email>
      <url>http://noellemarie.com</url>
      <organization>Noelle Marie</organization>
      <organizationUrl>http://noellemarie.com</organizationUrl>
      <roles>
        <role>tester</role>
      </roles>
      <timezone>-5</timezone>
      <properties>
        <gtalk>some.name@gmail.com</gtalk>
      </properties>
    </contributor>
  </contributors>
  ...
</project>
```

Contributors contain the same set of elements than developers sans the `id` element.

Environment Settings

Issue Management

This defines the defect tracking system (*Bugzilla*, *TestTrack*, *ClearQuest*, etc) used. Although there is nothing stopping a plugin from using this information for something, its primarily used for generating project documentation.

```
<project>
  ...
  <issueManagement>
    <system>Bugzilla</system>
    <url>http://127.0.0.1/bugzilla</url>
  </issueManagement>
  ...
</project>
```

Continuous Integration Management

Continuous integration build systems based upon triggers or timings (such as, hourly or daily) have grown in favor over manual builds in the past few years. As build systems

have become more standardized, so have the systems that run the trigger those builds. Although the majority of the configuration is up to the specific program used (Continuum, Cruise Control, etc.), there are a few configurations which may take place within the POM. Maven has captured a few of the recurring settings within the set of notifier elements. A notifier is the manner in which people are notified of certain build statuses. In the following example, this POM is setting a notifier of type `mail` (meaning email), and configuring the email address to use on the specified triggers `sendOnError`, `sendOnFailure`, and not `sendOnSuccess` or `sendOnWarning`.

```
<project>
  ...
  <ciManagement>
    <system>continuum</system>
    <url>http://127.0.0.1:8080/continuum</url>
    <notifiers>
      <notifier>
        <type>mail</type>
        <sendOnError>true</sendOnError>
        <sendOnFailure>true</sendOnFailure>
        <sendOnSuccess>false</sendOnSuccess>
        <sendOnWarning>false</sendOnWarning>
        <configuration><address>continuum@127.0.0.1</address></configuration>
      </notifier>
    </notifiers>
  </ciManagement>
  ...
</project>
```

Mailing Lists

Mailing lists are a great tool for keeping in touch with people about a project. Most mailing lists are for developers and users.

```
<project>
  ...
  <mailingLists>
    <mailingList>
      <name>User List</name>
      <subscribe>user-subscribe@127.0.0.1</subscribe>
      <unsubscribe>user-unsubscribe@127.0.0.1</unsubscribe>
      <post>user@127.0.0.1</post>
      <archive>http://127.0.0.1/user/</archive>
      <otherArchives>
        <otherArchive>http://base.google.com/base/1/127.0.0.1</otherArchive>
      </otherArchives>
    </mailingList>
  </mailingLists>
  ...
</project>
```

- **subscribe**, **unsubscribe**: There elements specify the email addresses which are used for performing the relative actions To subscribe to the user list above, a user would send an email to user-subscribe@127.0.0.1.
- **archive**: This element specifies the url of the archive of old mailing list emails, if one exists. If there are mirrored archives, they can be specified under otherArchives.
- **post**: The email address which one would use in order to post to the mailing list. Note that not all mailing lists have the ability to post to (such as a build failure list).

SCM

SCM (Software Configuration Management, also called Source Code/Control Management or, succinctly, version control) is an integral part of any healthy project. If your Maven project uses an SCM system (it does, doesn't it?) then here is where you would place that information into the POM.

```
<project>
  ...
  <scm>
    <connection>scm:svn:http://127.0.0.1/svn/my-project</connection>
   <developerConnection>scm:svn:https://127.0.0.1/svn/my-project</developerConnection>
    <tag>HEAD</tag>
    <url>http://127.0.0.1/websvn/my-project</url>
  </scm>
  ...
</project>
```

- **connection**, **developerConnection**: The two connection elements convey to how one is to connect to the version control system through Maven. Where connection requires read access for Maven to be able to find the source code (for example, an update), developerConnection requires a connection that will give write access. The Maven project has spawned another project named Maven SCM, which creates a common API for any SCMs that wish to implement it. The most popular are CVS and Subversion, however, there is a growing list of other supported SCMs (*/scm/scms-overview.html*). All SCM connections are made through a common URL structure.

