Vincent Massol & Jason van Zyl

Better Builds *with* Maven

How-to Guide *for* Maven 2.0

Mergere Library Press • March 2006

Better Builds with Maven

The How-to Guide for Maven 2.0

John Casey
Vincent Massol
Brett Porter
Carlos Sanchez
Jason Van Zyl

Better Builds with Maven. The How-to Guide for Maven 2.0

© 2006 Mergere, Inc.

The contents of this publication are protected by U.S. copyright law and international treaties. Unauthorized reproduction of this publication or any portion of it is strictly prohibited.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this book or from the use of programs and source code that may accompany it. In no event shall the publisher and the authors be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this book.

Printed: March 2006 in the USA

Acknowledgments

I'm very lucky to have been able to write this book with the core Maven team. They are all great developers and working with them on Maven since 2002 has been an endless source of discoveries and enlightening. Jason Van Zyl and I were initially supposed to write half of the book but we soon realized that there was a substantial amount of work to be done on the code to stabilize Maven and the plugins before we could write about it. This is when Brett Porter, Carlos Sanchez and John D. Casey stepped up to the plate and jumped on board to help us. They ended up writing a big portion of the book and improving the overall book's quality by several folds. Thank you guys! We owe you. I'd like to thank Jason of course who's guided Maven all those years and who's had the foresight and courage to rewrite Maven 2 from scratch, taking into account all learnings from the past.

A special thank goes to Jesse McConnell who's helped me a lot write the J2EE chapter and especially the Web Services part. Thanks also to all our reviewers who provided great feedback and fixed our errors. They are all part of the vibrant Maven community whether as committers or contributors. In no special order, I'd like to thank Stephane Nicoll, Napoleon Esmundo C. Ramirez, Felipe Leme, Jerome Lacoste, Bill Dudney and David Blevins.

A big thank to Mergere which sponsored this book and a company I admire for really understanding open source. Thanks to Winston Damarillo and Gordon King for making this book possible and thanks to Natalie Burdick for driving it relentlessly to completion, which is not a small feat when you have to manage a bunch of hardcore open source developers who continually lapse into coding when they should be writing instead! Delivering a quality book would not have been possible without the professional help of Lisa Malgeri and Elena Renard who did all layouts, copy editing and more generally transformed our technical prose into proper and readable English. Thank you to Joakim Erdfelt for the graphical images in our Figures.

A huge thank to Pivolis for allowing me to work part time on the book even though the return on investment was far from being guaranteed! Once more Francois Hisquin and Jean-Yves Grisi have proved that they have their employees' satisfaction at heart.

Last but not least, all my love goes to my wife Marie-Albane and my 3 young kids who kept loving me unfalteringly even though I spent a good portion of our play-time to write this book.

Vincent Massol

I'd like to start by thanking everyone who has been involved in the Maven community over the years, for sticking with it through some of the early development and for holding the development team to a higher standard. Maven is the cumulative work of nearly 40 committers and an even larger number of contributors who've taken the time to give feedback, contribute ideas, answer questions, test features, and submit fixes.

Jason van Zyl deserves particular credit and has my constant admiration for having the foresight to start the project and for continuing to stick unwaveringly to his vision throughout its development.

I'd also like to thank Paul Russell and the development team during my time at Fairfax Digital, for trusting my judgment to use Maven on our projects, and without whom I would never have been able to get involved in the project in the first place. Both I and the Maven project owe you all a debt of gratitude.

I'm grateful to have had the opportunity to finally write a book about Maven, and to do so with such a number of my talented friends and colleagues. To the authors Jason, Vincent, John and Carlos - congratulations on a job well done. To Natalie Burdick, for her dedication to see this through, despite having all of the least enjoyable tasks of keeping the development on track, doing additional copy editing and organizing its release. To Lisa Malgeri and Elena Renard, for their copy editing, formatting and development of the book's template, whose work made this book as professional and polished as it is.

I'm very thankful to the team at Mergere and Simula Labs for not only sponsoring the development of the book and making it available free of charge, but for the opportunity to work in such a tremendous environment. It is a rare thing to be able to work with such a friendly group of people that are among the brightest in their field, doing what you enjoy and being able to see your our own personal vision realized.

Personally, I'd like to say how much I appreciate all of my family, friends, and those of St Paul's Anglican church for all of their care,support, help and encouragement in everything that I do.

My most heartfelt thanks and all of my love go to my wife Laura, for loving me as I am despite all my flaws, for her great patience and sacrifice in allowing me to work at all hours from home, and for leaving her native Michigan to live in a bizarre foreign land. I continue to hope that it was for me and not just because of the warmer climate.

Brett Porter

I would like to thank professor Fernando Bellas for encouraging my curiosity about the open source world, and the teammates during my time at Softgal, for accepting my crazy ideas about open source.

Also, I'd like to thank my family for their continuous support, especially my parents and my brother for helping me whenever I needed. Thanks also to all the people in Galicia for that delicious food I miss so much when traveling around the world.

Carlos Sanchez

Many thanks to Jesse McConnell for his contributions to the book. It is much appreciated.

All of us would like to thank Lisa Malgeri, Elena Renard and Joakim Erdfelt for their many contributions to the book.

Finally, we would like to thank all the reviewers who greatly enhanced the content and quality of this book: Natalie Burdick, Stephane Nicoll, Napoleon Esmundo C. Ramirez, Felipe Leme, Jerome Lacoste, Bill Dudney, David Blevins, Lester Ecarma, Ruel Loehr, Mark Hobson, Tim O'Brien, Chris Berry, Abel Rodriguez, Jerome Lacoste, Fabrice Bellingard, Allan Ramirez, Emmanuel Venisse and John Tolentino.

Vincent, Jason, John, Brett and Carlos

About the Authors

Vincent Massol has been an active participant in the Maven community as both a committer and a member of the Project Management Committee (PMC) since Maven's early days in 2002. Vincent has directly contributed to Maven's core, as well as to various Maven plugins. In addition to his work on Maven, he founded the Jakarta Cactus project-a simple testing framework for server-side Java code and the Cargo project-a J2EE container manipulation framework. Vincent lives and works in Paris, where he is the technical director of Pivolis, a company which specializes in collaborative offshore software development using Agile methodologies. This is Vincent's third book; he is a co-author of JUnit in Action, published by Manning in 2003 (ISBN 1-930-11099-5) and Maven: A Developer's Notebook, published by O'Reilly in 2005 (ISBN 0-596-00750-7).

Jason Van Zyl: As chief architect and co-founder of Mergere, Inc., Jason van Zyl focuses on improving the Software Development Infrastructure associated with medium to large scale projects, which has led to the founding of the Apache Maven project. He continues to work directly on Maven and serves as the Chair of the Apache Maven Project Management Committee.

Brett Porter has been involved in the Apache Maven project since early 2003, discovering Maven while searching for a simpler way to define a common build process across projects. Immediately hooked, Brett became increasingly involved in the project's development, joining the Maven Project Management Committee (PMC) and directing traffic for both the 1.0 and 2.0 major releases. Additionally, Brett has become involved in a variety of other open source projects, and is a Member of the Apache Software Foundation. Brett is a co-founder and the Director of Engineering at Mergere, Inc., where he hopes to be able to make the lives of other developers easier. He is grateful to work and live in the suburbs of Sydney, Australia.

John Casey became involved in the Maven community in early 2002, when he began looking for something to make his job as Ant “buildmeister” simpler. He was invited to become a Maven committer in 2004, and in 2005, John was elected to the Maven Project Management Committee (PMC). Since 2004, his focus in the Maven project has been the development of Maven 2. Build management and open source involvement have been common threads throughout his professional career, and today a large part of John's job focus is to continue the advancement of Maven as a premier software development tool. John lives in Gainesville, Florida with his wife, Emily. When he's not working on Maven, John enjoys amateur astrophotography, roasting coffee, and working on his house.

Carlos Sanchez received his Computer Engineering degree in the University of Coruña, Spain, and started early in the open source technology world. He created his own company, CSSC, specializing in open source consulting, supporting both European and American companies to deliver pragmatic solutions for a variety of business problems in areas like e-commerce, financial, telecommunications and, of course, software development. He enjoys cycling and raced competitively when he was younger.

Table of Contents

List of Tables

List of Figures

Preface

Welcome to *Better Builds with Maven*, an indispensable guide to understand and use Maven 2.0.

Maven 2 is a product that offers immediate value to many users and organizations. As you will soon find, it does not take long to realize those benefits. Perhaps, reading this book will take you longer. Maven works equally well for small and large projects, but Maven shines in helping teams operate more effectively by allowing team members to focus on what the stakeholders of a project require -- leaving the build infrastructure to Maven!

This guide is not meant to be an in-depth and comprehensive resource but rather an introduction, which provides a wide range of topics from understanding Maven's build platform to programming nuances.

This guide is intended for Java developers who wish to implement the project management and comprehension capabilities of Maven 2 and use it to to make their day-to-day work easier and generally help with the comprehension of any Java-based project. We hope that this book will be useful for Java project managers as well.

For first time users, it is recommended that you step through the material in a sequential fashion. For users more familiar with Maven (including Maven 1.x), this guide endeavors to provide a quick solution for the need at hand.

Organization

Here is how the book is organized. The first two chapters are geared toward a new user of Maven 2, they discuss what Maven is and get you started with your first Maven project. Chapter 3 builds on that and shows you how to build a real-world project. Chapter 4 shows you how to build a J2EE application and deploy it. Chapter 5 focuses on developing plugins for Maven. Chapter 6 discusses project monitoring issues and reporting, Chapter 7 discusses using Maven in a team development environment and Chapter 8 shows you how to migrate Ant builds to Maven.

Chapter 1, Introducing Maven, goes through the background and philosophy behind Maven and defines what Maven is.

Chapter 2, Getting Started with Maven, gives detailed instructions on creating your first project, and compiling and packaging it. After reading this chapter, you should be up and running with Maven.

Chapter 3, Creating Applications with Maven, illustrates Maven's best practices and advanced uses by working on a real-world example application. In this chapter you will learn to setup a directory structure for a typical application and the basics of managing the application's development with Maven.

Chapter 4, Building J2EE Applications, shows how to create the build for a full-fledged J2EE application, shows how to use Maven to build J2EE archives (JAR, WAR, EAR, EJB, Web Services), and how to use Maven to deploy J2EE archives to a container. At this stage you'll pretty much become an expert Maven user.

Chapter 5, Developing Custom Maven Plugins, focuses on the task of writing custom plugins. It starts by describing fundamentals, including a review of plugin terminology and the basic mechanics of the the Maven plugin framework. From there, the chapter covers the tools available to simplify the life of the plugin developer. Finally, it discusses the various ways that a plugin can interact with the Maven build environment and explore some examples.

Chapter 6, Assessing Project Health with Maven, discusses Maven's monitoring tools, reporting tools, and how to use Maven to generate a Web site for your project. In this chapter, you will be revisiting the Proficio application that was developed in Chapter 3, and learning more about the health of the project.

Chapter 7, Team Collaboration with Maven, looks at Maven as a set of practices and tools that enable effective team communication and collaboration. These tools aid the team to organize, visualize, and document for reuse the artifacts that result from a software project. You will learn how to use Maven to ensure successful team development.

Chapter 8, Migrating to Maven, explains a migration path from an existing build in Ant to Maven. After reading this chapter, you will be able to take an existing Ant-based build, split it into modular components if needed, compile and test the code, create JARs, and install those JARs in your local repository using Maven. At the same time, you will be able to keep your current build working.

Errata

We have made every effort to ensure that there are no errors in the text or in the code. However, we are human, so occasionally something will come up that none of us caught prior to publication. To find the errata page for this book, go to http://library.mergere.com and locate the title in the title lists. Then, on the book details page, click the Book Errata link. On this page you will be able to view all errata that have been submitted for this book and posted by Maven editors. You can also click the Submit Errata link on this page to notify us of any errors that you might have found.

How To Contact Us

We want to hear about any errors you find in this book. Simply email the information to community@mergere.com. We'll check the information and, if appropriate, post an update to the book's errata page and fix the problem in subsequent editions of the book.

How To Download the Source Code

All of the source code used in this book is available for download at http://library.mergere.com. Once at the site, simply locate the appropriate Download link on the page and you can obtain all the source code for the book.

We offer source code for download, errata, and technical support from the Mergere Web site at http://library.mergere.com.

So if you have Maven 2.0 installed, then you're ready to go.

Introducing Maven

This chapter covers:

- An overview of Maven
- Maven's objectives
- Maven's principles:
  - Convention over configuration
  - Reuse of build logic
  - Declarative execution
  - Coherent organization of dependencies
  - Maven's benefits

Things should be made as simple as possible, but not any simpler.

- Albert Einstein

1.1. Maven Overview

Maven provides a comprehensive approach to managing software projects. From compilation, to distribution, to documentation, to team collaboration, Maven provides the necessary abstractions that encourage reuse and take much of the work out of project builds.

1.1.1. What is Maven?

Maven is a *project management framework*, but this doesn't tell you much about Maven. It's the most obvious three-word definition of Maven the authors could come up with, but the term *project management framework* is a meaningless abstraction that doesn't do justice to the richness and complexity of Maven. Too often technologists rely on abstract phrases to capture complex topics in three or four words, and with repetition phrases such as *project management* and *enterprise software* start to lose concrete meaning.

When someone wants to know what Maven is, they will usually ask “What exactly is Maven?”, and they expect a short, sound-bite answer. “Well it is a build tool or a scripting framework” Maven is more than three boring, uninspiring words. It is a combination of ideas, standards, and software, and it is impossible to distill the definition of Maven to simply digested sound-bites. Revolutionary ideas are often difficult to convey with words. If you are interested in a fuller, richer definition of Maven read this introduction; it will prime you for the concepts that are to follow. If you are reading this introduction just to find something to tell your manager[1], you can stop reading now and skip to Chapter 2.

If Maven isn't a “project management framework". what is it? Here's one attempt at a description: Maven is a set of standards, a repository format, and a piece of software used to manage and describe projects. It defines a standard life cycle for building, testing, and deploying project artifacts. It provides a framework that enables easy reuse of common build logic for all projects following Maven's standards. The Maven project at the Apache Software Foundation is an open source community which produces software tools that understand a common declarative *Project Object Model* (POM). This book focuses on the core tool produced by the Maven project, Maven 2, a framework that greatly simplifies the process of managing a software project.

You may have been expecting a more straightforward answer. Perhaps you picked up this book because someone told you that Maven is a build tool. Don't worry, Maven can be the build tool you are looking for, and many developers who have approached Maven as another build tool have come away with a finely tuned build system. While you are free to use Maven as “just another build tool”, to view it in such limited terms is akin to saying that a web browser is nothing more than a tool that reads hypertext.

Maven and the technologies related to the Maven project are beginning to have a transformative effect on the Java community.

In addition to solving straightforward, first-order problems such as simplifying builds, documentation, distribution, and the deployment process, Maven also brings with it some compelling second-order benefits.

1 You can tell your manager: “Maven is a declarative project management tool that decreases your overall time to market by effectively leveraging your synergies. It simultaneously reduces your headcount and leads to remarkable operational efficiencies.”

As more and more projects and products adopt Maven as a foundation for project management, it becomes easier to build relationships between projects and to build systems that navigate and report on these relationships. Maven's standard formats enable a sort of "Semantic Web" for programming projects. Maven's standards and central repository have defined a new naming system for projects. Using Maven has made it easier to add external dependencies and publish your own components.

So to answer the original question: Maven is many things to many people. It is a set of standards and an approach as much as it is a piece of software. It is a way of approaching a set of software as a collection of interdependent components which can be described in a common format. It is the next step in the evolution of how individuals and organizations collaborate to create software systems. Once you get up to speed on the fundamentals of Maven, you will wonder how you ever developed without it.

1.1.2. Maven's Origins

Maven was borne of the practical desire to make several projects at the Apache Software Foundation (ASF) work in the same, predictable way. Prior to Maven, every project at the ASF had a different approach to compilation, distribution, and Web site generation. The ASF was effectively a series of isolated islands of innovation. There were some common themes to each separate build, but different communities were creating different build systems of varied complexity and there was no reuse of build logic between projects. The build process for Tomcat was different from the build process used by Struts, the Turbine developers used a different approach to site generation than the developers of Jakarta Commons. Etc.

This lack of a common approach to building software meant that every new project tended to copy and paste another project's build system. Ultimately, this copy and paste approach to build reuse reached a critical tipping point at which the amount of work required to maintain this collection of build systems was distracting from the central task of developing high-quality software. In addition, the barrier to entry for a project with a difficult build system was extremely high; projects such as Jakarta Taglibs had (and continue to have) a tough time attracting developer interest because it could take an hour to configure everything in just the right way. Instead of focusing on creating good component libraries or MVC frameworks, developers were building yet another build system.

Maven entered the scene by way of the Turbine project, and it immediately sparked interest as a sort of Rosetta Stone for software project management. Developers within the Turbine project could freely move between subcomponents, knowing clearly how they all worked just by understanding how one of them worked. If developers spent time understanding how one project was built, they would not have to go through this process again when they moved on to the next project. People stopped figuring out creative ways to compile, test, and package, and started focusing on component development.

The same effect extended to testing, generating documentation, generating metrics and reports, and deploying. If you followed the Maven Way, your project gained a build by default. Soon after the creation of Maven other projects, such as Jakarta Commons, the Codehaus community started to adopt Maven 1 as a foundation for project management.

Many people come to Maven familiar with Ant so it's a natural association, but Maven is an entirely different creature from Ant. Maven is not just a build tool, and not just a replacement for Ant.

Whereas Ant provides a toolbox for scripting builds, Maven is about the application of patterns in order to facilitate project management through the reuse of common build strategies. Make files and Ant are alternatives to Maven.

However, if your project currently relies on an existing Ant build script that must be maintained, existing Ant scripts and Make files can be complementary to Maven and used through Maven's plugin architecture. Plugins allow developers to call existing Ant scripts and Make files and incorporate those existing functions into the Maven build life cycle.

1.1.3. What Does Maven Provide?

It provides a comprehensive model that can be applied to all software projects. Maven provides a common project “language”, and the software tool named Maven is just a secondary side-effect of this model. Projects and systems that use this standard declarative approach tend to be more transparent, more reusable, more maintainable, and easier to comprehend.

Maven provides a useful abstraction in the same way an automobile provides an abstraction most of us are familiar with. When you purchase a new car, the car provides a known interface; if you've learned how to drive a Jeep, you can easily drive a Camry. Maven takes a similar approach to software projects: if you can build one Maven project you can build them all, and if you can apply a testing plugin to one project, you can apply it to all projects. You describe your project using Maven's model, and you gain access to expertise and best-practices of an entire industry.

An individual Maven project's structure and contents are declared in a Project Object Model (POM), which forms the basis of the entire Maven system. The key value to developers from Maven is that it takes a declarative approach rather than developers creating the build process themselves, referred to as "building the build". Maven allows developers to declare goals and dependencies and rely on Maven’s default structures and plugin capabilities to build the project. Much of the project management and build orchestration (compile, test, assemble, install) is effectively delegated to the POM and the appropriate plugins. Developers can build any given project without having to understand how the individual plugins work (scripts in the Ant world).

Given the highly inter-dependent nature of projects in open source, Maven’s ability to standardize locations for source files, documentation, and output, to provide a common layout for project documentation, and to retrieve project dependencies from a shared storage area makes the building process much less time consuming, and much more transparent.

Maven provides you with:

- A comprehensive model for software projects
- Tools that interact with this Declarative Model

Organizations and projects that adopt Maven benefit from:

- **Coherence** - Maven allows organizations to standardize on a set of best practices. Because Maven projects adhere to a standard model they are less opaque. The definition of this term from the American Heritage dictionary captures the meaning perfectly: “Marked by an orderly, logical, and aesthetically consistent relation of parts.“
- **Reusability** - Maven is built upon a foundation of reuse. When you adopt Maven you are effectively reusing the best practices of an entire industry.
- **Agility** - Maven lowers the barrier to reuse not only of build logic but of components. When using Maven it is easier to create a component and integrate it into a multi-project build. It is easier for developers to jump between different projects without a steep learning curve that accompanies a custom, home-grown build system.
- **Maintainability** - Organizations that adopt Maven can stop building the build, and start focusing on the application. Maven projects are more maintainable because they have fewer surprises and follow a common model.

Without these characteristics, it is improbable that multiple individuals will work productively together on a project. Without visibility it is unlikely one individual will know what another has accomplished and it is likely that useful code will not be reused. When code is not reused it is very hard to create a maintainable system.

When everyone is constantly searching to find all the different bits and pieces that make up a project, there is little chance anyone is going to comprehend the project as a whole. As a result you end up with a lack of shared knowledge along with the commensurate degree of frustration among team members. This is a natural effect when processes don't work the same way for everyone.

1.2. Maven's Principles

According to Christopher Alexander "*patterns help create a shared language for communicating insight and experience about problems and their solutions*". The following principles were inspired by Christopher Alexander's idea of creating a shared language:

- Convention over configuration
- Declarative execution
- Reuse of build logic
- Coherent organization of dependencies

Maven provides a shared language for software development projects. As mentioned earlier Maven creates a sense of structure so that problems can be approached in terms of this structure. Each of these principles allows developers to talk about their projects at a higher level of abstraction allowing more effective communication and allowing team members to get on with the important work of creating value at the application level. This chapter will examine each of these principles in detail and these principles will be demonstrated in action in the following chapter when you create your first Maven project.

1.2.1. Convention Over Configuration

One of the central tenets of Maven is to provide sensible default strategies for the most common tasks so that you don't have to think about these mundane details. This is not to say that you can't override them, but the use of these sensible default strategies is encouraged and should be strayed from only when absolutely necessary.

This notion is known as "convention over configuration" and has been popularized by the Ruby on Rails (ROR) community and specifically encouraged by ROR's creator David Heinemeier Hansson who summarizes the notion as follows:

> "Rails is opinionated software. It eschews placing the old ideals of software in a primary position. One of those ideals is flexibility, the notion that we should try to accommodate as many approaches as possible, that we shouldn't pass judgment on one form of development over another. Well, Rails does, and I believe that's why it works.
>
> With Rails, you trade flexibility at the infrastructure level to gain flexibility at the application level. If you are happy to work along the golden path that I've embedded in Rails, you gain an immense reward in terms of productivity that allows you to do more, sooner, and better at the application level.
>
> One characteristic of opinionated software is the notion of "convention over configuration". If you follow basic conventions, such as classes are singular and tables are plural (a person class relates to a people table), you're rewarded by not having to configure that link. The class automatically knows which table to use for persistence. We have a ton of examples like that, which all add up to make a huge difference in daily use."[2]

2 O'Reilly interview with DHH

David Heinemeier Hansson articulates very well what Maven has aimed to accomplish since its inception (note that David Heinemeier Hansson in no way endorses the use of Maven, he probably doesn't even know what Maven is and wouldn't like if he did because it's not written in Ruby yet!). Using standard conventions saves time, makes it easier to communicate to others, and allows you to create value in your applications faster with less effort. You shouldn't need to spend a lot of time getting your development infrastructure functioning.

With Maven you slot the various pieces in where it asks and Maven will take care of almost all of the mundane aspects for you. You don't want to spend time fiddling with building, generating documentation, or deploying. All of these things should simply work, and this is what Maven provides for you.

There are three primary conventions that Maven employs to promote a familiar development environment:

- Standard directory layout for projects
- The concept of a single Maven project producing a single output
- Standard naming conventions

Let's elaborate on each of these points in the following sections.

Standard directory layout for projects

The first convention used by Maven is a standard directory layout for project sources, project resources, configuration files, generated output, and documentation. These components are generally referred to as project content.

Maven encourages a common arrangement of project content so that once you are familiar with the standard locations, you will be able to navigate within any Maven project you encounter in the future.

It is a very simple idea but it can save you a lot of time. If this saves you 30 minutes for each new project you look at, even if you only look at a few new projects a year that's time better spent on your application. First time users often complain about Maven forcing you to do things a certain way and the formalization of the directory structure is the source of most of the complaints.

You can override any of Maven's defaults to create a directory layout of your choosing, but, when you do this, you need to ask yourself if the extra configuration that comes with customization is really worth it. If you have no choice in the matter due to organizational policy or integration issues with existing systems you might be forced to use a directory structure that diverges from Maven's defaults.

In this case, you will be able to adapt your project to your customized layout at a cost, increased complexity of your project's POM If you do have a choice then why not harness the collective knowledge that has built up as a result of using this convention? You will be able to look at other projects and immediately understand the project layout. Follow the standard directory layout, and you will make it easier to communicate about your project. You will see clear examples of the standard directory structure in the next chapter but you can also take a look in Appendix B for a full listing of the standard conventions.

One primary output per project

The second convention used by Maven is the concept of a Maven project producing one primary output. To illustrate this point consider a set of sources for a client/server-based application that contained the client code, the server code, and some shared utility code.

You could produce a single JAR file which contained all the compiled classes, but Maven would encourage you to have three separate projects: a project for the client portion of the application, a project for the server portion of the application, and a project for the shared utility code. In this scenario the code contained in each separate project has a different role to play, or concern, and they should be separated.

The *separation of concerns* (SoC) principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendibility and reusability.

If you have placed all the sources together in a single project. the boundaries between our three separate concerns can easily become blurred and a simple desire to reuse the utility code could prove to be difficult. Having the utility code in a separate project, in a separate JAR file, is much easier to reuse. Maven pushes you to think clearly about the separation of concerns when setting up your projects because modularity leads to reuse.

Standard naming conventions

The third convention, a set of conventions really, is a standard naming convention for directories and a standard naming convention for the primary output of a project. The conventions provide clarity and immediate comprehension. This is important if there are multiple projects involved, because the naming convention keeps them separate in a logical, easy to see manner.

This is illustrated in the *Coherent Organization of Dependencies* section, later in this chapter.

A simple example of the naming convention might be `commons-logging-1.2.jar`. It is immediately obvious that this is version 1.2 of Commons Logging. If the JAR were named `commons-logging.jar` you would not really have any idea what version of Commons Logging you were dealing with and in a lot of cases not even the manifest in the JAR gives any indication.

The intent behind the naming conventions employed by Maven is that you can tell exactly what you are looking at by, well, looking at it. It doesn't make much sense to exclude pertinent information when you have it at hand to use.

Systems that cannot cope with information rich artifacts like `commons-logging-1.2.jar` are inherently flawed because there will come a time when something is misplaced and you'll track down a `ClassNotFound` exception which resulted from the wrong version of a JAR file being used. It's happened to all of us and it doesn't have to.

1.2.2. Reuse of Build Logic

As you have already learned, Maven encourages a separation of concerns because it promotes reuse. Maven puts this principle into practice by encapsulating build logic into coherent modules called *plugins*. Maven can be thought of as a framework which coordinates the execution of plugins in a well defined way.

In Maven there is a plugin for compiling source code, a plugin for running tests, a plugin for creating JARs, a plugin for creating Javadocs, and many others. Even from this short list of examples you can see that a plugin in Maven has a very specific role to play in the grand scheme of things. One important concept to keep in mind is that everything accomplished in Maven is the result of a plugin executing. Plugins are the key building blocks for everything in Maven.

The execution of Maven's plugins is coordinated by Maven's build life cycle in a declarative fashion with inputs from Maven's Project Object Model (POM), specifically from the plugin configurations contained in the POM.

1.2.3. Declarative Execution

Everything in Maven is driven in a declarative fashion using *Maven's Project Object Model* (POM) and specifically the plugin configurations contained in the POM.

Maven's project object model (POM)

Maven is project-centric by design, and the POM is Maven's description of a single project. Without POM, Maven is useless - the POM is Maven's currency. It is the POM that drives execution in Maven and this approach can be described as model driven execution.

The POM below is an example of what you could use to build and test a project. The POM is an XML document and looks like the following (very) simplified example:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This POM will allow you to compile, test, and generate basic documentation. You, being the observant reader, will ask how this is possible using a 15 line file. It's a very good and essential question in fact, and the answer lies in Maven's implicit use of its Super POM.

Maven's Super POM carries with it all the default conventions that Maven encourages, and is the analog of the Java language's `java.lang.Object` class.

In Java, all objects have the implicit parent of `java.lang.Object`. Likewise, in Maven all POMs have an implicit parent which is Maven's Super POM. The Super POM can be rather intimidating at first glance so if you wish to find out more about it you can refer to Appendix B. The key feature to remember is the Super POM contains important default information so you don't have to repeat this information in the POMs you create.

The POM contains every important piece of information about your project. The POM shown above is a very simple POM, but still displays the key elements every POM contains.

- `project` - This is the top-level element in all Maven `pom.xml` files.
- `modelVersion` - This required element indicates the version of the object model this POM is using. The version of the model itself changes very infrequently, but it is mandatory in order to ensure stability when Maven introduces new features or other model changes.
- `groupId` - This element indicates the unique identifier of the organization or group that created the project. The `groupId` is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example `org.apache.maven.plugins` is the designated `groupId` for all Maven plugins.
- `artifactId` - This element indicates the unique base name of the primary artifact being generated by this project. A typical artifact produced by Maven would have the form `<artifactId>-<version>.<extension>` (for example, `myapp-1.0.jar`). Additional artifacts such as source bundles also use the `artifactId` as part of their file name.
- `packaging` - This element indicates the package type to be used by this artifact (JAR, WAR, EAR, etc.). This not only means that the artifact produced is a JAR, WAR, or EAR, but also indicates a specific life cycle to use as part of the build process. The life cycle is a topic dealt with later in the chapter. For now, just keep in mind that the selected packaging of a project plays a part in customizing the build life cycle. The default value for the packaging element is jar so you do not have to specify this in most cases.
- `version` - This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development.
- `name` - This element indicates the display name used for the project. This is often used in Maven's generated documentation, and during the build process for your project, or other projects that use it as a dependency.
- `url` - This element indicates where the project's site can be found.
- `description` - This element provides a basic description of your project.

For a complete reference of the elements available for use in the POM please refer to the POM reference at http://maven.apache.org/maven-model/maven.html.

Maven's build life cycle

Software projects generally follow a similar, well-trodden paths: preparation, compilation, testing, packaging, installation. The path that Maven moves along to accommodate an infinite variety of projects is called *the build life cycle*. In Maven, the build life cycle consists of a series of phases where each phase can perform one or more actions, or goals, related to that phase. For example, the compile phase invokes a certain set of goals to compile a set of classes.

In Maven you do day-to-day work by invoking particular phases in this standard build life cycle. For example, you tell Maven that you want to compile, or test, or package, or install. The actions you need to have performed are stated at a high level and Maven deals with the details behind the scenes. It is important to note that each phase in the life cycle will be executed up to and including the phase you specify. If you tell Maven to `compile`, the `validate, initialize, generate-sources, process-sources, generate-resources`, and `compile` phases will execute.

The standard build life cycle consists of many phases and these can be thought of as extension points. When you need to add some functionality to the build life cycle you do so with a plugin. Maven plugins provide reusable build logic that can be slotted into the standard build life cycle. Any time you need to customize the way your project builds you either employ the use of an existing plugin or create a custom plugin for the task at hand. Chapter 2 includes a demonstrates the customization of a Maven build with a Maven plugin.

1.2.4. Coherent Organization of Dependencies

We are now going to delve into how Maven resolves dependencies and discuss the intimately connected concepts of dependencies, artifacts, and repositories. If you recall, our example POM has a single dependency listed, which is for `Junit`:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This POM states that your project has a dependency on JUnit, which is straightforward, but you may be asking yourself “Where does that dependency come from?” and “Where is the JAR?” The answers to those questions are not readily apparent without some explanation of how Maven dependencies, artifacts. and repositories work. In “Maven-speak” an artifact is a specific piece of software.

In Java, the most common artifact is a JAR file, but a Java artifact could also be a WAR, SAR, or EAR file. A dependency is a reference to an specific artifact that resides in a repository. In order for Maven to attempt to satisfy a dependency, Maven needs to know what repository to look in as well as the dependency's coordinates. A dependency is uniquely identified by the following identifiers: `groupId`, `artifactId` and `version`.

At a basic level, we can describe the process of dependency management as Maven reaching out into the world, grabbing a dependency, and providing this dependency to your software project. There is more going on behind the scenes, but the key concept is that Maven dependencies are declarative.

In the POM you are not specifically telling Maven where the dependencies are physically, you are simply telling Maven what a specific project expects.

Maven takes the dependency coordinates you provide in a POM, and it supplies these coordinates to its own internal dependency mechanisms. With Maven, you stop focusing on a collection of JAR files; instead you deal with logical dependencies. Your project doesn't require `junit-3.8.1.jar`, it depends on version 3.8.1 of the `junit` artifact produced by the `junit` group. Dependency Management is one of the most easily understood and powerful features in Maven.

When a dependency is declared, Maven tries to satisfy that dependency by looking in all of the remote repositories that are available, within the context of your project, for artifacts that match the dependency request. If a matching artifact is located, Maven transports it from that remote repository to your local repository for general use.

Maven has two types of repositories: local and remote. Maven usually interacts with your local repository, but when a declared dependency is not present in your local repository Maven searches all the remote repositories it has access to in an attempt to find what's missing. Read the following sections for specific details about where Maven searches for these dependencies.

Local Maven repository

When you install and run Maven for the first time your local repository will be created and populated with artifacts as a result of dependency requests. By default, Maven creates your local repository in `~/.m2/repository`. You must have a local repository in order for Maven to work. The following folder structure shows the layout of a local Maven repository that has a few locally installed dependency artifacts such as `junit-3.8.1.jar`:

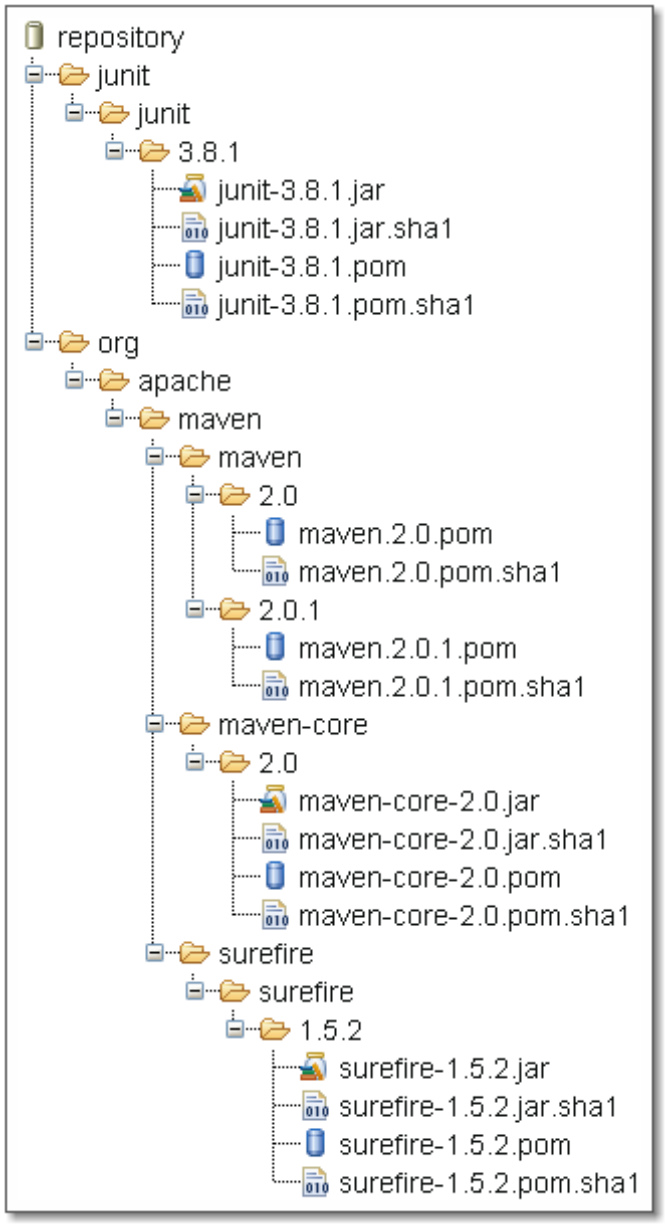


Figure 1-1: Artifact movement from remote to local repository

Take a closer look at one of the artifacts that appeared in your local repository, so you understand how the layout works. In theory a repository is just an abstract storage mechanism, but in practice the repository is a directory structure in your file system. We'll stick with our JUnit example and examine the `junit-3.8.1.jar` artifact that landed in your local repository.

Above you can see the directory structure that is created when the JUnit dependency is resolved. On the next page is the general pattern used to create the repository layout:

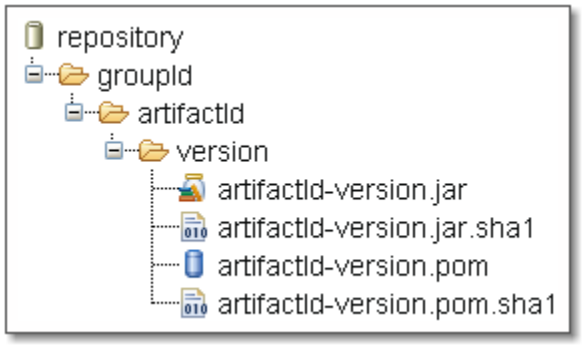


Figure 1-2: General pattern for the repository layout

If the `groupId` is a fully qualified domain name (something Maven encourages) such as `x.y.z` then you would end up with a directory structure like the following:

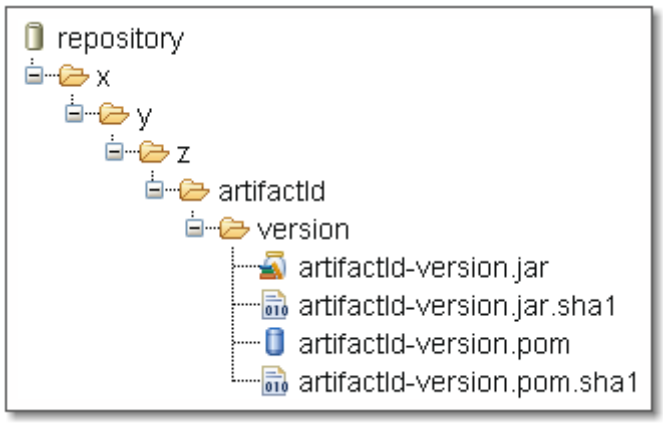


Figure 1-3: Sample directory structure

In the first directory listing you can see that Maven artifacts themselves are stored in a directory structure that corresponds to Maven's `groupId` of `org.apache.maven`.

Locating dependency artifacts

When satisfying dependencies, Maven attempts to locate a dependency's artifact using the following process. First, Maven will generate a path to the artifact in your local repository; for example, Maven will attempt to find the artifact with a `groupId` of "`junit`", `artifactId` of "`junit`", and a version of "`3.8.1`" in `~/.m2/repository/junit/unit/3.8.1/junit-3.8.1.jar`. If this file is not present, it will be fetched from a remote repository.

By default, Maven will attempt to fetch an artifact from the central Maven repository at http://www.ibiblio.org/maven2. If your project's POM contains more than one remote repository, Maven will attempt to download an artifact from each remote repository in the order defined in your POM. Once the dependency is satisfied, the artifact is downloaded and installed in your local repository.

From this point forward, every project with a POM that references the same dependency will use this single copy installed in your local repository. In other words, you don't store a copy of `junit-3.8.1.jar` for each project that needs it; all project referencing this dependency share a single copy of this JAR.

Your local repository is one-stop-shopping for all artifacts that you need regardless of how many projects you are working on. Before Maven, the common pattern in most project was to store JAR files in a project's subdirectory. If you were coding a web application, you would check in the 10-20 JAR files your project relies upon into a lib directory, and you build would add these dependencies to your classpath.

While this approach works for a few project, it doesn't scale easily to support an application with a great number of small components. With Maven, if your project has ten web applications which all depend on version 1.2.6 of the Spring Framework, there is no need to store the various spring JAR files in your project. Each project relies upon a specific artifact via the dependencies listed in a POM, and it is a trivial process to upgrade all of your ten web applications to Spring 2.0 by change your dependency declarations.

Instead of having to add the Spring 2.0 JARs to every project, you simply changed some configuration. Storing artifacts in your SCM along with your project may seem appealing but it is incompatible with the concept of small, modular project arrangements. Dependencies are not you projects code, and they don't deserve to be versioned in an SCM. Declare you dependencies and let Maven take care of details like compilation and testing classpaths.

1.3. Maven's Benefits

A successful technology takes away a burden, rather than imposing one. You don't have to worry about whether or not it's going to work; you don't have to jump through hoops trying to get it to work; it should rarely, if ever, be a part of your thought process. Like the engine in your car or the processor in your laptop, a useful technology just works, in the background, shielding you from complexity and allowing you to focus on your specific task.

Maven provides such a technology for project management, and, in doing so, it simplifies the process of development. To summarize, Maven is a set of standards, Maven is a repository, Maven is a framework, and Maven is software. Maven is also a vibrant, active open-source community that produces software focused on project management. Using Maven is more than just downloading another JAR file and a set of scripts, it is the adoption of processes that allow you to take your software development to the next level.

Getting Started with Maven

This chapter covers:

- Preparing to use Maven
- Creating your first project
- Compiling application sources
- Compiling test sources and running unit tests
- Packaging an installation to your local repository
- Handling classpath resources
- Using Maven plugins

> The key to performance is elegance, not battalions of special cases. The terrible temptation to tweak should be resisted unless the payoff is really noticeable.
>
> *- Jon Bentley and Doug McIlroy*

2.1. Preparing to Use Maven

In this chapter, it is assumed that you are a first time Maven user and have already set up Maven on your local system. If you have not set up Maven yet, then please refer to Maven's Download and Installation Instructions before continuing. Depending on where your machine is located, it may be necessary to make a few more preparations for Maven to function correctly. If you are behind a firewall, then you will have to set up Maven to understand that. To do this, create a `<your-home-directory>/.m2/settings.xml` file with the following content:

```
<settings>
 <proxies>
  <proxy>
   <active>true</active>
   <protocol>http</protocol>
   <host>proxy.mycompany.com</host>
   <port>8080</port>
   <username>your-username</username>
   <password>your-password</password>
  </proxy>
 </proxies>
</settings>
```

If Maven is already in use at your workplace, ask your administrator if there if there is an internal Maven proxy. If there is an active Maven proxy running, then note the URL and let Maven know you will be using a proxy. Create a `<your-home-directory>/.m2/settings.xml` file with the following content.

```
 <mirrors>
  <mirror>
   <id>maven.mycompany.com</id>
   <name>My Company's Maven Proxy</name>
   <url>http://maven.mycompany.com/maven2</url>
   <mirrorOf>central</mirrorOf>
  </mirror>
 </mirrors>
</settings>
```

In its optimal mode, Maven requires network access, so for now simply assume that the above settings will work. The settings.xml file will be explained in more detail in the following chapter and you can refer to the Maven Web site for the complete details on the `settings.xml` file. Now you can perform the following basic check to ensure Maven is working correctly:

```
mvn -version
```

If Maven's version is displayed, then you should be all set to create your first Maven project.

2.2. Creating Your First Maven Project

To create your first project, you will use Maven's **Archetype** mechanism. An archetype is defined as an original pattern or model from which all other things of the same kind are made. In Maven, an archetype is a template of a project, which is combined with some user input to produce a fully-functional Maven project. This chapter will show you how the archetype mechanism works, but if you would like more information about archetypes, please refer to the Introduction to Archetypes.

To create the Quick Start Maven project, execute the following:

```
C:\mvnbook> mvn archetype:create -DgroupId=com.mycompany.app \
-DartifactId=my-app
```

You will notice a few things happened when you executed this command. First, you will notice that a directory named `my-app` has been created for the new project, and this directory contains your `pom.xml`, which looks like the following:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</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>
```

At the top level of every project is your `pom.xml` file. Whenever you see a directory structure which contains a `pom.xml` file you know you are dealing with a Maven project. After the archetype generation has completed you will notice that the following directory structure has been created, and it in fact adheres to Maven's standard directory layout discussed in Chapter 1.

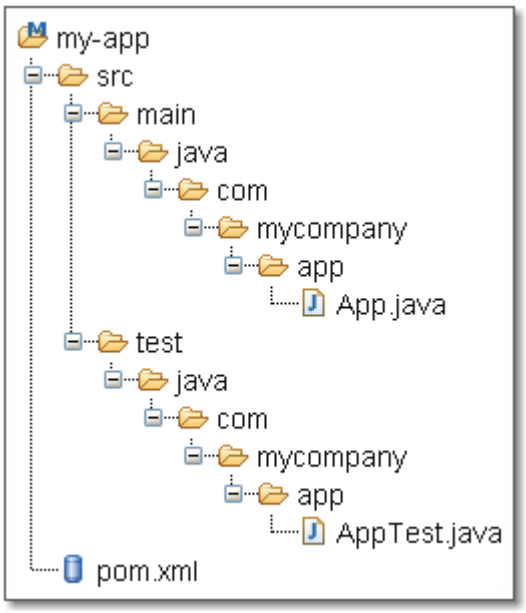


Figure 2-1: Directory structure after archetype generation

The `src` directory contains all of the inputs required for building, testing, documenting, and deploying the project (source files, configuration files, various descriptors such as assembly descriptors, the site, and so on). In this first stage you have Java source files only, but later in the chapter you will see how the standard directory layout is employed for other project content.

Now that you have a POM, some application sources, and some test sources, you are ready to build your project.

2.3. Compiling Application Sources

As mentioned in the introduction, at a very high level, you tell Maven what you need, in a declarative way, in order to accomplish the desired task. Before you issue the command to compile the application sources, note that manifest in this one simple command are Maven's four foundational principles:

- Convention over configuration
- Reuse of build logic
- Declarative execution
- Coherent organization of dependencies

These principles are ingrained in all aspects of Maven, but the following analysis of the simple compile command shows you the four principles in action and makes clear their fundamental importance in simplifying the development of a project.

Change into the `<my-app>` directory. The `<my-app>` directory is the base directory, `${basedir}`, for the `my-app` project. Then, in one fell swoop, you will compile your application sources using the following command:

```
C:\mvnbook\my-app> mvn compile
```

After executing this command you should see output similar to the following:

```
[INFO]------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]  task-segment: [compile]
[INFO]------------------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-resources-plugin: checking for
updates from central
...
[INFO] artifact org.apache.maven.plugins:maven-compiler-plugin: checking for
updates from central
...
[INFO] [resources:resources]
...
[INFO] [compiler:compile]
Compiling 1 source file to c:\mvnbook\my-app\target\classes
[INFO]------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------------
[INFO] Total time: 3 minutes 54 seconds
[INFO] Finished at: Fri Sep 23 15:48:34 GMT-05:00 2005
[INFO] Final Memory: 2M/6M
[INFO]------------------------------------------------------------------------
```

Now let's dissect what actually happened and see where Maven's four principles come into play with the execution of this seemingly simple command.

How did Maven know where to look for sources in order to compile them? And how did Maven know where to put the compiled classes? This is where Maven's principle of “convention over configuration” comes into play. By default, application sources are placed in `src/main/java`. This default value (though not visible in the POM above) was, in fact, inherited from the Super POM. Even the simplest of POMs knows the default location for application sources. This means you don't have to state this location at all in any of your POMs, if you use the default location for application sources. You can, of course, override this default location, but there is very little reason to do so. The same holds true for the location of the compiled classes which, by default, is target/classes.

What actually compiled the application sources? This is where Maven's second principle of “reusable build logic” comes into play. The standard compiler plugin, along with its default configuration, is the tool used to compile your application sources. The same build logic encapsulated in the compiler plugin will be executed consistently across any number of projects.

Although you now know that the compiler plugin was used to compile the application sources, how was Maven able to decide to use the compiler plugin, in the first place? You might be guessing that there is some background process that maps a simple command to a particular plugin. In fact, there is a form of mapping and it is called Maven's default **build life cycle**.

So, now you know how Maven finds application sources, what Maven uses to compile the application sources, and how Maven invokes the compiler plugin. The next question is, how was Maven able to retrieve the compiler plugin? After all, if you poke around the standard Maven installation, you won't find the compiler plugin since it is not shipped with the Maven distribution. Instead, Maven downloads plugins when they are needed.

The first time you execute this (or any other) command, Maven will download all the plugins and related dependencies it needs to fulfill the command. From a clean installation of Maven this can take quite a while (in the output above, it took almost 4 minutes with a broadband connection). The next time you execute the same command again, because Maven already has what it needs, it won't download anything new. Therefore, Maven will execute the command much more quickly.

As you can see from the output, the compiled classes were placed in `target/classes`, which is specified by the standard directory layout. If you're a keen observer you'll notice that using the standard conventions makes the POM above very small, and eliminates the requirement for you to explicitly tell Maven where any of your sources are, or where your output should go. By following the standard Maven conventions you can get a lot done with very little effort!

2.4. Compiling Test Sources and Running Unit Tests

Now that you're successfully compiling your application's sources, you probably have unit tests that you want to compile and execute as well (after all, programmers always write and execute their own unit tests *nudge nudge, wink wink*).

Again, simply tell Maven you want to test your sources. This implies that all prerequisite phases in the life cycle will be performed to ensure that testing will be successful. Use the following simple command to test:

```
C:\mvnbook\my-app> mvn test
```

After executing this command you should see output similar to the following:

```
[INFO]------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]  task-segment: [test]
[INFO]------------------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-surefire-plugin: checking for
updates from central
...
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
Compiling 1 source file to C:\Test\Maven2\test\my-app\target\test-classes
...
[INFO] [surefire:test]
[INFO] Setting reports dir: C:\Test\Maven2\test\my-app\target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.mycompany.app.AppTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0 sec

Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0

[INFO]------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------------
[INFO] Total time: 15 seconds
[INFO] Finished at: Thu Oct 06 08:12:17 MDT 2005
[INFO] Final Memory: 2M/8M
[INFO]------------------------------------------------------------------------
```

Some things to notice about the output:

- Maven downloads more dependencies this time. These are the dependencies and plugins necessary for executing the tests (recall that it already has the dependencies it needs for compiling and won't download them again).
- Before compiling and executing the tests, Maven compiles the main code (all these classes are up-to-date, since we haven't changed anything since we compiled last).

If you simply want to compile your test sources (but not execute the tests), you can execute the following command:

```
C:\mvnbook\my-app> mvn test-compile
```

However, remember that it isn't necessary to run this every time; `mvn` test will always run the `compile` and `test-compile` phases first, as well as all the others defined before it.

Now that you can compile the application sources, compile the tests, and execute the tests, you'll want to move on to the next logical step, how to package your application.

2.5. Packaging and Installation to Your Local Repository

Making a JAR file is straightforward and can be accomplished by executing the following command:

```
C:\mvnbook\my-app> mvn package
```

If you take a look at the POM for your project,you will notice the packaging element is set to jar. This is how Maven knows to produce a JAR file from the above command (you'll read more about this later). Take a look in the the target directory and you will see the generated JAR file.

Now, you'll want to install the artifact (the JAR file) you've generated into your local repository. It can then be used by other projects as a dependency. The directory `<your-home-directory>/.m2/repository` is the default location of the repository. To install, execute the following command:

```
C:\mvnbook\my-app> mvn install
```

Upon executing this command you should see the following output:

```
[INFO]------------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]  task-segment: [install]
[INFO]------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
Compiling 1 source file to <dir>/my-app/target/classes
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
Compiling 1 source file to <dir>/my-app/target/test-classes
[INFO] [surefire:test]
[INFO] Setting reports dir: <dir>/my-app/target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.mycompany.app.AppTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.001 sec

Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0

[INFO] [jar:jar]
[INFO] Building jar: <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing c:\mvnbook\my-app\target\my-app-1.0-SNAPSHOT.jar to <local-
repository>\com\mycompany\app\my-app\1.0-SNAPSHOT\my-app-1.0-SNAPSHOT.jar
[INFO]------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Tue Oct 04 13:20:32 GMT-05:00 2005
[INFO] Final Memory: 3M/8M
[INFO]------------------------------------------------------------------------
```

Note that the Surefire plugin (which executes the test) looks for tests contained in files with a particular naming convention. By default, the following tests are included:

- `**/*Test.java`
- `**/Test*.java`
- `**/*TestCase.java`

Conversely, the following tests are excluded:

- `**/Abstract*Test.java`
- `**/Abstract*TestCase.java`

You have now completed the process for setting up, building, testing, packaging, and installing a typical Maven project. For projects that are built with Maven, this covers the majority of tasks users perform, and if you've noticed, everything done up to this point has been driven by an 18-line POM.

Of course, there is far more functionality available to you from Maven without requiring any additions to the POM, as it currently stands. In contrast, to get any more functionality out of an Ant build script, you must keep making error-prone additions.

So, what other functionality can you leverage, given Maven's re-usable build logic? With even the simplest POM, there are a great number of Maven plugins that work out of the box. This chapter will cover one in particular, as it is one of the highly-prized features in Maven. Without any work on your part, this POM has enough information to generate a Web site for your project! Though you will typically want to customize your Maven site, if you're pressed for time and just need to create a basic Web site for your project, simply execute the following command:

```
C:\mvnbook\my-app> mvn site
```

There are plenty of other stand-alone goals that can be executed as well, for example:

```
C:\mvnbook\my-app> mvn clean
```

This will remove the target directory with the old build data before starting, so it is fresh. Perhaps you'd like to generate an `IntelliJ` IDEA descriptor for the project:

```
C:\mvnbook\my-app> mvn idea:idea
```

This can be run over the top of a previous IDEA project. In this case, it will update the settings rather than starting fresh.

Or, alternatively you might like to generate an Eclipse descriptor:

```
C:\mvnbook\my-app> mvn eclipse:eclipse
```

2.6. Handling Classpath Resources

Another common use case which requires no changes to the POM shown previously, is the packaging of resources into a JAR file. For this common task, Maven again uses the standard directory layout. This means that by adopting Maven's standard conventions, you can package resources within JARs, simply by placing those resources in a standard directory structure.

In the example following, you need to add the directory `src/main/resources`. That is where you place any resources you wish to package in the JAR. The rule employed by Maven is that all directories or files placed within the `src/main/resources` directory are packaged in your JAR with the exact same structure, starting at the base of the JAR.

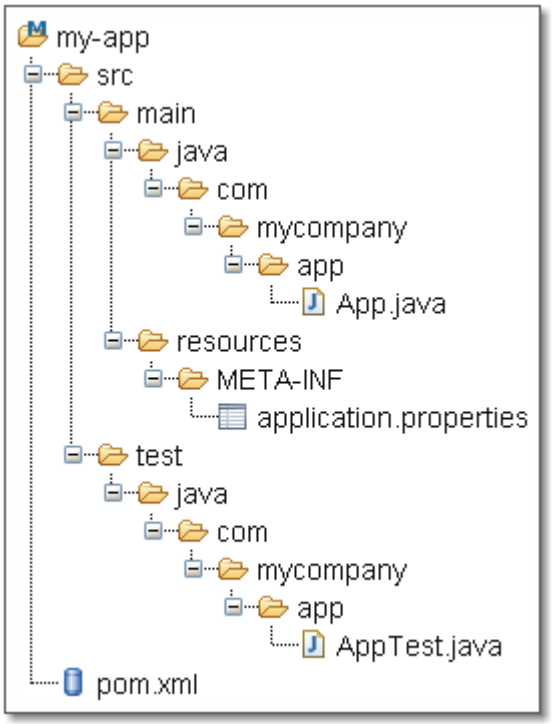


Figure 2-2: Directory structure after adding the resources directory

You can see in the preceding example that there is a `META-INF` directory with an `application.properties` file within that directory. If you unpacked the JAR that Maven created you would see the following:

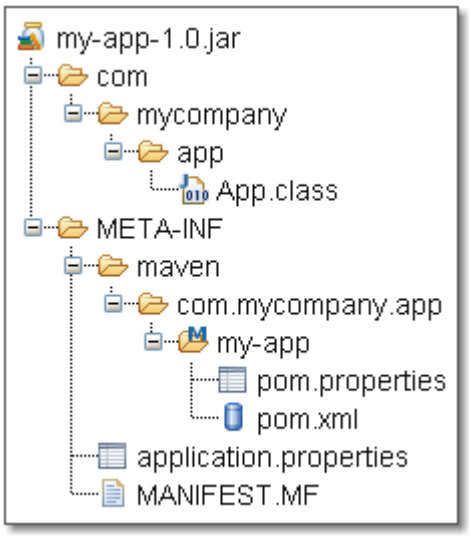


Figure 2-3: Directory structure of the JAR file created by Maven

The original contents of `src/main/resources` can be found starting at the base of the JAR and the `application.properties` file is there in the `META-INF` directory. You will also notice some other files there like `META-INF/MANIFEST.MF`, as well as a `pom.xml` and `pom.properties` file. These come standard with the creation of a JAR in Maven. You can create your own manifest if you choose, but Maven will generate one by default if you don't.

The `pom.xml` and `pom.properties` files are packaged up in the JAR so that each artifact produced by Maven is self-describing and also allows you to utilize the metadata in your own application, should the need arise. One simple use might be to retrieve the version of your application. Operating on the POM file would require you to use Maven utilities, but the properties can be utilized using the standard Java APIs.

2.6.1. Handling Test Classpath Resources

To add resources to the classpath for your unit tests, follow the same pattern as you do for adding resources to the JAR, except place resources in the `src/test/resources directory`. At this point you have a project directory structure that should look like the following:

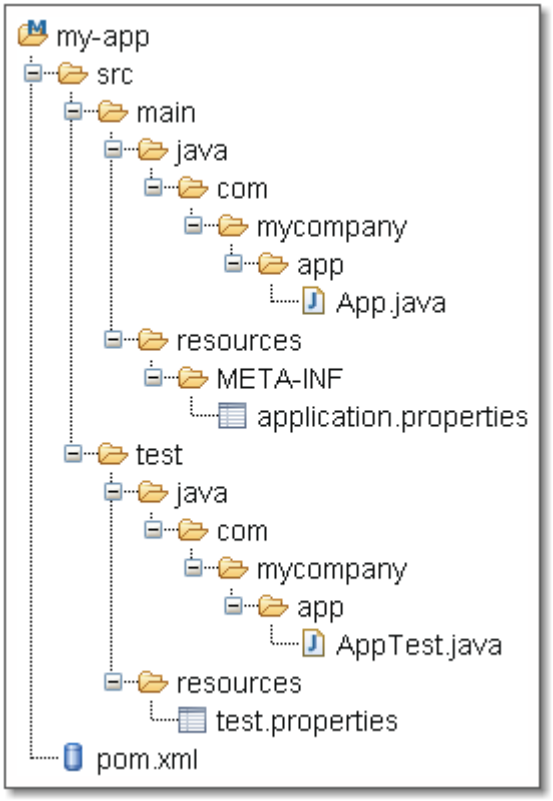


Figure 2-4: Directory structure after adding test resources

In a unit test, you could use a simple snippet of code like the following for access to the resource required for testing:

```
[...]
// Retrieve resource
InputStream is = getClass().getResourceAsStream( "/test.properties" );

// Do something with the resource

[...]
```

2.6.2. Filtering Classpath Resources

Sometimes a resource file will need to contain a value that can be supplied at build time, only. To accomplish this in Maven, put a reference to the property that will contain the value into your resource file using the syntax `${<property name>}`. The property can be either one of the values defined in your `pom.xml`, a value defined in the user's `settings.xml`, a property defined in an external properties file, or a system property.

To have Maven filter resources when copying, simply set filtering to `true` for the resource directory in your `pom.xml`:

```
<project>
```

```
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</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>
  <build>
   <resources>
    <resource>
     <directory>src/main/resources</directory>
     <filtering>true</filtering>
    </resource>
   </resources>
  </build>
</project>
```

You'll notice that the build, resources, and resource elements have been added, which weren't there before. In addition, the POM has to explicitly state that the resources are located in the `src/main/resources` directory. All of this information was previously provided as default values. To override the default value for filtering and set it to `true`, this information had to be added to the `pom.xml`.

To reference a property defined in your `pom.xml`, the property name uses the names of the XML elements that define the value. So `${project.name}` refers to the name of the project, `${project.version}` refers to the version of the project, and `${project.build.finalName}` refers to the final name of the file created, when the built project is packaged. In fact, any element in your POM is available when filtering resources.

To continue the example, create an `src/main/resources/application.properties` file, whose values will be supplied when the resource is filtered as follows:

```
# application.properties
application.name=${project.name}
application.version=${project.version}
```

With that in place, you can execute the following command (`process-resources` is the build life cycle phase where the resources are copied and filtered):

```
mvn process-resources
```

The `application.properties` file under target/classes, which will eventually go into the JAR looks like this:

```
# application.properties
application.name=Maven Quick Start Archetype
application.version=1.0-SNAPSHOT
```

To reference a property defined in an external file, all you need to do is add a reference to this external file in your `pom.xml`. First, create an external properties file and call it `src/main/filters/filter.properties`:

```
# filter.properties
my.filter.value=hello!
```

Next, add a reference to this new file in the `pom.xml` file:

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

Then, add a reference to this property in the `application.properties` file as follows:

```
# application.properties
application.name=${project.name}
application.version=${project.version}
message=${my.filter.value}
```

The next execution of the `mvn process-resources` command will put the new property value into `application.properties`. As an alternative to defining the `my.filter.value` property in an external file, you could have defined it in the properties section of your `pom.xml` and you'd get the same effect (notice you don't need the references to `src/main/filters/filter.properties` either):

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</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>
 <build>
  <resources>
   <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
   </resource>
  </resources>
 </build>
 <properties>
  <my.filter.value>hello</my.filter.value>
 </properties>
</project>
```

Filtering resources can also retrieve values from system properties; either the system properties built into Java (like `java.version` or `user.home`), or properties defined on the command line using the standard Java -D parameter. To continue the example, change the `application.properties` file to look like the following:

```
# application.properties
java.version=${java.version}
command.line.prop=${command.line.prop}
```

Now, when you execute the following command (note the definition of the `command.line.prop` property on the command line), the `application.properties` file will contain the values from the system properties.

```
mvn process-resources "-Dcommand.line.prop=hello again"
```

2.6.3. Preventing Filtering of Binary Resources

Sometimes there are classpath resources that you want included in your JAR, but you do not want them filtered. This is most often the case with binary resources, for example image files.

If you had a `src/main/resources/images` that you didn't want to be filtered, then you would create one resource entry that handled the filtering of resources with an exclusion on the resources you wanted unfiltered. In addition you would add another resource entry, with filtering disabled, and an inclusion of your images directory. The build element would look like the following:

```
<project>
...
 <build>
  <resources>
   <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
    <excludes>
     <exclude>images/**</exclude>
    </excludes>
   </resource>
   <resource>
    <directory>src/main/resources</directory>
    <includes>
     <include>images/**</include>
    </include>
   </resource>
  </resources>
 </build>
...
</project>
```

2.7. Using Maven Plugins

As noted earlier in the chapter, to customize the build for a Maven project, you must include additional Maven plugins, or configure parameters for the plugins already included in the build.

For example, you may want to configure the Java compiler to allow JDK 5.0 sources. This is as simple as adding this following to your POM:

```
<project>
...
 <build>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.0</version>
    <configuration>
     <source>1.5</source>
     <target>1.5</target>
    </configuration>
   </plugin>
  </plugins>
 </build>
...
</project>
```

You'll notice that all plugins in Maven 2 look very similar to a dependency,and in some ways they are. This plugin will be downloaded and installed automatically, if it is not present on your local system, in much the same way that a dependency would be handled. To illustrate the similarity between plugins and dependencies, the `groupId` and version elements have been shown, but in most cases these elements are not required.

If you do not specify a `groupId`, then Maven will default to looking for the plugin with the `org.apache.maven.plugins` or the `org.codehaus.mojo groupId` label. You can specify an additional `groupId` to search within your POM, or `settings.xml`.

If you do not specify a version then Maven will attempt to use the latest released version of the specified plugin. This is often the most convenient way to use a plugin, but you may want to specify the version of a plugin to ensure reproducibility. For the most part, plugin developers take care to ensure that new versions of plugins are backward compatible so you are usually OK with the latest release, but if you find something has changed - you can lock down a specific version.

The configuration element applies the given parameters to every goal from the compiler plugin. In the above case, the compiler plugin is already used as part of the build process and this just changes the configuration.

If you want to find out what the plugin's configuration options are, use the `mvn help:describe` command. If you want to see the options for the maven-compiler-plugin shown previously, use the following command:

```
mvn help:describe -DgroupId=org.apache.maven.plugins \
 -DartifactId=maven-compiler-plugin -Dfull=true
```

You can also find out what plugin configuration is available by using the Maven Plugin Reference section at http://maven.apache.org/plugins/ and navigating to the plugin and goal you are using.

2.8. Summary

After reading this chapter, you should be up and running with Maven. If someone throws a Maven project at you, you'll know how to exercise the basic features of Maven: creating a project, compiling a project, testing a project, and packaging a project. You should also have some insight into how Maven handles dependencies and provides an avenue for customization using Maven plugins. In seventeen pages, you've seen how you can use Maven to build your project, and if you were just looking for a build tool, you could safely stop reading this book now. You might need to refer to the next chapter for more information about customizing your build to fit your project's unique needs.

If someone gave you a project using Maven tomorrow, you would know enough to compile, test, and build it. By learning how to build a Maven project, you have gained access to every single project uses Maven. You've learned a new language and you've taken Maven for a test drive.

If you are interested in learning how Maven builds upon the concepts introduced in the Introduction and the practical tools introduced in Chapter 2, read on. The next few chapters provide you with the tools to customize Maven's behavior and use Maven to manage interdependent software projects. The chapter focused on Maven as a concrete build tool, it laid a good foundation for the rest of this book.

Creating Applications with Maven

This chapter covers:

- Setting Up an Application Directory Structure
- Using Project Inheritance
- Managing Dependencies
- Using Snapshots
- Using Version Ranges
- Managing Plugins
- Utilizing the Build Life Cycle
- Using Profiles
- Deploying your Application
- Creating a Web Site for your Application

Walking on water and developing software from a specification are easy if both are frozen.

- Edward V. Berard

3.1. Introduction

In the second chapter you stepped though the basics of setting up a simple project. Now you will delve in a little deeper, using a real-world example. In this chapter, you are going to learn about some of Maven's best practices and advanced uses by working on a small application to manage frequently asked questions (FAQ). In doing so you will be guided through the specifics of setting up an application and managing that structure.

The application that you are going to create is called Proficio, which is Latin for "help". So, lets start by discussing the ideal directory structure for Proficio.

3.2. Setting Up an Application Directory Structure

In setting up the directory structure for Proficio, it is important to keep in mind that Maven emphasizes the practice of discrete, coherent, and modular builds. The natural outcome of this practice, are discrete and coherent components, which enables code reusability, a necessary goal for every software development project. The guiding principle in determining how best to decompose your application is called the *Separation of Concerns* (SoC).

SoC refers to the ability to identify, encapsulate, and operate on the pieces of software that are relevant to a particular concept, goal, task, or purpose. Concerns are the primary motivation for organizing and decomposing software into smaller, more manageable and comprehensible parts, each of which addresses one or more concerns.

As such, you will see that the Proficio sample application is made up of several modules:

- **Proficio Model**: The data model for the Proficio application, which consists of all the classes that will be used by Proficio as a whole.
- **Proficio API**: The application programming interface for Proficio, which consists of a set of interfaces. The interfaces for the APIs of major components, like the store, are also kept here.
- **Proficio Core**: The implementation of the API.
- **Proficio Stores**: The module which itself houses all the store modules. Proficio has a very simple `memory-based` store and a simple `XStream-based` store.
- **Proficio CLI**: The code which provides a command line interface to Proficio.

These are default naming conventions that Maven uses, but you are free to name your modules in any fashion your team decides. The only real criterion to which to adhere is that your team has a single convention, and they it clearly understands that convention, and can easily identify what a particular module does simply by looking at its name.

In examining the top-level POM for Proficio, you can see in the modules element all the modules that make up the Proficio application. A module is a reference to another Maven project, which really means a reference to another POM. This setup is typically referred to as a multi-module build and this is how it looks in the top-level Proficio POM:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.apache.maven.proficio</groupId>
 <artifactId>proficio</artifactId>
 <packaging>pom</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>Maven Proficio</name>
 <url>http://maven.apache.org</url>
...
  <modules>
    <module>proficio-model</module>
    <module>proficio-api</module>
    <module>proficio-core</module>
    <module>proficio-stores</module>
  </modules>
...
</project>
```

An important feature to note in the POM above is the value of the version element, which you can see is `1.0-SNAPSHOT`. For an application that has multiple modules it is very common that you will release all the modules together and so it makes sense that all the modules have a common application version. It is recommended that you specify the application version in the top-level POM and use that version across all the modules that make up your application.

Currently there is some variance on the Maven web site when referring to directory structures that contain more then one Maven project. In Maven 1.x these were commonly referred to as multi-project builds and some of this vestigial terminology carried over to the Maven 2.x documentation, but the Maven team is now trying to consistently refer to these setups as multi-module builds.

You should take note of the `packaging` element, which has a value of `pom`. For POMs that contain modules, the packaging must be set to value of `pom`: this tells Maven that you're going to be walking through a set of modules in a structure similar to the example being covered here. If you were to look at Proficio's directory structure you would see the following:

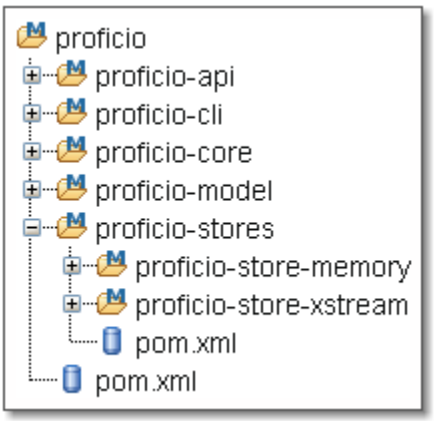


Figure 3-1: Proficio directory structure

You may have noticed that the module elements in the POM match the names of the directories that you see above in the Proficio directory structure. Looking at the module names is how Maven steps into the right directory to process the POM located there. Let's take a quick look at the modules and examine the packaging for each of them:

Table 3-1: Module packaging types

Module	Packaging
`proficio-api`	`jar`
`proficio-cli`	`jar`
`proficio-core`	`jar`
`proficio-model`	`jar`
`proficio-stores`	`pom`

The packaging type in most cases is the default of jar, but the interesting thing here is that we have a another project with a packaging type of `pom`, which is the `proficio-stores` module. If you take a look at the POM for the `proficio-stores` module you will see a set of modules contained there:

```
<project>
   <parent>
    <groupId>org.apache.maven.proficio</groupId>
    <artifactId>proficio</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>proficio-stores</artifactId>
  <name>Maven Proficio Stores</name>
  <packaging>pom</packaging>
  <modules>
    <module>proficio-store-memory</module>
    <module>proficio-store-xstream</module>
  </modules>
</project>
```

Examine the directory structure inside the `proficio-stores` directory and you will see the following:

Figure 3-2: Proficio-stores directory

Whenever Maven sees a POM with a packaging of type `pom` Maven knows to look for a set of modules and to then process each of those modules. You can nest sets of projects like this to any level, organizing your projects in groups according to concern, just as has been done with Proficio's multiple storage mechanisms, which are all placed in one directory.

3.3. Using Project Inheritance

One of the most powerful features in Maven is project inheritance. Using project inheritance allows you to do things like state your organizational information, state your deployment information, or state your common dependencies - all in a single place. Being the observant user, you have probably taken a peek at all the POMs in each of the projects that make up the Proficio project and noticed the following at the top of each of the POMs:

```
...
  <parent>
   <groupId>org.apache.maven.proficio</groupId>
   <artifactId>proficio</artifactId>
   <version>1.0-SNAPSHOT</version>
  </parent>
...
```

This is the snippet in each of the POMs that lets you draw on the resources stated in the specified top-level POM and from which you can inherit down to any level you wish - enabling you to add resources where it makes sense in the hierarchy of your projects. Let's examine a case where it makes sense to put a resource in the top-level POM, using our the top-level POM for the sample Proficio application.

If you look at the top-level POM for Proficio, you will see that in the dependencies section there is a declaration for JUnit version 3.8.1. In this case the assumption being made is that JUnit will be used for testing in all our child projects. So, by stating the dependency in the top-level POM once, you never have to declare this dependency again, in any of your child POMs. The dependency is stated as following:

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

What specifically happens for each child POM, is that each one inherits the dependencies section of the top-level POM. So, if you take a look at the POM for the `proficio-core` module you will see the following, note specifically there is no visible dependency declaration for JUnit:

```
<project>
  <parent>
    <groupId>org.apache.maven.proficio</groupId>
    <artifactId>proficio</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>proficio-core</artifactId>
  <packaging>jar</packaging>
  <name>Maven Proficio Core</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven.proficio</groupId>
      <artifactId>proficio-api</artifactId>
    </dependency>
    <dependency>
      <groupId>org.codehaus.plexus</groupId>
      <artifactId>plexus-container-default</artifactId>
    </dependency>
  </dependencies>
</project>
```

In order for you to see what happens during the inheritance process you will need to use the handy `mvn help:effective-pom` command. This command will show you the final result for a target POM. After you move into the `proficio-core` module and run the command, take a look at the resulting POM you will see the JUnit version 3.8.1 dependency:

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

You will have noticed that the POM that you see when using the `mvn help:effective-pom` is bigger then you expected. But remember from Chapter 2 that the Super POM sits at the top of the inheritance hierarchy. So in this case, the `proficio-core` project inherits from the top-level Proficio project, which in turn inherits from the Super POM. Looking at the effective POM includes everything and is useful to view when trying to figure out what is going on when you are having problems.

3.4. Managing Dependencies

When you are building applications you have many dependencies to manage and the number of dependencies only increases over time, making dependency management difficult to say the least. Maven's strategy for dealing with this problem is to combine the power of project inheritance with specific dependency management elements in the POM.

When you write applications which consist of multiple, individual projects, it is likely that some of those projects will share common dependencies. When this happens it is critical that the same version of a given dependency is used for all your projects, so that the final application works correctly.

You don't want, for example, to end up with multiple versions of a dependency on the classpath when your application executes, as the results can be far from desirable. You want to make sure that all the versions, of all your dependencies, across all of your projects are in alignment so that your testing accurately reflects what you will deploy as your final result. In order to manage, or align, versions of dependencies across several projects, you use the dependency management section in the top-level POM of an application.

To illustrate how this mechanism works, let's look at the dependency management section of the Proficio top-level POM:

```
<project>
...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-model</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-api</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-core</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.codehaus.plexus</groupId>
        <artifactId>plexus-container-default</artifactId>
        <version>1.0-alpha-9</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
...
</project>
```

Note that the `${project.version}` specification is the version specified by the top-level POM's version element, which is the application version.

As you can see within the dependency management section we have several Proficio dependencies and a dependency for the Plexus IoC container. There is an important distinction to be made between the dependencies element contained within the `dependencyManagment` element and the top-level dependencies element in the POM.

The dependencies element contained within the `dependencyManagement` element is used only to state the preference for a version and by itself does not affect a project's dependency graph, whereas the top-level dependencies element does affect the dependency graph. If you take a look at the POM for the `proficio-api` module, you will see a single dependency declaration and that it does not specify a version:

```
<project>
...
  <dependencies>
    <dependency>
      <groupId>org.apache.maven.proficio</groupId>
      <artifactId>proficio-model</artifactId>
    </dependency>
  </dependencies>
</project>
```

The version for this dependency is derived from the `dependencyManagement` element which is inherited from the Proficio top-level POM. The `dependencyManagement` declares a stated preference for the `1.0-SNAPSHOT` (stated as `${project.version}`) for `proficio-model` so that version is injected into the dependency above, to make it complete. The dependencies stated in the `dependencyManagement` only come into play when a dependency is declared without a version.

3.5. Using Snapshots

While you are developing an application with multiple modules, it is usually the case that each of the modules are in flux. Your APIs might be undergoing some change or your implementations are undergoing change and are being fleshed out, or you may be doing some refactoring. Your build system needs to be able to easily deal with this real-time flux, and this is where Maven's concept of a snapshot comes into play. A snapshot in Maven is an artifact which has been prepared using the most recent sources available. If you look at the top-level POM for Proficio you will see a snapshot version specified:

```
<project>
...
 <version>1.0-SNAPSHOT</version>
 <dependencyManagement>
    <dependencies>
     <dependency>
       <groupId>org.apache.maven.proficio</groupId>
       <artifactId>proficio-model</artifactId>
       <version>${project.version}</version>
      </dependency>
      <dependency>
       <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-api</artifactId>
       <version>${project.version}</version>
     </dependency>
      <dependency>
       <groupId>org.codehaus.plexus</groupId>
       <artifactId>plexus-container-default</artifactId>
       <version>1.0-alpha-9</version>
     </dependency>
   </dependencies>
 </dependencyManagement>
...
</project>
```

Specifying a snapshot version for a dependency means that Maven will look for new versions of that dependency without you having to manually specify a new version. Snapshot dependencies are assumed to be changing, so Maven will attempt to update them. By default Maven will look for snapshots on a daily basis, but you can use the `-U` command line option to force the search for updates. Controlling how snapshots work will be explained in detail in Chapter 7. When you specify a non-snapshot version of a dependency Maven will download that dependency once and never attempt to retrieve it again.

3.6. Resolving Dependency Conflicts and Using Version Ranges

With the introduction of transitive dependencies in Maven 2, it became possible to simplify your own POM by only including the dependencies you need directly, and to allow Maven to calculate the full dependency graph. However, as the graph grows, it is inevitable that two or more artifacts will require different versions of a particular dependency. In this case, Maven must choose which version to provide.

In Maven 2.0, the version selected is the one declared “nearest” to the top of the tree - that is, the least number of dependencies that were traversed to get this version. A dependency in the POM being built will be used over anything else.

However, this has limitations:

- The version chosen may not have all the features required by the other dependency.
- If multiple versions are selected at the same depth, then the result is undefined.

While further dependency management features are scheduled for the next release of Maven at the time of writing, there are ways to manually resolve these conflicts as the end user of a dependency, and more importantly ways to avoid it as the author of a reusable library.

To manually resolve conflicts, you can remove the incorrect version from the tree, or you can override both with the correct version. Removing the incorrect version requires identifying the source of the incorrect version by running Maven with the -X flag (for more information on how to do this, see section 6.9 in Chapter 6). For example, if you run `mvn -X test` on the `proficio-core` module, the output will contain something similar to:

```
proficio-core:1.0-SNAPSHOT
  junit:3.8.1 (selected for test)
  plexus-container-default:1.0-alpha-9 (selected for compile)
    plexus-utils:1.0.4 (selected for compile)
    classworlds:1.1-alpha-2 (selected for compile)
    junit:3.8.1 (not setting... to compile; local scope test wins)
  proficio-api:1.0-SNAPSHOT (selected for compile)
    proficio-model:1.0-SNAPSHOT (selected for compile)
    plexus-utils:1.1 (selected for compile)
```

Once the path to the version has been identified, you can exclude the dependency from the graph by adding an exclusion to the dependency that introduced it. In this example, `plexus-utils` occurs twice, and Proficio requires version 1.1 be used. To ensure this, modify the plexus-container-default dependency in the `proficio-core/pom.xml` file as follows:

```
...
<dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-container-default</artifactId>
 <version>1.0-alpha-9</version>
 <exclusions>
  <exclusion>
   <groupId>org.codehaus.plexus</groupId>
   <artifactId>plexus-utils</artifactId>
  </exclusion>
 </exclusions>
</dependency>
...
```

This ensures that Maven ignores the 1.0.4 version of `plexus-utils` in the dependency graph, so that the 1.1 version is used instead.

The alternate way to ensure that a particular version of a dependency is used, is to include it directly in the POM, as follows:

```
...
<dependencies>
 <dependency>
  <groupId>org.codehaus.plexus</groupId>
  <artifactId>plexus-utils</artifactId>
  <version>1.1</version>
  <scope>runtime</scope>
 </dependency>
</dependencies>
...
```

However, this approach is **not** recommended unless you are producing an artifact that is bundling its dependencies and is not used as a dependency itself (for example, a WAR file). The reason for this is that it distorts the true dependency graph, which will accumulate if this project is reused as a dependency itself.

You'll notice here that the runtime scope is used. This is because in this situation, the dependency is used only for packaging, not for compilation. In fact, if it were required for compilation, for stability it would always be declared in the current POM as a dependency - regardless of whether another dependency introduces it.

Neither of these solutions are ideal, but it is possible to improve the quality of your own dependencies to reduce the risk of these issues occurring with your own build artifacts. This is extremely important if you are publishing a build, for a library or framework, that will be used widely by others. This is achieved using version ranges instead.

When a version is declared as `1.1`, as shown above for `plexus-utils`, this indicates that the preferred version of the dependency is `1.1`, but that other versions may be acceptable. Maven has no knowledge about what versions will definitely work, so in the case of a conflict with another dependency, Maven assumes that all versions are valid and uses the “nearest dependency” technique described previously to determine which version to use.

However, you may require a feature that was introduced in `plexus-utils` version 1.1. In this case, the dependency should be specified as follows:

```
<dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-utils</artifactId>
 <version>[1.1,)</version>
</dependency>
```

What this means is that, while the nearest dependency technique will still be used in the case of a conflict, the version that is used must fit the range given. If the nearest version does not match, then the next nearest will be tested, and so on. Finally, if none of them match, or there were no conflicts originally, the version you are left with is `[1.1,)`. This means that the latest version, that is greater than or equal to 1.1, will be retrieved from the repository.

The notation used above is set notation, and table 3-2 shows some of the values that can be used.

Table 3-2: Examples of version ranges

Range	Meaning
`(,1.0]`	Less than or equal to 1.0
`[1.2,1.3]`	Between 1.2 and 1.3 (inclusive)
`[1.0,2.0)`	Greater than or equal to 1.0, but less than 2.0
`[1.5,)`	Greater than or equal to 1.5
`(,1.1),(1.1,)`	Any version, except 1.1

By being more specific through the use of version ranges, it is possible to make the dependency mechanism more reliable for your builds and to reduce the number of exception cases that will be required. However, you also need to avoid being too specific. If two ranges in a dependency graph do not intersect at all, the build will fail.

To understand how ranges work, it is necessary to understand how versions are compared. You can see how a version is partitioned by Maven in figure 3-1.

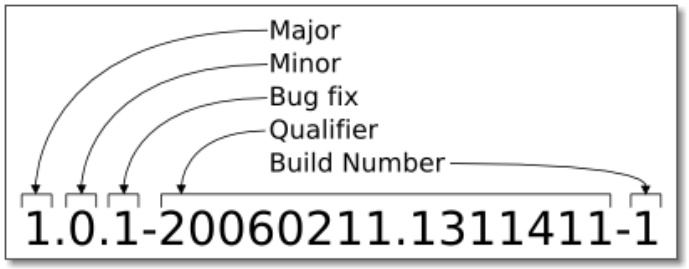


Figure 3-3: Version parsing

As you can see, a version is broken down into five parts: the major, minor and bug fix releases, then the qualifier and finally a build number. In the current version scheme, the snapshot (as shown above) is a special case where the qualifier and build number are both allowed. In a regular version, you can provide only the qualifier, or only the build number. It is intended that the qualifier indicates a version prior to release (for example, `alpha-1`, `beta-1`, `rc1`), while the build number is an increment after release to indicate patched builds.

With regard to ordering, the elements are considered in sequence to determine which is greater - first by major version, second - if the major versions were equal - by minor version, third by bug fix version, fourth by qualifier (using string comparison), and finally, by build number. If a qualifier exists, it means that the version is older than if the qualifier didn't. Conversely, if the build number exists, that is considered newer than if it doesn't. In some cases, the versions will not match this syntax. In those case, the two versions are compared entirely as strings.

Because all of these elements are considered part of the version, the ranges do not differentiate. So, if you use the range `[1.1,)`, and the versions `1.1` and `1.2-beta-1` exist, then `1.2-beta-1` will be selected. Often this is not desired, so to avoid such a situation, you must structure your releases accordingly, through the use of a separate repository.

Whether you use snapshots until the final release, or release betas as milestones along the way, you should deploy them to a snapshot repository as is discussed in Chapter 7 of this book. This will ensure that the beta versions are used in a range only if the project has declared the snapshot (or development) repository explicitly.

A final note relates to how version updates are determined when a range is in use. This mechanism is identical to that of the snapshots that you learned in section 3.6. By default, the repository is checked once a day for updates to the versions of artifacts in use. However, this can be configured per-repository to be on a more regular interval, or forced from the command line using the `-U` option for a particular Maven execution.

If it will be configured for a particular repository, the `updatePolicy` value is changed for releases. For example:

```
...
<repository>
...
 <releases>
  <updatePolicy>interval:60</updatePolicy>
 </releases>
</repository>
```

3.7. Utilizing the Build Life Cycle

In Chapter 2 Maven was described as a framework that coordinates the execution of its plugins in a well defined way or path, which is actually Maven's default build life cycle. Maven's default build life cycle will suffice for a great number of projects without any augmentation but, of course, all projects have different requirements and it is sometimes necessary to augment the life cycle to satisfy these requirements.

For example, Proficio has a requirement to generate Java sources from a model. Maven accommodates this requirement by allowing the declaration of a plugin, which binds itself to a standard phase in Maven's default life cycle, the `generate-sources` phase.

Plugins in Maven are created with a specific task in mind, which means the plugin is bound to a specific phase in the default life cycle, typically. In Proficio, the Modello plugin is used to generate the Java sources for Proficio's data model. If you look at the POM for the `proficio-model` you will see the `plugins` element with a configuration for the Modello plugin:

```
<project>
  <parent>
    <groupId>org.apache.maven.proficio</groupId>
    <artifactId>proficio</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>proficio-model</artifactId>
  <packaging>jar</packaging>
  <name>Proficio Model</name>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.modello</groupId>
        <artifactId>modello-maven-plugin</artifactId>
        <version>1.0-alpha-5</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <version>1.0.0</version>
          <packageWithVersion>false</packageWithVersion>
          <model>src/main/mdo/proficio.mdo</model>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

This is very similar to the declaration for the maven-compiler-plugin that you saw in Chapter 2, but here you see an additional executions element. A plugin in Maven may have several goals, so you need to specify which goal in the plugin you wish to run, by specifying the goal in the executions element.