  ```
  scm:[provider]:[provider_specific]
  ```

  Where provider is the type of SCM system. For example, connecting to a CVS repository may look like this:

  ```
  scm:cvs:pserver:127.0.0.1:/cvs/root:my-project
  ```

- **tag**: Specifies the tag that this project lives under. HEAD (meaning, the SCM root) should be the default.
- **url**: A publicly browsable repository. For example, via ViewCVS.

  ```
  <project>
    ...
    <prerequisites>
  ```

```
      <maven>2.0.4</maven>
    </prerequisites>
    ...
  </project>
```

- **prerequisites**: The POM may have certain prerequisites in order to execute correctly. For example, perhaps there was a fix in Maven 2.0.3 that you need in order to deploy using sftp. Here is where you give the prerequisites to building. If these are not met, Maven will fail the build before even starting. The only element that exists as a prerequisite in POM 4.0 is the maven element, which takes a minimum version number.

Repositories

Repositories are collections of artifacts which adhere to the Maven repository directory layout. In order to be a Maven 2 repository artifact, a POM file must live within the structure `$BASE_REPO/groupId/artifactId/version/artifactId-version.pom`. `$BASE_REPO` can be local (file structure) or remote (base URL); the remaining layout will be the same. Repositories exist as a place to collect and store artifacts. Whenever a project has a dependency upon an artifact, Maven will first attempt to use a local copy of the specified artifact. If that artifact does not exist in the local repository, it will then attempt to download from a remote repository. The repository elements within a POM specify those alternate repositories to search.

The repository is one of the most powerful features of the Maven community. The default central Maven repository lives on *http://repo1.maven.org/maven2*. Another source for artifacts not yet in iBiblio is the Codehaus snapshots repo.

```
<project>
  ...
  <repositories>
    <repository>
      <releases>
        <enabled>false</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
      </snapshots>
      <id>codehausSnapshots</id>
      <name>Codehaus Snapshots</name>
      <url>http://snapshots.maven.codehaus.org/maven2</url>
      <layout>default</layout>
    </repository>
  </repositories>
  <pluginRepositories>
    ...
  </pluginRepositories>
```

```
  ...
</project>
```

- **releases**, **snapshots**: These are the policies for each type of artifact, Release or snapshot. With these two sets, a POM has the power to alter the policies for each type independent of the other within a single repository. For example, one may decide to enable only snapshot downloads, possibly for development purposes.
- **enabled**: `true` or `false` for whether this repository is enabled for the respective type (`releases` or `snapshots`).
- **updatePolicy**: This element specifies how often updates should attempt to occur. Maven will compare the local POMs timestamp (stored in a repositorys maven-metadata file) to the remote. The choices are: `always`, `daily` (default), `interval:X` (where X is an integer in minutes) or `never`.
- **checksumPolicy**: When Maven deploys files to the repository, it also deploys corresponding checksum files. Your options are to `ignore`, `fail`, or `warn` on missing or incorrect checksums.
- **layout**: In the above description of repositories, it was mentioned that they all follow a common layout. This is mostly correct. Maven 2 has a default layout for its repositories; however, Maven 1.x had a different layout. Use this element to specify which if it is `default` or `legacy`.

Plugin Repositories

Repositories are home to two major types of artifacts. The first are artifacts that are used as dependencies of other artifacts. These are the majority of plugins that reside within central. The other type of artifact is plugins. Maven plugins are themselves a special type of artifact. Because of this, plugin repositories may be separated from other repositories (although, I have yet to hear a convincing argument for doing so). In any case, the structure of the `pluginRepositories` element block is similar to the repositories element. The `pluginRepository` elements each specify a remote location of where Maven can find new plugins.

Distribution Management

Distribution management acts precisely as it sounds: it manages the distribution of the artifact and supporting files generated throughout the build process. Starting with the last elements first:

```
<project>
  ...
  <distributionManagement>
    ...
    <downloadUrl>http://mojo.codehaus.org/my-project</downloadUrl>
    <status>deployed</status>
  </distributionManagement>
```

```
  ...
</project>
```

- **downloadUrl**: is the url of the repository from whence another POM may point to in order to grab this POMs artifact. In the simplest terms, we told the POM how to upload it (through repository/url), but from where can the public download it? This element answers that question.
- **status**: Warning! Like a baby bird in a nest, the status should never be touched by human hands! The reason for this is that Maven will set the status of the project when it is transported out to the repository. Its valid types are as follows.
  - —**none**: No special status. This is the default for a POM.
  - —**converted**: The manager of the repository converted this POM from an earlier version to Maven 2.
  - —**partner**: This could just as easily have been called synched. This means that this artifact has been synched with a partner repository.
  - —**deployed**: By far the most common status, meaning that this artifact was deployed from a Maven 2 instance. This is what you get when you manually deploy using the command-line deploy phase.
  - —**verified**: This project has been verified, and should be considered finalized.

Repository

Where as the repositories element specifies in the POM the location and manner in which Maven may download remote artifacts for use by the current project, distributionManagement specifies where (and how) this project will get to a remote repository when it is deployed. The repository elements will be used for snapshot distribution if the snapshotRepository is not defined.

```
<project>
  ...
  <distributionManagement>
    <repository>
      <uniqueVersion>false</uniqueVersion>
      <id>corp1</id>
      <name>Corporate Repository</name>
      <url>scp://repo1/maven2</url>
      <layout>default</layout>
    </repository>
    <snapshotRepository>
      <uniqueVersion>true</uniqueVersion>
      <id>propSnap</id>
      <name>Propellors Snapshots</name>
      <url>sftp://propellers.net/maven</url>
      <layout>legacy</layout>
    </snapshotRepository>
    ...
  </distributionManagement>
```

```
  ...
</project>
```

- **id**, **name**: The `id` is used to uniquely identify this repository amongst many, and the `name` is a human readable form.
- **uniqueVersion**: The unique version takes a `true` or `false` value to denote whether artifacts deployed to this repository should get a uniquely generated version number, or use the version number defined as part of the address.
- **url**: This is the core of the repository element. It specifies both the location and the transport protocol to be used to transfer a built artifact (and POM file, and checksum data) to the repository.
- **layout**: These are the same types and purpose as the layout element defined in the repository element. They are `default` and `legacy`.

Site Distribution

More than distribution to the repositories, `distributionManagement` is responsible for defining how to deploy the projects site and documentation.

```
<project>
  ...
  <distributionManagement>
    ...
    <site>
      <id>mojo.website</id>
      <name>Mojo Website</name>
      <url>scp://beaver.codehaus.org/home/projects/mojo/public_html/</url>
    </site>
    ...
  </distributionManagement>
  ...
</project>
```

- **id**, **name**, **url**: These elements are similar to their counterparts above in the `distributionManagement` `repository` element.

Relocation

```
<project>
  ...
  <distributionManagement>
    ...
    <relocation>
      <groupId>org.apache</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0</version>
      <message>We have moved the Project under Apache</message>
    </relocation>
    ...
  </distributionManagement>
  ...
</project>
```

Projects are not static; they are living things (or dying things, as the case may be). A common thing that happens as projects grow, is that they are forced to move to more suitable quarters. For example, when your next wildly successful open source project moves under the Apache umbrella, it would be good to give your users as heads-up that the project is being renamed to `org.apache:my-project:1.0`. Besides specifying the new address, it is also good form to provide a message explaining why.

Profiles

A new feature of the POM 4.0 is the ability of a project to change settings depending on the environment where it is being built. A `profile` element contains both an optional activation (a profile trigger) and the set of changes to be made to the POM if that profile has been activated. For example, a project built for a test environment may point to a different database than that of the final deployment. Or dependencies may be pulled from different repositories based upon the JDK version used. The elements of profiles are as follows:

```
<project>
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>...</activation>
      <build>...</build>
      <modules>...</modules>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <dependencies>...</dependencies>
      <reporting>...</reporting>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
    </profile>
  </profiles>
</project>
```

Activation

Activations are the key of a profile. The power of a profile comes from its ability to modify the basic POM only under certain circumstances. Those circumstances are specified via an `activation` element.

```
<project>
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
```

```
            <family>Windows</family>
            <arch>x86</arch>
            <version>5.1.2600</version>
          </os>
          <property>
            <name>mavenVersion</name>
            <value>2.0.3</value>
          </property>
          <file>
            <exists>${basedir}/file2.properties</exists>
            <missing>${basedir}/file1.properties</missing>
          </file>
        </activation>
        ...
      </profile>
    </profiles>
  </project>
```

Activation occurs when all specified criteria have been met, though not all are required at once.

- **jdk**: `activation` has a built in, Java-centric check in the `jdk` element. This will activate if the test is run under a jdk version number that matches the prefix given. In the above example, `1.5.0_06` will match.
- **os**: The `os` element can define some operating system specific properties shown above.
- **property**: The `profile` will activate if Maven detects a property (a value which can be dereferenced within the POM by `${name}`) of the corresponding `name=value` pair.
- **file**: Finally, a given filename may activate the `profile` by the `existence` of a file, or if it is `missing`.

The `activation` element is not the only way that a `profile` may be activated. The `settings.xml` file's `activeProfile` element may contain the profile's `id`. They may also be activated explicitly through the command line via a comma separated list after the `P` flag (e.g. `-P test`).

To see which profile will activate in a certain build, use the `maven-help-plugin`.

```
mvn help:active-profiles
```

The BaseBuild Element Set *(revisited)*

As mentioned above, the reason for the two types of build elements reside in the fact that it does not make sense for a profile to configure build directories or extensions as it does in the top level of the POM. Reguardless of in which environment the project is built, some values will remain constant, such as the directory structure of the source code. *If you find your project needing to keep two sets of code for different environments, it may be prudent to investigate refactoring the project into two or more seperate projects.*

Summary

The Maven 2 POM is big. However, its size is also a testament to its versatility. The ability to abstract all of the aspects of a project into a single artifact is powerful, to say the least. Gone are the days of dozens of disparate build scripts and scattered documentation concerning each individual project. Along with Mavens other stars that make up the Maven galaxy a well defined build lifecycle, easy to write and maintain plugins, centralized repositories, system-wide and user-based configurations, as well as the increasing number of tools to make developers jobs easier to maintain complex projects the POM is the large, but bright, center.

CHAPTER 15

Appendix: Settings Details

Introduction

Quick Overview

The `settings` element in the `settings.xml` file contains elements used to define values which configure Maven execution in various ways, like the `pom.xml`, but should not be bundled to any specific project, or distributed to an audience. These include values such as the local repository location, alternate remote repository servers, and authentication information. There are two locations where a `settings.xml` file may live:

- The Maven install: `$M2_HOME/conf/settings.xml`
- A user's install: `${user.dir}/.m2/settings.xml`

Here is an overview of the top elements under `settings`:

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

Settings Details

Simple Values

Half of the top-level `settings` elements are simple values, representing a range of values which describe elements of the build system that are active full-time.

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>${user.dir}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <usePluginRegistry>false</usePluginRegistry>
  <offline>false</offline>
  <pluginGroups>
    <pluginGroup>org.codehaus.mojo</pluginGroup>
  </pluginGroups>
  ...
</settings>
```