3.8. Using Profiles

Profiles are Maven's way of letting you create environmental variations in the build life cycle to accommodate things like building on different platforms, building with different JVMs, testing with different databases, or referencing the local file system. Typically you try to encapsulate as much as possible in the POM to ensure that builds are portable, but sometimes you simply have to take into consideration variation across systems and this is why profiles were introduced in Maven.

Profiles are specified using a subset of the elements available in the POM itself (plus one extra section), and can be activated in several ways. Profiles modify the POM at build time, and are meant to be used in complementary sets to give equivalent-but-different parameters for a set of target environments (providing, for example, the path of the application server root in the development, testing, and production environments).

As such, profiles can easily lead to differing build results from different members of your team. However, used properly, you can still preserve build portability with profiles. You can define profiles in one of the following three places:

- The Maven settings file (typically `<your -home-directory>/.m2/settings.xml`)
- A file in the the same directory as the POM, called `profiles.xml`
- The POM itself

In any of the above three profile sites, you can define the following elements:

- repositories
- pluginRepositories
- dependencies
- plugins
- properties (not actually available in the main POM, but used behind the scenes)
- modules
- reporting
- dependencyManagement
- distributionManagement

A subset of the build element, which consists of:

- defaultGoal
- resources
- testResources
- finalName

There are several ways that you can activate profiles:

- Profiles can be specified explicitly using the `-P` CLI option. This option takes an argument that contains a comma-delimited list of profile-ids. When this option is specified, no profiles other than those specified in the option argument will be activated. For example:

```
mvn -Pprofile1,profile2 install
```

- Profiles can be activated in the Maven settings, via the `activeProfiles` section. This section takes a list of `activeProfile` elements, each containing a profile-id. Note that you must have defined the profiles in your `settings.xml` file as well. For example:

```
<settings>
…
  <profiles>
    <profile>
      <id>profile1</id>
      …
        </profile>
        </profiles>

    <activeProfiles>
        <activeProfile>profile1</activeProfile>
   </activeProfiles>
...
</settings>
```

- Profiles can be triggered automatically based on the detected state of the build environment. These activators are specified via an **activation** section in the profile itself. Currently, this detection is limited to prefix-matching of the JDK version, the presence of a system property, or the value of a system property. Here are some examples:

```
<profile>
  <id>profile1</id>
...
   <activation>
     <jdk>1.4</jdk>
   </activation>
</profile>
```

This activator will trigger the profile when the JDK's version starts with "1.4" (eg. "1.4.0_08", "1.4.2_07", "1.4").

```
<profile>
   <id>profile1</id>
...
  <activation>
   <property>
      <name>debug</name>
    </property>
      </activation>
</profile>
```

This will activate the profile when the system property "debug" is specified with any value.

```
<profile>
  <id>profile1</id>
 ...
  <activation>
    <property>
      <name>environment</name>
      <value>test</value>
    </property>
  </activation>
</profile>
```

This last example will activate the profile when the system property "environment" is specified with the value "test".

Now that you are familiar with profiles, you are going to use them to create tailored assemblies: an assembly of Proficio which uses the memory-based store and an assembly of Proficio which uses the XStream-based store. The assemblies will be created in the `proficio-cli` module and the profiles used to control the creation of our tailored assemblies are defined there as well.

If you take a look at the POM for the `proficio-cli` module you will see the profile definitions:

```
<project>
...
 <!-- Profiles for the two assemblies to create for deployment -->
 <profiles>
  <!-- Profile which creates an assembly using the memory based store -->
   <profile>
    <id>memory</id>
     <build>
      <plugins>
        <plugin>
         <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
           <descriptors>
            <descriptor>src/main/assembly/assembly-store-memory.xml</descriptor>
           </descriptors>
          </configuration>
        </plugin>
      </plugins>
     </build>
    <activation>
      <property>
       <name>memory</name>
        </property>
     </activation>
   </profile>
    <!-- Profile which creates an assembly using the xstream based store -->
   <profile>
    <id>xstream</id>
     <build>
       <plugins>
         <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
           <descriptors>
            <descriptor>src/main/assembly/assembly-store-xstream.xml</descriptor>
              </descriptors>
            </configuration>
          </plugin>
        </plugins>
      </build>
      <activation>
        <property>
          <name>xstream</name>
        </property>
      </activation>
    </profile>
  </profiles>
</project>
```

You can see there are two profiles: one with an id of `memory` and another with an id of `xstream`. In each of these profiles you are configuring the assembly plugin to point at the assembly descriptor that will create a tailored assembly. You will also notice that the profiles are activated using a system property.

If you wanted to create the assembly using the memory-based store you would execute the following:

```
mvn -Dmemory clean assembly:assembly
```

If you wanted to create the assembly using the XStream-based store you would execute the following:

```
mvn -Dxstream clean assembly:assembly
```

Both of the assemblies are created in the target directory and if you use the `jar tvf` command on the resulting assemblies you will see that the memory-based assembly only contains the `proficio-store-memory-1.0-SNAPSHOT.jar` file and the XStream-based store only contains the `proficio-store-xstream-1.0-SNAPSHOT.jar` file. This is a very simple example but illustrates how you can customize the execution of the life cycle using profiles to suit any requirement you might have.

3.9. Deploying your Application

Now that you have an application assembly, you'll want to share it with as many people as possible! So, it is now time to deploy your application assembly.

Currently Maven supports several methods of deployment include simple file-based deployment, SSH2 deployment, SFTP deployment, FTP deployment, and external SSH deployment. In order to deploy you need to correctly configure your `distributionManagement` element in your POM, which would typically be your top-level POM so that all child POMs can inherit this information. Here are some examples of how to configure your POM via the various deployment mechanisms.

3.9.1. Deploying to the File System

To deploy to the file system you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>file://${basedir}/target/deploy</url>
    </repository>
  </distributionManagement>
...
</project>
```

3.10. Deploying with SSH2

To deploy to an SSH2 server you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>scp://sshserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>
...
</project>
```

3.11. Deploying with SFTP

To deploy to an SFTP server you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>sftp://ftpserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>
...
</project>
```

3.11.1. Deploying with an External SSH

Now, the first three methods illustrated are included with Maven, so only the `distributionManagement` element is required, but to use an external SSH command to deploy you must configure not only the `distributionManagment` element, but also a build extension.

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>scpexe://sshserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>

  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ssh-external</artifactId>
         <version>1.0-alpha-6</version>
      </extension>
    </extensions>
  </build>
...
</project>
```

The build extension specifies the use of the Wagon external SSH provider, which does the work of moving your files to the remote server. Wagon is the general purpose transport mechanism used throughout Maven.

3.11.2. Deploying with FTP

To deploy with FTP you must also specify a build extension. To deploy to an FTP server you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>ftp://ftpserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ftp</artifactId>
         <version>1.0-alpha-6</version>
      </extension>
    </extensions>
  </build>
...
</project>
```

Once you have configured your POM accordingly, and you are ready to initiate deployment, simply execute the following command:

```
mvn deploy
```

3.12. Creating a Web Site for your Application

Now that you have walked though the process of building, testing and deploying Proficio, it is time for you to see how a standard web site is created for an application. For applications like Proficio it is recommended that you create a source directory at the top-level of the directory structure to store the resources that are used go generate the web site. If you take a look you will see that we have something similarly the following:

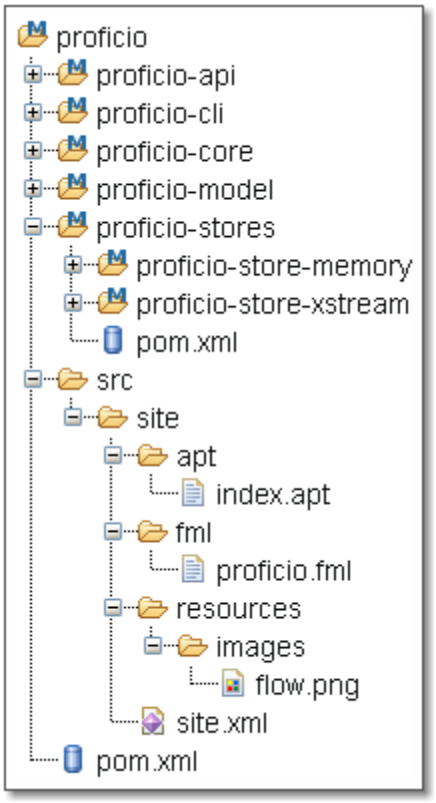


Figure 3-4: The site directory structure

Everything that you need to generate the site resides within the `src/site` directory. Within the `src/site` directory, there is a subdirectory for each of the supported documentation formats that you are using for your site and the very important site descriptor. Maven supports a number of different documentation formats to accommodate various needs and preferences.

Currently the most well supported formats available are:

- The XDOC format, which is a simple XML format used widely at Apache.
- The APT format (Almost Plain Text), which is a wiki-like format that allows you to write simple, structured documents (like this) very quickly. A full reference of the APT Format is available.
- The FML format, which is the FAQ format. A simple XML format for managing FAQs.
- The DocBook Simple format, which is a less complex version of the full DocBook format.

Maven also has limited support for:

- The Twiki format, which is a popular Wiki markup format.
- The Confluence format, which is another popular Wiki markup format.
- The DocBook format.

We will look at a few of the more well supported formats further on in the chapter, but you should become familiar with the site descriptor as it is used to:

- Configure the appearance of the banner
- Configure the skin used for the site
- Configure the format of the publish date
- Configure the links displayed below the banner
- Configure additional information to be fed into the <head/> element the generate pages
- Configure the menu items displayed in the navigation column
- Configure the appearance of project reports

If you look in the `src/site` directory of the Proficio application and look at the site descriptor you will see the following:

```
<project name="Proficio">

  <bannerLeft>
    <name>Proficio</name>
    <href>http://maven.apache.org/</href>
  </bannerLeft>

  <bannerRight>
    <name>Proficio</name>
    <src>http://maven.apache.org/images/apache-maven project.png</src>
  </bannerRight>

  <skin>
    <groupId>org.apache.maven.skins</groupId>
    <artifactId>maven-default-skin</artifactId>
    <version>1.0-SNAPSHOT</version>
  </skin>

  <publishDate format="dd MMM yyyy" />
  <body>
    <links>
      <item name="Apache" href="http://www.apache.org/"/>
      <item name="Maven" href="http://maven.apache.org/"/>
      <item name="Continuum" href="http://maven.apache.org/continuum"/>
    </links>
    <head>
      <meta name="faq" content="proficio"/>
    </head>

    <menu name="Quick Links">
      <item name="Features" href="/maven-features.html"/>
    </menu>

    <menu name="About Proficio">
      <item name="What is Proficio?" href="/what-is-maven.html"/>
    </menu>

    ${reports}
  </body>
</project>
```

This is a fairly standard site descriptor, but you should know about the specifics of each of the elements in the site descriptor:

Table 3-3: Site descriptor

Site Descriptor Element	Description
`bannerLeft and bannerRight`	These elements take a name, `href` and optional `src` element, which can be used for images.
`skin`	This element looks very much like a dependency (the same mechanism is used to retrieve the skin) and controls which skin is used for the site.
`publishDate`	The format of the publish date is that of the `SimpleDateFormat` class in Java.
`body/links`	The link elements control the references that are displayed below the banner and take a simple name and `href`.
`body/head`	The head element allows you to insert anything in the head element of generated pages. You may wish to add metadata, or script snippets for activating tools like Google Analytics.
`body/menu`	The menu elements control the list of menus and their respective menu items that are displayed in the navigation column of the site. You can have any number of menu items each containing any number of links.
`body/${reports}`	The inclusion of the `${reports}` reference controls whether the project reports are displayed in the web site. You might exclude ${reports} reference for a site that consists purely of user documentation for example.

One of the most popular features of Maven are the standard reports that can be produced with little effort. If you simply include the `${reports}` reference in your site descriptor you will, by default, have the standard project information reports generated and displayed automatically for you. The standard project information reports consist of the following:

- Dependencies Report
- Mailing Lists Report
- *Continous* Integration Report
- Source Repository Report
- Issue Tracking Report
- Project Team Report
- License

Even though the standard reports are very useful, often you will want to customize the projects reports that are created and displayed in your web site. The reports created and displayed are controlled in the build/reports element in the POM. You may want to be more selective about the reports that you generate and to do so you need to list each report that you want to include as part of the site generation. You do so by configuring the plugin as follows:

```
<project>
...
  <reporting>
...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <reportSets>
        <reportSet>
          <reports>
            <report>dependencies</report>
            <report>project-team</report>
            <report>mailing-list</report>
            <report>cim</report>
            <!--
            Issue tracking report will be omitted
            <report>issue-tracking</report>
            -->
            <report>license</report>
            <report>scm</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
...
  <reporting>
...
</project>
```

Now that you have a good grasp on what formats are supported, how the site descriptor works, and how to configure reports it's time for you to generate the site. You can do so by executing the following command:

```
mvn site
```

After executing this command you will end up with a directory structure (generated inside the target directory) with the generated content that looks like this:

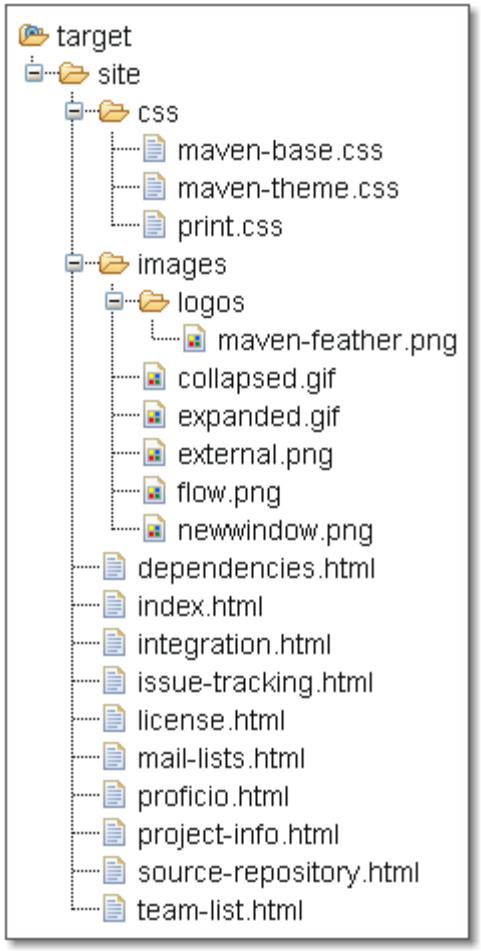


Figure 3-5: The `target` directory

If you look at the generated site with your browser this is what you will see:

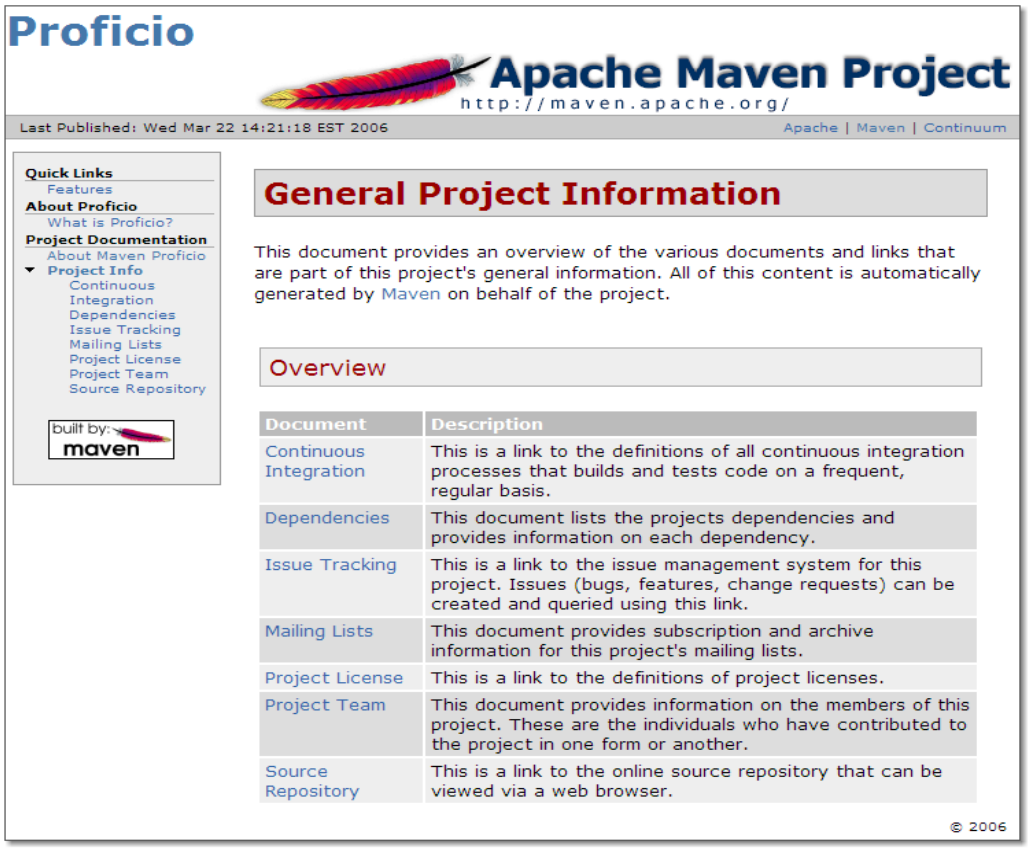


Figure 3-6: The sample generated site

If you have resources like images or PDFs that you want to be able to reference in your documents you can use the `src/site/resources` to store them and when the site is generated the content of `src/site/resources` will be copied to the top-level directory of the site.

If you remember the directory structure for the the sources of our site you will have noticed the `src/site/resources` directory and that it contains an images directory and as you can see in the directory listing above it is located within the images directory of the generated site. Keeping in mind this simple rule you can add any resources you wish to your site.

3.13. Summary

In this chapter you have learned to setup a directory structure for a typical application and learned the basics of managing the application's development with Maven. You should now have a grasp of how project inheritance works, how to manage your application's dependencies, how to make small modifications to Maven's build life cycle, how to deploy your application, and how to create a simple Web site for your application. You are now prepared to move on and learn about more advanced application directory structures like the J2EE example you will see in Chapter 4 and more advanced uses of Maven like creating your own plugins, augmenting your site to view quality metrics, and using Maven in a collaborative environment.

Building J2EE Applications

This chapter covers:

- Organizing the directory structure
- Building J2EE archives (EJB, WAR, EAR, Web Services)
- Setting up in-place Web development
- Deploying J2EE archives to a container
- Automating container start/stop

Keep your face to the sun and you will never see the shadows.

- Helen Keller

4.1. Introduction

J2EE (or Java EE as it is now called) applications are everywhere. Whether you are using the full J2EE stack with EJBs or only using Web applications with frameworks such as Spring or Hibernate, it's likely that you are using J2EE in some of your projects. As a consequence the Maven community has developed plugins to cover every aspect of building J2EE applications. This chapter will take you through the journey of creating the build for a full-fledged J2EE application called DayTrader. You'll learn not only how to create a J2EE build but also how to create a productive development environment (especially for Web application development) and how to deploy J2EE modules into your container.

4.2. Introducing the DayTrader Application

DayTrader is a real world application developed by IBM and then donated to the Apache Geronimo project. Its goal is to serve as both a functional example of a full-stack J2EE 1.4 application and as a test bed for running performance tests. This chapter demonstrates how to use Maven on a real application to show how to address the complex issues related to automated builds. Through this example, you'll learn how to build EARs, EJBs, Web services, and Web applications. As importantly, you'll learn how to automate configuration and deployment of J2EE application servers.

The functional goal of the DayTrader application is to buy and sell stock, and it's architecture is shown in Figure .

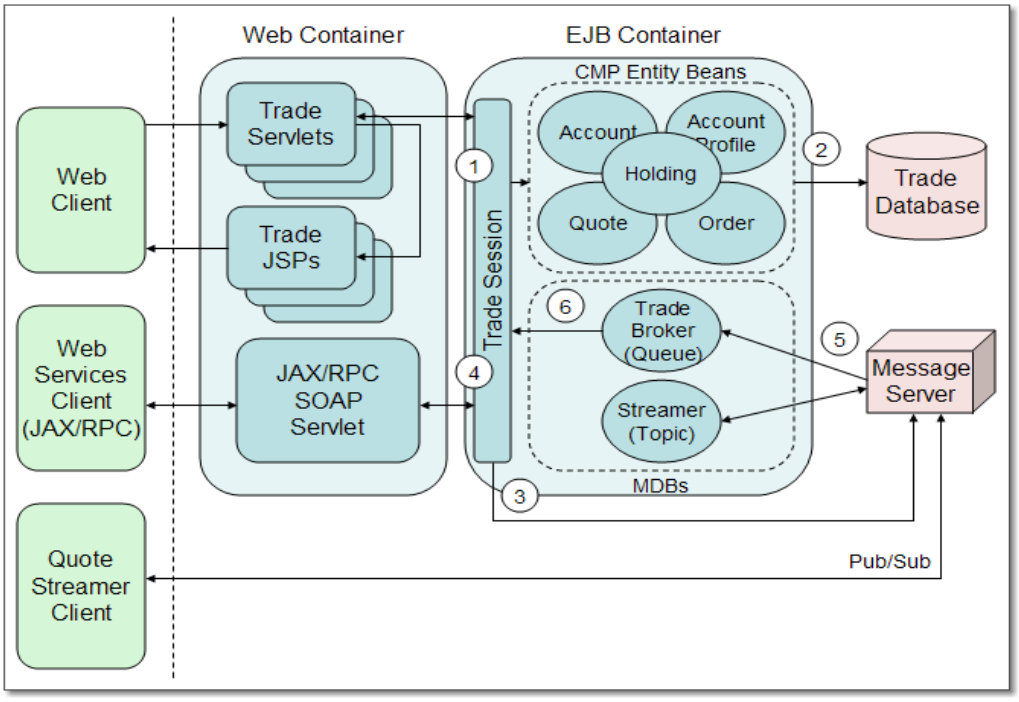


Figure 4-1: Architecture of the DayTrader application

There are 4 layers in the architecture:

- The Client layer offers 3 ways to access the application: using a browser, using Web services, and using the Quote Streamer. The Quote Streamer is a Swing GUI application that monitors quote information about stocks in real-time as the price changes.
- The Web layer offers a view of the application for both the Web client and the Web services client. It uses servlets and JSPs.
- The EJB layer is where the business logic is. The Trade Session is a stateless session bean that offers the business services such as login, logout, get a stock quote, buy or sell a stock, cancel an order, and so on. It uses container-managed persistence (CMP) entity beans for storing the business objects (`Order`, `Account`, `Holding`,`Quote` and `AccountProfile`), and Message-Driven Beans (MDB) to send purchase orders and get quote changes.
- The Data layer consists of a database used for storing the business objects and the status of each purchase, and a JMS Server for interacting with the outside world.

A typical “buy stock” use case consists of the following steps that were shown in Figure :

1. The user gives a buy order (by using the Web client or the Web services client). This request is handled by the Trade Session bean.
2. A new “open” order is saved in the database using the CMP Entity Beans.
3. The order is then queued for processing in the JMS Message Server.
4. The creation of the “open” order is confirmed for the user.
5. Asynchronously the order that was placed on the queue is processed and the purchase completed. Once this happens the Trade Broker MDB is notified
6. The Trade Broker calls the Trade Session bean which in turn calls the CMP entity beans to mark the order as “completed". The user is notified of the completed order on a subsequent request.

4.3. Organizing the DayTrader Directory Structure

The first step to organizing the directory structure is deciding what build modules are required. The easy answer is to follow Maven's artifact guideline: **one module = one main artifact**. Thus you simply need to figure out what artifacts you need. Looking again at Figure you can see that the following modules will be needed:

- A module producing an EJB which will contain all of the server-side EJBs.
- A module producing a WAR which will contain the Web application.
- A module producing a JAR that will contain the Quote Streamer client application.
- A module producing another JAR that will contain the Web services client application.

In addition you may need another module producing an EAR which will contain the EJB and WAR produced from the other modules. This EAR will be used to easily deploy the server code into a J2EE container.

Note that this is the minimal number of modules required. It is possible to come up with more. For example, you may want to split the WAR module into 2 WAR modules: one for the browser client and one for the Web services client. Best practices suggest to do this only when the need arises. If there isn't a strong need you may find that managing several modules can be more cumbersome than useful. On the other hand, it is important to split the modules when it is appropriate for flexibility. For example, if you needed to physically locate the WARs in separate servlet containers to distribute the load.

The next step is to give these modules names and map them to a directory structure. As a general rule, it is better to find functional names for modules. However, it is usually easier to choose names that represent a technology instead. For the DayTrader application the following names were chosen:

- `ejb` - the module containing the EJBs
- `web` - the module containing the Web application
- `streamer` - the module containing the client side streamer application
- `wsappclient` - the module containing the Web services client application
- `ear` - the module producing the EAR which packages the EJBs and the Web application

There are two possible layouts that you can use to organize these modules: a flat directory structure and a nested one. Let's discuss the pros and cons of each layout.

Figure 4-2 shows these modules in a **flat directory structure**. It is flat because you're locating all the modules in the same directory.

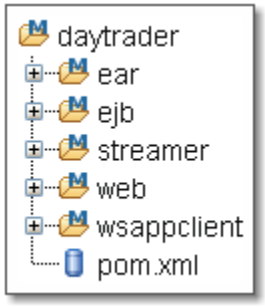


Figure 4-2: Module names and a simple flat directory structure

The top-level `daytrader/pom.xml` file contains the POM elements that are shared between all of the modules. This file also contains the list of modules that Maven will build when executed from this directory (see the Chapter 3, Creating Applications with Maven, for more details):

```
[...]
<modules>
  <module>ejb</module>
  <module>web</module>
  <module>streamer</module>
  <module>wsappclient</module>
  <module>ear</module>
</modules>
[...]
```

This is the easiest and most flexible structure to use, and is the structure used in this chapter. However, if you have many modules in the same directory you may consider finding commonalities between them and create subdirectories to partition them. Note that in this case the modules are still separate, not nested within each other. For example, you might separate the client side modules from the server side modules in the way shown in Figure 4-3.

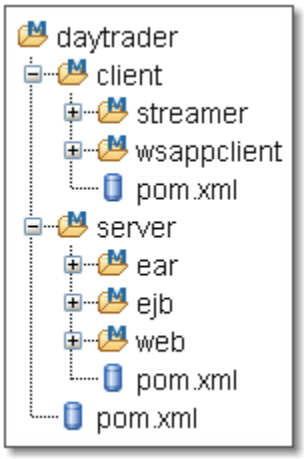


Figure 4-3: Modules split according to a server-side vs client-side directory organization

As before, each directory level containing several modules contains a `pom.xml` file containing the shared POM elements and the list of modules underneath.

The other alternative is to use a **nested directory structure**, as shown in Figure 4-4. In this case, the `ejb` and `web` modules are nested in the ear module. This makes sense as the EAR artifact is composed of the EJB and WAR artifacts produced by the `ejb` and `web` modules. Having this nested structure clearly shows how nested modules are linked to their parent.

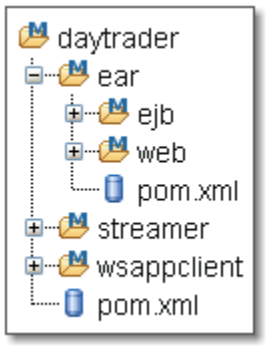


Figure 4-4: Nested directory structure for the EAR, EJB and Web modules

However, even though the nested directory structure seems to work quite well here, it has several drawbacks:

- Eclipse users will have issues with this structure as Eclipse doesn't yet support nested projects. You'd need to consider the three modules as one project, but then you'll be restricted in several ways. For example, the three modules wouldn't be able to have different natures (Web application project, EJB project, EAR project).
- It doesn't allow flexible packaging. For example, the `ejb` or `web` modules might depend on a utility JAR and this JAR may be also required for some other EAR. Or the `ejb` module might be producing a client EJB JAR which is not used by the EAR, but by some client-side application.

  These examples show that there are times when there is not a clear parent for a module. In those cases using a nested directory structure should be avoided. In addition, the nested strategy doesn't fit very well with the Assembler role as described in the J2EE specification.

  The Assembler has a pool of modules and its role is to package those modules for deployment. Depending on the target deployment environment the Assembler may package things differently: one EAR for one environment or two EARs for another environment where a different set of machines are used, etc. A flat layout is more neutral with regard to assembly and should thus be preferred.

Now that you have decided on the directory structure for the DayTrader application, you're going to create the Maven build for each module, starting with the EJB module.

4.4. Building an EJB Project

Let's create a build for the `ejb` module.

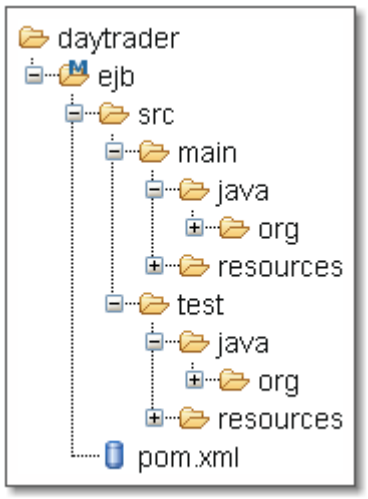


Figure 4-5: Directory structure for the DayTrader `ejb` module

Figure 4-5 shows a canonical directory structure for EJB projects:

- Runtime Java source code in `src/main/java`.
- Runtime classpath resources in c. More specifically, the standard `ejb-jar.xml` deployment descriptor is in `src/main/resources/META-INF/ejb-jar.xml`. Any container-specific deployment descriptor should also be placed in this directory.
- Unit tests in `src/test/java` and classpath resources for the unit tests in `src/test/resources`. Unit tests are tests that execute in isolation from the container. Tests that require the container to run are called integration tests and are covered at the end of this chapter.

Now, take a look at the content of this project's `pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>daytrader-ejb</artifactId>
  <name>Apache Geronimo DayTrader EJB Module</name>
  <packaging>ejb</packaging>
  <description>DayTrader EJBs</description>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-j2ee_1.4_spec</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.3</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ejb-plugin</artifactId>
        <configuration>
           <generateClient>true</generateClient>
           <clientExcludes>
             <clientExclude>**/ejb/*Bean.class</clientExclude>
           </clientExcludes>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

As you can see, you're extending a parent POM using the parent element. This is because the DayTrader build is a multi-module build and you are gathering common POM elements in a parent `daytrader/pom.xml` file.

Now run maven `-N` install in `daytrader/` in order to install the parent POM in your local repository and make it available to all modules when you build them:

```
C:\dev\m2book\code\j2ee\daytrader>mvn -N install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building DayTrader :: Performance Benchmark Sample
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\pom.xml to
C:\[...]\.m2\repository\org\apache\geronimo\samples\daytrader\daytrader\1.0\daytrad
er-1.0.pom
```

The `ejb/pom.xml` file is is a standard POM file except for three items:

- You need to tell Maven that this project is an EJB project so that it generates an EJB JAR when the package phase is called. This is done by specifying:

  ```
  <packaging>ejb</packaging>
  ```

- As you're compiling J2EE code you need to have the J2EE specifications JAR in the project's build classpath. This is achieved by specifying a dependency element on the J2EE JAR. You could instead specify a dependency on Sun's J2EE JAR. However, this JAR is not redistributable and as such cannot be found on ibiblio. Fortunately, the Geronimo project has made the J2EE JAR available under an Apache license and this JAR can be found on ibiblio.

  You should note that you're using a `provided` scope instead of the default compile scope. The reason is that this dependency will already be present in the environment (being the J2EE application server) where your EJB will execute. You make this clear to Maven by using the `provided` scope; this prevents the EAR module from including the J2EE JAR when it is packaged. Even though this dependency is provided at runtime, it still needs to be listed in the POM so that the code can be compiled.

- Lastly, the `pom.xml` contains a configuration to tell the Maven EJB plugin to generate a Client EJB JAR file when `mvn install` is called. The Client will be used in a later examples when building the web module. By default the EJB plugin does not generate the client JAR, so you must explicitly tell it to do so:

  ```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-ejb-plugin</artifactId>
    <configuration>
      <generateClient>true</generateClient>
      <clientExcludes>
        <clientExclude>**/ejb/*Bean.class</clientExclude>
      </clientExcludes>
    </configuration>
  </plugin>
  ```

The EJB plugin has a default set of files to exclude from the client EJB JAR: `**/*Bean.class`, `**/*CMP.class`, `**/*Session.class` and `**/package.html`.

In this example, you need to override the defaults using a `clientExclude` element because it happens that there are some required non-EJB files matching the default `**/*Bean.class` pattern and which need to be present in the generated client EJB JAR. Thus you're specifying a pattern that only excludes from the generated client EJB JAR all EJB implementation classes located in the `ejb` package (`**/ejb/*Bean.class`). Note that it's also possible to specify a list of files to include using `clientInclude` elements.

You're now ready to execute the build. Relax and type `mvn install`:

```
C:\dev\m2book\code\j2ee\daytrader\ejb>mvn install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Building DayTrader :: EJBs
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 49 source files to C:\dev\m2book\code\j2ee\daytrader\ejb\target\classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
Compiling 1 source file to C:\dev\m2book\code\j2ee\daytrader\ejb\target\test-
classes
[INFO] [surefire:test]
[INFO] Setting reports dir: C:\dev\m2book\code\j2ee\daytrader\ejb\target/surefire-
reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running org.apache.geronimo.samples.daytrader.FinancialUtilsTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,02 sec

Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0

[INFO] [ejb:ejb]
[INFO] Building ejb daytrader-ejb-1.0
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\ejb\
       target\daytrader-ejb-1.0.jar
[INFO] Building ejb client daytrader-ejb-1.0-client
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\ejb\
       target\daytrader-ejb-1.0-client.jar
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\ejb\
       target\daytrader-ejb-1.0.jar to
       C:\[...]\.m2\repository\org\apache\geronimo\samples\
       daytrader\daytrader-ejb\1.0\daytrader-ejb-1.0.jar
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\ejb\
       target\daytrader-ejb-1.0-client.jar to
       C:\[...]\.m2\repository\org\apache\geronimo\samples\
       daytrader\daytrader-ejb\1.0\daytrader-ejb-1.0-client.jar
```

Maven has created both the EJB JAR and the client EJB JAR and installed them in your local Maven repository.

The EJB plugin has several other configuration elements that you can use to suit your exact needs. Please refer to the EJB plugin documentation on http://maven.apache.org/plugins/maven-ejb-plugin/.

> Early adopters of EJB3 may be interested to know how Maven supports EJB3. At the time of writing, the EJB3 specification is still not final. There is a working prototype of an EJB3 Maven plugin, however in the future it will be added to the main EJB plugin after the specification is finalized. Stay tuned!

4.5. Building an EJB Module With Xdoclet

If you've been developing a lot of EJBs (version 1 and 2) you have probably used XDoclet to generate all of the EJB interfaces and deployment descriptors for you. Using XDoclet is easy: by adding Javadoc annotations to your classes, you can run the XDoclet processor to generate those files for you. When writing EJBs it means you simply have to write your EJB implementation class and XDoclet will generate the Home interface, the Remote and Local interfaces, the container-specific deployment descriptors, and the `ejb-jar.xml` descriptor.

> Note that if you're an EJB3 user, you can safely skip this section – you won't need it!

Here's an extract of the `TradeBean` session EJB using Xdoclet:

```
/**
 * Trade Session EJB manages all Trading services
 *
 * @ejb.bean
 *     display-name="TradeEJB"
 *     name="TradeEJB"
 *     view-type="remote"
 *     impl-class-name=
 *         "org.apache.geronimo.samples.daytrader.ejb.TradeBean"
 * @ejb.home
 *     generate="remote"
 *     remote-class=
 *         "org.apache.geronimo.samples.daytrader.ejb.TradeHome"
 * @ejb.interface
 *     generate="remote"
 *     remote-class=
 *         "org.apache.geronimo.samples.daytrader.ejb.Trade"
 * […]
*/
public class TradeBean implements SessionBean
{
    […]
    /**
     * Queue the Order identified by orderID to be processed in a
     * One Phase commit […]
     *
     * @ejb.interface-method
     *     view-type="remote"
     * @ejb.transaction
     *     type="RequiresNew"
     *[…]
     */
    public void queueOrderOnePhase(Integer orderID)
        throws javax.jms.JMSException, Exception
    […]
```

To demonstrate XDoclet, create a copy of the DayTrader `ejb` module called `ejb-xdoclet`. As you can see in Figure 4-6, the project's directory structure is the same as in Figure 4-5, but you don't need the ejb-jar.xml file anymore as it's going to be generated by Xdoclet.

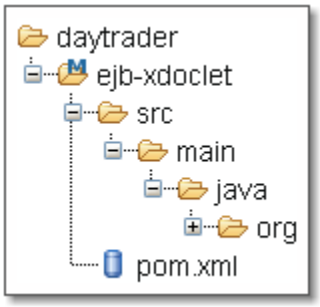


Figure 4-6: Directory structure for the DayTrader `ejb` module when using Xdoclet

The other difference is that you only need to keep the *Bean.java classes and remove all of the Home, Local and Remote interfaces as they'll also get generated.

Now you need to tell Maven to run XDoclet on your project. Since XDoclet generates source files, this has to be run before the compilation phase occurs. This is achieved by using the Maven XDoclet plugin and binding it to the generate-sources life cycle phase. Here's the portion of the pom.xml that configures the plugin:

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>xdoclet-maven-plugin</artifactId>
  <executions>
   <execution>
    <phase>generate-sources</phase>
     <goals>
      <goal>xdoclet</goal>
    </goals>
     <configuration>
      <tasks>
       <ejbdoclet verbose="true" force="true" ejbSpec="2.1" destDir=
           "${project.build.directory}/generated-sources/xdoclet">
         <fileset dir="${project.build.sourceDirectory}">
           <include name="**/*Bean.java"></include>
           <include name="**/*MDB.java"></include>
         </fileset>
         <homeinterface/>
         <remoteinterface/>
         <localhomeinterface/>
         <localinterface/>
         <deploymentdescriptor
             destDir="${project.build.outputDirectory}/META-INF"/>
       </ejbdoclet>
      </tasks>
    </configuration>
  </execution>
 </executions>
</plugin>
```

The XDoclet plugin is configured within an execution element. This is required by Maven to bind the xdoclet goal to a phase. The plugin generates sources by default in ${project.build.directory}/generated-sources/xdoclet (you can configure this using the generatedSourcesDirectory configuration element).

It also tells Maven that this directory contains sources that will need to be compiled when the compile phase executes. Finally, in the tasks element you use the ejbdoclet Ant task provided by the XDoclet project (for reference documentation see http://xdoclet.sourceforge.net/xdoclet/ant/xdoclet/modules/ejb/EjbDocletTask.html).

In practice you can use any XDoclet task (or more generally any Ant task) within the tasks element, but here the need is to use the ejbdoclet task to instrument the EJB class files. In addition, the XDoclet plugin will also trigger Maven to download the XDoclet libraries from Maven's remote repository and add them to the execution classpath.

Executing `mvn install` now automatically executes XDoclet and compiles the generated files:

```
C:\dev\m2book\code\j2ee\daytrader\ejb-xdoclet>mvn install
[…]
[INFO] [xdoclet:xdoclet {execution: default}]
[INFO] Initializing DocletTasks!!!
[INFO] Executing tasks
10 janv. 2006 16:53:50 xdoclet.XDocletMain start
INFO: Running <homeinterface/>
Generating Home interface for
  'org.apache.geronimo.samples.daytrader.ejb.TradeBean'.
[…]
INFO: Running <remoteinterface/>
Generating Remote interface for
  'org.apache.geronimo.samples.daytrader.ejb.TradeBean'.
[…]
10 janv. 2006 16:53:50 xdoclet.XDocletMain start
INFO: Running <localhomeinterface/>
Generating Local Home interface for
  'org.apache.geronimo.samples.daytrader.ejb.AccountBean'.
[…]
10 janv. 2006 16:53:51 xdoclet.XDocletMain start
INFO: Running <localinterface/>
Generating Local interface for
  'org.apache.geronimo.samples.daytrader.ejb.AccountBean'.
[…]
10 janv. 2006 16:53:51 xdoclet.XDocletMain start
INFO: Running <deploymentdescriptor/>
Generating EJB deployment descriptor (ejb-jar.xml).
[…]
[INFO] [ejb:ejb]
[INFO] Building ejb daytrader-ejb-1.0
[…]
```

You might also want to try XDoclet2. It's based on a new architecture but the tag syntax is backward-compatible in most cases. There's also a Maven 2 plugin for XDoclet2 at http://xdoclet.codehaus.org/Maven2+Plugin. However, it should be noted that XDoclet2 is a work in progress and is not yet fully mature, nor does it boast all the plugins that XDoclet1 has.

4.6. Deploying EJBs

Now that you know how to build an EJB project, you will learn how to deploy it. Later, you will also learn how to test it automatically, in the *Testing J2EE Applications* section of this chapter. Let's discover how you can automatically start a container and deploy your EJBs into it.

First, you will need to have Maven start the container automatically. To do so you're going to use the Maven plugin for Cargo. Cargo is a framework for manipulating containers. It offers generic APIs (Java, Ant, Maven 1, Maven 2, Netbeans, IntelliJ IDEA, etc.) for performing various actions on containers such as starting, stopping, configuring them and deploying modules to them. In this example, the JBoss container will be used.

Edit the `ejb/pom.xml` file and add the following Cargo plugin configuration:

```
<build>
  <plugins>
    [...]
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <configuration>
        <container>
          <containerId>jboss4x</containerId>
          <zipUrlInstaller>
            <url>http://internap.dl.sourceforge.net/
              sourceforge/jboss/jboss-4.0.2.zip</url>
            <installDir>${installDir}</installDir>
          </zipUrlInstaller>
        </container>
      </configuration>
    </plugin>
  </plugins>
</build>
```

If you want to debug Cargo's execution, you can use the log element to specify a file where Cargo logs will go and you can also use the output element to specify a file where the container's output will be dumped. For example:

```
<container>
  <containerId>jboss4x</containerId>
  <output>${project.build.directory}/jboss4x.log</output>
  <log>${project.build.directory}/cargo.log</log>
  [...]
```

See http://cargo.codehaus.org/Debugging for full details.

In the container element you tell the Cargo plugin that you want to use JBoss 4.x (`containerId` element) and that you want Cargo to download the JBoss 4.0.2 distribution from the specified URL and install it in `${installDir}`. The location where Cargo should install JBoss is a user-dependent choice and this is why the `${installDir}` property was introduced. In order to build this project you need to create a Profile where you define the `${installDir}` property's value.

As explained in Chapter 3, you can define a profile in the POM, in a `profiles.xml` file, or in a `settings.xml` file. In this case, as the content of the Profile is user-dependent you wouldn't want to define it in the POM. Nor should the content be shared with other Maven projects at large, in a `settings.xml` file. Thus the best place is to create a `profiles.xml` file in the `ejb/` directory:

```
<profilesXml>
  <profiles>
    <profile>
      <id>vmassol</id>
      <properties>
        <installDir>c:/apps/cargo-installs</installDir>
      </properties>
    </profile>
  </profiles>
  <activeProfiles>
    <activeProfile>vmassol</activeProfile>
  </activeProfiles>
</profilesXml>
```

This sample `profiles.xml` file defines a profile named `vmassol`, activated by default and in which the `${installDir}` property points to `c:/apps/cargo-installs`.

> It's also possible to tell Cargo that you already have JBoss installed locally. In that case replace the `zipURLInstaller` element with a home element. For example: `<home>c:/apps/jboss-4.0.2</home>`

That's all you need to have a working build and to deploy the EJB JAR into JBoss. The Cargo plugin does all the work: it provides a default JBoss configuration (using port 8080 for example), it detects that the Maven project is producing an EJB from the packaging element and it automatically deploys it when the container is started.

Of course, the EJB JAR should first be created, so run `mvn package` to generate it, then start JBoss and deploy the EJB JAR by running `mvn cargo:start` (or `mvn package cargo:start` to do it all at once):

```
C:\dev\m2book\code\j2ee\daytrader\ejb>mvn cargo:start
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'cargo'.
[INFO] ------------------------------------------------------------------------
[INFO] Building DayTrader :: EJBs
[INFO]    task-segment: [cargo:start]
[INFO] ------------------------------------------------------------------------
[INFO] [cargo:start]
[INFO] [talledLocalContainer] Parsed JBoss version = [4.0.2]
[INFO] [talledLocalContainer] JBoss 4.0.2 starting...
[INFO] [talledLocalContainer] JBoss 4.0.2 started on port [8080]
[INFO] Press Ctrl-C to stop the container...
```

That's it! JBoss is running, and the EJB JAR has been deployed.

> As you have told Cargo to download and install JBoss, the first time you execute `cargo:start` it will take some time, especially if you are on a slow connection. Subsequent calls will be fast as Cargo will not download JBoss again.

If the container was already started and you wanted to just deploy the EJB, you would run the cargo:deploy goal. Finally, to stop the container call `mvn cargo:stop`.

Cargo has many other configuration options such as the possibility of using an existing container installation, modifying various container parameters, deploying on a remote machine, and more. Check the documentation at http://cargo.codehaus.org/Maven2+plugin.

4.7. Building a Web Application Project

Now, let's focus on building the DayTrader web module. The layout is the same as for a JAR module (see the first two chapters of this book), except that there is an additional `src/main/webapp` directory for locating Web application resources such as HTML pages, JSPs, WEB-INF configuration files, etc. (see Figure 4-7).

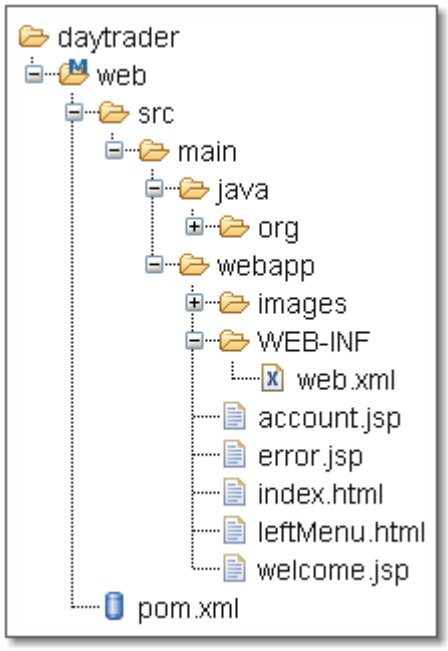


Figure 4-7: Directory structure for the DayTrader web module showing some Web application resources

As usual everything is specified in the `pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>daytrader-web</artifactId>
  <name>DayTrader :: Web Application</name>
  <packaging>war</packaging>
  <description>DayTrader Web</description>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.samples.daytrader</groupId>
      <artifactId>daytrader-ejb</artifactId>
      <version>1.0</version>
      <type>ejb-client</type>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-j2ee_1.4_spec</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

You start by telling Maven that it's building a project generating a WAR:

```
<packaging>war</packaging>
```

Next, you specify the required dependencies. The reason you are building this web module after the `ejb` module is because the web module's servlets call the EJBs. Therefore, you need to add a dependency on the `ejb` module in `web/pom.xml`:

```
<dependency>
  <groupId>org.apache.geronimo.samples.daytrader</groupId>
  <artifactId>daytrader-ejb</artifactId>
  <version>1.0</version>
  <type>ejb-client</type>
</dependency>
```

Note that you're specifying a type of `ejb-client` and not `ejb`. This is because the servlets are a client of the EJBs. Therefore, the servlets only need the EJB client JAR in their classpath to be able to call the EJBs. This is why you told the EJB plugin to generate a client JAR earlier on in `ejb/pom.xml`.

Depending on the main EJB JAR would also work, but it's not necessary and would increase the size of the WAR file. It's always cleaner to depend on the minimum set of required classes, for example to prevent coupling.

> If you add a dependency on a WAR, then the WAR you generate will be overlaid with the

content of that dependent WAR, allowing the aggregation of multiple WAR files. However, only files not in the existing Web application will be added, and files such as `web.xml` won't be merged. An alternative is to use the `uberwar` goal from the Cargo Maven Plugin (see http://cargo.codehaus.org/Merging+WAR+files).

The final dependency listed is the J2EE JAR as your web module uses servlets and calls EJBs. Again, the Geronimo J2EE specifications JAR is used with a provided scope (as seen previously when building the EJB).

As you know, Maven 2 supports transitive dependencies. When it generates your WAR, it recursively adds your module's dependencies, unless their scope is test or provided. This is why we defined the J2EE JAR using a `provided` scope in the web module's `pom.xml`. Otherwise it would have surfaced in the `WEB-INF/lib` directory of the generated WAR.

The configuration is very simple because the defaults from the WAR plugin are being used. As seen in the introduction, it's a good practice to use the default conventions as much as possible, as it reduces the size of the `pom.xml` file and reduces maintenance.

Running `mvn` install generates the WAR and installs it in your local repository:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn install
[...]
[INFO] [war:war]
[INFO] Exploding webapp...
[INFO] Copy webapp resources to
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Assembling webapp daytrader-web in
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Generating war
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
[INFO] Building war:
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
[INFO] [install:install]
[INFO] Installing
       C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war
       to C:\[...]\.m2\repository\org\apache\geronimo\samples\daytrader\
       daytrader-web\1.0\daytrader-web-1.0.war
```

Table 4-1 lists some other parameters of the WAR plugin that you may wish to configure.

Table 4-1: WAR plugin configuration properties

Configuration property	Default value	Description
`warSourceDirectory`	`${basedir}/src/main/webapp`	Location of Web application resources to include in the WAR.
`webXml`	The `web.xml` file found in `${warSourceDirectory}/WEB-INF/web.xml`	Specify where to find the `web.xml` file.
`warSourceIncludes/warSourceExcludes`	All files are included	Specify the files to include/exclude from the generated WAR.
`warName`	`${project.build.finalName}`	Name of the generated WAR.

For the full list, see the reference documentation for the WAR plugin at http://maven.apache.org/plugins/maven-war-plugin/.

4.8. Improving Web Development Productivity

If you're doing Web development you know how painful it is to have to package your code in a WAR and redeploy it every time you want to try out a change you made to your HTML, JSP or servlet code. Fortunately, Maven can help. There are two plugins that can alleviate this problem: the Cargo plugin and the Jetty6 plugin. You'll discover how to use the Jetty6 plugin in this section as you've already seen how to use the Cargo plugin in a previous section.

The Jetty6 plugin creates a custom Jetty 6 configuration that is wired to your source tree. The plugin is configured by default to look for resource files in `src/main/webapp`, and it adds the compiled classes in target/classes to its execution classpath. The plugin monitors the source tree for changes, including the `pom.xml` file, the `web.xml` file, the `src/main/webapp` tree, the project dependencies and the compiled classes and classpath resources in target/classes. If any change is detected, the plugin reloads the Web application in Jetty.

A typical usage for this plugin is to develop the source code in your IDE and have the IDE configured to compile classes in target/classes (this is the default when the Maven IDE plugins are used to set up your IDE project). Thus any recompilation in your IDE will trigger a redeploy of your Web application in Jetty, providing an extremely fast turnaround time for development.

Let's try the Jetty6 plugin on the DayTrader web module. Add the following to the `web/pom.xml` file:

```
[…]
<build>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty6-plugin</artifactId>
      <configuration>
        <scanIntervalSeconds>10</scanIntervalSeconds>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>org.apache.geronimo.specs</groupId>
          <artifactId>geronimo-j2ee_1.4_spec</artifactId>
          <version>1.0</version>
          <scope>provided</scope>
        </dependency>
      </dependencies>
    </plugin>
[…]
```

The `scanIntervalSeconds` configuration property tells the plugin to monitor for changes every 10 seconds. The reason for the dependency on the J2EE specification JAR is because Jetty is a servlet engine and doesn't provide the EJB specification JAR. Since the Web application earlier declared that the specification must be provided through the provided scope, adding this dependency to the plugin adds it to the classpath for Jetty.

You execute the Jetty6 plugin by typing `mvn jetty6:run`:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn jetty6:run
[…]
[INFO] [jetty6:run]
[INFO] Configuring Jetty for project:
       Apache Geronimo DayTrader Web Module
[INFO] Webapp source directory is:
C:\dev\m2book\code\j2ee\daytrader\web\src\main\webapp
[INFO] web.xml file located at: C:\dev\m2book\code\j2ee\daytrader\
web\src\main\webapp\WEB-INF\web.xml
[INFO] Classes located at: C:\dev\m2book\code\j2ee\daytrader\
web\target\classes
[INFO] tmp dir for webapp will be
C:\dev\m2book\code\j2ee\daytrader\web\target\jetty-tmp
[INFO] Starting Jetty Server ...
[INFO] No connectors configured, using defaults:
org.mortbay.jetty.nio.SelectChannelConnector listening on 8080
with maxIdleTime 30000
0 [main] INFO org.mortbay.log - Logging to
org.slf4j.impl.SimpleLogger@1242b11 via org.mortbay.log.Slf4jLog
[INFO] Context path = /daytrader-web
[INFO] Webapp directory =
C:\dev\m2book\code\j2ee\daytrader\web\src\main\webapp
[INFO] Setting up classpath ...
[INFO] Finished setting up classpath
[INFO] Started configuring web.xml, resource base=
C:\dev\m2book\code\j2ee\daytrader\web\src\main\webapp
[INFO] Finished configuring web.xml
681 [main] INFO org.mortbay.log - Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Starting scanner at interval of 10 seconds.
```

As you can see, Maven pauses as Jetty is now started. Your Web application has been deployed and the plugin is waiting, listening for changes. Open a browser with the `http://localhost:8080/daytrader-web/register.jsp` URL as shown in Figure 4-8 to see the Web application running.

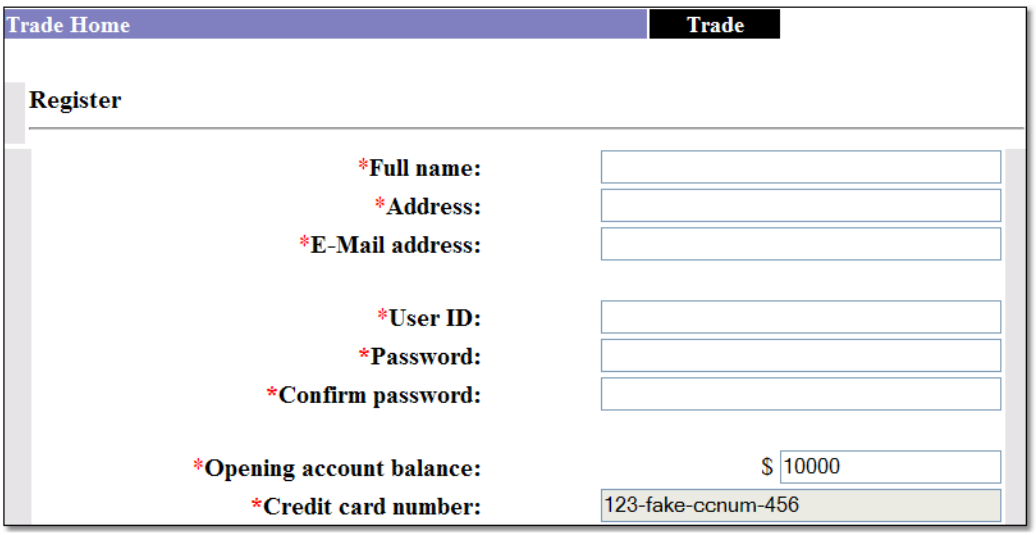


Figure 4-8: DayTrader JSP registration page served by the Jetty6 plugin

Note that the application will fail if you open a page that calls EJBs. The reason is that we have only deployed the Web application here, but the EJBs and all the back end code has not been deployed. In order to make it work you'd need to have your EJB container started with the DayTrader code deployed in it. In practice it's easier to deploy a full EAR as you'll see below.

Now let's try to modify the content of this JSP by changing the opening account balance. Edit `web/src/main/webapp/register.jsp`, search for "10000" and replace it with "90000" (a much better starting amount!):

```
<TD colspan="2" align="right">$<B> </B><INPUT size="20" type="text"
   name="money" value='<%= money==null ? "90000" : money %>'></TD>
```

Now simply refresh your browser (usually the F5 key) and the new value will appear as shown in Figure 4-9:

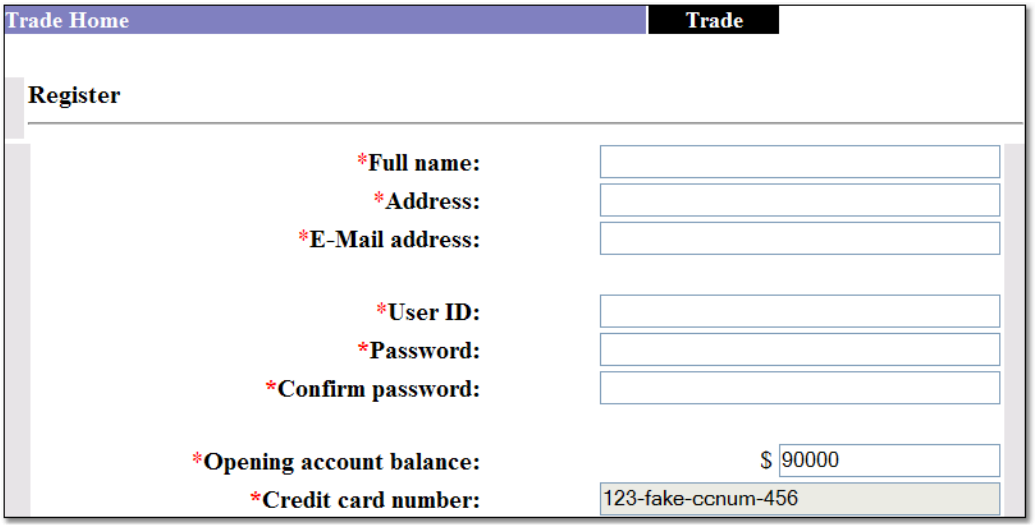


Figure 4-9: Modified registration page automatically reflecting our source change

That's nifty, isn't it? What happened is that the Jetty6 plugin realized the page was changed and it redeployed the Web application automatically. The Jetty container automatically recompiled the JSP when the page was refreshed.

There are various configuration parameters available for the Jetty6 plugin such as the ability to define Connectors and Security realms. For example if you wanted to run Jetty on port 9090 with a user realm defined in `etc/realm.properties`, you would use:

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty6-plugin</artifactId>
  <configuration>
    […]
    <connectors>
      <connector implementation=
          "org.mortbay.jetty.nio.SelectChannelConnector">
        <port>9090</port>
        <maxIdleTime>60000</maxIdleTime>
      </connector>
    </connectors>
    <userRealms>
      <userRealm implementation=
          "org.mortbay.jetty.security.HashUserRealm">
        <name>Test Realm</name>
        <config>etc/realm.properties</config>
      </userRealm>
    </userRealms>
  </configuration>
</plugin>
```

You can also configure the context under which your Web application is deployed by using the `contextPath` configuration element. By default the plugin uses the module's `artifactId` from the POM.

It's also possible to pass in a `jetty.xml` configuration file using the `jettyConfig` configuration element. In that case anything in the `jetty.xml` file will be applied first. For a reference of all configuration options see the Jetty6 plugin documentation at http://jetty.mortbay.org/jetty6/maven-plugin/index.html.

Now imagine that you have an awfully complex Web application generation process, that you have custom plugins that do all sorts of transformations to Web application resource files, possibly generating some files, and so on. The strategy above would not work as the Jetty6 plugin would not know about the custom actions that need to be executed to generate a valid Web application. Fortunately there's a solution.

The WAR plugin has an exploded goal which produces an expanded Web application in the target directory. Calling this goal ensures that the generated Web application is the correct one. The Jetty6 plugin also contains two goals that can be used in this situation:

- `etty6:run-war`: The plugin first runs the package phase which generates the WAR file. Then the plugin deploys the WAR file to the Jetty server and it performs hot redeployments whenever the WAR is rebuilt (by calling `mvn` package from another window, for example) or when the `pom.xml` file is modified.
- `jetty6:run-exploded`: The plugin runs the package phase as with the `jetty6:run-war` goal. Then it deploys the unpacked Web application located in `target/` (whereas the `jetty6:run-war` goal deploys the WAR file). The plugin then watches the following files: `WEB-INF/lib`, `WEB-INF/classes`, `WEB-INF/web.xml` and `pom.xml`; any change to those files results in a hot redeployment.

To demonstrate, execute `mvn jetty6:run-exploded` goal on the web module:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn jetty6:run-exploded
[...]
[INFO] [war:war]
[INFO] Exploding webapp...
[INFO] Copy webapp resources to
C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Assembling webapp daytrader-web in
C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0
[INFO] Generating war C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-
1.0.war
[INFO] Building war: C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-
1.0.war
[INFO] [jetty6:run-exploded]
[INFO] Configuring Jetty for project: DayTrader :: Web Application
[INFO] Starting Jetty Server ...
0 [main] INFO org.mortbay.log - Logging to org.slf4j.impl.SimpleLogger@78bc3b via
org.mortbay.log.Slf4jLog
[INFO] Context path = /daytrader-web
2214 [main] INFO org.mortbay.log - Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Scanning ...
[INFO] Scan complete at Wed Feb 15 11:59:00 CET 2006
[INFO] Starting scanner at interval of 10 seconds.
```

As you can see the WAR is first assembled in the `target` directory and the Jetty plugin is now waiting for changes to happen. If you open another shell and run `mvn` package you'll see the following in the first shell's console:

```
[INFO] Scan complete at Wed Feb 15 12:02:31 CET 2006
[INFO] Calling scanner listeners ...
[INFO] Stopping webapp ...
[INFO] Reconfiguring webapp ...
[INFO] Restarting webapp ...
[INFO] Restart completed.
[INFO] Listeners completed.
[INFO] Scanning ...
```

You're now ready for productive web development. No more excuses!

4.9. Deploying Web Applications

You have already seen how to deploy a Web application for in-place Web development in the previous section, so now the focus will be on deploying a packaged WAR to your target container. This example uses the Cargo Maven plugin to deploy to any container supported by Cargo (see http://cargo.codehaus.org/Containers). This is very useful when you're developing an application and you want to verify it works on several containers.

First, edit the web module's `pom.xml` file and add the Cargo configuration:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      <containerId>${containerId}</containerId>
      <zipUrlInstaller>
        <url>${url}</url>
        <installDir>${installDir}</installDir>
      </zipUrlInstaller>
    </container>
    <configuration>
      <properties>
        <cargo.servlet.port>8280</cargo.servlet.port>
      </properties>
    </configuration>
  </configuration>
</plugin>
```

As you can see this is a configuration similar to the one you have used to deploy your EJBs in the *Deploying EJBs* section of this chapter. There are two differences though:

- Two new properties have been introduced (`containerId` and `url`) in order to make this build snippet generic. Those properties will be defined in a Profile.
- A `cargo.servlet.port` element has been introduced to show how to configure the containers to start on port 8280 instead of the default 8080 port. This is very useful if you have containers already running your machine and you don't want to interfere with them.

As seen in the Deploying EJBs section the `installDir` property is user-dependent and should be defined in a `profiles.xml` file. However, the `containerId` and `url` properties should be shared for all users of the build. Thus, add the following profiles to the `web/pom.xml` file:

```
[...]
  </build>
  <profiles>
    <profile>
      <id>jboss4x</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <containerId>jboss4x</containerId>
        <url>http://ovh.dl.sourceforge.net/sourceforge/jboss/jboss4.0.2.zip</url>
      </properties>
    </profile>
    <profile>
      <id>tomcat5x</id>
      <properties>
        <containerId>tomcat5x</containerId>
        <url>http://www.apache.org/dist/jakarta/tomcat-5/v5.0.30/bin/
          jakarta-tomcat-5.0.30.zip</url>
      </properties>
    </profile>
  </profiles>
</project>
```

You have defined two profiles: one for JBoss and one for Tomcat and the JBoss profile is defined as active by default (using the `activation` element). You could add as many profiles as there are containers you want to execute your Web application on.

Executing `mvn install cargo:start` generates the WAR, starts the JBoss container and deploys the WAR into it:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn install cargo:start
[...]
[INFO] [cargo:start]
[INFO] [talledLocalContainer] Parsed JBoss version = [4.0.2]
[INFO] [talledLocalContainer] JBoss 4.0.2 starting...
[INFO] [talledLocalContainer] JBoss 4.0.2 started on port [8280]
[INFO] Press Ctrl-C to stop the container...
```

To deploy the WAR using Tomcat tell Maven to execute the `tomcat5x` profile by typing `mvn cargo:start -Ptomcat5x`:

```
C:\dev\m2book\code\j2ee\daytrader\web>mvn cargo:start -Ptomcat5x
[...]
[INFO] [cargo:start]
[INFO] [talledLocalContainer] Tomcat 5.0.30 starting...
[INFO] [CopyingLocalDeployer] Deploying
       [C:\dev\m2book\code\j2ee\daytrader\web\target\daytrader-web-1.0.war]
       to [C:\[...]\Temp\cargo\50866\webapps]...
[INFO] [talledLocalContainer] Tomcat 5.0.30 started on port [8280]
[INFO] Press Ctrl-C to stop the container...
```

This is useful for development and to test that your code deploys and works. However, once this is verified you'll want a solution to deploy your WAR into an integration platform. One solution is to have your container running on that integration platform and to perform a remote deployment of your WAR to it.

To deploy the DayTrader's WAR to a running JBoss server on machine `remoteserver` and executing on port 80, you would need the following Cargo plugin configuration in `web/pom.xml`:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      <containerId>jboss4x</containerId>
      <type>remote</type>
    </container>
    <configuration>
      <type>runtime</type>
      <properties>
        <cargo.hostname>${remoteServer}</cargo.hostname>
        <cargo.servlet.port>${remotePort}</cargo.servlet.port>
        <cargo.remote.username>${remoteUsername}</cargo.remote.username>
        <cargo.remote.password>${remotePassword}</cargo.remote.password>
      </properties>
    </configuration>
  </configuration>
</plugin>
```

When compared to the configuration for a local deployment above, the changes are:

- A remote container and configuration type to tell Cargo that the container is remote and not under Cargo's management,
- Several configuration properties (especially a user name and password allowed to deploy on the remote JBoss container) to specify all the details required to perform the remote deployment. All the properties introduced need to be declared inside the POM for those shared with other users and in the `profiles.xml` file (or the `settings.xml` file) for those user-dependent. Note that there was no need to specify a deployment URL as it is computed automatically by Cargo.

Check the Cargo reference documentation for all details on deployments at http://cargo.codehaus.org/Deploying+to+a+running+container.

At this stage you have learned how to build and deploy EJBs and WARs. Let's now discover how to build Web services modules.

4.10. Building a Web Services Client Project

Web services are a part of many J2EE applications, and Maven's ability to integrate toolkits can make them easier to add to the build process. For example, the Maven plugin called *Axis Tools plugin* takes WSDL files and generates the Java files needed to interact with the Web services it defines. As the name suggests, the plugin uses the Axis framework (http://ws.apache.org/axis/java/), and this will be used from DayTrader's `wsappclient` module.

Axis generates the following:

Table 4-2: Axis generated classes

WSDL clause	Java class(es) generated
For each entry in the type section	A Java class A holder if this type is used as an in-out/out parameter
For each port type	A Java interface
For each binding	A stub class
For each service	A service interface A service implementation (the locator)

For more details on the generation process, see http://ws.apache.org/axis/java/user-guide.html#WSDL2JavaBuildingStubsSkeletonsAndDataTypesFromWSDL.

Figure 4-10 shows the directory structure of the `wsappclient` module. As you may notice, the WSDL files `are in src/main/wsdl`, which is the default used by the Axis Tools plugin:

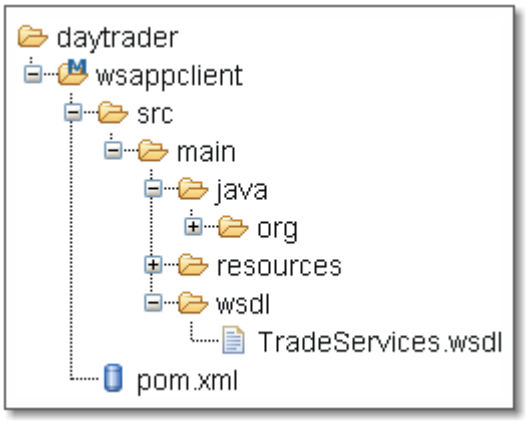


Figure 4-10: Directory structure of the `wsappclient` module

The location of WSDL source can be customized using the `sourceDirectory` property. For example:

```
[...]
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>axistools-maven-plugin</artifactId>
  <configuration>
    <sourceDirectory>
      src/main/resources/META-INF/wsdl
    </sourceDirectory>
  </configuration>
[...]
```

In order to generate the Java source files from the `TradeServices.wsdl` file, the `wsappclient/pom.xml` file must declare and configure the Axis Tools plugin:

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>axistools-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>wsdl2java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Note that there's no need to define a phase in the execution element as the `wsdl2java` goal is bound to the `generate-sources` phase by default.

At this point f you were to execute the build, it would fail. This is because after the sources are generated, you will require a dependency on Axis and Axis JAXRPC in your `pom.xml`. While you might expect the Axis Tools plugin to define this for you, it is required for two reasons: it allows you to control what version of the dependency to use regardless of what the Axis Tools plugin was built against, and more importantly, it allows users of your project to automatically get the dependency transitively. Similarly, any tools that report on the POM will be able to recognize the dependency.

As before, you need to add the J2EE specifications JAR to compile the project's Java sources. Thus add the following two dependencies to your POM:

```
<dependencies>
  <dependency>
    <groupId>axis</groupId>
    <artifactId>axis</artifactId>
    <version>1.2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>axis</groupId>
    <artifactId>axis-jaxrpc</artifactId>
    <version>1.2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-j2ee_1.4_spec</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

The Axis JAR depends on the Mail and Activation Sun JARs which cannot be redistributed by Maven. Thus, they are not present on ibiblio and you'll need to install them manually. Run `mvn install` and Maven will fail and print the installation instructions.

Running the build with `mvn install` leads to:

```
C:\dev\m2book\code\j2ee\daytrader\wsappclient>mvn install
[...]
[INFO] [axistools:wsdl2java {execution: default}]
[INFO] about to add compile source root
[INFO] processing wsdl:
       C:\dev\m2book\code\j2ee\daytrader\wsappclient\
       src\main\wsdl\TradeServices.wsdl
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 13 source files to
C:\dev\m2book\code\j2ee\daytrader\wsappclient\target\classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [surefire:test]
[INFO] No tests to run.
[INFO] [jar:jar]
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\wsappclient\
       target\daytrader-wsappclient-1.0.jar
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\wsappclient\
       target\daytrader-wsappclient-1.0.jar to
       C:\[...]\.m2\repository\org\apache\geronimo\samples\daytrader\
       daytrader-wsappclient\1.0\daytrader-wsappclient-1.0.jar
[...]
```

Note that the `daytrader-wsappclient` JAR now includes the class files compiled from the generated source files, in addition to the sources from the standard source directory.

The Axis Tools plugin boasts several other goals including java2wsdl that is useful for generating the server-side WSDL file from handcrafted Java classes. The generated WSDL file could then be injected into the Web Services client module to generate client-side Java files. But that's another story... The Axis Tools reference documentation can be found at http://mojo.codehaus.org/axistools-maven-plugin/.

4.11. Building an EAR Project

You have now built all the individual modules. It's time to package the server module artifacts (EJB and WAR) into an EAR for convenient deployment. The ear module's directory structure can't be any simpler... it solely consists of a `pom.xml` file (see Figure 4-11).

Figure 4-11: Directory structure of the `ear` module

As usual the magic happens in the `pom.xml` file. Start by defining that this is an EAR project by using the `packaging` element:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>daytrader-ear</artifactId>
  <name>DayTrader :: Enterprise Application</name>
  <packaging>ear</packaging>
  <description>DayTrader EAR</description>
```

Next, define all of the dependencies that need to be included in the generated EAR:

```
<dependencies>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-wsappclient</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-web</artifactId>
    <version>1.0</version>
    <type>war</type>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-ejb</artifactId>
    <version>1.0</version>
    <type>ejb</type>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader-streamer</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
```

Finally, you need to configure the Maven EAR plugin by giving it the information it needs to automatically generate the `application.xml` deployment descriptor file. This includes the display name to use, the description to use, and the J2EE version to use. It is also necessary to tell the EAR plugin which of the dependencies are Java modules, Web modules, and EJB modules. At the time of writing, the EAR plugin supports the following module types: `ejb`, `war`, `jar`, `ejb-client`, `rar`, `ejb3`, `par`, `sar` and `wsr`.

By default, all dependencies are included, with the exception of those that are optional, or those with a scope of test or provided. However, it is often necessary to customize the inclusion of some dependencies such as shown in this example:

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ear-plugin</artifactId>
        <configuration>
          <displayName>Trade</displayName>
          <description>
            DayTrader Stock Trading Performance Benchmark Sample
          </description>
          <version>1.4</version>
          <modules>
            <javaModule>
              <groupId>org.apache.geronimo.samples.daytrader</groupId>
              <artifactId>daytrader-streamer</artifactId>
              <includeInApplicationXml>true</includeInApplicationXml>
            </javaModule>
            <javaModule>
              <groupId>org.apache.geronimo.samples.daytrader</groupId>
              <artifactId>daytrader-wsappclient</artifactId>
              <includeInApplicationXml>true</includeInApplicationXml>
            </javaModule>
            <webModule>
              <groupId>org.apache.geronimo.samples.daytrader</groupId>
              <artifactId>daytrader-web</artifactId>
              <contextRoot>/daytrader</contextRoot>
            </webModule>
          </modules>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Here, the `contextRoot` element is used for the `daytrader-web` module definition to tell the EAR plugin to use that context root in the generated `application.xml` file.

You should also notice that you have to specify the `includeInApplicationXml` element in order to include the streamer and `wsappclient` libraries into the EAR. By default, only EJB client JARs are included when specified in the Java modules list.

It is also possible to configure where the JARs' Java modules will be located inside the generated EAR. For example, if you wanted to put the libraries inside a lib subdirectory of the EAR you would use the `bundleDir` element:

```
<javaModule>
  <groupId>org.apache.geronimo.samples.daytrader</groupId>
  <artifactId>daytrader-streamer</artifactId>
  <includeInApplicationXml>true</includeInApplicationXml>
  <bundleDir>lib</bundleDir>
</javaModule>
<javaModule>
  <groupId>org.apache.geronimo.samples.daytrader</groupId>
  <artifactId>daytrader-wsappclient</artifactId>
  <includeInApplicationXml>true</includeInApplicationXml>
  <bundleDir>lib</bundleDir>
</javaModule>
```

In order not to have to repeat the `bundleDir` element for each Java module definition you can instead use the `defaultJavaBundleDir` element:

```
   [...]
 <defaultBundleDir>lib</defaultBundleDir>
 <modules>
   <javaModule>
     [...]
   </javaModule>
 [...]
```

There are some other configuration elements available in the EAR plugin which you can find out by checking the reference documentation on http://maven.apache.org/plugins/maven-ear-plugin.

The streamer module's build is not described in this chapter because it's a standard build generating a JAR. However the ear module depends on it and thus you'll need to have the Streamer JAR available in your local repository before you're able to run the ear module's build. Run `mvn install` in `daytrader/streamer`.

To generate the EAR, run `mvn install`:

```
C:\dev\m2book\code\j2ee\daytrader\ear>mvn install
[…]
[INFO] [ear:generate-application-xml]
[INFO] Generating application.xml
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [ear:ear]
[INFO] Copying artifact [jar:org.apache.geronimo.samples.daytrader:
       daytrader-streamer:1.0] to [daytrader-streamer-1.0.jar]
[INFO] Copying artifact [jar:org.apache.geronimo.samples.daytrader:
       daytrader-wsappclient:1.0] to
       [daytrader-wsappclient-1.0.jar]
[INFO] Copying artifact [war:org.apache.geronimo.samples.daytrader:
       daytrader-web:1.0] to [daytrader-web-1.0.war]
[INFO] Copying artifact [ejb:org.apache.geronimo.samples.daytrader:
       daytrader-ejb:1.0] to[daytrader-ejb-1.0.jar]
[INFO] Copying artifact
       [ejb-client:org.apache.geronimo.samples.daytrader:
       daytrader-ejb:1.0] to [daytrader-ejb-1.0-client.jar]
[INFO] Could not find manifest file:
       C:\dev\m2book\code\j2ee\daytrader\ear\src\main\application\
       META-INF\MANIFEST.MF - Generating one
[INFO] Building jar: C:\dev\m2book\code\j2ee\daytrader\ear\
        target\daytrader-ear-1.0.ear
[INFO] [install:install]
[INFO] Installing C:\dev\m2book\code\j2ee\daytrader\ear\
       target\daytrader-ear-1.0.ear to
       C:\[...]\.m2\repository\org\apache\geronimo\samples\
       daytrader\daytrader-ear\1.0\daytrader-ear-1.0.ear
```

You should review the generated `application.xml` to prove that it has everything you need:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd"
    version="1.4">
  <description>
    DayTrader Stock Trading Performance Benchmark Sample
  </description>
  <display-name>Trade</display-name>
  <module>
    <java>daytrader-streamer-1.0.jar</java>
  </module>
  <module>
    <java>daytrader-wsappclient-1.0.jar</java>
  </module>
  <module>
    <web>
      <web-uri>daytrader-web-1.0.war</web-uri>
      <context-root>/daytrader</context-root>
    </web>
  </module>
  <module>
    <ejb>daytrader-ejb-1.0.jar</ejb>
  </module>
</application>
```

This looks good. The next section will demonstrate how to deploy this EAR into a container.