- **localRepository**: This value is the path of this build system's local repository. The default value is `$user.dir/.m2/repository`. This element is especially useful for a main build server allowing all logged-in users to build from a common local repository.
- **interactiveMode**: `true` if Maven should attempt to interact with the user for input, `false` if not. Defaults to `true`.
- **usePluginRegistry**: `true` if Maven should use the `${user.dir}/.m2/plugin-registry.xml` file to manage plugin versions, defaults to `false`. *Note that for the current version of Maven 2.0, the plugin-registry.xml file should not be depended upon. Consider it dormant for now.*
- **offline**: `true` if this build system should operate in offline mode, defaults to `false`. This element is useful for build servers which cannot connect to a remote repository, either because of network setup or security reasons.
- **pluginGroups**: This element contains a list of `pluginGroup` elements, each contains a groupId. The list is searched when a plugin is used and the groupId is not provided in the command line. This list contains `org.apache.maven.plugins` by default. For example, given the above settings the Maven command-line may execute `org.codehaus.mojo:castor-maven-plugin:generate` with the truncated command:

```
mvn castor-maven-plugin:generate
```

Servers

The `distributionManagement` element of the POM defines the repositories for deployment. However, certain settings such as `username` and `password` should not be distrib-

uted along with the `pom.xml`. This type of information should exist on the build server in the `settings.xml`.

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>server001</id>
      <username>my_login</username>
      <password>my_password</password>
      <privateKey>${usr.home}/.ssh/id_dsa</privateKey>
      <passphrase>some_passphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
  </servers>
  ...
</settings>
```

- **id**: This is the ID of the server *(not of the user to login as)* that matches the `distributionManagement` `repository` element's `id`.
- **username**, **password**: These elements appear as a pair denoting the login and password required to authenticate to this server.
- **privateKey**, **passphrase**: Like the previous two elements, this pair specifies a path to a private key (default is `$usr.home/.ssh/id_dsa`) and a `passphrase`, if required. The `passphrase` and `password` elements may be externalized in the future, but for now they must be set plain-text in the `settings.xml` file.
- **filePermissions**, **directoryPermissions**: When a repository file or directory is created on deployment, these are the permissions to use. The legal values of each is a three digit number corrosponding to *nix file permissions, ie. 664, or 775.

Mirrors

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <mirrors>
    <mirror>
      <id>planetmirror.com</id>
      <name>PlanetMirror Australia</name>
      <url>http://downloads.planetmirror.com/pub/maven2</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
```

```
  ...
</settings>
```

- **id**, **name**: The unique identifier and readable name of this mirror. The `id` is used to differentiate between `mirror` elements.
- **url**: The base URL of this mirror. The build system will use prepend this URL to connect to a repository rather than the default server URL.
- **mirrorOf**: The `id` of the server that this is a mirror of. For example, to point to a mirror of the Maven `central` server (*http://repo1.maven.org/maven2*), set this element to `central`. This must not match the mirror `id`.

Proxies

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

- **id**: The unique identifier for this proxy. This is used to differentiate between `proxy` elements.
- **active**: `true` if this proxy is active. This is useful for declaring a set of proxies, but only one may be active at a time.
- **protocol**, **host**, **port**: The `protocol://host:port` of the proxy, seperated into discrete elements.
- **username**, **password**: These elements appear as a pair denoting the login and password required to authenticate to this proxy server.
- **nonProxyHosts**: This is a list of hosts which should not be proxied. The delimiter of the list is the expected type of the proxy server; the example above is pipe delimited - comma delimited is also common.

Profiles

The `profile` element in the `settings.xml` is a truncated version of the `pom.xml` `profile` element. It consists of the `activation`, `repositories`, `pluginRepositories` and `properties` elements. The `profile` elements only include these four elements because they concerns themselves with the build system as a whole (which is the role of the `settings.xml` file), not about individual project object model settings.