4.12. Deploying a J2EE Application

You have already learned how to deploy EJBs and WARs into a container individually. Deploying EARs follows the same principle. In this example, you'll deploy the DayTrader EAR into Geronimo. Geronimo is somewhat special among J2EE containers in that deploying requires calling the *Deployer* tool with a deployment plan.

A plan is an XML file containing configuration information such as how to map CMP entity beans to a specific database, how to map J2EE resources in the container, etc. Like any other container, Geronimo also supports having this deployment descriptor located within the J2EE archives you are deploying.

However, it is recommended that you use an external plan file so that the deployment configuration is independent from the archives getting deployed, enabling the Geronimo plan to be modified to suit the deployment environment.

> The DayTrader application does not deploy correctly when using the JDK 5 or newer. You'll need to use the JDK 1.4 for this section and the following.

To get started, store the deployment plan in `ear/src/main/deployment/geronimo/plan.xml`, as shown on Figure 4-2.

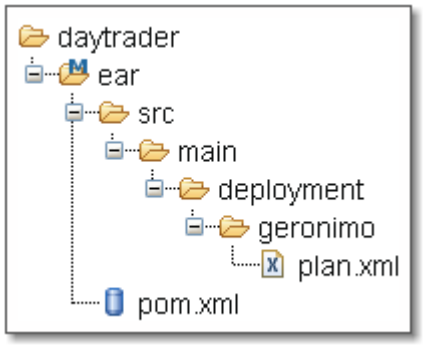


Figure 4-12: Directory structure of the `ear` module showing the Geronimo deployment plan

How do you perform the deployment with Maven? One option would be to use Cargo as demonstrated earlier in the chapter. You would need the following `pom.xml` configuration snippet:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      <containerId>geronimo1x</containerId>
      <zipUrlInstaller>
        <url>http://www.apache.org/dist/geronimo/1.0/
             geronimo-tomcat-j2ee-1.0.zip</url>
        <installDir>${installDir}</installDir>
      </zipUrlInstaller>
    </container>
    <deployer>
      <deployables>
        <deployable>
          <properties>
            <plan>${basedir}/src/main/deployment/dayTrader-plan.xm</plan>
          </properties>
        </deployable>
      </deployables>
    </deployer>
  </configuration>
</plugin>
```

However, in this section you'll learn how to use the Maven Exec plugin. This plugin can execute any process. You'll use it to run the Geronimo Deployer tool to deploy your EAR into a running Geronimo container. Even though it's recommended to use a specific plugin like the Cargo plugin, learning how to use the Exec plugin is useful in situations where you want to do something slightly different, or when Cargo doesn't support the container you want to deploy into. Modify the `ear/pom.xml` to configure the Exec plugin:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <configuration>
    <executable>java</executable>
    <arguments>
      <argument>-jar</argument>
      <argument>${geronimo.home}/bin/deployer.jar</argument>
      <argument>--user</argument>
      <argument>system</argument>
      <argument>--password</argument>
      <argument>manager</argument>
      <argument>deploy</argument>
      <argument>
        ${project.build.directory}/${project.build.finalName}.ear
      </argument>
      <argument>
        ${basedir}/src/main/deployment/geronimo/plan.xml
      </argument>
   </arguments>
 </configuration>
</plugin>
```

You may have noticed that you're using a `geronimo.home` property that has not been defined anywhere. As you've seen in the EJB and WAR deployment sections above and in previous chapters it's possible to create properties that are defined either in a properties section of the POM or in a Profile. As the location where Geronimo is installed varies depending on the user, put the following profile in a `profiles.xml` or `settings.xml` file:

```
<profiles>
  <profile>
    <id>vmassol</id>
    <properties>
      <geronimo.home>c:/apps/geronimo-1.0-tomcat</geronimo.home>
    </properties>
  </profile>
</profiles>
```

At execution time, the Exec plugin will transform the executable and arguments elements above in the following command line:

```
java -jar c:/apps/geronimo-1.0-tomcat/bin/deployer.jar
-user system -password manager deploy
C:\dev\m2book\code\j2ee\daytrader\ear\target/daytrader-ear-1.0.ear
C:\dev\m2book\code\j2ee\daytrader\ear/src/main/deployment/geronimo/plan.xml
```

First, start your preinstalled version of Geronimo and run `mvn exec:exec`:

```
C:\dev\m2book\code\j2ee\daytrader\ear>mvn exec:exec
[...]
[INFO] [exec:exec]
[INFO]     Deployed Trade
[INFO]
[INFO]       `-> daytrader-web-1.0-SNAPSHOT.war
[INFO]
[INFO]       `-> daytrader-ejb-1.0-SNAPSHOT.jar
[INFO]
[INFO]       `-> daytrader-streamer-1.0-SNAPSHOT.jar
[INFO]
[INFO]       `-> daytrader-wsappclient-1.0-SNAPSHOT.jar
[INFO]
[INFO]       `-> TradeDataSource
[INFO]
[INFO]       `-> TradeJMS
```

You can now access the DayTrader application by opening your browser to http://localhost:8080/daytrader/.

You will need to make sure that the DayTrader application is not already deployed before running the exec:exec goal or it will fail. Since Geronimo 1.0 comes with the DayTrader application bundled, you should first stop it, by creating a new execution of the `Exec` plugin or run the following:

```
C:\apps\geronimo-1.0-tomcat\bin>deploy stop
  geronimo/daytrader-derby-tomcat/1.0/car
```

If you need to undeploy the DayTrader version that you've built above you'll use the “Trade” identifier instead:

```
C:\apps\geronimo-1.0-tomcat\bin>deploy undeploy Trade
```

4.13. Testing J2EE Applications

In this last section you'll learn how to automate functional testing of the EAR built previously. At the time of writing, Maven only supports integration and functional testing by creating a separate module. To achieve this, create a functional-tests module as shown in Figure 4-12.

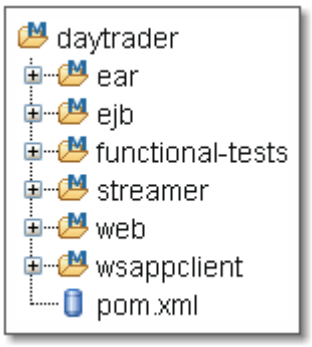


Figure 4-13: The new functional-tests module amongst the other DayTrader modules

You need to add this module to the list of modules in the `daytrader/pom.xml` so that it's built along with the others. Functional tests can take a long time to execute, so you can define a profile to build the functional-tests module only on demand. For example, modify the `daytrader/pom.xml` file as follows:

```
<modules>
  <module>ejb</module>
  <module>web</module>
  <module>streamer</module>
  <module>wsappclient</module>
  <module>ear</module>
</modules>
<profiles>
  <profile>
    <id>functional-test</id>
    <activation>
      <property>
        <name>enableCiProfile</name>
        <value>true</value>
      </property>
    </activation>
    <modules>
      <module>functional-tests</module>
    </modules>
  </profile>
</profiles>
```

For more information on the `ciProfile` configuration, see Chapter 7.

This means that running `mvn install` will not build the functional-tests module, but running `mvn install-Pfunctional-test` will.

Now, take a look in the functional-tests module itself. Figure 4-1 shows how it is organized:

- Functional tests are put in `src/it/java`,
- Classpath resources required for the tests are put in `src/it/resources` (this particular example doesn't have any resources).
- The Geronimo deployment Plan file is located in `src/deployment/geronimo/plan.xml`.

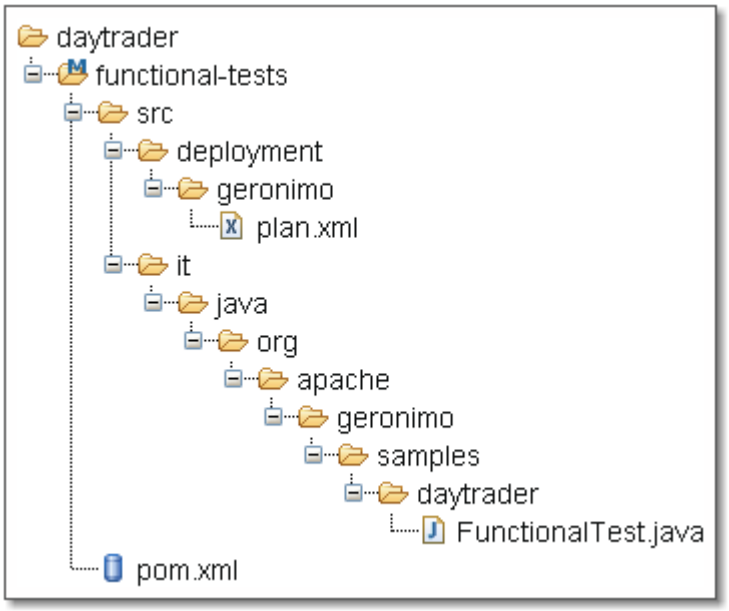


Figure 4-14: Directory structure for the functional-tests module

As this module does not generate an artifact, the packaging should be defined as `pom`. However, the compiler and Surefire plugins are not triggered during the build life cycle of projects with a `pom` packaging, so these need to be configured in the `functional-tests/pom.xml` file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.geronimo.samples.daytrader</groupId>
    <artifactId>daytrader</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>daytrader-tests</artifactId>
  <name>DayTrader :: Functional Tests</name>
  <packaging>pom</packaging>
  <description>DayTrader Functional Tests</description>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.samples.daytrader</groupId>
      <artifactId>daytrader-ear</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>ear</type>
      <scope>provided</scope>
    </dependency>
    [...]
  </dependencies>
  <build>
    <testSourceDirectory>src/it</testSourceDirectory>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <executions>
          <execution>
            <phase>integration-test</phase>
            <goals>
              <goal>test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
</project>
```

As you can see there is also a dependency on the `daytrader-ear` module. This is because the EAR artifact is needed to execute the functional tests. It also ensures that the `daytrader-ear` module is built before running the functional-tests build when the full DayTrader build is executed from the top-level in `daytrader/`.

For integration and functional tests, you will usually utilize a real database in a known state. To set up your database you can use the DBUnit Java API (see http://dbunit.sourceforge.net/). However, in the case of the DayTrader application, Derby is the default database configured in the deployment plan, and it is started automatically by Geronimo. In addition, there's a DayTrader Web page that loads test data into the database, so DBUnit is not needed to perform any database operations.

You may be asking how to start the container and deploy the DayTrader EAR into it. You're going to use the Cargo plugin to start Geronimo and deploy the EAR into it.

As the Surefire plugin's test goal has been bound to the integration-test phase above, you'll bind the Cargo plugin's start and deploy goals to the preintegration-test phase and the stop goal to the post-integration-test phase, thus ensuring the proper order of execution.

Start by adding the Cargo dependencies to the `functional-tests/pom.xml` file:

```
<project>
  [...]
  <dependencies>
    [...]
    <dependency>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-core-uberjar</artifactId>
      <version>0.8</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-ant</artifactId>
      <version>0.8</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Then create an execution element to bind the Cargo plugin's start and deploy goals:

```
<build>
  <plugins>
    [...]
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <configuration>
        <wait>false</wait>
        <container>
          <containerId>geronimo1x</containerId>
          <zipUrlInstaller>
            <url>http://www.apache.org/dist/geronimo/1.0/
              geronimo-tomcat-j2ee-1.0.zip</url>
            <installDir>${installDir}</installDir>
          </zipUrlInstaller>
        </container>
      </configuration>
      <executions>
        <execution>
          <id>start-container</id>
          <phase>pre-integration-test</phase>
          <goals>
            <goal>start</goal>
            <goal>deploy</goal>
          </goals>
          <configuration>
            <deployer>
              <deployables>
                <deployable>
                  <groupId>org.apache.geronimo.samples.daytrader</groupId>
                  <artifactId>daytrader-ear</artifactId>
                  <type>ear</type>
                  <properties>
                    <plan>${basedir}/src/deployment/geronimo/plan.xml</plan>
                  </properties>
                  <pingURL>http://localhost:8080/daytrader</pingURL>
                </deployable>
              </deployables>
            </deployer>
          </configuration>
        </execution>
        [...]
```

The `deployer` element is used to configure the Cargo plugin's deploy goal. It is configured to deploy the EAR using the Geronimo Plan file. In addition, a `pingURL` element is specified so that Cargo will ping the specified URL till it responds, thus ensuring that the EAR is ready for servicing when the tests execute.

Last, add an execution element to bind the Cargo plugin's stop goal to the post-integration-test phase:

```
            [...]
            <execution>
              <id>stop-container</id>
              <phase>post-integration-test</phase>
              <goals>
                <goal>stop</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </project>
```

The functional test scaffolding is now ready. The only thing left to do is to add the tests in src/it/java.

An alternative to using Cargo's Maven plugin is to use the Cargo Java API directly from your tests, by wrapping it in a JUnit `TestSetup` class to start the container in `setUp`() and stop it in `tearDown`().

You're going to use the HttpUnit testing framework (http://httpunit.sourceforge.net/) to call a Web page from the DayTrader application and check that it's working. Add the JUnit and HttpUnit dependencies, with both defined using a test scope, as you're only using them for testing:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>httpunit</groupId>
  <artifactId>httpunit</artifactId>
  <version>1.6.1</version>
  <scope>test</scope>
</dependency>
```

Next, add a JUnit test class called `src/it/java/org/apache/geronimo/samples/daytrader/FunctionalTest.java`. In the class, the `http://localhost:8080/daytrader` URL is called to verify that the returned page has a title of "DayTrader":

```
package org.apache.geronimo.samples.daytrader;

import junit.framework.*;
import com.meterware.httpunit.*;

public class FunctionalTest extends TestCase
{
    public void testDisplayMainPage() throws Exception
    {
        WebConversation wc = new WebConversation();
        WebRequest request =  new GetMethodWebRequest(
            "http://localhost:8080/daytrader");
        WebResponse response = wc.getResponse(request);
        assertEquals("DayTrader", response.getTitle());
    }
}
```

It's time to reap the benefits from your build. Change directory into functional-tests, type mvn install and relax:

```
C:\dev\m2book\code\j2ee\daytrader\functional-tests>mvn install
[...]
[INFO] [cargo:start {execution: start-container}]
[INFO] [cargo:deploy {execution: start-container}]
[INFO] [surefire:test {execution: default}]
[INFO] Setting reports dir: C:\dev\m2book\code\j2ee\daytrader\functional-
tests\target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running org.apache.geronimo.samples.daytrader.FunctionalTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,531 sec
[INFO] [cargo:stop {execution: stop-container}]
```

4.14. Summary

You have learned from chapters 1 and 2 how to build any type of application and this chapter has demonstrated how to build J2EE applications. In addition you've discovered how to automate starting and stopping containers, deploying J2EE archives and implementing functional tests. At this stage you've pretty much become an expert Maven user! The following chapters will show even more advanced topics such as how to write Maven plugins, how to gather project health information from your builds, how to effectively set up Maven in a team, and more.

Developing Custom Maven Plugins

This chapter covers:

- How plugins execute in the Maven life cycle
- Tools and languages available to aid plugin developers
- Implementing a basic plugin using Java and Ant
- Working with dependencies, source directories, and resources from a plugin
- Attaching an artifact to the project

> For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled.
>
> *- Richard Feynman*

5.1. Introduction

As described in Chapter 2, Maven is actually a platform that executes plugins within a build life cycle, in order to perform the tasks necessary to build a project. Maven's core APIs handle the “heavy lifting” associated with loading project definitions (POMs), resolving project dependencies, and organizing and running plugins. The actual functional tasks, or work, of the build process are executed by the set of plugins associated with the phases of a project's build life-cycle. This makes Maven's plugin framework extremely important as a means of not only building a project, but also extending a project's build to incorporate new functionality, such as integration with external tools and systems.

With most projects, the plugins provided “out of the box” by Maven are enough to satisfy the needs of most build processes (see Appendix A for a list of default plugins used to build a typical project). Even if a project requires a special task to be performed, it is still likely that a plugin already exists to perform this task. Such supplemental plugins can be found at the Apache Maven project the loosely affiliated CodeHaus Mojo project, or even at the web sites of third-party tools offering Maven integration by way of their own plugins (for a list of some additional plugins available for use, refer to the Plugin Matrix. However, if your project requires tasks that have no corresponding plugin, it may be necessary to write a custom plugin to integrate these tasks into the build life cycle.

This chapter will focus on the task of writing custom plugins. It starts by describing fundamentals, including a review of plugin terminology and the basic mechanics of the the Maven plugin framework. From there, the chapter will cover the tools available to simplify the life of the plugin developer. Finally, it will discuss the various ways that a plugin can interact with the Maven build environment and explore some examples.

5.2. A Review of Plugin Terminology

Before delving into the details of how Maven plugins function and how they are written, let's begin by reviewing the terminology used to describe a plugin and its role in the build.

A *mojo* is the basic unit of work in the Maven application. It executes an atomic build task that represents a single step in the build process. Each mojo can leverage the rich infrastructure provided by Maven for loading projects, resolving dependencies, injecting runtime parameter information, and more. When a number of mojos perform related tasks, they are packaged together into a *plugin*.

Just like Java packages, *plugins* provide a grouping mechanism for multiple mojos that serve similar functions within the build life cycle. For example, the maven-compiler-plugin incorporates two mojos: `compile` and `testCompile`. In this case, the common theme for these tasks is the function of compiling code. Packaging these mojos inside a single plugin provides a consistent access mechanism for users, allowing shared configuration to be added to a single section of the POM. Additionally, it enables these mojos to share common code more easily.

Recall that a mojo represents a single task in the build process. Correspondingly, the build process for a project is comprised of set of mojos executing in a particular, well-defined order. This ordering is called the build *life cycle*, and is defined as a set of task categories, called *phases*. When Maven executes a build, it traverses the phases of the life cycle in order, executing all the associated mojos at each phase of the build.

This association of mojos to phases is called *binding* and is described in detail below.

Together with phase binding, the ordered execution of Maven's life cycle gives coherence to the build process, sequencing the various build operations. While Maven does in fact define three different life-cycles, the discussion in this chapter is restricted to the default life cycle, which is used for the majority of build activities (the other two life cycles deal with cleaning a project's work directory and generating a project web site). A discussion of all three build life cycles can be found in Appendix A.

Most mojos fall into a few general categories, which correspond to the phases of the build life cycle. As a result, mojos have a natural phase binding which determines when a task should execute within the life cycle. Since phase bindings provide a grouping mechanism for mojos within the life cycle, successive phases can make assumptions about what work has taken place in the previous phases. Therefore, to ensure compatibility with other plugins, it is important to provide the appropriate phase binding for your mojos.

While mojos usually specify a default phase binding, they can be bound to any phase in the life cycle. Indeed, a given mojo can even be bound to the life cycle multiple times during a single build, using the plugin executions section of the project's POM. Each execution can specify a separate phase binding for its declared set of mojos. However, before a mojo can execute, it may still require that certain activities have already been completed, so be sure to check the documentation for a mojo before you re-bind it.

In some cases, a mojo may be designed to work outside the context of the build life cycle. Such mojos may be meant to check out a project from version control, or even create the directory structure for a new project. These mojos are meant to be used by way of direct invocation, and as such, will not have a life-cycle phase binding at all since they don't fall into any natural category within a typical build process. Think of these mojos as tangential to the the Maven build process, since they often perform tasks for the POM maintainer, or aid integration with external development tools.

5.3. Bootstrapping into Plugin Development

In addition to understanding Maven's plugin terminology, you will also need a good understanding of how plugins are structured and how they interact with their environment. As a plugin developer, you must understand the mechanics of life-cycle phase binding and *parameter injection*. Understanding this framework will enable you to extract the Maven build-state information that each mojo requires, in addition to determining its appropriate phase binding.

5.3.1. The Plugin Framework

Maven provides a rich framework for its plugins, including a well-defined build life cycle, dependency management, and parameter resolution and injection. Using the life cycle, Maven also provides a well-defined procedure for building a project's sources into a distributable archive, plus much more. Binding to a phase of the Maven life cycle allows a mojo to make assumptions based upon what has happened in the preceding phases. Using Maven's parameter injection infrastructure, a mojo can pick and choose what elements of the build state it requires in order to execute its task. Together, parameter injection and life-cycle binding form the cornerstone for all mojo development.

Participation in the build life cycle

Most plugins consist entirely of mojos that are bound at various phases in the life cycle according to their function in the build process. As a specific example of how plugins work together through the life cycle, consider a very basic Maven build: a project with source code that should be compiled and archived into a jar file for redistribution. During this build process, Maven will execute a default life cycle for the 'jar' packaging. The 'jar' packaging definition assigns the following life-cycle phase bindings:

Table 5-1: Life-cycle bindings for jar packaging

Life-cycle Phase	Mojo	Plugin
`process-resources`	`resources`	`maven-resources-plugin`
`compile`	`compile`	`maven-compiler-plugin`
`process-test-resources`	`testResources`	`maven-resources-plugin`
`test-compile`	`testCompile`	`maven-compiler-plugin`
`test`	`test`	`maven-surefire-plugin`
`package`	`jar`	`maven-jar-plugin`
`install`	`install`	`maven-install-plugin`
`deploy`	`deploy`	`maven-deploy-plugin`

When you command Maven to execute the package phase of this life cycle, at least two of the above mojos will be invoked. First, the compile mojo from the `maven-compiler-plugin` will compile the source code into binary class files in the output directory. Then, the jar mojo from the maven-jar-plugin will harvest these class files and archive them into a jar file.

Only those mojos with tasks to perform are executed during this build. Since our hypothetical project has no “non-code” resources, none of the mojos from the maven-resources-plugin will be executed. Instead, each of the resource-related mojos will discover this lack of non-code resources and simply opt out without modifying the build in any way. This is not a feature of the framework, but a requirement of a well-designed mojo. In good mojo design, determining when *not* to execute, is often as important as the modifications made during execution itself.

If this basic Maven project also includes source code for unit tests, then two additional mojos will be triggered to handle unit testing. The `testCompile` mojo from the maven-compiler-plugin will compile the test sources, then the test mojo from the maven-surefire-plugin will execute those compiled tests. These mojos were always present in the life-cycle definition, but until now they had nothing to do and therefore, did not execute.

Depending on the needs of a given project, many more plugins can be used to augment the default life-cycle definition, providing functions as varied as deployment into the repository system, validation of project content, generation of the project's website, and much more. Indeed, Maven's plugin framework ensures that almost anything can be integrated into the build life cycle. This level of extensibility is part of what makes Maven so powerful.

Accessing build information

In order for mojos to execute effectively, they require information about the state of the current build. This information comes in two categories:

- **Project information** – which is derived from the project POM, in addition to any programmatic modifications made by previous mojo executions.
- **Environment information** – which is more static, and consists of the user- and machine-level Maven settings, along with any system properties that were provided when Maven was launched.

To gain access to the current build state, Maven allows mojos to specify parameters whose values are extracted from the build state using expressions. At runtime, the expression associated with a parameter is resolved against the current build state, and the resulting value is injected into the mojo, using a language-appropriate mechanism. Using the correct parameter expressions, a mojo can keep its dependency list to a bare minimum, thereby avoiding traversal of the entire build-state object graph.

For example, a mojo that applies patches to the project source code will need to know where to find the project source and patch files. This mojo would retrieve the list of source directories from the current build information using the following expression:

```
${project.compileSourceRoots}
```

Then, assuming the patch directory is specified as mojo configuration inside the POM, the expression to retrieve that information might look as follows:

```
${patchDirectory}
```

For more information about which mojo expressions are built into Maven, and what methods Maven uses to extract mojo parameters from the build state, see Appendix A.

The plugin descriptor

Though you have learned about binding mojos to life-cycle phases and resolving parameter values using associated expressions, until now you have not seen exactly how a life-cycle binding occurs. That is to say, how do you associate mojo parameters with their expression counterparts, and once resolved, how do you instruct Maven to inject those values into the mojo instance? Further, how do you instruct Maven to instantiate a given mojo in the first place?

The answers to these questions lie in the plugin *descriptor*. The Maven plugin descriptor is a file that is embedded in the plugin jar archive, under the path `/META-INF/maven/plugin.xml`. The descriptor is an XML file that informs Maven about the set of mojos that are contained within the plugin. It contains information about the mojo's implementation class (or its path within the plugin jar), the life-cycle phase to which the mojo should be bound, the set of parameters the mojo declares, and more.

Within this descriptor, each declared mojo parameter includes information about the various expressions used to resolve its value, whether it is editable, whether it is required for the mojo's execution, and the mechanism for injecting the parameter value into the mojo instance. For the complete plugin descriptor syntax, see Appendix A.

The plugin descriptor is very powerful in its ability to capture the wiring information for a wide variety of mojos. However, this flexibility comes at a price. To accommodate the extensive variability required from the plugin descriptor, it uses a complex syntax. Writing a plugin descriptor by hand demands that plugin developers understand low-level details about the Maven plugin framework – details that the developer will not use, except when configuring the descriptor. This is where Maven's plugin development tools come into play. By abstracting many of these details away from the plugin developer, Maven's development tools expose only relevant specifications in a format convenient for a given plugin's implementation language.

5.3.2. Plugin Development Tools

To simplify the creation of plugin descriptors, Maven provides plugin tools to parse mojo metadata from a variety of formats. This metadata is embedded directly in the mojo's source code where possible, and its format is specific to the mojo's implementation language. In short, Maven's plugin-development tools remove the burden of maintaining mojo metadata by hand. These plugin-development tools are divided into the following two categories:

- **The plugin extractor framework** – which knows how to parse the metadata formats for every language supported by Maven. This framework generates both plugin documentation and the coveted plugin descriptor; it consists of a framework library which is complemented by a set of provider libraries (generally, one per supported mojo language).
- **The `maven-plugin-plugin`** – which uses the plugin extractor framework, and orchestrates the process of extracting metadata from mojo implementations, adding any other plugin-level metadata through its own configuration (which can be modified in the plugin's POM); the maven-plugin-plugin simply augments the standard jar life cycle mentioned previously as a resource-generating step (this means the standard process of turning project sources into a distributable jar archive is modified only slightly, to generate the plugin descriptor).

Of course, the format used to write a mojo's metadata is dependent upon the language in which the mojo is implemented. Using Java, it's a simple case of providing special javadoc annotations to identify the properties and parameters of the mojo. For example, the clean mojo in the maven-clean-plugin provides the following class-level javadoc annotation:

```
/**
 * @goal clean
 */
public class CleanMojo extends AbstractMojo
```

This annotation tells the plugin-development tools the mojo's name, so it can be referenced from life-cycle mappings, POM configurations, and direct invocations (as from the command line). The clean mojo also defines the following:

```
/**
 * Be verbose in the debug log-level?
 *
 * @parameter expression="${clean.verbose}" default=value="false"
 */
private boolean verbose;
```

Here, the annotation identifies this field as a mojo parameter. Moreover, it specifies that this parameter can be configured from the POM using:

```
<configuration>
  <verbose>true</verbose>
</configuration>
```

You may notice that this configuration name isn't explicitly specified in the annotation; it's implicit when using the `@parameter` annotation. However, this parameter annotation also specifies two attributes, default-value and expression. The first specifies that this parameter's default value should be set to true. The second specifies that this parameter can also be configured from the command line as follows:

```
-Dclean.verbose=true
```

At first, it might seem counter-intuitive to initialize the default value of a Java field using a javadoc annotation, especially when you could just declare the field as follows:

```
private boolean verbose = true;
```

But consider what would happen if the default value you wanted to inject contained a parameter expression. For instance, consider the following field annotation from the resources mojo in the maven-resources-plugin:

```
/**
 * Directory containing the classes.
 *
 * @parameter default-value="${project.build.outputDirectory}"
 */
private File classesDirectory;
```

In this case, it's impossible to initialize the Java field with the value you need, namely the `java.io.File` instance, which references the output directory for the current project. When the mojo is instantiated, this value is resolved based on the POM and injected into this field. Since the plugin tools can also generate documentation about plugins based on these annotations, it's a good idea to consistently specify the parameter's default value in the metadata, rather than in the Java field initialization code.

For a complete list of javadoc annotations available for specifying mojo metadata, see Appendix A.

Remember, these annotations are specific to mojos written in Java. If you choose to write mojos in another language, like Ant, then the mechanism for specifying mojo metadata such as parameter definitions will be different. However, the underlying principles remain the same.

Choose your mojo implementation language

Through its flexible plugin descriptor format and invocation framework, Maven can accommodate mojos written in virtually any language. For example, Maven currently supports mojos written in Java, Ant, and Beanshell. Whatever language you use, Maven lets you select pieces of the build state to inject as mojo parameters. This relieves you of the burden associated with traversing a large object graph in your code, and minimizes the number of dependencies you will have on Maven's core APIs.

For many mojo developers, Java is the language of choice. Since it provides easy reuse of third-party APIs from within your mojo, and because many Maven-built projects are written in Java, it also provides good alignment of skill sets when developing mojos from scratch. Simple javadoc annotations give the plugin processing plugin (the maven-plugin-plugin) the instructions required to generate a descriptor for your mojo. Plugin parameters can be injected via either field reflection or setter methods. Since Beanshell behaves in a similar way to standard Java, this technique also works well for Beanshell-based mojos.

However, in certain cases you may find it easier to use Ant scripts to perform build tasks. To make Ant scripts reusable, Maven can wrap an Ant build target and use it as if it were a mojo. This is especially important during migration, when translating a project build from Ant to Maven (refer to Chapter 8 for more discussion about migrating from Ant to Maven). During the early phases of such a migration, it is often simpler to wrap existing Ant build targets with Maven mojos and bind them to various phases in the life cycle. Ant-based plugins can consist of multiple mojos mapped to a single build script, individual mojos each mapped to separate scripts, or any combination thereof. In these cases, mojo mappings and parameters definitions are declared via an associated metadata file. This pairing of the build script and accompanying metadata file follows a naming convention that allows the maven-plugin-plugin to correlate the two files and create an appropriate plugin descriptor.

Since Java is currently the easiest language for plugin development, this chapter will focus primarily on plugin development in this language. In addition, due to the mitigation value of Ant-based mojos when converting a build to Maven, this chapter will also provide an example of basic plugin development using Ant.

5.3.3. A Note on the Examples in this Chapter

When learning how to interact with the different aspects of Maven from within a mojo, it's important to keep the examples clean and relatively simple. Otherwise, you risk confusing the issue at hand – namely, the particular feature of the mojo framework currently under discussion. Therefore, the examples in this chapter will focus on a relatively simple problem space: gathering and publishing information about a particular build. Such information might include details about the system environment, the specific snapshot versions of dependencies used in the build, and so on.

To facilitate these examples, you will need to work with an external project, called `maven-buildinfo`, which is used to read and write build information metadata files. This project can be found in the source code that accompanies this book. You can install it using the following simple command:

```
mvn install
```

5.4. Developing Your First Mojo

For the purposes of this chapter, you will look at the development effort surrounding a sample project, called Guinea Pig. This development effort will have the task of maintaining information about builds that are deployed to the development repository, for the purposes of debugging. This information should capture relevant details about the environment used to build the Guinea Pig artifacts, which will be deployed to the Maven repository system. Capturing this information is key, since it can have a critical effect on the build process and the composition of the resulting Guinea Pig artifacts. In addition to simply capturing build-time information, you will need to disseminate the build to the rest of the development team, eventually publishing it alongside the project's artifact in the repository for future reference (refer to Chapter 7 for more details on how teams use Maven).

5.4.1. `BuildInfo` Example: Capturing Information with a Java Mojo

To begin, consider a case where the POM contains a profile, which will be triggered by the value of a given system property – say, if the system property `os.name` is set to the value Linux (for more information on profiles, refer to Chapter 3). When triggered, this profile adds a new dependency on a Linux-specific library, which allows the build to succeed in that environment. When this profile is *not* triggered, a default profile injects a dependency on a windows-specific library. For simplicity, this dependency is used only during testing, and has no impact on transitive dependencies for users of this project.

Here, the values of system properties used in the build are clearly very important. If you have a test dependency which contains a defect, and this dependency is injected by one of the aforementioned profiles, then the value of the triggering system property – and the profile it triggers – could reasonably determine whether the build succeeds or fails. Therefore, it makes sense to publish the value of this particular system property in a build information file so that others can see the aspects of the environment that affected this build.

Using the archetype plugin to generate a stub plugin project

To jump-start the process of writing a new plugin, it's helpful to use the archetype plugin to create a simple stub project. Once you have the plugin's project structure in place, writing your custom mojo is simple. To generate a stub plugin project for the `buildinfo` plugin, simply execute the following:

```
mvn archetype:create -DgroupId=com.mergere.mvnbook.plugins \
    -DartifactId=maven-buildinfo-plugin \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-mojo \
    -DarchetypeVersion=1.0-alpha-4
```

This will create a project with the standard layout under a new subdirectory called `maven-buildinfo-plugin` within the current working directory. Inside, you'll find a basic POM and a sample mojo. For the purposes of this plugin, you will need to modify the POM as follows:

- Change the name element to `Maven BuildInfo Plugin`.
- Remove the `url` element, since this plugin doesn't currently have an associated web site.

You will modify the POM again later, as you know more about your mojos' dependencies. However, this simple version will suffice for now.

Finally, since you will be creating your own mojo from scratch, you should remove the sample mojo. It can be found in the plugin's project directory, under the following path:

```
src/main/java/org/codehaus/mojo/MyMojo.java.
```

The mojo

You can handle this scenario using the following, fairly simple Java-based mojo:

```
[...]
/**
 * Write environment information for the current build to file.
 * @goal extract
 * @phase package
 */
public class WriteBuildInfoMojo extends AbstractMojo {

    /**
     * Determines which system properties are added to the file.
     * This is a comma-delimited list.
     * @parameter expression="${systemProperties}"
     */
   private String systemProperties;

   /**
     * The location to write the buildinfo file.
     * @parameter expression="${buildinfo.outputFile}" default-
value="${project.build.outputDirectory}/${project.artifactId}-${project.version}-
buildinfo.xml"
     * @required
      */
    private File outputFile;

   public void execute() throws MojoExecutionException {
        BuildInfo buildInfo = new BuildInfo();

        Properties sysprops = System.getProperties();

        if ( systemProperties != null )
        {
            String[] keys = systemProperties.split( "," );
            for ( int i = 0; i < keys.length; i++ )
            {
                String key = keys[i].trim();

                String value = sysprops.getProperty( key,
                    BuildInfoConstants.MISSING_INFO_PLACEHOLDER );

                buildInfo.addSystemProperty( key, value );
            }
        }

        try
```

```
        {
            BuildInfoUtils.writeXml( buildInfo, outputFile );
        }
        catch ( IOException e )
        {
            throw new MojoExecutionException(
                "Error writing buildinfo XML file. Reason: " +
                    e.getMessage(),
                e );
        }
    }
}
```

While the code for this mojo is fairly straightforward, it's worthwhile to take a closer look at the javadoc annotations. In the class-level javadoc comment, there are two special annotations:

```
/**
 * @goal extract
 * @phase package
 */
```

The first annotation, `@goal`, tells the plugin tools to treat this class as a mojo named extract. When you invoke this mojo, you will use this name. The second annotation tells Maven where in the build life cycle this mojo should be executed. In this case, you're collecting information from the environment with the intent of distributing it alongside the main project artifact in the repository. Therefore, it makes sense to execute this mojo in the package phase, so it will be ready to attach to the project artifact. In general, attaching to the package phase also gives you the best chance of capturing all of the modifications made to the build state before the jar is produced.

Aside from the class-level comment, you have several field-level javadoc comments, which are used to specify the mojo's parameters. Each offers a slightly different insight into parameter specification, so they will be considered separately. First, consider the `systemProperties` parameter:

```
/**
* @parameter expression="${buildinfo.systemProperties}"
*/
```

This is one of the simplest possible parameter specifications. Using the `@parameter` annotation by itself, with no attributes, will allow this mojo field to be configured using the plugin configuration specified in the POM. However, you may want to allow a user to specify which system properties to include in the build information file. This is where the expression attribute comes into play. Using the expression attribute, you can specify the name of this parameter when it's referenced from the command line. In this case, the expression attribute allows you to specify a list of system properties on-the-fly, as follows:

```
localhost $ mvn buildinfo:java-write \
  -Dbuildinfo.systemProperties=java.version,user.dir
```

Finally, the `outputFile` parameter presents a slightly more complex example of parameter annotation. However, since you have more specific requirements for this parameter, the complexity is justified. Take another look:

```
/**
* The location to write the buildinfo file.
*
* @parameter expression="${buildinfo.outputFile}" default-
value="${project.build.outputDirectory}/${project.artifactId}-${project.version}-
buildinfo.xml"
*
* @required
 */
```

In this case, the mojo cannot function unless it knows where to write the build information file, as execution without an output file would be pointless. To ensure that this parameter has a value, the mojo uses the `@required` annotation. If this parameter has no value when the mojo is configured, the build will fail with an error. In addition, you want the mojo to use a certain value – calculated from the project's information – as a default value for this parameter. In this example, you can see why the normal Java field initialization is not used. The default output path is constructed directly inside the annotation, using several expressions to extract project information on-demand.

The plugin POM

Once the mojo has been written, you can construct an equally simple POM which will allow you to build the plugin, as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mergere.mvnbook.plugins</groupId>
  <artifactId>maven-buildinfo-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>

  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>2.0</version>
    </dependency>
    <dependency>
      <groupId>com.mergere.mvnbook.shared</groupId>
      <artifactId>buildinfo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

This POM declares the project's identity and its two dependencies.

Note the dependency on the `buildinfo` project, which provides the parsing and formatting utilities for the build information file. Also, note the packaging – specified as maven-plugin – which means that this plugin build will follow the maven-plugin life-cycle mapping. This mapping is a slightly modified version of the one used for the jar packaging, which simply adds plugin descriptor extraction and generation to the build process.

Binding to the life cycle

Now that you have a method of capturing build-time environmental information, you need to ensure that every build captures this information. The easiest way to guarantee this is to bind the extract mojo to the life cycle, so that every build triggers it. This involves modification of the standard jar life-cycle, which you can do by adding the configuration of the new plugin to the Guinea Pig POM, as follows:

```
<build>
 ...
  <plugins>
    <plugin>
      <groupId>com.mergere.mvnbook.plugins</groupId>
      <artifactId>maven-buildinfo-plugin</artifactId>
      <executions>
        <execution>
          <id>extract</id>
          <configuration>
            <systemProperties>
              <systemProperty>os.name</systemProperty>
            </systemProperties>
          </configuration>
          <goals>
            <goal>extract</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
 ...
</build>
```

The above binding will execute the extract mojo from your new `maven-buildinfo-plugin` during the package phase of the life cycle, and capture the `os.name` system property.

The output

Now that you have a mojo and a POM, you can build the plugin and try it out! First, build the buildinfo plugin with the following commands:

```
> C:\book-projects\maven-buildinfo-plugin
> mvn clean install
```

Next, test the plugin by building Guinea Pig with the `buildinfo` plugin bound to its life cycle as follows:

```
> C:\book-projects\guinea-pig
 > mvn package
```

When the Guinea Pig build executes, you should see output similar to the following:

```
[...]

[INFO] [buildinfo:extract {execution:extract}]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------

[...]
```

Under the target directory, there should be a file named:

```
guinea-pig-1.0-SNAPSHOT-buildinfo.xml
```

In the file, you will find information similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?><buildinfo>
  <systemProperties>
    <os.name>Linux</os.name>
  </systemProperties>
</buildinfo>
```

Formatting aside, the output of of the generated build information is clear enough. Your mojo has captured the version of Java being used to execute the build, a property that can have profound effects on binary compatibility.

5.4.2. `BuildInfo` Example: Notifying Other Developers with an Ant Mojo

Now that some important information has been captured, you need to share it with others in your team when the resulting project artifact is deployed. It's important to remember that in the Maven world, “deployment” is defined as injecting the project artifact into the Maven repository system. For now, it might be enough to send a notification e-mail to the project development mailing list, so that other team members have access to it.

Of course, such a task could be handled using a Java-based mojo and the JavaMail API from Sun. However, given the amount of setup and code required, and the dozens of well-tested, mature tasks available for build script use (including one specifically for sending e-mails), it's simpler to use Ant.

After writing the Ant target to send the notification e-mail, you just need to write a mojo definition to wire the new target into Maven's build process.

The Ant target

To leverage the output of the mojo from the previous example – the build information file – you can use that content as the body of the e-mail. From here, it's a simple matter of specifying where the e-mail should be sent, and how. Your new mojo will be in a file called `notify.build.xml`, and should look similar to the following:

```
<project>
  <target name="notify-target">
    <mail from="maven@localhost" replyto="${listAddr}"
            subject="Build Info for Deployment of ${project.name}"
          mailhost="${mailHost}" mailport="${mailPort}"
        messagefile="${buildinfo.outputFile}">

      <to>${listAddr}</to>

    </mail>
  </target>
</project>
```

If you're familiar with Ant, you'll notice that this mojo expects several project properties. Information like the to: address will have to be dynamic; therefore, it should be extracted directly from the POM for the project we're building. To ensure these project properties are in place within the Ant Project instance, simply declare mojo parameters for them.

The mojo metadata file

Unlike the prior Java examples, metadata for an Ant mojo is stored in a separate file, which is associated to the build script using a naming convention. In this example, the build script was called `notify.build.xml`. The corresponding metadata file will be called `notify.mojos.xml` and should appear as follows:

```
<pluginMetadata>
  <mojos>
    <mojo>
      <call>notify-target</call>
      <goal>notify</goal>
      <phase>deploy</phase>
      <description><![CDATA[
        Email environment information from the current build to the
        development mailing list when the artifact is deployed.
      ]]></description>
      <parameters>
        <parameter>
          <name>buildinfo.outputFile</name>
          <defaultValue>
              ${project.build.directory}/${project.artifactId}-
${project.version}-buildinfo.xml
            </defaultValue>
          <required>true</required>
          <readonly>false</readonly>
        </parameter>
        <parameter>
          <name>listAddr</name>
          <required>true</required>
        </parameter>
        <parameter>
          <name>project.name</name>
          <defaultValue>${project.name}</defaultValue>
          <required>true</required>
          <readonly>true</readonly>
        </parameter>
        <parameter>
          <name>mailHost</name>
          <expression>${mailHost}</expression>
          <defaultValue>localhost</defaultValue>
          <required>false</required>
        </parameter>
        <parameter>
          <name>mailPort</name>
          <expression>${mailPort}</expression>
          <defaultValue>25</defaultValue>
          <required>false</required>
        </parameter>
      </parameters>
    </mojo>
  </mojos>
</pluginMetadata>
```

At first glance, the contents of this file may appear different than the metadata used in the Java mojo; however, upon closer examination, you will see many similarities.

First of all, since you now have a good concept of the types of metadata used to describe a mojo, the overall structure of this file should be familiar. As with the Java example, mojo-level metadata describes details such as phase binding and mojo name.

Also, metadata specify a list of parameters for the mojo, each with its own information like name, expression, default value, and more. The expression syntax used to extract information from the build state is exactly the same, and parameter flags such as required are still present, but expressed in XML.

When this mojo is executed, Maven still must resolve and inject each of these parameters into the mojo; the difference here is the mechanism used for this injection. In Java, parameter injection takes place either through direct field assignment, or through JavaBeans-style setXXX() methods. In an Ant-based mojo however, parameters are injected as properties and references into the Ant Project instance.

The rule for parameter injection in Ant is as follows: if the parameter's type is java.lang.String (the default), then its value is injected as a property; otherwise, its value is injected as a project reference. In this example, all of the mojo's parameter types are java.lang.String. If one of the parameters were some other object type, you'd have to add a <type> element alongside the <name> element, in order to capture the parameter's type in the specification.

Finally, notice that this mojo is bound to the deploy phase of the life cycle. This is an important point in the case of this mojo, because you're going to be sending e-mails to the development mailing list. Any build that runs must be deployed for it to affect other development team members, so it's pointless to spam the mailing list with notification e-mails every time a jar is created for the project. Instead, by binding the mojo to the deploy phase of life cycle, the notification e-mails will be sent only when a new artifact becomes available in the remote repository.

As with the Java example, a more in-depth discussion of the metadata file for Ant mojos is available in Appendix A.

Modifying the plugin POM for Ant mojos

Since Maven 2.0 shipped without support for Ant-based mojos (support for Ant was added later in version 2.0.2), some special configuration is required to allow the maven-plugin-plugin to recognize Ant mojos. Fortunately, Maven allows POM-specific injection of plugin-level dependencies in order to accommodate plugins that take a framework approach to providing their functionality.

The maven-plugin-plugin is a perfect example, with its use of the `MojoDescriptorExtractor` interface from the `maven-plugin-tools-api` library. This library defines a set of interfaces for parsing mojo descriptors from their native format and generating various output from those descriptors – including plugin descriptor files. The maven-plugin-plugin ships with the Java and Beanshell provider libraries which implement the above interface.

This allows developers to generate descriptors for Java- or Beanshell-based mojos with no additional configuration. However, to develop an Ant-based mojo, you will have to add support for Ant mojo extraction to the `maven-plugin-plugin`.

To accomplish this, you will need to add a dependency on the maven-plugin-tools-ant library to the `maven-plugin-plugin` using POM configuration as follows:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <dependencies>
          <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-tools-ant</artifactId>
            <version>2.0.2</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
 [...]
</project>
```

Additionally, since the plugin now contains an Ant-based mojo, it requires a couple of new dependencies, the specifications of which should appear as follows:

```
<dependencies>
  [...]
  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-script-ant</artifactId>
    <version>2.0.2</version>
  </dependency>
  <dependency>
    <groupId>ant</groupId>
    <artifactId>ant</artifactId>
    <version>1.6.5</version>
  </dependency>
  [...]
</dependencies>
```

The first of these new dependencies is the mojo API wrapper for Ant build scripts, and it is always necessary for embedding Ant scripts as mojos in the Maven build process. The second new dependency is, quite simply, a dependency on the core Ant library (whose necessity should be obvious). If you don't have Ant in the plugin classpath, it will be quite difficult to execute an Ant-based plugin.

Binding the notify mojo to the life cycle

Once the plugin descriptor is generated for the Ant mojo, it behaves like any other type of mojo to Maven. Even its configuration is the same. Adding a life-cycle binding for the new Ant mojo in the Guinea Pig POM should appear as follows:

```
<build>
  [...]
  <plugins>
    <plugin>
      <artifactId>maven-buildinfo-plugin</artifactId>
      <executions>
        <execution>
          <id>extract</id>
          [...]
        </execution>
        <execution>
          <id>notify</id>
          <goals>
            <goal>notify</goal>
          </goals>
          <configuration>
            <listAddr>dev@guineapig.codehaus.org</listAddr>
          </configuration>
        </execution>
      </executions>
    </plugin>
    [...]
  </plugins>
</build>
```

The existing `<execution>` section – the one that binds the extract mojo to the build – is not modified. Instead, a new section for the notify mojo is created. This is because an execution section can address only one phase of the build life cycle, and these two mojos should not execute in the same phase (as mentioned previously).

In order to tell the notify mojo where to send this e-mail, you should add a configuration section to the new execution section, which supplies the `listAddr` parameter value.

Now, execute the following command:

```
> mvn deploy
```

The build process executes the steps required to build and deploy a jar - except in this case, it will also extract the relevant environmental details during the package phase, and send them to the Guinea Pig development mailing list in the deploy phase. Again, notification happens in the deploy phase only, because non-deployed builds will have no effect on other team members.

5.5. Advanced Mojo Development

The preceding examples showed how to declare basic mojo parameters, and how to annotate the mojo with a name and a preferred phase binding. The next examples cover more advanced topics relating to mojo development. The following sections do not build on one another, and are not required for developing basic mojos. However, if you want to know how to develop plugins that manage dependencies, project source code and resources, and artifact attachments, then read on!

5.5.1. Accessing Project Dependencies

Many mojos perform tasks that require access to a project's dependencies. For example, the compile mojo in the maven-compiler-plugin must have a set of dependency paths in order to build the compilation classpath. In addition, the test mojo in the maven-surefire-plugin requires the project's dependency paths so it can execute the project's unit tests with a proper classpath. Fortunately, Maven makes it easy to inject a project's dependencies.

To enable a mojo to work with the set of artifacts that comprise the project's dependencies, only the following two changes are required:

- First, the mojo must tell Maven that it requires the project dependency set.
- Second, the mojo must tell Maven that it requires the project's dependencies be *resolved* (this second requirement is critical, since the dependency resolution process is what populates the set of artifacts that make up the project's dependencies).

Injecting the project dependency set

As described above, if the mojo works with a project's dependencies, it must tell Maven that it requires access to that set of artifacts. As with all declarations, this is specified via a mojo parameter definition and should use the following syntax:

```
/**
 * The set of dependencies required by the project
 * @parameter default-value="${project.artifacts}"
 * @required
 * @readonly
 */
private java.util.Set dependencies;
```

This declaration should be familiar to you, since it defines a parameter with a default value that is required to be present before the mojo can execute. However, this declaration has another annotation, which might not be as familiar: `@readonly`. This annotation tells Maven not to allow the user to configure this parameter directly, namely it *disables* configuration via the POM under the following section:

```
<configuration>
  <dependencies>...</dependencies>
</configuration>
```

It also *disables* configuration via system properties, such as:

```
-Ddependencies=[...]
```

So, you may be wondering, “How exactly *can* I configure this parameter?” The answer is that the mojos parameter value is derived from the dependencies section of the POM, so you configure this parameter by modifying that section directly.

If this parameter could be specified separately from the main dependencies section, users could easily break their builds – particularly if the mojo in question compiled project source code.

In this case, direct configuration could result in a dependency being present for compilation, but being unavailable for testing. Therefore, the `@readonly` annotation functions to force users to configure the POM, rather than configuring a specific plugin only.

Requiring dependency resolution

Having declared a parameter that injects the projects dependencies into the mojo, the mojo is missing one last important step. To gain access to the project's dependencies, your mojo must declare that it needs them.

Maven provides a mechanism that allows a mojo to specify whether it requires the project dependencies to be resolved, and if so, at which scope. Maven 2 will not resolve project dependencies until a mojo requires it. Even then, Maven will resolve only the dependencies that satisfy the requested scope. In other words, if a mojo declares that it requires dependencies for the compile scope, any dependencies specific to the test scope will remain unresolved. However, if later in the build process, Maven encounters another mojo that declares a requirement for test-scoped dependencies, it will force all of the dependencies to be resolved (test is the widest possible scope, encapsulating all others).

It's important to note that your mojo can require any valid dependency scope to be resolved prior to its execution.

If you've used Maven 1, you'll know that one of its major problems is that it always resolves all project dependencies before invoking the first goal in the build (for clarity, Maven 2.0 uses the term 'mojo' as roughly equivalent to the Maven 1.x term 'goal'). Consider the case where a developer wants to clean the project directory using Maven 1.x. If the project's dependencies aren't available, the clean process will fail – though not because the clean goal requires the project dependencies. Rather, this is a direct result of the rigid dependency resolution design in Maven 1.x.

Maven 2 addresses this problem by deferring dependency resolution until the project's dependencies are actually required. If a mojo doesn't need access to the dependency list, the build process doesn't incur the added overhead of resolving them.

Returning to the example, if your mojo needs to work with the project's dependencies, it will have to tell Maven to resolve them. Failure to do so will cause an empty set to be injected into the mojo's dependencies parameter.

You can declare the requirement for the test-scoped project dependency set using the following class-level annotation:

```
/**
 * @requiresDependencyResolution test
 [...]
 */
```

Now, the mojo should be ready to work with the dependency set.

`BuildInfo` example: logging dependency versions

Turning once again to the `maven-buildinfo-plugin`, you will want to log the versions of the dependencies used during the build. This is critical when the project depends on snapshot versions of other libraries. In this case, knowing the specific set of snapshots used to compile a project can lend insights into why other builds are breaking. For example, one of the dependency libraries may have a newer snapshot version available.

To that end, you'll add the dependency-set injection code discussed previously to the extract mojo in the `maven-buildinfo-plugin`, so it can log the exact set of dependencies that were used to produce the project artifact.

This will result in the addition of a new section in the `buildinfo` file, which enumerates all the dependencies used in the build, along with their versions – including those dependencies that are resolved transitively. Once you have access to the project dependency set, you will need to iterate through the set, adding the information for each individual dependency to your `buildinfo` object.

The code required is as follows:

```
if ( dependencies != null && !dependencies.isEmpty() )
{
    for ( Iterator it = dependencies.iterator(); it.hasNext(); )
    {
        Artifact artifact = (Artifact) it.next();
        ResolvedDependency rd = new ResolvedDependency();

        rd.setGroupId( artifact.getGroupId() );
        rd.setArtifactId( artifact.getArtifactId() );
        rd.setResolvedVersion( artifact.getVersion() );
        rd.setOptional( artifact.isOptional() );
        rd.setScope( artifact.getScope() );
        rd.setType( artifact.getType() );

        if ( artifact.getClassifier() != null )
        {
            rd.setClassifier( artifact.getClassifier() );
        }

        buildInfo.addResolvedDependency( rd );
    }
}
```

When you re-build the plugin and re-run the Guinea Pig build, the extract mojo should produce the same `buildinfo` file, with an additional section called `resolvedDependencies` that looks similar to the following:

```
<resolvedDependencies>
  <resolvedDependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <resolvedVersion>3.8.1</resolvedVersion>
    <optional>false</optional>
    <type>jar</type>
    <scope>test</scope>
  </resolvedDependency>
  [...]
  <resolvedDependency>
    <groupId>com.mergere.mvnbook.guineapig</groupId>
    <artifactId>guinea-pig-api</artifactId>
    <resolvedVersion>1.0-20060210.094434-1</resolvedVersion>
    <optional>false</optional>
    <type>jar</type>
    <scope>compile</scope>
  </resolvedDependency>
  [...]
</resolvedDependencies>
```

The first dependency listed here, `junit`, has a static version of 3.8.1. This won't add much insight for debuggers looking for changes from build to build, but consider the next dependency: `guinea-pig-api`. This dependency is part of the example development effort, and is still listed with the version `1.0-alpha-SNAPSHOT` in the POM. The actual snapshot version used for this artifact in a previous build could yield tremendous insight into the reasons for a current build failure, particularly if the newest snapshot version is different.

If you were using a snapshot version from the local repository which has not been deployed, the `resolvedVersion` in the output above would be `1.0-alpha-SNAPSHOT`. This is because snapshot time-stamping happens on deployment only.

5.5.2. Accessing Project Sources and Resources

In certain cases, it's possible that a plugin may be introduced into the build process when a profile is activated. If this plugin adds resources like images, or new source code directories to the build, it can have dramatic effects on the resulting project artifact. For instance, when a project is built in a JDK 1.4 environment, it may be necessary to augment a project's code base with an additional source directory. Once this new source directory is in place, the compile mojo will require access to it, and other mojos may need to produce reports based on those same source directories. Therefore, it's important for mojos to be able to access and manipulate both the source directory list and the resource definition list for a project.

Adding a source directory to the build

Although the POM supports only a single `sourceDirectory` entry, Maven's concept of a project can accommodate a whole list of directories. This can be very useful when plugins generate source code, or simply need to augment the basic project code base. Maven's project API bridges this gap, allowing plugins to add new source directories as they execute. It requires access to the current `MavenProject` instance only, which can be injected into a mojo using the following code:

```
/**
 * Project instance, used to add new source directory to the build.
 * @parameter default-value="${project}"
 * @required
 * @readonly
 */
private MavenProject project;
```

This declaration identifies the project field as a required mojo parameter that will inject the current `MavenProject` instance into the mojo for use. As in the prior project dependencies discussion, this parameter also adds the `@readonly` annotation. This annotation tells Maven that users cannot modify this parameter, instead, it refers to a part of the build state that should always be present (a more in-depth discussion of this annotation is available in section 3.6, Chapter 3 of this book). The current project instance is a great example of this; any normal build will have a current project, and no other project contains current state information for this build.

It is possible that some builds won't have a current project, as in the case where the maven-archetype-plugin is used to create a stub of a new project. However, mojos require a current project instance to be available, unless declared otherwise. Maven will fail the build if it doesn't have a current project instance and it encounters a mojo that requires one. So, if you expect your mojo to be used in a context where there is no POM – as in the case of the archetype plugin – then simply add the class-level annotation: `@requiresProject` with a value of false, which tells Maven that it's OK to execute this mojo in the absence of a POM.

Once the current project instance is available to the mojo, it's a simple matter of adding a new source root to it, as in the following example:

```
project.addCompileSourceRoot( sourceDirectoryPath );
```

Mojos that augment the source-root list need to ensure that they execute ahead of the compile phase. The generally-accepted binding for this type of activity is in the generate-sources life-cycle phase. Further, when generating source code, the accepted default location for the generated source is in:

```
${project.build.directory}/generated-sources/<plugin-prefix>
```

While conforming with location standards like this is not required, it does improve the chances that your mojo will be compatible with other plugins bound to the same life cycle.

Adding a resource to the build

Another common practice is for a mojo to generate some sort of non-code resource, which will be packaged up in the same jar as the project classes. This could be some sort of descriptor for binding the project artifact into an application framework, as in the case of Maven itself, or it could be added to help identify some key attribute about the artifact to other developers.

Whatever the purpose of the mojo, the process of adding a new resource directory to the current build is straightforward. However, it does require a new type of parameter declaration: a requirement based on another component within the Maven application.

Just as the current project instance is required in order to add a new source root, the project is also required to achieve this. The project parameter is declared as follows:

```
/**
 * Project instance, used to add new source directory to the build.
 * @parameter default-value="${project}"
 * @required
 * @readonly
 */
private MavenProject project;
```

This declaration will inject the current project instance into the mojo, as discussed previously. However, to make it simpler to add resources to a project, the mojo also needs access to the `MavenProjectHelper` component. This component is part of the Maven application, which means it's always present; so your mojo simply needs to ask for it. The project helper component can be injected as follows:

```
/**
 * project-helper instance, used to make addition of resources
 * simpler.
 * @component
 */
private MavenProjectHelper helper;
```

Right away, you should notice something very different about this parameter. Namely, that it's not a parameter at all! In fact, this is what Maven calls a *component requirement* (it's a dependency on an internal component of the running Maven application). To be clear, the project helper is not a build state; it is a utility.

Component requirements are simple to declare; in most cases, the unadorned `@component` annotation – like the above code snippet – is adequate. Component requirements are not available for configuration by users.

Normally, the Maven application itself is well-hidden from the mojo developer. However, in some special cases, Maven components can make it much simpler to interact with the build process. For example, the `MavenProjectHelper` is provided to standardize the process of augmenting the project instance, and abstract the associated complexities away from the mojo developer. It provides methods for attaching artifacts and adding new resource definitions to the current project.

A complete discussion of Maven's architecture – and the components available – is beyond the scope of this chapter; however, the `MavenProjectHelper` component is worth mentioning here, as it is particularly useful to mojo developers.

With these two objects at your disposal, adding a new resource couldn't be easier. Simply define the resources directory to add, along with inclusion and exclusion patterns for resources within that directory, and then call a utility method on the project helper. The code should look similar to the following:

```
String directory = "relative/path/to/some/directory";
List includes = Collections.singletonList("**/*");
List excludes = null;

helper.addResource(project, directory, includes, excludes);
```

The prior example instantiates the resource's directory, inclusion patterns, and exclusion patterns as local variables, for the sake of brevity. In a typical case, these values would come from other mojo parameters, which may or may not be directly configurable.

Again, it's important to understand where resources should be added during the build life cycle. Resources are copied to the classes directory of the build during the process-resources phase. If your mojo is meant to add resources to the eventual project artifact, it will need to execute ahead of this phase. The most common place for such activities is in the generate-resources life-cycle phase. Again, conforming with these standards improves the compatibility of your plugin with other plugins in the build.

Accessing the source-root list

Just as some mojos add new source directories to the build, others must read the list of active source directories, in order to perform some operation on the source code. The classic example is the compile mojo in the maven-compiler-plugin, which actually compiles the source code contained in these root directories into classes in the project output directory. Other examples include javadoc mojo in the maven-javadoc-plugin, and the jar mojo in the maven-source-plugin. Gaining access to the list of source root directories for a project is easy; all you have to do is declare a single parameter to inject them, as in the following example:

```
/**
 * List of source roots containing non-test code.
 * @parameter default-value="${project.compileSourceRoots}"
 * @required
 * @readonly
 */
private List sourceRoots;
```

Similar to the parameter declarations from previous sections, this parameter declaration states that Maven does not allow users to configure this parameter directly; instead, they have to modify the `sourceDirectory` element in the POM, or else bind a mojo to the life-cycle phase that will add an additional source directory to the build. The parameter is also required for this mojo to execute; if it's missing, the entire build will fail.

Now that the mojo has access to the list of project source roots, it can iterate through them, applying whatever processing is necessary. Returning to the `buildinfo` example, it could be critically important to track the list of source directories used in a particular build, for eventual debugging purposes. If a certain profile injects a supplemental source directory into the build (most likely by way of a special mojo binding), then this profile would dramatically alter the resulting project artifact when activated. Therefore, in order to incorporate list of source directories to the `buildinfo` object, you need to add the following code:

```
for ( Iterator it = sourceRoots.iterator(); it.hasNext(); )
{
    String sourceRoot = (String) it.next();

    buildInfo.addSourceRoot( makeRelative( sourceRoot ) );
}
```

One thing to note about this code snippet is the `makeRelative()` method. By the time the mojo gains access to them, source roots are expressed as absolute file-system paths. In order to make this information more generally applicable, any reference to the path of the project directory in the local file system should be removed. This involves subtracting `${basedir}` from the source-root paths. To be clear, the `${basedir}` expression refers to the location of the project directory in the local file system.

When you add this code to the extract mojo in the `maven-buildinfo-plugin`, it will add a corresponding section to the `buildinfo` file that looks like the following:

```
<sourceRoots>
  <sourceRoot>src/main/java</sourceRoot>
  <sourceRoot>some/custom/srcDir</sourceRoot>
</sourceRoots>
```

Since a mojo using this code to access project source-roots does not actually modify the build state in any way, it can be bound to any phase in the life cycle. However, as in the case of the extract mojo, it's better to bind it to a later phase like package if capturing a complete picture of the project is important. Remember, binding this mojo to an early phase of the life cycle increases the risk of another mojo adding a new source root in a later phase. In this case however, binding to any phase later than compile should be acceptable, since compile is the phase where source files are converted into classes.

Accessing the resource list

Non-code resources complete the picture of the raw materials processed by a Maven build. You've already learned that mojos can modify the list of resources included in the project artifact; now, let's learn about how a mojo can access the list of resources used in a build. This is the mechanism used by the resources mojo in the maven-resources-plugin, which copies all non-code resources to the output directory for inclusion in the project artifact.

Much like the source-root list, the resources list is easy to inject as a mojo parameter. The parameter appears as follows:

```
/**
 * List of Resource objects for the current build, containing
 * directory, includes, and excludes.
 * @parameter default-value="${project.resources}"
 * @required
 * @readonly
 */
private List resources;
```

Just like the source-root injection parameter, this parameter is declared as required for mojo execution and cannot be edited by the user. In this case, the user has the option of modifying the value of the list by configuring the resources section of the POM.

As noted before with the dependencies parameter, allowing direct configuration of this parameter could easily produce results that are inconsistent with other resource-consuming mojos. It's also important to note that this list consists of Resource objects, which in fact contain information about a resource root, along with some matching rules for the resource files it contains.

Since the resources list is an instance of `java.util.List`, and Maven mojos must be able to execute in a JDK 1.4 environment that doesn't support Java generics, mojos must be smart enough to cast list elements as `org.apache.maven.model.Resource` instances.

Since mojos can add new resources to the build programmatically, capturing the list of resources used to produce a project artifact can yield information that is vital for debugging purposes. For instance, if an activated profile introduces a mojo that generates some sort of supplemental framework descriptor, it can mean the difference between an artifact that can be deployed into a server environment and an artifact that cannot. Therefore, it is important that the `buildinfo` file capture the resource root directories used in the build for future reference. It's a simple task to add this capability, and can be accomplished through the following code snippet:

```
if ( resources != null && !resources.isEmpty() )
{
    for ( Iterator it = resources.iterator(); it.hasNext(); )
    {
        Resource resource = (Resource) it.next();
        String resourceRoot = resource.getDirectory();

        buildInfo.addResourceRoot( makeRelative( resourceRoot ) );
    }
}
```

As with the prior source-root example, you'll notice the `makeRelative()` method. This method converts the absolute path of the resource directory into a relative path, by trimming the `${basedir}` prefix. All POM paths injected into mojos are converted to their absolute form first, to avoid any ambiguity. It's necessary to revert resource directories to relative locations for the purposes of the `buildinfo` plugin, since the `${basedir}` path won't have meaning outside the context of the local file system.

Adding this code snippet to the extract mojo in the `maven-buildinfo-plugin` will result in a `resourceRoots` section being added to the `buildinfo` file. That section should appear as follows:

```
<resourceRoots>
  <resourceRoot>src/main/resources</resourceRoot>
  <resourceRoot>target/generated-resources/xdoclet</resourceRoot>
</resourceRoots>
```

Once more, it's worthwhile to discuss the proper place for this type of activity within the build life cycle. Since all project resources are collected and copied to the project output directory in the process-resources phase, any mojo seeking to catalog the resources used in the build should execute at least as late as the process-resources phase. This ensures that any resource modifications introduced by mojos in the build process have been completed. Like the vast majority of activities, which may be executed during the build process, collecting the list of project resources has an appropriate place in the life cycle.

Note on testing source-roots and resources

All of the examples in this advanced development discussion have focused on the handling of source code and resources, which must be processed and included in the final project artifact. It's important to note however, that for every activity examined that relates to source-root directories or resource definitions, a corresponding activity can be written to work with their test-time counterparts.

This chapter does not discuss test-time and compile-time source roots and resources as separate topics; instead, due to the similarities, the key differences are summarized in the table below. The concepts are the same; only the parameter expressions and method names are different.

Table 5-2: Key differences between compile-time and test-time mojo activities

Activity	Change This	To This
Add testing source root	`project.addCompileSourceRoot()`	`project.addTestSourceRoot()`
Get testing source roots	`${project.compileSourceRoots}`	`${project.testSourceRoots}`
Add testing resource	`helper.addResource()`	`helper.addTestResource()`
Get testing resources	`${project.resources}`	`${project.testResources}`

5.5.3. Attaching Artifacts for Installation and Deployment

Occasionally, mojos produce new artifacts that should be distributed alongside the main project artifact in the Maven repository system. These artifacts are typically a derivative action or side effect of the main build process. Maven treats these *derivative* artifacts as attachments to the main project artifact, in that they are never distributed without the project artifact being distributed. Classic examples of attached artifacts are source archives, javadoc bundles, and even the `buildinfo` file produced in the examples throughout this chapter.

Once an artifact attachment is deposited in the Maven repository, it can be referenced like any other artifact. Usually, an artifact attachment will have a `classifier`, like `sources` or `javadoc`, which sets it apart from the main project artifact in the repository. Therefore, this classifier must also be specified when declaring the dependency for such an artifact, by using the `classifier` element for that dependency section within the POM.

When a mojo, or set of mojos, produces a derivative artifact, an extra piece of code must be executed in order to attach that artifact to the project artifact. Doing so guarantees that attachment will be distributed when the install or deploy phases are run. This extra step, which is still missing from the `maven-buildinfo-plugin` example, can provide valuable information to the development team, since it provides information about how each snapshot of the project came into existence.

While an e-mail describing the build environment is transient, and only serves to describe the latest build, the distribution of the `buildinfo` file via Maven's repository will provide a more permanent record of the build for each snapshot in the repository, for historical reference.

Including an artifact attachment involves adding two parameters and one line of code to your mojo. First, you'll need a parameter that references the current project instance as follows:

```
/**
 * Project instance, to which we want to add an attached artifact.
 * @parameter default-value="${project}"
 * @required
 * @readonly
 */
private MavenProject project;
```

This is the target of the attachment activities, in that it contains the project artifact that the build is in the process of creating. However, for convenience you should also inject the following reference to `MavenProjectHelper`, which will make the process of attaching the `buildinfo` artifact a little easier:

```
/**
 * This helper class makes adding an artifact attachment simpler.
 * @component
 */
private MavenProjectHelper helper;
```

See Section 5.6.2 for a discussion about `MavenProjectHelper` and component requirements.

Once you include these two fields in the extract mojo within the maven-buildinfo-plugin, the process of attaching the generated buildinfo file to the main project artifact can be accomplished by adding the following code snippet:

```
helper.attachArtifact( project, "xml",
                               "buildinfo", outputFile );
```

From the prior examples, the meaning and requirement of project and outputFile references should be clear. However, there are also two somewhat cryptic string values being passed in: "xml" and "buildinfo". These values represent the artifact extension and classifier, respectively.

By specifying an extension of "xml", you're telling Maven that the file in the repository should be named using a.xml extension. By specifying the "buildinfo" classifier, you're telling Maven that this artifact should be distinguished from other project artifacts by using this value in the classifier element of the dependency declaration. It identifies the file as being produced by the the maven-buildinfo-plugin, as opposed to another plugin in the build process which might produce another XML file with different meaning. This serves to attach meaning beyond simply saying, "This is an XML file".

Now that you've added code to distribute the buildinfo file, you can test it by re-building the plugin, then running Maven to the install life-cycle phase on our test project. If you build the Guinea Pig project using this modified version of the maven-buildinfo-plugin, you should see the buildinfo file appear in the local repository alongside the project jar, as follows:

```
> mvn install
> cd C:\Documents and Settings\jdcasey\.m2\repository
> cd org\codehaus\guineapig\guinea-pig-core\1.0-SNAPSHOT
> dir

guinea-pig-core-1.0-SNAPSHOT-buildinfo.xml
guinea-pig-core-1.0-SNAPSHOT.jar
guinea-pig-core-1.0-SNAPSHOT.pom
```

Now, the maven-buildinfo-plugin is ready for action. It can extract relevant details from a running build and generate a buildinfo file based on these details. From there, it can attach the buildinfo file to the main project artifact so that it's distributed whenever Maven installs or deploys the project.

Finally, when the project is deployed, the maven-buildinfo-plugin can also generate an e-mail that contains the buildinfo file contents, and route that message to other development team members on the project development mailing list.

5.6. Summary

In its unadorned state, Maven represents an implementation of the 80/20 rule. Using the default life-cycle mapping, Maven can build a basic project with little or no modification – thus covering the 80% case. However, in certain circumstances, a project requires special tasks in order to build successfully. Whether they be code-generation, reporting, or verification steps, Maven can integrate these custom tasks into the build process through its extensible plugin framework. Since the build process for a project is defined by the plugins – or more accurately, the mojos – that are bound to the build life cycle, there is a standardized way to inject new behavior into the build by binding new mojos at different life-cycle phases.

In this chapter, you've learned that it's a relatively simple to create a mojo, which can extract relevant parts of the build state in order to perform a custom build-process task – even to the point of altering the set of source-code directories used to build the project. Working with project dependencies and resources is equally as simple. Finally, you've also learned how a plugin generated file can be distributed alongside the project artifact in Maven's repository system, enabling you to attach custom artifacts for installation or deployment.

Many plugins already exist for Maven use, only a tiny fraction of which are a part of the default life-cycle mapping. If your project requires special handling, chances are good that you can find a plugin to address this need at the Apache Maven project, the Codehaus Mojo project, or the project web site of the tools with which your project's build must to integrate. If not, developing a custom Maven plugin is an easy next step.

Mojo development can be as simple or as complex (to the point of embedding nested Maven processes within the build) as you need it to be. Using the plugin mechanisms described in this chapter, you can integrate almost any tool into the build process.

However, remember that whatever problem your custom-developed plugin solves, it's unlikely to be a requirement unique to your project. So, if you have the means, please consider contributing back to the Maven community by providing access to your new plugin. It is in great part due to the re-usable nature of its plugins that Maven can offer such a powerful build platform.

Assessing Project Health with Maven

This chapter covers:

- How Maven relates to project health
- Organizing documentation and developer reports
- Selecting the right tools to monitor the health of your project
- How to incorporate reporting tools
- Tips for how to use tools more effectively

Life is not an exact science, it is an art.

- Samuel Butler

6.1. What Does Maven Have to do With Project Health?

In the introduction, it was pointed out that Maven's application of patterns provides visibility and comprehensibility. It is these characteristics that assist you in assessing the health of your project.

Through the POM, Maven has access to the information that makes up a project, and using a variety of tools, Maven can analyze, relate, and display that information in a single place. Because the POM is a *declarative* model of the project, new tools that can assess its health are easily integrated. In this chapter, you'll learn how to use a number of these tools effectively.

When referring to health, there are two aspects to consider:

- **Code quality** - determining how well the code works, how well it is tested, and how well it adapts to change.
- **Project vitality** - finding out whether there is any activity on the project, and what the nature of that activity is.

Maven takes all of the information you need to know about your project and brings it together under the project Web site. The next three sections demonstrate how to set up an effective project Web site.

It is important not to get carried away with setting up a fancy Web site full of reports that nobody will ever use (especially when reports contain failures they don't want to know about!). For this reason, many of the reports illustrated can be run as part of the regular build in the form of a “check” that will fail the build if a certain condition is not met.

But, why have a site, if the build fails its checks? The Web site also provides a permanent record of a project's health, which everyone can see at any time. It provides additional information to help determine the reasons for a failed build, and whether the conditions for the checks are set correctly. This is important, because if the bar is set too high, there will be too many failed builds. This is unproductive as minor changes are prioritized over more important tasks, to get a build to pass. Conversely, if the bar is set too low, the project will meet only the lowest standard and go no further.

In this chapter, you will be revisiting the Proficio application that was developed in Chapter 3, and learning more about the health of the project. The code that concluded Chapter 3 is also included in `Code_Ch03.zip` for convenience as a starting point. To begin, unzip the `Code_Ch03.zip` file into `C:\mvnbook` or your selected working directory, and then run `mvn install` from the `proficio` subdirectory to ensure everything is in place.

6.2. Adding Reports to the Project Web site

This section builds on the information on project Web sites in Chapter 2 and Chapter 3, and now shows how to integrate project health information.

To start, review the project Web site shown in figure 6-1.

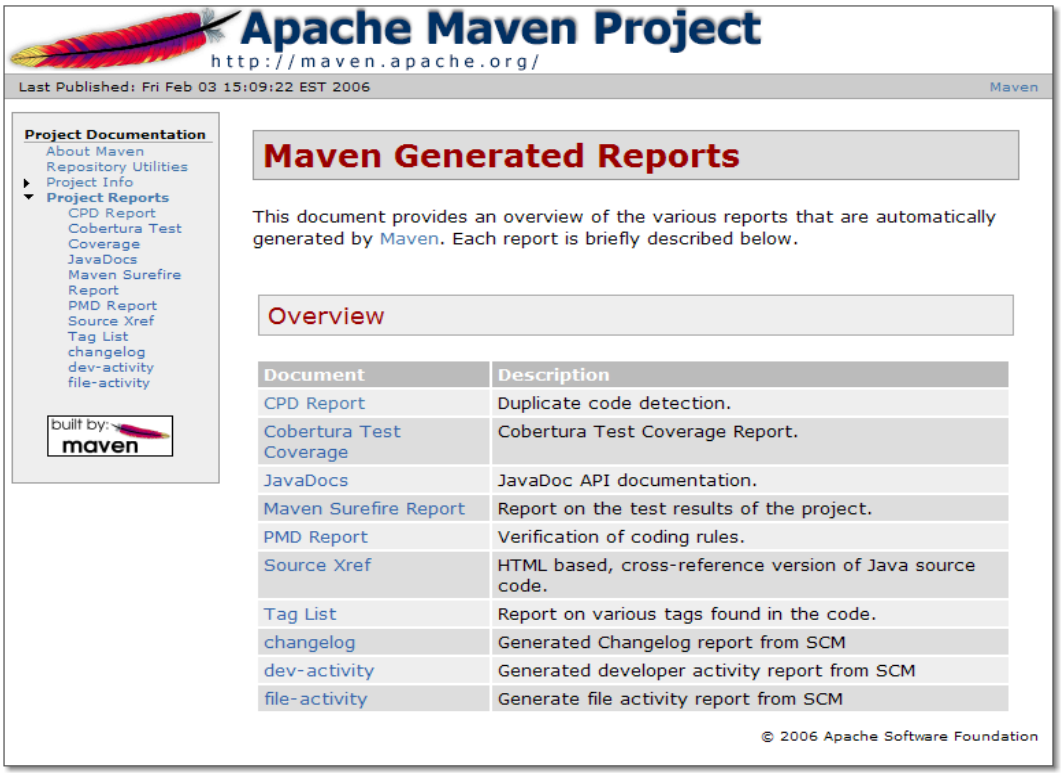


Figure 6-1: The reports generated by Maven

You can see that the navigation on the left contains a number of reports. The *Project Info* menu lists the standard reports Maven includes with your site by default, unless you choose to disable them. These reports are useful for sharing information with others, and to reference as links in your mailing lists, SCM, issue tracker, and so on. For newcomers to the project, having these standard reports means that those familiar with Maven Web sites will always know where to find the information they need.

The second menu (shown opened in figure 6-1), *Project Reports*, is the focus of the rest of this chapter. These reports provide a variety of insights into the quality and vitality of the project.

On a new project, this menu doesn't appear as there are no reports included. However, adding a new report is easy. For example, you can add the Surefire report to the sample application, by including the following section in `proficio/pom.xml`:

```
...
 <reporting>
  <plugins>
  < plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-report-plugin</artifactId>
   </plugin>
  </plugins>
 </reporting>
...
</project>
```

This adds the report to the top level project, and as a result, it will be inherited by all of the child modules. You can now run the following site task in the `proficio-core` directory to regenerate the site.