If a profile is active from `settings`, its values will override any equivalently ID'd profiles in a POM or `profiles.xml` file.

Activation

Activations are the key of a profile. Like the POM's profiles, the power of a profile comes from its ability to modify some values only under certain circumstances; those circumstances are specified via an `activation` element.

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.3</value>
        </property>
        <file>
          <exists>${basedir}/file2.properties</exists>
          <missing>${basedir}/file1.properties</missing>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

Activation occurs when all specified criteria have been met, though not all are required at once.

- **jdk**: `activation` has a built in, Java-centric check in the `jdk` element. This will activate if the test is run under a jdk version number that matches the prefix given. In the above example, `1.5.0_06` will match.
- **os**: The `os` element can define some operating system specific properties shown above.
- **property**: The `profile` will activate if Maven detects a property (a value which can be dereferenced within the POM by `${name}`) of the corresponding `name=value` pair.
- **file**: Finally, a given filename may activate the `profile` by the `existence` of a file, or if it is `missing`.

The `activation` element is not the only way that a `profile` may be activated. The `settings.xml` file's `activeProfile` element may contain the profile's `id`. They may also be activated explicitly through the command line via a comma separated list after the `P` flag (e.g. `-P test`).

To see which profile will activate in a certain build, use the `maven-help-plugin`.

```
mvn help:active-profiles
```

Properties

Maven properties are value placeholder, like properties in Ant. Their values are accessible anywhere within a POM by using the notation `${X}`, where `X` is the property. They come in five different styles, all accessible from the `settings.xml` file:

1. `env.X`: Prefixing a variable with "env." will return the shells environment variable. For example, `${env.PATH}` contains the $path environment variable (`%PATH%` in Windows).
2. `project.x`: A dot (.) notated path in the POM will contain the corresponding elements value. For example: `<project><version>1.0</version></project>` is accessible via `${project.version}`.
3. `settings.x`: A dot (.) notated path in the `settings.xml` will contain the corresponding elements value. For example: `<settings><offline>false</offline></settings>` is accessible via `${settings.offline}`.
4. Java System Properties: All properties accessible via `java.lang.System.getProperties()` are available as POM properties, such as `${java.home}`.
5. `x`: Set within a <properties /> element or an external files, the value may be used as `${someVar}`.

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
```

```
      <properties>
        <user.install>${user.dir}/our-project</user.install>
      </properties>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

The property `$user.install` is accessible from a POM if this profile is active.

Repositories

Repositories are remote collections of projects from which Maven uses to populate the local repository of the build system. It is from this local repository that Maven calls it plugins and dependencies. Different remote repositories may contain different projects, and under the active profile they may be searched for a matching release or snapshot artifact.

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <repositories>
        <repository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>http://snapshots.maven.codehaus.org/maven2</url>
          <layout>default</layout>
        </repository>
      </repositories>
      <pluginRepositories>
        ...
      </pluginRepositories>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

- **releases**, **snapshots**: These are the policies for each type of artifact, Release or snapshot. With these two sets, a POM has the power to alter the policies for each type independent of the other within a single repository. For example, one may decide to enable only snapshot downloads, possibly for development purposes.
- **enabled**: `true` or `false` for whether this repository is enabled for the respective type (`releases` or `snapshots`).
- **updatePolicy**: This element specifies how often updates should attempt to occur. Maven will compare the local POMs timestamp (stored in a repositorys maven-metadata file) to the remote. The choices are: `always`, `daily` (default), `interval:X` (where X is an integer in minutes) or `never`.
- **checksumPolicy**: When Maven deploys files to the repository, it also deploys corresponding checksum files. Your options are to `ignore`, `fail`, or `warn` on missing or incorrect checksums.
- **layout**: In the above description of repositories, it was mentioned that they all follow a common layout. This is mostly correct. Maven 2 has a default layout for its repositories; however, Maven 1.x had a different layout. Use this element to specify which if it is `default` or `legacy`.