```
C:\mvnbook\proficio\proficio-core> mvn site
```

This can now be found in the file `target/site/surefire-report.html`, and is shown in figure 6-2.

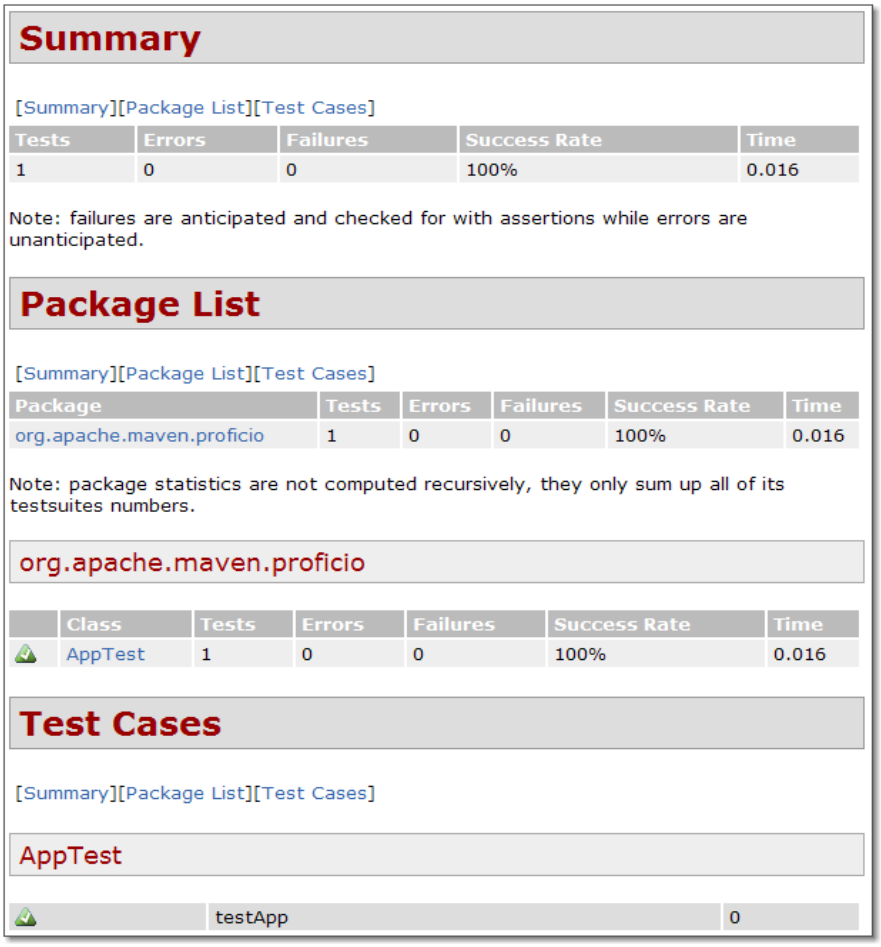


Figure 6-2: The Surefire report

As you may have noticed in the summary, the report shows the test results of the project.

For a quicker turn around, the report can also be run individually using the following standalone goal:

```
C:\mvnbook\proficio\proficio-core> mvn surefire-report:report
```

That's all there is to generating the report! This is possible thanks to key concepts of Maven discussed in Chapter 2: through a **declarative project model**, Maven knows where the tests and test results are, and due to using **convention over configuration**, the defaults are sufficient to get started with a useful report.

6.3. Configuration of Reports

Before stepping any further into using the project Web site, it is important to understand how the report configuration is handled in Maven.

You might recall from Chapter 2 that a plugin is configured using the configuration element inside the plugin declaration in `pom.xml`, for example:

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

Configuration for a reporting plugin is very similar, however it is added to the reporting section of the POM. For example, the report can be modified to only show test failures by adding the following configuration in `pom.xml`:

```
...
<reporting>
 <plugins>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-report-plugin</artifactId>
  <configuration>
  <showSuccess>false</showSuccess>
  </configuration>
 </plugin>
 </plugins>
</reporting>
...
```

The addition of the plugin element triggers the inclusion of the report in the Web site, as seen in the previous section, while the configuration can be used to modify its appearance or behavior.

If a plugin contains multiple reports, they will all be included.

However, some reports apply to both the site, and the build. To continue with the Surefire report, consider if you wanted to create a copy of the HTML report in the directory `target/surefire-reports` every time the build ran. To do this, the plugin would need to be configured in the build section instead of, or in addition to, the `reporting` section:

```
...
<build>
 <plugins>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-report-plugin</artifactId>
  <configuration>
  <outputDirectory>
   ${project.build.directory}/surefire-reports
  </outputDirectory>
  </configuration>
  <executions>
  <execution>
   <phase>test</phase>
   <goals>
   <goal>report</goal>
   </goals>
  </execution>
  </executions>
 </plugin>
 </plugins>
</build>
...
```

“Executions” such as this were introduced in Chapter 3. The plugin is included in the build section to ensure that the configuration, even though it is not specific to the execution, is used only during the build, and not site generation.

> Plugins and their associated configuration that are declared in the build section are not used during site generation.

However, what if the location of the Surefire XML reports that are used as input (and would be configured using the `reportsDirectory` parameter) were different to the default location? Initially, you might think that you'd need to configure the parameter in both sections. Fortunately, this isn't the case – adding the configuration to the reporting section is sufficient.

> Any plugin configuration declared in the reporting section is also applied to those declared in the build section.

When you configure a reporting plugin, always place the configuration in the reporting section – unless one of the following is true:

1. The reports will not be included in the site
2. The configuration value is specific to the build stage

When you are configuring the plugins to be used in the reporting section, by default all reports available in the plugin are executed once. However, there are cases where only some of the reports that the plugin produces will be required, and cases where a particular report will be run more than once, each time with a different configuration.

Both of these cases can be achieved with the `reportSets` element, which is the reporting equivalent of the executions element in the build section. Each report set can contain configuration, and a list of reports to include. For example, consider if you had run Surefire twice in your build, once for unit tests and once for a set of performance tests, and that you had had generated its XML results to target/surefire-reports/unit and `target/surefire-reports/perf` respectively.

To generate two HTML reports for these results, you would include the following section in your `pom.xml`:

```
...
<reporting>
 <plugins>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-report-plugin</artifactId>
  <reportSets>
  <reportSet>
   <id>unit</id>
   <configuration>
   <reportsDirectory>
    ${project.build.directory}/surefire-reports/unit
   </reportsDirectory>
   <outputName>surefire-report-unit</outputName>
   </configuration>
   <reports>
   <report>report</report>
   </reports>
  </reportSet>
  <reportSet>
   <id>perf</id>
   <configuration>
   <reportsDirectory>
    ${project.build.directory}/surefire-reports/perf
   </reportsDirectory>
   <outputName>surefire-report-perf</outputName>
   </configuration>
   <reports>
   <report>report</report>
   </reports>
  </reportSet>
  </reportSets>
 </plugin>
 </plugins>
</reporting>
...
```

Running `mvn` site with this addition will generate two Surefire reports: target/site/surefire-report-unit.html and `target/site/surefire-report-perf.html`.

However, as with executions, running `mvn surefire-report:report` will not use either of these configurations. When a report is executed individually, Maven will use only the configuration that is specified in the plugin element itself, outside of any report sets.

The reports element in the report set is a required element. If you want all of the reports in a plugin to be generated, they must be enumerated in this list. The reports in this list are identified by the goal names that would be used if they were run from the command line.

It is also possible to include only a subset of the reports in a plugin. For example, to generate only the mailing list and license pages of the standard reports, add the following to the reporting section of the `pom.xml` file:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-project-info-reports-plugin</artifactId>
 <reportSets>
 <reportSet>
  <reports>
  <report>mailing-list</report>
  <report>license</report>
  </reports>
 </reportSet>
 </reportSets>
</plugin>
...
```

While the defaults are usually sufficient, this customization will allow you to configure reports in a way that is just as flexible as your build.

6.4. Separating Developer Reports From User Documentation

After adding a report, there's something subtly wrong with the project Web site. On the entrance page there are usage instructions for Proficio, which are targeted at an end user, but in the navigation there are reports about the health of the project, which are targeted at the developers.

This may be confusing for the first time visitor, who isn't interested in the state of the source code, and an inconvenience to the developer who doesn't want to wade through end user documentation to find out the current state of a project's test coverage.

This approach to balancing these competing requirements will be vary, depending on the project. Consider the following:

- The commercial product, where the end user documentation is on a completely different server than the developer information, and most likely doesn't use Maven to generate it;
- The open source graphical application, where the developer information is available, but quite separate to the end user documentation;
- The open source reusable library, where much of the source code and Javadoc reference is of interest to the end user.

To determine the correct balance, each section of the site needs to be considered; in some cases down to individual reports. Table 6-1 lists the content that a project Web site may contain, and the content's characteristics.

Table 6-1: Project Web site content types

Content	Description	Updated	Distributed	Separated
News, FAQs and general Web site	This is the content that is considered part of the Web site rather than part of the documentation.	Yes	No	Yes
End user documentation	This is documentation for the end user including usage instructions and guides. It refers to a particular version of the software.	Yes	Yes	No
Source code reference material	This is reference material (for example, Javadoc) that in a library or framework is useful to the end user, but usually not distributed or displayed in an application.	No	Yes	No
Project health and vitality reports	These are the reports discussed in this chapter that display the current state of the project to the developers.	Yes	No	No

In the table, the **Updated** column indicates whether the content is regularly updated, regardless of releases. This is true of the news and FAQs, which are based on time and the current state of the project. Some standard reports, like mailing list information and the location of the issue tracker and SCM are updated also. It is also true of the project quality and vitality reports, which are continuously published and not generally of interest for a particular release. However, source code references should be given a version and remain unchanged after being released.

The situation is different for end user documentation. It is good to update the documentation on the Web site between releases, and to maintain only one set of documentation.

Features that are available only in more recent releases should be marked to say when they were introduced. It is important not to include documentation for features that don't exist in the last release, as it is confusing for those reading the site who expect it to reflect the latest release.

The best compromise between not updating between releases, and not introducing incorrect documentation, is to branch the end user documentation in the same way as source code. You can maintain a stable branch, that can be updated between releases without risk of including new features, and a development branch where new features can be documented for when that version is released.

The **Distributed** column in the table indicates whether that form of documentation is typically distributed with the project. This is typically true for the end user documentation. For libraries and frameworks, the Javadoc and other reference material are usually distributed for reference as well. Sometimes these are included in the main bundle, and sometimes they are available for download separately.

The **Separated** column indicates whether the documentation can be a separate module or project. While there are some exceptions, the source code reference material and reports are usually generated from the modules that hold the source code and perform the build. For a single module library, including the end user documentation in the normal build is reasonable as it is closely tied to the source code reference.

However, in most cases, the documentation and Web site should be kept in a separate module dedicated to generating a site. This avoids including inappropriate report information and navigation elements.

This separated documentation may be a module of the main project, or maybe totally independent. You would make it a module when you wanted to distribute it with the rest of the project, but make it an independent project when it forms the overall site with news and FAQs, and is not distributed with the project.

It is important to note that none of these are restrictions placed on a project by Maven. While these recommendations can help properly link or separate content according to how it will be used, you are free to place content wherever it best suits your project.

In Proficio, the site currently contains end user documentation and a simple report. In the following example, you will learn how to separate the content and add an independent project for the news and information Web site.

The current structure of the project is shown in figure 6-3.

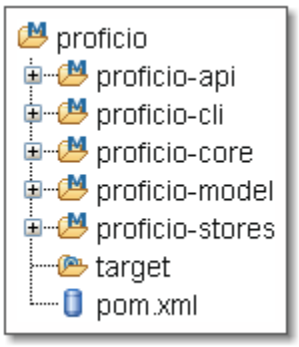


Figure 6-3: The initial setup

The first step is to create a module called user-guide for the end user documentation. In this case, a module is created since it is not related to the source code reference material. This is done using the site archetype :

```
C:\mvnbook\proficio> mvn archetype:create -DartifactId=user-guide \
          -DgroupId=com.mergere.mvnbook.proficio \
          -DarchetypeArtifactId=maven-archetype-site-simple
```

This archetype creates a very basic site in the user-guide subdirectory, which you can later add content to. The resulting structure is shown in figure 6-4.

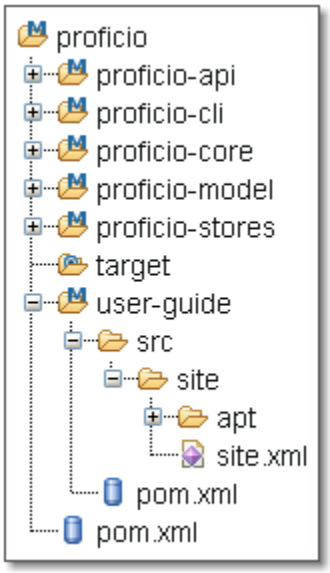


Figure 6-4: The directory layout with a user guide

The next step is to ensure the layout on the Web site is correct. Previously, the URL and deployment location were set to the root of the Web site: http://library.mergere.com/mvnbook/proficio. Under the current structure, the development documentation would go to that location, and the user guide to http://library.mergere.com/mvnbook/proficio/user-guide.

In this example, the development documentation will be moved to a `/reference/version` subdirectory so that the top level directory is available for a user-facing web site.

> Adding the version to the development documentation, while optional, is useful if you are maintaining multiple public versions, whether to maintain history or to maintain a release and a development preview.

First, edit the top level `pom.xml` file to change the site deployment `url`:

```
...
<distributionManagement>
 <site>
...
 <url>
 scp://mergere.com/www/library/mvnbook/proficio/reference/${project.version}
 </url>
 </site>
</distributionManagement>
...
```

Next, edit the `user-guide/pom.xml` file to set the site deployment `url` for the module:

```
...
 <distributionManagement>
  <site>
  <id>mvnbook.site</id>
  <url>
   scp://mergere.com/www/library/mvnbook/proficio/user-guide
 </url>
 </site>
</distributionManagement>
...
```

There are now two sub-sites ready to be deployed:

- http://library.mergere.com/mvnbook/proficio/user-guide/
- http://library.mergere.com/mvnbook/proficio/reference/1.0-SNAPSHOT/

You will not be able to deploy the Web site to the location `scp://mergere.com/www/library/mvnbook/proficio/user-guide` and it is included here only for illustrative purposes.

Now that the content has moved, a top level site for the project is required. This will include news and FAQs about the project that change regularly.

As before, you can create a new site using the archetype. This time, run it one directory above the `proficio` directory, in the `ch06-1` directory.

```
C:\mvnbook> mvn archetype:create -DartifactId=proficio-site \
      -DgroupId=com.mergere.mvnbook.proficio \
      -DarchetypeArtifactId=maven-archetype-site-simple
```

The resulting structure is shown in figure 6-5.

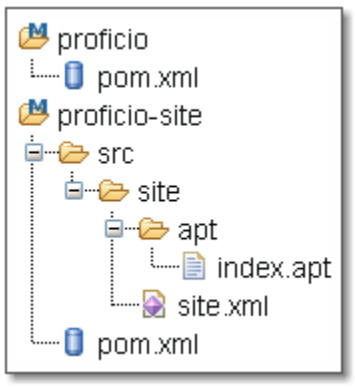


Figure 6-5: The new Web site

You will need to add the same elements to the POM for the `url` and `distributionManagement` as were set originally for `proficio/pom.xml` as follows:

```
[...]
<url>http://library.mergere.com/mvnbook/proficio</url>
[...]
<distributionManagement>
 <site>
 <id>mvnbook.website</id>
 <url>scp://mergere.com/www/library/mvnbook/proficio</url>
 </site>
</distributionManagement>
[...]
```

Next, replace the `src/site/apt/index.apt` file with a more interesting news page, like the following:

```
 -----
Proficio
 -----
Joe Blogs
 -----
2 February 2006
 -----

Proficio

 Proficio is super.

* News

 * <16 Jan 2006> - Proficio project started
```

Finally, add some menus to `src/site/site.xml` that point to the other documentation as follows:

```
...
<menu name="Documentation">
 <item name="User's Guide" href="/user-guide/" />
</menu>

<menu name="Reference">
 <item name="API" href="/reference/1.0-SNAPSHOT/apidocs/" />
 <item name="Developer Info" href="/reference/1.0-SNAPSHOT/" />
</menu>
...
```

You can now run `mvn` site in `proficio-site` to see how the separate site will look. If you deploy both sites to a server using `mvn` site-deploy as you learned in Chapter 3, you will then be able to navigate through the links and see how they relate.

> Note that you haven't produced the `apidocs` directory yet, so that link won't work even if the site is deployed. Generating reference documentation is covered in section 6.6 of this chapter.

The rest of this chapter will focus on using the developer section of the site effectively and how to build in related conditions to regularly monitor and improve the quality of your project.

6.5. Choosing Which Reports to Include

Choosing which reports to include, and which checks to perform during the build, is an important decision that will determine the effectiveness of your build reports. Report results and checks performed should be accurate and conclusive – every developer should know what they mean and how to address them.

> In some instances, the performance of your build will be affected by this choice. In particular, the reports that utilize unit tests often have to re-run the tests with new parameters. While future versions of Maven will aim to streamline this, it is recommended that these checks be constrained to the continuous integration and release environments if they cause lengthy builds. See Chapter 7, *Team Collaboration with Maven*, for more information.

Table 6-2 covers the reports discussed in this chapter and reasons to use them. For each report, there is also a note about whether it has an associated visual report (for project site inclusion), and an applicable build check (for testing a certain condition and failing the build if it doesn't pass).

You can use this table to determine which reports apply to your project specifically and limit your reading to just those relevant sections of the chapter, or you can walk through all of the examples one by one, and look at the output to determine which reports to use.

While these aren't all the reports available for Maven, the guidelines should help you to determine whether you need to use other reports.

> You may notice that many of these tools are Java-centric. While this is certainly the case at present, it is possible in the future that reports for other languages will be available, in addition to the generic reports such as those for dependencies and change tracking.

Table 6-2: Report highlights

Report	Description	Visual	Check	Notes
Javadoc	Produces an API reference from Javadoc.	Yes	N/A	✔ Useful for most Java software ✔ Important for any projects publishing a public API
JXR	Produces a source cross reference for any Java code.	Yes	N/A	✔ Companion to Javadoc that shows the source code ✔ Important to include when using other reports that can refer to it, such as Checkstyle ✗ Doesn't handle JDK 5.0 features
Checkstyle	Checks your source code against a standard descriptor for formatting issues.	Yes	Yes	✔ Use to enforce a standard code style. ✔ Recommended to enhance readability of the code. ✗ Not useful if there are a lot of errors to be fixed – it will be slow and the result unhelpful.

Report	Description	Visual	Check	Notes
PMD	Checks your source code against known rules for code smells.	Yes	Yes	✔ Should be used to improve readability and identify simple and common bugs. ✔ Some overlap with Checkstyle rules
CPD	Part of PMD, checks for duplicate source code blocks that indicates it was copy/pasted.	Yes	No	✔ Can be used to identify lazy copy/pasted code that might be refactored into a shared method. ✔ Avoids issues when one piece of code is fixed/updated and the other forgotten
Tag List	Simple report on outstanding tasks or other markers in source code	Yes	No	✔ Useful for tracking TODO items ✔ Very simple, convenient set up ✔ Can be implemented using Checkstyle rules instead.
Cobertura	Analyze code statement coverage during unit tests or other code execution.	Yes	Yes	✔ Recommended for teams with a focus on tests ✔ Can help identify untested or even unused code. ✗ Doesn't identify all missing or inadequate tests, so additional tools may be required.
Surefire Report	Show the results of unit tests visually.	Yes	Yes	✔ Recommended for easier browsing of results. ✔ Can also show any tests that are long running and slowing the build. ✔ Check already performed by surefire:test.
Dependency Convergence	Examine the state of dependencies in a multiple module build	Yes	No	✔ Recommended for multiple module builds where consistent versions are important. ✔ Can help find snapshots prior to release.
Clirr	Compare two versions of a JAR for binary compatibility	Yes	Yes	✔ Recommended for libraries and frameworks with a public API ✔ Also important for reviewing changes to the internal structure of a project that are still exposed publicly.
Changes	Produce release notes and road maps from issue tracking systems	Yes	N/A	✔ Recommended for all publicly released projects. ✔ Should be used for keeping teams up to date on internal projects also.

6.6. Creating Reference Material

Source code reference materials are usually the first reports configured for a new project, because it is often of interest to the end user of a library or framework, as well as to the developer of the project itself.

The two reports this section illustrates are:

- JXR – the Java source cross reference, and,
- Javadoc – the Java documentation tool

You can get started with JXR on the example project very quickly, by running the following command:

```
C:\mvnbook\proficio\proficio-core> mvn jxr:jxr
```

You should now see a `target/site/jxr.html` created. This page links to the two sets of cross references: one for the main source tree, and one for the unit tests.

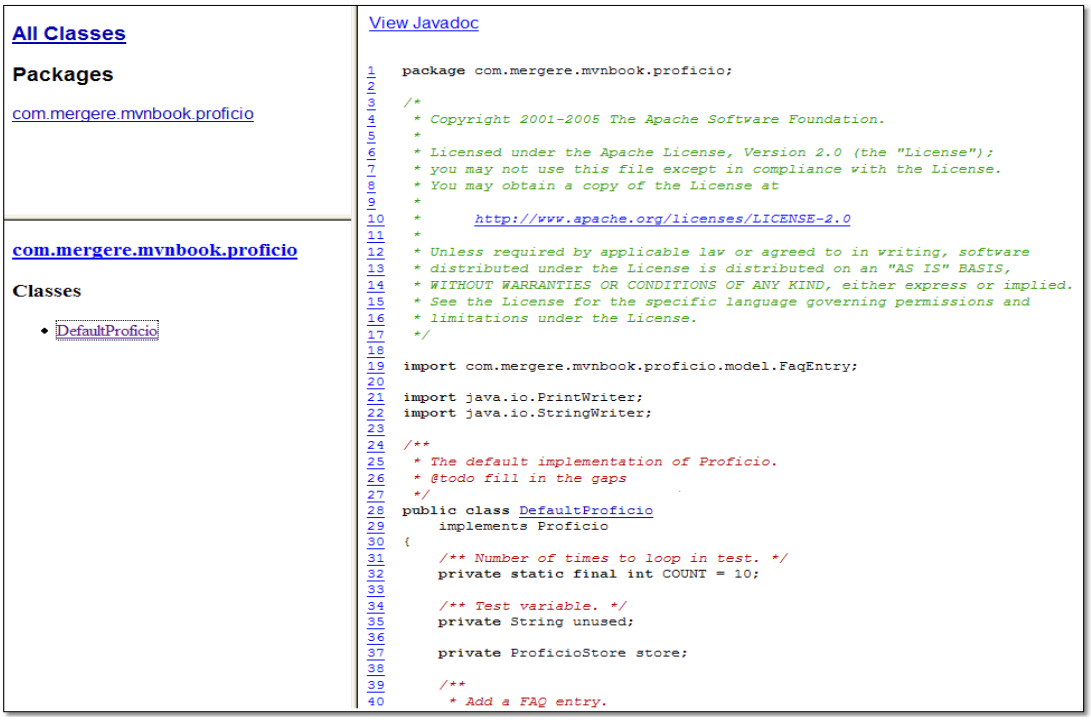


Figure 6-6: An example source code cross reference

Figure 6-6 shows an example of the cross reference. Those familiar with Javadoc will recognize the framed navigation layout, however the content pane is now replaced with a syntax-highlighted, cross-referenced Java source file for the selected class. The hyper links in the content pane can be used to navigate to other classes and interfaces within the cross reference.

A useful way to leverage the cross reference is to use the links given for each line number in a source file to point team mates at a particular piece of code. Or, if you don't have the project open in your IDE, the links can be used to quickly find the source belonging to a particular exception.

Including JXR as a permanent fixture of the site for the project is simple, and can be done by adding the following to `proficio/pom.xml`:

```
...
<reporting>
 <plugins>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jxr-plugin</artifactId>
 </plugin>
...
 </plugins>
</reporting>
...
```

You can now run `mvn` site in `proficio-core` and see the *`Source Xref`* item listed in the *`Project Reports`* menu of the generated site.

In most cases, the default JXR configuration is sufficient, however if you'd like a list of available configuration options, see the plugin reference at http://maven.apache.org/plugins/maven-jxr-plugin/.

Now that you have a source cross reference, many of the other reports demonstrated in this chapter will be able to link to the actual code to highlight an issue. However, browsing source code is too cumbersome for the developer if they only want to know about how the API works, so an equally important piece of reference material is the Javadoc report.

> A Javadoc report is only as good as your Javadoc! Make sure you document the methods you intend to display in the report, and if possible use `Checkstyle` to ensure they are documented.

Using Javadoc is very similar to the JXR report and most other reports in Maven. Again, you can run it on its own using the following command:

```
C:\mvnbook\proficio\proficio-core> mvn javadoc:javadoc
```

Since it will be included as part of the project site, you should include it in `proficio/pom.xml` as a site report to ensure it is run every time the site is regenerated:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-javadoc-plugin</artifactId>
</plugin>
...
```

The end result is the familiar Javadoc output, in `target/site/apidocs`.

Unlike JXR, the Javadoc report is quite configurable, with most of the command line options of the Javadoc tool available.

One useful option to configure is links. In the `online` mode, this will link to an external Javadoc reference at a given URL.

For example, the following configuration, when added to `proficio/pom.xml`, will link both the JDK 1.4 API documentation and the Plexus container API documentation used by Proficio:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-javadoc-plugin</artifactId>
 <configuration>
 <links>
  <link>http://java.sun.com/j2se/1.4.2/docs/api</link>
  <link>http://plexus.codehaus.org/ref/1.0-alpha-9/apidocs</link>
 </links>
 </configuration>
</plugin>
...
```

If you regenerate the site in `proficio-core` with `mvn` site again, you'll see that all references to the standard JDK classes such as `java.lang.String` and `java.lang.Object`, are linked to API documentation on the Sun website, as well as any references to classes in Plexus.

Setting up Javadoc has been very convenient, but it results in a separate set of API documentation for each library in a multi-module build. Since it is preferred to have discreet functional pieces separated into distinct modules, but conversely to have the Javadoc closely related, this is not sufficient.

One option would be to introduce links to the other modules (automatically generated by Maven based on dependencies, of course!), but this would still limit the available classes in the navigation as you hop from module to module. Instead, the Javadoc plugin provides a way to produce a single set of API documentation for the entire project.

Edit the configuration of the existing Javadoc plugin in `proficio/pom.xml` by adding the following line:

```
...
<configuration>
 <aggregate>true</aggregate>
...
</configuration>
...
```

When built from the top level project, this simple change will produce an aggregated Javadoc and ignore the Javadoc report in the individual modules. This setting must go into the reporting section so that it is used for both reports and if the command is executed separately. However, this setting is always ignored by the javadoc:jar goal, ensuring that the deployed Javadoc corresponds directly to the artifact with which it is deployed for use in an IDE.

Try running `mvn` clean javadoc:javadoc in the `proficio` directory to produce the aggregated Javadoc in `target/site/apidocs/index.html`.

Now that the sample application has a complete reference for the source code, the next section will allow you to start monitoring and improving its health.

6.7. Monitoring and Improving the Health of Your Source Code

There are several factors that contribute to the health of your source code:

- **Accuracy** – whether the code does what it is expected to do
- **Robustness** – whether the code gracefully handles exceptional conditions
- **Extensibility** – how easily the code can be changed without affecting accuracy or requiring changes to a large amount of other code
- **Readability** – how easily the code can be understood (in a team environment, this is important for both the efficiency of other team members and also to increase the overall level of code comprehension, which in turn reduces the risk that its accuracy will be affected by change)

Maven has reports that can help with each of these health factors, and this section will look at three:

- PMD (http://pmd.sf.net/)
- Checkstyle (http://checkstyle.sf.net/)
- Tag List

PMD takes a set of either predefined or user-defined rule sets and evaluates the rules across your Java source code. The result can help identify bugs, `copy-and-pasted` code, and violations of a coding standard. Figure 6-7 shows the output of a PMD report on `proficio-core`, which is obtained by running `mvn pmd:pmd`.

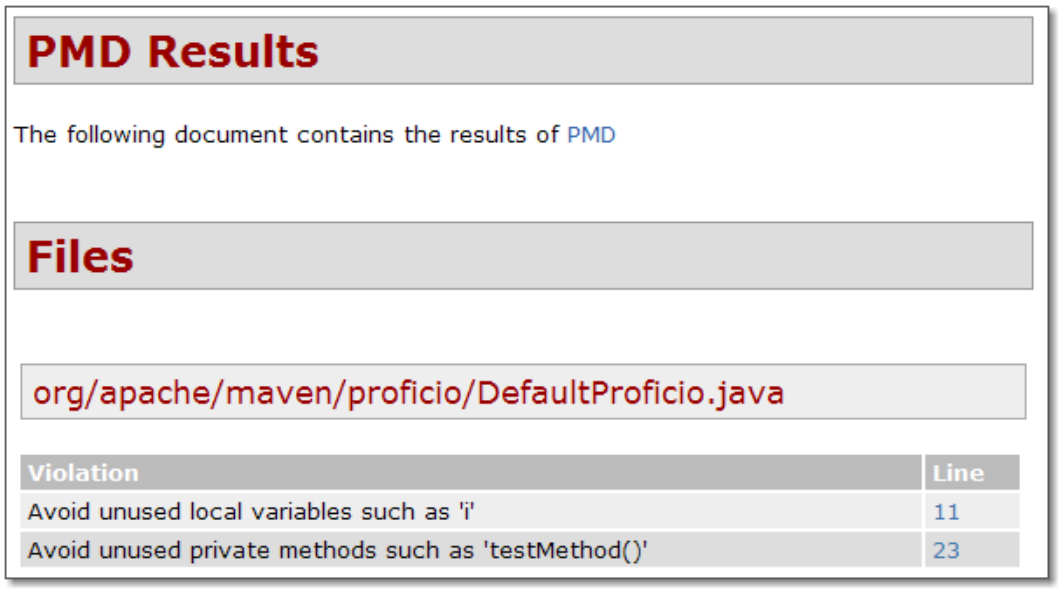


Figure 6-7: An example PMD report

As you can see, some source files are identified as having problems that could be addressed, such as unused methods and variables. Also, since the JXR report was included earlier, the line numbers in the report are linked to the actual source code so you can browse the issues.

Adding the default PMD report to the site is just like adding any other report – you can include it in the reporting section in the `proficio/pom.xml` file:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-pmd-plugin</artifactId>
</plugin>
...
```

The default PMD report includes the *basic*, *unused code,* and *imports* rule sets. The “basic” rule set includes checks on empty blocks, unnecessary statements and possible bugs – such as incorrect loop variables. The “unused code” rule set will locate unused private fields, methods, variables and parameters. The “imports” rule set will detect duplicate, redundant or unused import declarations.

Adding new rule sets is easy, by passing the `rulesets` configuration to the plugin. However, if you configure these, you must configure *all* of them – including the defaults explicitly. For example, to include the default rules, and the `finalizer` rule sets, add the following to the plugin configuration you declared earlier:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-pmd-plugin</artifactId>
 <configuration>
 <rulesets>
  <ruleset>/rulesets/basic.xml</ruleset>
  <ruleset>/rulesets/imports.xml</ruleset>
  <ruleset>/rulesets/unusedcode.xml</ruleset>
  <ruleset>/rulesets/finalizers.xml</ruleset>
 </rulesets>
 </configuration>
</plugin>
...
```

You may find that you like some rules in a rule set, but not others. Or, you may use the same rule sets in a number of projects. In either case, you can choose to create a custom rule set. For example, you could create a rule set with all the default rules, but exclude the “unused private field” rule. To try this, create a file in the `proficio-core` directory of the sample application called `src/main/pmd/custom.xml`, with the following content:

```
<?xml version="1.0"?>
<ruleset name="custom">
 <description>
 Default rules, no unused private field warning
 </description>
 <rule ref="/rulesets/basic.xml" />
 <rule ref="/rulesets/imports.xml" />
 <rule ref="/rulesets/unusedcode.xml">
 <exclude name="UnusedPrivateField" />
 </rule>
</ruleset>
```

To use this rule set, change the configuration by overriding it in the `proficio-core/pom.xml` file, adding:

```
...
<reporting>
 <plugins>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <configuration>
  <rulesets>
   <ruleset>${basedir}/src/main/pmd/custom.xml</ruleset>
  </rulesets>
  </configuration>
 </plugin>
 </plugins>
</reporting>
...
```

For more examples on customizing the rule sets, see the instructions on the PMD Web site at http://pmd.sf.net/howtomakearuleset.html. It is also possible to write your own rules if you find that existing ones do not cover recurring problems in your source code.

One important question is how to select appropriate rules. For PMD, try the following guidelines from the Web site at http://pmd.sf.net/bestpractices.html:

- Pick the rules that are right for you. There is no point having hundreds of violations you won't fix.
- Start small, and add more as needed. basic, `unusedcode`, and imports are useful in most scenarios and easily fixed. From this starting, select the rules that apply to your own project.

If you've done all the work to select the right rules and are correcting all the issues being discovered, you need to make sure it stays that way.

Try this now by running `mvn pmd:check` on `proficio-core`. You'll see that the build fails with the following 3 errors:

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Proficio Core
[INFO]  task-segment: [pmd:check]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing pmd:check
[INFO] [pmd:pmd]
[INFO] [pmd:check]
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] You have 3 PMD violations.
[INFO] ------------------------------------------------------------------------
```

Before correcting these errors, you should include the check in the build, so that it is regularly tested. This is done by binding the goal to the build life cycle. To do so, add the following section to the `proficio/pom.xml` file:

```
<build>
 <plugins>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <executions>
  <execution>
   <goals>
   <goal>check</goal>
   </goals>
  </execution>
  </executions>
 </plugin>
...
 </plugins>
</build>
```

> You may have noticed that there is no configuration here, but recall from *Configuring Reports and Checks* section of this chapter that the reporting configuration is applied to the build as well.

By default, the `pmd:check` goal is run in the verify phase, which occurs after the packaging phase. If you need to run checks earlier, you could add the following to the execution block to ensure that the check runs just after all sources exist:

```
<phase>process-sources</phase>
```

To test this new setting, try running `mvn verify` in the `proficio-core` directory. You will see that the build fails. To correct this, fix the errors in the `src/main/java/com/mergere/mvnbook/proficio/DefaultProficio.java` file by adding a `//NOPMD` comment to the unused variables and method:

```
...
// Trigger PMD and checkstyle
int i; // NOPMD
...
int j; // NOPMD
...
private void testMethod() // NOPMD
{
}
...
```

If you run `mvn verify` again, the build will succeed.

While this check is very useful, it can be slow and obtrusive during general development. For that reason, adding the check to a profile, which is executed only in an appropriate environment, can make the check optional for developers, but mandatory in an integration environment. See *Continuous Integration with Continuum* section in the next chapter for information on using profiles and continuous integration.

While the PMD report allows you to run a number of different rules, there is one that is in a separate report. This is the CPD, or `copy/paste detection` report, and it includes a list of duplicate code fragments discovered across your entire source base. An example report is shown in figure 6-8.This report is included by default when you enable the PMD plugin in your reporting section, and will appear as “CPD report” in the `Project Reports` menu.

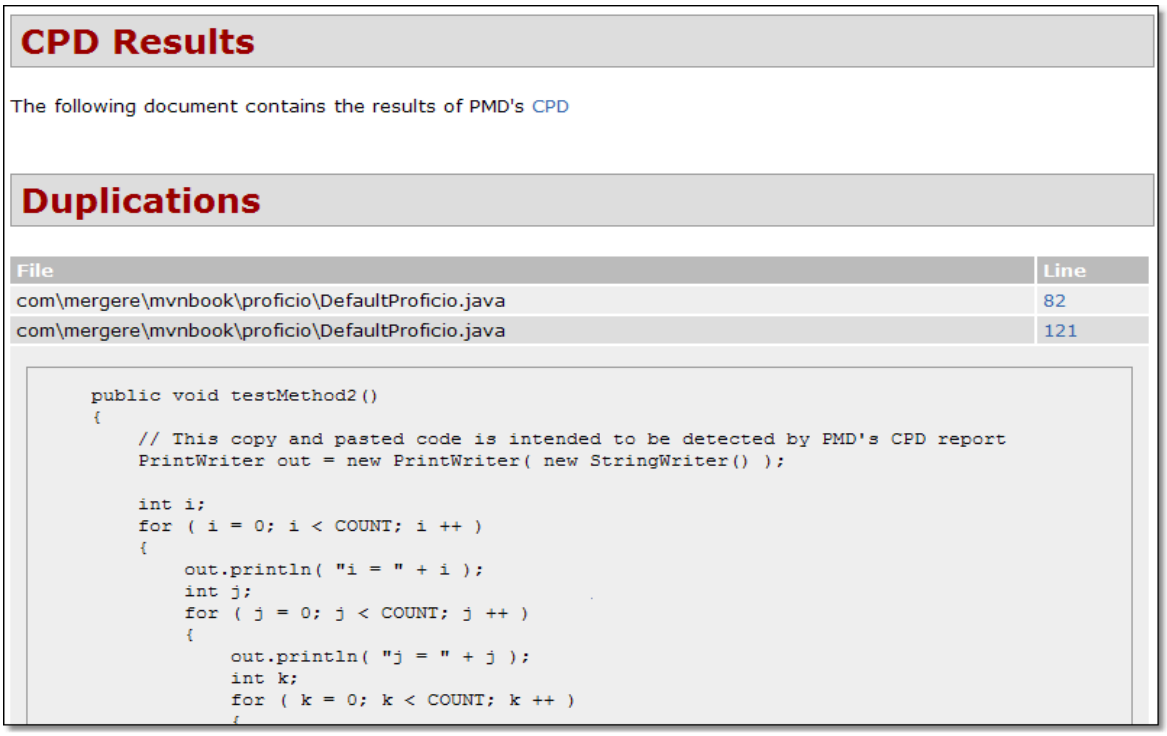


Figure 6-8: An example CPD report

In a similar way to the main check, `pmd:cpd-check` can be used to enforce a failure if duplicate source code is found. However, the CPD report contains only one variable to configure: `minimumTokenCount`, which defaults to 100. With this setting you can fine tune the size of the copies detected. This may not give you enough control to effectively set a rule for the source code, resulting in developers attempting to avoid detection by making only slight modifications, rather than identifying a possible factoring of the source code. Whether to use the report only, or to enforce a check will depend on the environment in which you are working.

There are other alternatives for copy and paste detection, such as Checkstyle, and a commercial product called Simian (http://www.redhillconsulting.com.au/products/simian/). Simian can also be used through Checkstyle and has a larger variety of configuration options for detecting duplicate source code.

Checkstyle is a tool that is, in many ways, similar to PMD. It was originally designed to address issues of format and style, but has more recently added checks for other code issues.

Depending on your environment, you may choose to use it in one of the following ways:

- Use it to check code formatting only, and rely on other tools for detecting other problems.
- Use it to check code formatting and selected other problems, and still rely on other tools for greater coverage.
- Use it to check code formatting and to detect other problems exclusively

This section focuses on the first usage scenario. If you need to learn more about the available modules in Checkstyle, refer to the list on the Web site at http://checkstyle.sf.net/availablechecks.html.

Figure 6-9 shows the Checkstyle report obtained by running `mvn checkstyle:checkstyle` from the `proficio-core` directory. Some of the extra summary information for overall number of errors and the list of checks used has been trimmed from this display.

Summary

Files	Infos	Warnings	Errors
2	0	0	107

Files

Files	I	W	E
com/mergere/mvnbook/proficio/DefaultProficio.java	0	0	107

Details

com/mergere/mvnbook/proficio/DefaultProficio.java

Violation	Message	Line
	Line has trailing spaces.	30
	'{' should be on the previous line.	30
	Missing a Javadoc comment.	37
	Method 'addFaqEntry' is not designed for extension - needs to be abstract, final or empty.	43
	'(' is followed by whitespace.	43
	Parameter entry should be final.	43

Figure 6-9: An example Checkstyle report

You'll see that each file with notices, warnings or errors are listed in a summary, and then the errors are shown, with a link to the corresponding source line – if the JXR report was enabled.

That's a lot of errors! By default, the rules used are those of the Sun Java coding conventions, but Proficio is using the Maven team's code style.

This style is also bundled with the Checkstyle plugin, so to include the report in the site and configure it to use the Maven style, add the following to the reporting section of `proficio/pom.xml`:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-checkstyle-plugin</artifactId>
 <configuration>
 <configLocation>config/maven_checks.xml</configLocation>
 </configuration>
</plugin>
```

Table 6-3 shows the configurations that are built into the Checkstyle plugin.

Table 6-3: Built-in Checkstyle configurations

Configuration	Description	Reference
`config/sun_checks.xml`	Sun Java Coding Conventions	http://java.sun.com/docs/codeconv/
`config/maven_checks.xml`	Maven team's coding conventions	http://maven.apache.org/guides/development/guide-m2-development.html#Maven%20Code%20Style
`config/turbine_checks.xml`	Conventions from the Jakarta Turbine project	http://jakarta.apache.org/turbine/common/code-standards.html
`config/avalon_checks.xml`	Conventions from the Apache Avalon project	No longer online – the Avalon project has closed. These checks are for backwards compatibility only.

The `configLocation` parameter can be set to a file within your build, a URL, or a resource within a special dependency also.

It is a good idea to reuse an existing Checkstyle configuration for your project if possible – if the style you use is common, then it is likely to be more readable and easily learned by people joining your project. The built-in Sun and Maven standards are quite different, and typically, one or the other will be suitable for most people. However, if you have developed a standard that differs from these, or would like to use the additional checks introduced in Checkstyle 3.0 and above, you will need to create a Checkstyle configuration.

While this chapter will not go into an example of how to do this, the Checkstyle documentation provides an excellent reference at http://checkstyle.sf.net/config.html.

The Checkstyle plugin itself has a large number of configuration options that allow you to customize the appearance of the report, filter the results, and to parameterize the Checkstyle configuration for creating a baseline organizational standard that can be customized by individual projects. It is also possible to share a Checkstyle configuration among multiple projects, as explained at http://maven.apache.org/plugins/maven-checkstyle-plugin/tips.html.

Before completing this section it is worth mentioning the Tag List plugin. This report, known as “Task List” in Maven 1.0, will look through your source code for known tags and provide a report on those it finds. By default, this will identify the tags TODO and `@todo` in the comments of your source code.