Plugin Repositories

Repositories are home to two major types of artifacts. The first are artifacts that are used as dependencies of other artifacts. These are the majority of plugins that reside within central. The other type of artifact is plugins. Maven plugins are themselves a special type of artifact. Because of this, plugin repositories may be separated from other repositories (although, I have yet to hear a convincing argument for doing so). In any case, the structure of the `pluginRepositories` element block is similar to the `repositories` element. The `pluginRepository` elements each specify a remote location of where Maven can find new plugins.

Active Profiles

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <activeProfiles>
    <activeProfile>env-test</activeProfile>
  </activeProfiles>
</settings>
```

The final piece of the `settings.xml` puzzle is the `activeProfiles` element. This contains a set of `activeProfile` elements, which each have a value of a `profile id`. Any `profile id` defined as an `activeProfile` will be active, reguardless of any environment settings. If no matching profile is found nothing will happen. For example, if `env-`

`test` is an `activeProfile`, a profile in a `pom.xml` (or `profile.xml` with a corrosponding `id` will be active. If no such profile is found then execution will continue as normal.

Summary

A short summary on the settings.

CHAPTER 16

Appendix: Properties

Properties

A large portion of Maven's portability may be attributed to its use of properties. Like properties in Ant or variables in Make, Maven properties represent a value to be set at runtime (as opposed to a hardcoded value like a String or Integer). This is a list of the most common properties in Maven's core.

This is the base directory of the currently running project... where the POM file lives.

- **basedir**

  These properties corrospond to values in the current project's POM settings.

```
project.*
 version
 build.*
   directory
```

- **project.groupId**
- **project.artifactId**
- **project.version**
- **project.packaging**
- **project.build.directory**
- **project.build.finalName**

These properties corrospond to values in the system's settings.xml file. Note that the settings.xml file can be in the Maven installation /conf directory, as well as the current user's home directory (${user.home}) /.m2 directory. Their values are merged, with the user settings winning.

```
settings.*
 localRepository
```

- **settings.localRepository**

```
Environment variable ... highly system specific, but some assumptions can often be made.
env.*
```

```
PATH
M2_HOME
JAVA_HOME
```

Java runtime's System.properties.

- **java.version** - Java Runtime Environment version
- **java.vendor** - Java Runtime Environment vendor
- **java.vendor.url** - Java vendor URL
- **java.home** - Java installation directory
- **java.vm.specification.version** - Java Virtual Machine specification version
- **java.vm.specification.vendor** - Java Virtual Machine specification vendor
- **java.vm.specification.name** - Java Virtual Machine specification name
- **java.vm.version** - Java Virtual Machine implementation version
- **java.vm.vendor** - Java Virtual Machine implementation vendor
- **java.vm.name** - Java Virtual Machine implementation name
- **java.specification.version** - Java Runtime Environment specification version
- **java.specification.vendor** - Java Runtime Environment specification vendor
- **java.specification.name** - Java Runtime Environment specification name
- **java.class.version** - Java class format version number
- **java.class.path** - Java class path
- **java.ext.dirs** - Path of extension directory or directories
- **os.name** - Operating system name
- **os.arch** - Operating system architecture
- **os.version** - Operating system version
- **file.separator** - File separator ("/" on UNIX, "\" on Windows)
- **path.separator** - Path separator (":" on UNIX, ";" on Windows)
- **line.separator** - Line separator ("\n" on UNIX and Windows)
- **user.name** - User's account name
- **user.home** - User's home directory
- **user.dir** - User's current working directory

Filtering

Filtering is an integral part of Maven resources, and is a chief reason why resources are seperated from source code (the next is seperation of concerns). Filtering is simple to do in Maven:

```
<project>
  ...
```

```
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>

  <properties>
    <some.property.name>property value!</some.property.name>
  </properties>
  ...
</project>
```

Now any file under src/main/resources which contains the denotion `${some.property.name}` will be replaced by the building property value... in the above case "property value!".

Tips and Tricks

Using maven-antrun-plugin to View Property Values

This is a simple way to view the values of any properties you may be curious. Echo the `${property.you.want}` through the goal configuration. Note that it is important that you bind the `antrun:run` goal to a phase (`validate` is best) and execute that, rather than run the `antrun:run` goal directly. This is because the lifecycle may load properties that are not loaded when executing a goal in solarity.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>prop-test</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <phase>validate</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>$PATH=${env.PATH}</echo>
              </tasks>
            </configuration>
          </execution>
```

```
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Executing `mvn validate` yeilds the following results on my computer:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.mycompany:prop-test:pom:1.0-SNAPSHOT
[INFO]    task-segment: [validate]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: default}]
[INFO] Executing tasks
    [echo] $PATH=/usr/local/bin:/sw/bin:/sw/sbin:/bin:/sbin:/usr/bin:/usr/sbin:/usr/X11R6/bin:/usr/local
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sat Apr 28 11:58:20 CDT 2007
[INFO] Final Memory: 2M/4M
[INFO] ------------------------------------------------------------------------
```

Synchronize versions/groups with an Inherited Property

Create a parent/multi-module POM with a specific version. Then for all inheriting projects, set their version via `dependencyManagement` as a property. For example:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.group</groupId>
  <artifactId>parent</artifactId>
  <version>1.1-alpha-4-SNAPSHOT</version>

  <modules>
    <module>sub-project-a</module>
    <module>sub-project-b</module>
  </modules>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>${groupId}</groupId>
        <artifactId>sub-project-a</artifactId>
        <version>${version}</version>
      </dependency>
      <dependency>
        <groupId>${groupId}</groupId>
        <artifactId>sub-project-b</artifactId>
        <version>${version}</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

```
  ...
</project>
```

This makes synchronizing versions easier, since inter-dependencies need not both with the task of synchronizing when the version changed, or bother with specifying groupIds.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.group</groupId>
    <artifactId>parent</artifactId>
    <version>1.1-alpha-4-SNAPSHOT</version>
  </parent>
  <artifactId>sub-project-a</artifactId>

  <dependencies>
    <dependency>
      <groupId>${groupId}</groupId>
      <artifactId>sub-project-b</artifactId>
    </dependency>
  </dependencies>
</project>
```

If you release using the `maven-release-plugin`, not only will all of your projects be incremeneted, but also all inter-dependencies will be synchronized as well.

GOTCHA: `maven-release-plugin` will not currently increment properties used as versions - eg. via the `properties` element. This may be fixed in later version of the plugin. For now, just stick with using the `${version}` property.

CHAPTER 17

Appendix: Plugin APIs

Plugin APIs

```
Log
   public abstract boolean isDebugEnabled();
   public abstract boolean isInfoEnabled();
   public abstract boolean isWarnEnabled();
   public abstract boolean isErrorEnabled();
   public abstract void debug(CharSequence charsequence);
   public abstract void debug(CharSequence charsequence, Throwable throwable);
   public abstract void debug(Throwable throwable);
   public abstract void info(CharSequence charsequence);
   public abstract void info(CharSequence charsequence, Throwable throwable);
   public abstract void info(Throwable throwable);
   public abstract void warn(CharSequence charsequence);
   public abstract void warn(CharSequence charsequence, Throwable throwable);
   public abstract void warn(Throwable throwable);
   public abstract void error(CharSequence charsequence);
   public abstract void error(CharSequence charsequence, Throwable throwable);
   public abstract void error(Throwable throwable);

 Mojo
  public abstract void execute() throws MojoExecutionException, MojoFailureException;
       public abstract Log getLog();

 AbstractMojo
   public Log getLog();
   public Map getPluginContext();

MojoExecutionException

MojoFailureException
```