To try this plugin, add the following to the reporting section of `proficio/pom.xml`:

```
...
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>taglist-maven-plugin</artifactId>
 <configuration>
 <tags>
  <tag>TODO</tag>
  <tag>@todo</tag>
  <tag>FIXME</tag>
  <tag>XXX</tag>
 </tags>
 </configuration>
</plugin>
...
```

This configuration will locate any instances of `TODO`, `@todo`, `FIXME`, or `XXX` in your source code. It is actually possible to achieve this using Checkstyle or PMD rules, however this plugin is a more convenient way to get a simple report of items that need to be addressed at some point later in time.

PMD, Checkstyle, and Tag List are just three of the many tools available for assessing the health of your project's source code. Some other similar tools, such as `FindBugs`, `JavaNCSS` and `JDepend`, have beta versions of plugins available from the http://mojo.codehaus.org/ project at the time of this writing, and more plugins are being added every day.

6.8. Monitoring and Improving the Health of Your Tests

One of the important (and often controversial) features of Maven is the emphasis on testing as part of the production of your code. In the build life cycle defined in Chapter 2, you saw that tests are run *before* the packaging of the library or application for distribution, based on the theory that you shouldn't even try to use something before it has been tested. There are additional testing stages that can occur after the packaging step to verify that the assembled package works under other circumstances.

As you learned in section 6.2, *Setting Up the Project Web Site*, it is easy to add a report to the Web site that shows the results of the tests that have been run. While the default Surefire configuration fails the build if the tests fail, the report (run either on its own, or as part of the site), will ignore these failures when generated to show the current test state. Failing the build is still recommended – but the report allows you to provide a better visual representation of the results. In addition to that, it can be a useful report for demonstrating the number of tests available and the time it takes to run certain tests for a package.

Knowing whether your tests pass is an obvious and important assessment of their health. Another critical technique is to determine how much of your source code is covered by the test execution. At the time of writing, for assessing coverage, Cobertura (http://cobertura.sf.net) is the open source tool best integrated with Maven. While you are writing your tests, using this report on a regular basis can be very helpful in spotting any holes in the test plan.

To see what Cobertura is able to report, run `mvn cobertura:cobertura` in the `proficio-core` directory of the sample application. Figure 6-10 shows the output that you can view in `target/site/cobertura/index.html`.

The report contains both an overall summary, and a line-by-line coverage analysis of each source file, in the familiar Javadoc style framed layout. For a source file, you'll notice the following markings:

- Unmarked lines are those that do not have any executable code associated with them. This includes method and class declarations, comments and white space.
- Each line with an executable statement has a number in the second column that indicates during the test run how many times a particular statement was run.
- Lines in red are statements that were not executed (if the count is 0), or for which all possible branches were not executed. For example, a branch is an if statement that can behave differently depending on whether the condition is true or false.

Unmarked lines with a green number in the second column are those that have been completely covered by the test execution.

Figure 6-10: An example Cobertura report

The complexity indicated in the top right is the cyclomatic complexity of the methods in the class, which measures the number of branches that occur in a particular method. High numbers (for example, over 10), might indicate a method should be re-factored into simpler pieces, as it can be hard to visualize and test the large number of alternate code paths. If this is a metric of interest, you might consider having PMD monitor it.

The Cobertura report doesn't have any notable configuration, so including it in the site is simple. Add the following to the reporting section of `proficio/pom.xml`:

```
...
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>cobertura-maven-plugin</artifactId>
</plugin>
...
```

If you now run `mvn` site under `proficio-core`, the report will be generated in `target/site/cobertura/index.html`.

While not required, there is another useful setting to add to the *build* section. Due to a hard-coded path in Cobertura, the database used is stored in the project directory as `cobertura.ser`, and is not cleaned with the rest of the project. To ensure that this happens, add the following to the build section of `proficio/pom.xml`:

```
...
<build>
 <plugins>
 <plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>cobertura-maven-plugin</artifactId>
  <executions>
  <execution>
   <id>clean</id>
   <goals>
   <goal>clean</goal>
   </goals>
  </execution>
  </executions>
 </plugin>
 </plugins>
</build>
...
```

If you now run `mvn` clean in `proficio`-core, you'll see that the `cobertura.ser` file is deleted, as well as the target directory.

The Cobertura plugin also contains a goal called `cobertura:check` that is used to ensure that the coverage of your source code is maintained at a certain percentage.

To configure this goal for Proficio, add a configuration and another execution to the build plugin definition you added above when cleaning the Cobertura database:

```
...
<configuration>
 <check>
 <totalLineRate>100</totalLineRate>
 <totalBranchRate>100</totalBranchRate>
 </check>
</configuration>
<executions>
...
 <execution>
 <id>check</id>
 <goals>
  <goal>check</goal>
 </goals>
 </execution>
</executions>
...
```

Note that the configuration element is *outside* of the executions. This ensures that if you run `mvn cobertura:check` from the command line, the configuration will be applied. This wouldn't be the case if it were associated with the life-cycle bound check execution.

If you now run `mvn` verify under `proficio-core`, the check will be performed.

> You'll notice that your tests are run twice. This is because Cobertura needs to instrument your class files, and the tests are re-run using those class files instead of the normal ones (however, these are instrumented in a separate directory, so are not packaged in your application). The Surefire report may also re-run tests if they were already run – both of these are due to a limitation in the way the life cycle is constructed that will be improved in future versions of Maven.

The rules that are being used in this configuration are 100% overall line coverage rate, and 100% branch coverage rate. You would have seen in the previous examples that there were some lines not covered, so running the check fails.

Normally, you would add unit tests for the functions that are missing tests, as in the Proficio example. However, looking through the report, you may decide that only some exceptional cases are untested, and decide to reduce the overall average required. You can do this for Proficio to have the tests pass by changing the setting in `proficio/pom.xml`:

```
...
<configuration>
 <check>
 <totalLineRate>80</totalLineRate>
...
```

If you run `mvn` verify again, the check passes.

These settings remain quite demanding though, only allowing a small number of lines to be untested. This will allow for some constructs to remain untested, such as handling checked exceptions that are unexpected in a properly configured system and difficult to test. It is just as important to allow these exceptions, as it is to require that the other code be tested. Remember, the easiest way to increase coverage is to remove code that handles untested, exceptional cases – and that's certainly not something you want!

The settings above are requirements for averages across the entire source tree. You may want to enforce this for each file individually as well, using `lineRate` and `branchRate`, or as the average across each package, using `packageLineRate` and `packageBranchRate`. It is also possible to set requirements on individual packages or classes using the `regexes` parameter. For more information, refer to the Cobertura plugin configuration reference at http://mojo.codehaus.org/cobertura-maven-plugin.

Choosing appropriate settings is the most difficult part of configuring any of the reporting metrics in Maven. Some helpful hints for determining the right code coverage settings are:

- Like all metrics, involve the whole development team in the decision, so that they understand and agree with the choice.
- Don't set it too low, as it will become a minimum benchmark to attain and rarely more.
- Don't set it too high, as it will discourage writing code to handle exceptional cases that aren't being tested.
- Set some known guidelines for what type of code can remain untested.
- Consider setting any package rates higher than the per-class rate, and setting the total rate higher than both.
- Remain flexible – consider changes over time rather than hard and fast rules. Choose to reduce coverage requirements on particular classes or packages rather than lowering them globally.

Cobertura is not the only solution available for assessing test coverage. The best known commercial offering is Clover, which is very well integrated with Maven as well. It behaves very similarly to Cobertura, and you can evaluate it for 30 days when used in conjunction with Maven. For more information, see the Clover plugin reference on the Maven Web site at http://maven.apache.org/plugins/maven-clover-plugin/.

Of course, there is more to assessing the health of tests than success and coverage. These reports won't tell you if all the features have been implemented – this requires functional or acceptance testing. It also won't tell you whether the results of untested input values produce the correct results. Tools like Jester (http://jester.sf.net), although not yet integrated with Maven directly, may be of assistance there. Jester mutates the code that you've already determined is covered and checks that it causes the test to fail when run a second time with the wrong code.

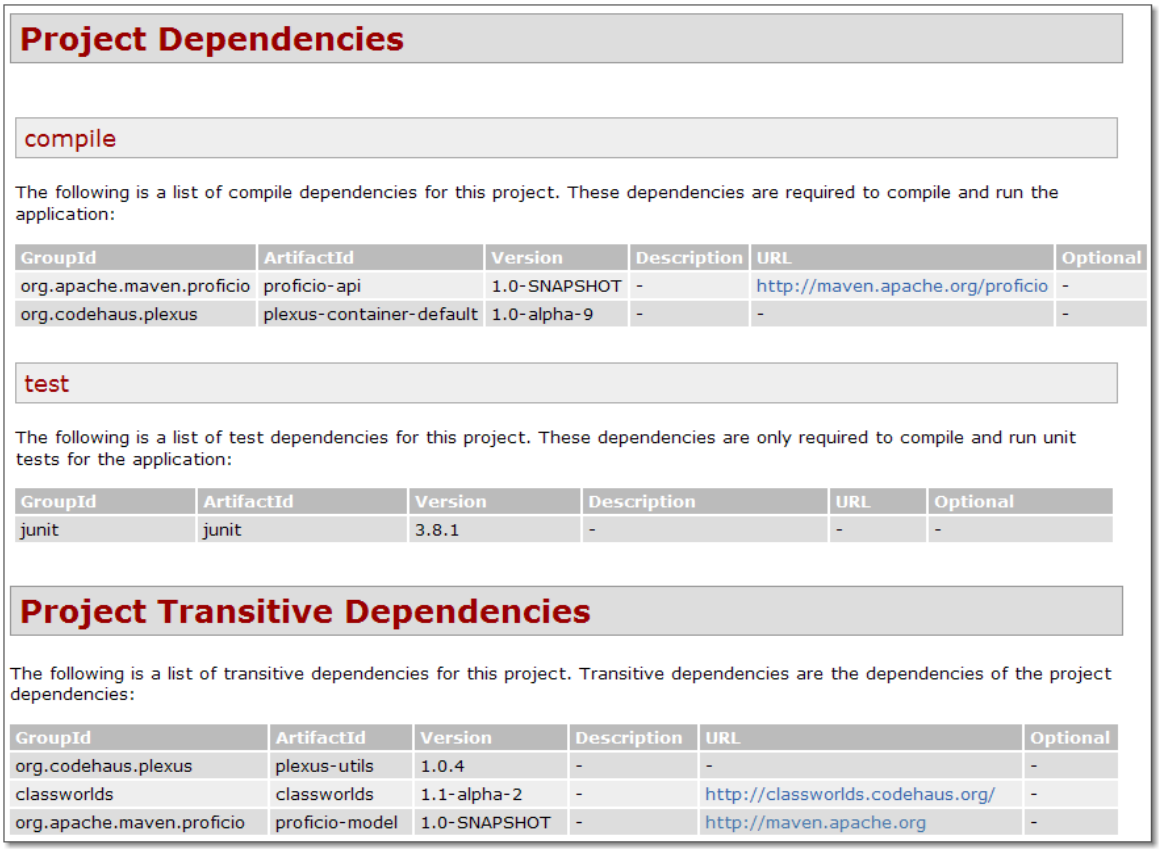
To conclude this section on testing, it is worth noting that one of the benefits of Maven's use of the Surefire abstraction is that the tools above will work for any type of runner introduced. For example, Surefire supports tests written with TestNG, and at the time of writing experimental JUnit 4.0 support is also available. In both cases, these reports work unmodified with those test types. If you have another tool that can operate under the Surefire framework, it is possible for you to write a provider to use the new tool, and get integration with these other tools for free.

6.9. Monitoring and Improving the Health of Your Dependencies

Many people use Maven primarily as a dependency manager. While this is only one of Maven's features, used well it is a significant time saver.

Maven 2.0 introduced transitive dependencies, where the dependencies of dependencies are included in a build, and a number of other features such as scoping and version selection. This brought much more power to Maven's dependency mechanism, but does introduce a drawback: poor dependency maintenance or poor scope and version selection affects not only your own project, but any projects that depend on your project. Left unchecked, the full graph of a project's dependencies can quickly balloon in size and start to introduce conflicts.

The first step to effectively maintaining your dependencies is to review the standard report included with the Maven site. If you haven't done so already, run `mvn site` in the `proficio-core` directory, and browse to the file generated in `target/site/dependencies.html`. The result is shown in figure 6-11.



Project Dependencies

compile

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

GroupId	ArtifactId	Version	Description	URL	Optional
org.apache.maven.proficio	proficio-api	1.0-SNAPSHOT	-	http://maven.apache.org/proficio	-
org.codehaus.plexus	plexus-container-default	1.0-alpha-9	-	-	-

test

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

GroupId	ArtifactId	Version	Description	URL	Optional
junit	junit	3.8.1	-	-	-

Project Transitive Dependencies

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

GroupId	ArtifactId	Version	Description	URL	Optional
org.codehaus.plexus	plexus-utils	1.0.4	-	-	-
classworlds	classworlds	1.1-alpha-2	-	http://classworlds.codehaus.org/	-
org.apache.maven.proficio	proficio-model	1.0-SNAPSHOT	-	http://maven.apache.org	-

Figure 6-11: An example dependency report

This report shows detailed information about your direct dependencies, but more importantly in the second section it will list all of the transitive dependencies included through those dependencies. It's here that you might see something that you didn't expect – an extra dependency, an incorrect version, or an incorrect scope – and choose to investigate its inclusion.

Currently, this requires running your build with debug turned on, such as `mvn -X package`. This will output the dependency tree as it is calculated, using indentation to indicate which dependencies introduce other dependencies, as well as comments about what versions and scopes are selected, and why. For example, here is the resolution process of the dependencies of `proficio-core` (some fields have been omitted for brevity):

```
proficio-core:1.0-SNAPSHOT
 junit:3.8.1 (selected for test)
 plexus-container-default:1.0-alpha-9 (selected for compile)
  plexus-utils:1.0.4 (selected for compile)
  classworlds:1.1-alpha-2 (selected for compile)
  junit:3.8.1 (not setting scope to: compile; local scope test wins)
 proficio-api:1.0-SNAPSHOT (selected for compile)
  proficio-model:1.0-SNAPSHOT (selected for compile)
```

Here you can see that, for example, `proficio-model` is introduced by `proficio-api`, and that plexus-container-default attempts to introduce `junit` as a compile dependency, but that it is overridden by the test scoped dependency in `proficio-core`.

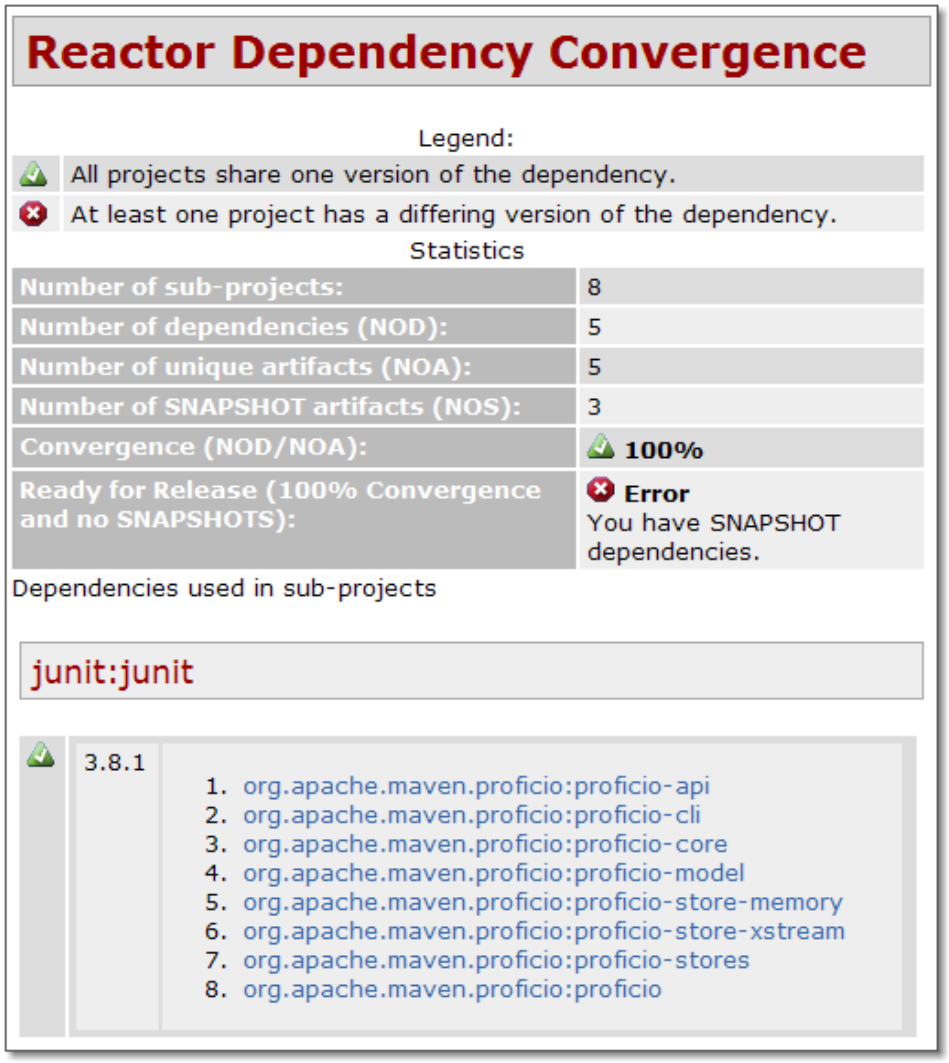
This can be quite difficult to read, so at the time of this writing there are two features in progress that are aimed at helping in this area:

- The Maven Repository Manager will allow you to navigate the dependency tree through the metadata stored in the `Ibiblio` repository.
- A dependency graphing plugin that will render a graphical representation of the information.

Another report that is available is the “Dependency Convergence Report”. This report is also a standard report, but appears in a multi-module build only. To see the report for the Proficio project, run `mvn` site from the base `proficio` directory. The file `target/site/dependency-convergence.html` will be created, and is shown in figure 6-12.

The report shows all of the dependencies included in all of the modules within the project. It also includes some statistics and reports on two important factors:

- Whether the versions of dependencies used for each module is in alignment. This helps ensure your build is consistent and reduces the probability of introducing an accidental incompatibility.
- Whether there are outstanding SNAPSHOT dependencies in the build, which indicates dependencies that are in development, and must be updated before the project can be released.



Reactor Dependency Convergence

Legend:

All projects share one version of the dependency.

At least one project has a differing version of the dependency.

Statistics

Number of sub-projects:	8
Number of dependencies (NOD):	5
Number of unique artifacts (NOA):	5
Number of SNAPSHOT artifacts (NOS):	3
Convergence (NOD/NOA):	100%
Ready for Release (100% Convergence and no SNAPSHOTS):	Error You have SNAPSHOT dependencies.

Dependencies used in sub-projects

junit:junit

3.8.1

1. org.apache.maven.proficio:proficio-api
2. org.apache.maven.proficio:proficio-cli
3. org.apache.maven.proficio:proficio-core
4. org.apache.maven.proficio:proficio-model
5. org.apache.maven.proficio:proficio-store-memory
6. org.apache.maven.proficio:proficio-store-xstream
7. org.apache.maven.proficio:proficio-stores
8. org.apache.maven.proficio:proficio

Figure 6-12: The dependency convergence report

These reports are passive – there are no associated checks for them. However, they can provide basic help in identifying the state of your dependencies once you know what to find. To improve your project's health and the ability to reuse it as a dependency itself, try the following recommendations for your dependencies:

- Look for dependencies in your project that are no longer used
- Check that the scope of your dependencies are set correctly (to test if only used for unit tests, or runtime if it is needed to bundle with or run the application but not for compiling your source code).
- Use a range of supported dependency versions, declaring the absolute minimum supported as the lower boundary, rather than using the latest available. You can control what version is actually used by declaring the dependency version in a project that packages or runs the application.

- Add exclusions to dependencies to remove poorly defined dependencies from the tree. This is particularly the case for dependencies that are optional and unused by your project.

Given the importance of this task, more tools are needed in Maven. Two that are in progress were listed above, but there are plans for more:

- A class analysis plugin that helps identify dependencies that are unused in your current project
- Improved dependency management features including different mechanisms for selecting versions that will allow you to deal with conflicting versions, specification dependencies that let you depend on an API and manage the implementation at runtime, and more.

6.10. Monitoring and Improving the Health of Your Releases

Releasing a project is one of the most important procedures you will perform, but it is often tedious and error prone. While the next chapter will go into more detail about how Maven can help automate that task and make it more reliable, this section will focus on improving the quality of the code released, and the information released with it.

An important tool in determining whether a project is ready to be released is Clirr (http://clirr.sf.net/). Clirr detects whether the current version of a library has introduced any binary incompatibilities with the previous release. Catching these before a release can eliminate problems that are quite difficult to resolve once the code is “in the wild”. An example Clirr report is shown in figure 6-13.

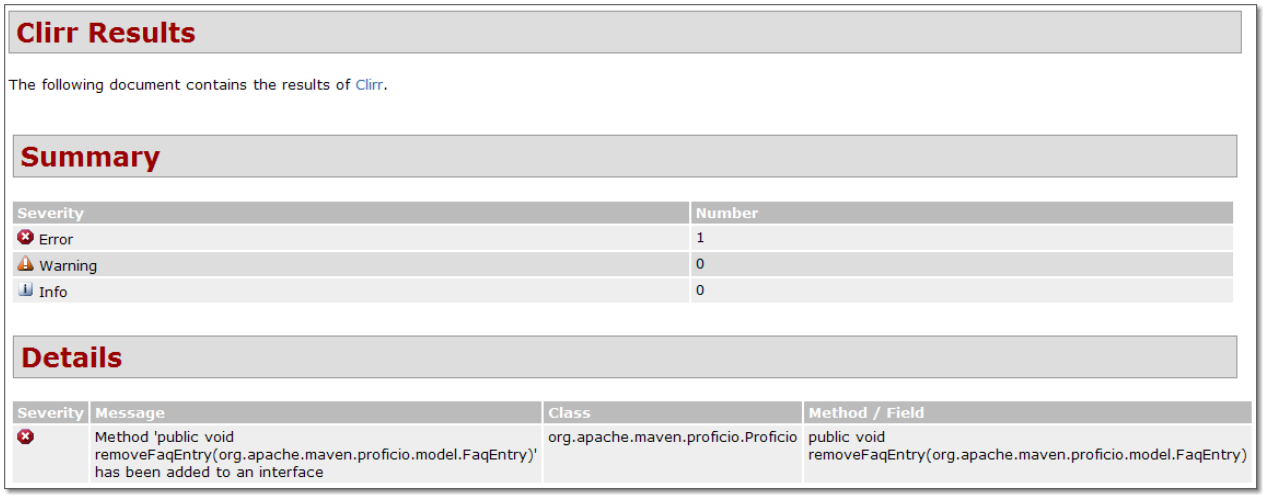


Figure 6-13: An example Clirr report

This is particularly important if you are building a library or framework that will be consumed by developers outside of your own project.

Libraries will often be substituted by newer versions to obtain new features or bug fixes, but then expected to continue working as they always have.

Because existing libraries are not recompiled every time a version is changed, there is no verification that a library is binary-compatible – incompatibility will be discovered only when there's a failure.

But does binary compatibility apply if you are not developing a library for external consumption? While it may be of less importance, the answer here is clearly – yes. As a project grows, the interactions between the project's own components will start behaving as if they were externally-linked. Different modules may use different versions, or a quick patch may need to be made and a new version deployed into an existing application.

This is particularly true in a Maven-based environment, where the dependency mechanism is based on the assumption of binary compatibility between versions. While methods of marking incompatibility are planned for future versions, Maven currently works best if any version of an artifact is backwards compatible, back to the first release.

By default, the Clirr report shows only errors and warnings. However, you can configure the plugin to show all informational messages, by setting the `minSeverity` parameter. This gives you an overview of all the changes since the last release, even if they are binary compatible. To see this in action, add the following to the reporting section of `proficio-api/pom.xml`:

```
...
<reporting>
 <plugins>
 <plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>clirr-maven-plugin</artifactId>
  <configuration>
  <minSeverity>info</minSeverity>
  </configuration>
 </plugin>
 </plugins>
</reporting>
...
```

If you run `mvn site` in `proficio-api`, the report will be generated in `target/site/clirr-report.html`. You can obtain the same result by running the report on its own using `mvn clirr:clirr`.

If you run either of these commands, you'll notice that Maven reports that it is using version 0.9 of `proficio-api` against which to compare (and that it is downloaded if you don't have it already):

```
...
[INFO] [clirr:clirr]
[INFO] Comparing to version: 0.9
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
...
```

This version is determined by looking for the newest release in repository, that is before the current development version.

You can change the version used with the `comparisonVersion` parameter. For example, to compare the current code to the 0.8 release, run the following command:

```
mvn clirr:clirr -DcomparisonVersion=0.8
```

These versions of `proficio-api` are retrieved from the repository, however you can see the original sources by extracting the `Codeh_Ch06-2.zip` file.

You'll notice there are a more errors in the report, since this early development version had a different API, and later was redesigned to make sure that version 1.0 would be more stable in the long run.

It is best to make changes earlier in the development cycle, so that fewer people are affected. The longer poor choices remain, the harder they are to change as adoption increases. Once a version has been released that is intended to remain binary-compatible going forward, it is almost always preferable to deprecate an old API and add a new one, delegating the code, rather than removing or changing the original API and breaking binary compatibility.

In this instance, you are monitoring the `proficio-api` component for binary compatibility changes only. This is the most important one to check, as it will be used as the interface into the implementation by other applications. If it is the only one that the development team will worry about breaking, then there is no point in checking the others – it will create noise that devalues the report's content in relation to the important components.

However, if the team is prepared to do so, it is a good idea to monitor as many components as possible. Even if they are designed only for use inside the project, there is nothing in Java preventing them from being used elsewhere, and it can assist in making your own project more stable.

Like all of the quality metrics, it is important to agree up front, on the acceptable incompatibilities, to discuss and document the practices that will be used, and to check them automatically. The Clirr plugin is also capable of automatically checking for introduced incompatibilities through the `clirr:check` goal.

To add the check to the `proficio-api/pom.xml` file, add the following to the build section:

```
...
<build>
 <plugins>
 <plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>clirr-maven-plugin</artifactId>
  <executions>
  <execution>
   <goals>
   <goal>check</goal>
   </goals>
  </execution>
  </executions>
 </plugin>
...
 </plugins>
</build>
...
```

If you now run `mvn verify`, you will see that the build fails due to the binary incompatibility introduced between the 0.9 preview release and the final 1.0 version. Since this was an acceptable incompatibility due to the preview nature of the 0.9 release, you can choose to exclude that from the report by adding the following configuration to the plugin:

```
...
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>clirr-maven-plugin</artifactId>
 <configuration>
 <excludes>
  <exclude>**/Proficio</exclude>
 </excludes>
 </configuration>
...
</plugin>
```

This will prevent failures in the Proficio class from breaking the build in the future. Note that in this instance, it is listed only in the *build* configuration, so the report still lists the incompatibility. This allows the results to be collected over time to form documentation about known incompatibilities for applications using the library.

A limitation of this feature is that it will eliminate a class entirely, not just the one acceptable failure. Hopefully a future version of Clirr will allow acceptable incompatibilities to be documented in the source code, and ignored in the same way that PMD does.

With this simple setup, you can create a very useful mechanism for identifying potential release disasters much earlier in the development process, and then act accordingly. While the topic of designing a strong public API and maintaining binary compatibility is beyond the scope of this book, the following articles and books can be recommended:

- Evolving Java-based APIs contains a description of the problem of maintaining binary compatibility, as well as strategies for evolving an API without breaking it.
- Effective Java describes a number of practical rules that are generally helpful to writing code in Java, and particularly so if you are designing a public API.

A similar tool to Clirr that can be used for analyzing changes between releases is JDiff. Built as a Javadoc doclet, it takes a very different approach, taking two source trees and comparing the differences in method signatures and Javadoc annotations. This can be useful in getting a greater level of detail than Clirr on specific class changes. However, it will not pinpoint potential problems for you, and so is most useful for browsing. It has a functional Maven 2 plugin, which is available at http://mojo.codehaus.org/jdiff-maven-plugin.

6.11. Viewing Overall Project Health

In the previous sections of this chapter, a large amount of information was presented about a project, each in discrete reports. Some of the reports linked to one another, but none related information from another report to itself, and few of the reports aggregated information across a multiple module build. Finally, none of the reports presented how the information changes over time other than the release announcements. These are all important features to have to get an overall view of the health of a project. While some attempts were made to address this in Maven 1.0 (for example, the Dashboard plugin), they did not address all of these requirements, and have not yet been implemented for Maven 2.0.

However, it should be noted that the Maven reporting API was written with these requirements in mind specifically, and as the report set stabilizes – summary reports will start to appear.

In the absence of these reports, enforcing good, individual checks that fail the build when they're not met, will reduce the need to gather information from various sources about the health of the project, as there is a constant background monitor that ensures the health of the project is being maintained.

6.12. Summary

The power of Maven's declarative project model is that with a very simple setup (often only 4 lines in `pom.xml`), a new set of information about your project can be added to a shared Web site to help your team visualize the health of the project. Best of all, the model remains flexible enough to make it easy to extend and customize the information published on your project web site.

However, it is important that your project information not remain passive. Most Maven plugins allow you to integrate rules into the build that check certain constraints on that piece information once it is well understood. The purpose, then, of the visual display is to aid in deriving the appropriate constraints to use.

How well this works in your own projects will depend on the development culture of your team. It is important that developers are involved in the decision making process regarding build constraints, so that they feel that they are achievable. In some cases, it requires a shift from a focus on time and deadlines, to a focus on quality. Once established, this focus and automated monitoring will have the natural effect of improving productivity and reducing time of delivery again.

The additions and changes to Proficio made in this chapter can be found in the `Code_Ch06-1.zip` source archive, and will be used as the basis for the next chapter.

The next chapter examines team development and collaboration, and incorporates the concepts learned in this chapter, along with techniques to ensure that the build checks are now automated, regularly scheduled, and run in the appropriate environment.

Team Collaboration with Maven

This chapter covers:

- How Maven helps teams
- How to set up consistent developer environments
- How to set up a shared internal Maven repository
- Continuous Integration
- Creating shared organization metadata and archetypes
- Releasing a project

> Collaboration on a book is the ultimate unnatural act.
>
> *- Tom Clancy*

7.1. The Issues Facing Teams

Software development as part of a team, whether it is 2 people or 200 people, faces a number of challenges to the success of the effort. Many of these challenges are out of any given technology's control – for instance finding the right people for the team, and dealing with differences in opinions.

However, one of the biggest challenges relates to the sharing and management of development information. While it's essential that team members receive all of the project information required to be productive, it's just as important that they don't waste valuable time researching and reading through too many information sources simply to find what they need.

This problem gets exponentially larger as the size of the team increases. As each member retains project information that isn't shared or commonly accessible, every other member (and particularly new members), will inevitably have to spend time obtaining this localized information, repeating errors previously solved or duplicating efforts already made. Even when it is not localized, project information can still be misplaced, misinterpreted, or forgotten, further contributing to the problem.

As teams continue to grow, it is obvious that trying to publish and disseminate all of the available information about a project would create a near impossible learning curve and generate a barrier to productivity.

This problem is particularly relevant to those working as part of a team that is distributed across different physical locations and timezones. However, although a distributed team has a higher communication overhead than a team working in a single location, the key to the information issue in both situations is to reduce the amount of communication necessary to obtain the required information in the first place.

A *Community-oriented Real-time Engineering* (CoRE) process excels with this information challenge. CoRE is based on accumulated learnings from open source projects that have achieved successful, rapid development, working on complex, component-based projects despite large, widely-distributed teams. Using the model of a community, CoRE emphasizes the relationship between project information and project members.

An organizational and technology-based framework, CoRE enables globally distributed development teams to cohesively contribute to high-quality software, in rapid, iterative cycles. This value is delivered to development teams by supporting project transparency, real-time stakeholder participation, and asynchronous engineering, which is enabled by the accessibility of consistently structured and organized information such as centralized code repositories, web-based communication channels and web-based project management tools.

Even though teams may be widely distributed, the fact that everyone has direct access to the other team members through the CoRE framework reduces the time required to not only share information, but also to incorporate feedback, resulting in shortened development cycles. The CoRE approach to development also means that new team members are able to become productive quickly, and that existing team members become more productive and effective.

While Maven is not tied directly to the CoRE framework, it does encompass a set of practices and tools that enable effective team communication and collaboration. These tools aid the team to organize, visualize, and document for reuse the artifacts that result from a software project.

As described in Chapter 6, Maven can gather and share the knowledge about the health of a project. In this chapter, this is taken a step further, demonstrating how Maven provides teams with real-time information on the builds and health of a project, through the practice of continuous integration.

This chapter also looks at the adoption and use of a consistent development environment, and the use of archetypes to ensure consistency in the creation of new projects.

7.2. How to Setup a Consistent Developer Environment

Consistency is important when establishing a shared development environment. Without it, the set up process for a new developer can be slow, error-prone and full of omissions. Additionally, because the environment will tend to evolve inconsistently once started that way, it will be the source of time-consuming development problems in the future.

While one of Maven's objectives is to provide suitable conventions to reduce the introduction of inconsistencies in the build environment, there are unavoidable variables that remain, such as different installation locations for software, multiple JDK versions, varying operating systems, and other discrete settings such as user names and passwords.

To maintain build consistency, while still allowing for this natural variability, the key is to minimize the configuration required by each individual developer, and to effectively define and declare them. In Maven, these variables relate to the user and installation settings files, and to user-specific profiles.

In Chapter 2, you learned how to create your own `settings.xml` file. This file can be stored in the `conf` directory of your Maven installation, or in the `.m2` subdirectory of your home directory (settings in this location take precedence over those in the Maven installation directory). The `settings.xml` file contains a number of settings that are user-specific, but also several that are typically common across users in a shared environment, such as proxy settings.

In a shared development environment, it's a good idea to leverage Maven's two different settings files to separately manage shared and user-specific settings. Common configuration settings are included in the installation directory, while an individual developer's settings are stored in their home directory.

The following is an example configuration file that you might use in the installation directory, `<maven home>/conf/settings.xml`:

```
<settings>
 <proxies>
  <proxy>
   <active>true</active>
   <protocol>http</protocol>
   <host>proxy</host>
   <port>8080</port>
  </proxy>
 </proxies>
 <servers>
  <server>
   <id>website</id>
   <username>${website.username}</username>
    <filePermissions>664</filePermissions>
   <directoryPermissions>775</directoryPermissions>
  </server>
 </servers>
 <profiles>
  <profile>
   <repositories>
    <repository>
     <id>internal</id>
     <name>Internal Repository</name>
     <url>http://repo.mycompany.com/internal/</url>
    </repository>
   </repositories>
   <pluginRepositories>
    <pluginRepository>
     <id>internal</id>
     <name>Internal Plugin Repository</name>
     <url>http://repo.mycompany.com/internal/</url>
    </pluginRepository>
   </pluginRepositories>
  </profile>
 </profiles>
 <activeProfiles>
  <activeProfile>property-overrides</activeProfile>
  <activeProfile>default-repositories</activeProfile>
 </activeProfiles>
 <pluginGroups>
  <pluginGroup>com.mycompany.plugins</pluginGroup>
 </pluginGroups>
</settings>
```

There are a number of reasons to include these settings in a shared configuration:

- If a proxy server is allowed, it would usually be set consistently across the organization or department.
- The server settings will typically be common among a set of developers, with only specific properties such as the user name defined in the user's settings. By placing the common configuration in the shared settings, issues with inconsistently-defined identifiers and permissions are avoided.
- The mirror element can be used to specify a mirror of a repository that is closer to you, which is typically one that has been set up within your own organization or department. See section 7.3 of this chapter for more information on creating a mirror of the central repository within your own organization.
- The profile defines those common, internal repositories that contain a given organization's or department's released artifacts. These repositories are independent of the central repository in this configuration. See section 7.3 for more information on setting up an internal repository.
- The active profiles listed enable the profile defined previously in every environment. Another profile, property-overrides is also enabled by default. This profile will be defined in the user's settings file to set the properties used in the shared file, such as `${website.username}`.
- The plugin groups are necessary only if an organization has plugins, which are run from the command line and not defined in the POM.

You'll notice that the local repository is omitted in the prior example. While you may define a standard location that differs from Maven's default (for example, `${user.home}/maven-repo`), it is important that you do not configure this setting in a way that shares a local repository, at a single physical location, across users. In Maven, the local repository is defined as the repository of single user.

The previous example forms a basic template that is a good starting point for the settings file in the Maven installation. Using the basic template, you can easily add and consistently roll out any new server and repository settings, without having to worry about integrating local changes made by individual developers. The user-specific configuration is also much simpler as shown below:

```
<settings>
 <profiles>
  <profile>
   <id>property-overrides</id>
   <properties>
    <website.username>myuser</website.username>
   </properties>
  </profile>
 </profiles>
</settings>
```

To confirm that the settings are installed correctly, you can view the merged result by using the following help plugin command:

```
C:\mvnbook> mvn help:effective-settings
```

Separating the shared settings from the user-specific settings is helpful, but it is also important to ensure that that the shared settings are easily and reliably installed with Maven, and when possible, easily updated. The following are a few methods to achieve this:

- Rebuild the Maven release distribution to include the shared configuration file and distribute it internally. A new release will be required each time the configuration is changed.
- Place the Maven installation on a read-only shared or network drive from which each developer runs the application. If this infrastructure is available, each execution will immediately be up-to-date. However, doing so will prevent Maven from being available off-line, or if there are network problems.
- Check the Maven installation into CVS, Subversion, or other source control management (SCM) system. Each developer can check out the installation into their own machines and run it from there. Retrieving an update from an SCM will easily update the configuration and/or installation, but requires a manual procedure.
- Use an existing desktop management solution, or other custom solution.

If necessary, it is possible to maintain multiple Maven installations, by one of the following methods:

- Using the `M2_HOME` environment variable to force the use of a particular installation.
- Adjusting the path or creating symbolic links (or shortcuts) to the desired Maven executable, if `M2_HOME` is not set.

Configuring the `settings.xml` file covers the majority of use cases for individual developer customization, however it applies to all projects that are built in the developer's environment. In some circumstances however, an individual will need to customize the build of an individual project. To do this, developers must use profiles in the `profiles.xml` file, located in the project directory. For more information on profiles, see Chapter 3.

Now that each individual developer on the team has a consistent set up that can be customized as needed, the next step is to establish a repository to and from which artifacts can be published and dependencies downloaded, so that multiple developers and teams can collaborate effectively.

7.3. Creating a Shared Repository

Most organizations will need to set up one or more shared repositories, since not everyone can deploy to the central Maven repository. To publish releases for use across different environments within their network, organization's will typically want to set up what is referred to as an *internal repository*. This internal repository is still treated as a *remote repository* in Maven, just as any other external repository would be. For an explanation of the different types of repositories, see Chapter 2.

Setting up an internal repository is simple. While any of the available transport protocols can be used, the most popular is HTTP. You can use an existing HTTP server for this, or create a new server using Apache HTTPd, Apache Tomcat, Jetty, or any number of other servers.

To set up your organization's internal repository using Jetty, create a new directory in which to store the files. While it can be stored anywhere you have permissions, in this example `C:\mvnbook\repository` will be used. To set up Jetty, download the Jetty 5.1.10 server bundle from the book's Web site and copy it to the repository directory. Change to that directory, and run:

```
C:\mvnbook\repository> java -jar jetty-5.1.10-bundle.jar 8081
```

You can now navigate to `http://localhost:8081/` and find that there is a web server running displaying that directory. Your repository is now set up.

The server is set up on your own workstation for simplicity in this example. However, you will want to set up or use an existing HTTP server that is in a shared, accessible location, configured securely and monitored to ensure it remains running at all times.

This chapter will assume the repositories are running from `http://localhost:8081/` and that artifacts are deployed to the repositories using the file system. However, it is possible to user a repository on another server with any combination of supported protocols including `http`, `ftp`, `scp`, `sftp` and more. For more information, refer to Chapter 3.

Later in this chapter you will learn that there are good reasons to run multiple, separate repositories, but rather than set up multiple web servers, you can store the repositories on this single server. For the first repository, create a subdirectory called internal that will be available at `http://localhost:8081/internal/`.

This creates an empty repository, and is all that is needed to get started.

```
C:\mvnbook\repository> mkdir internal
```

It is also possible to set up another repository (or use the same one) to mirror content from the Maven central repository. The setting for using this was shown in the mirror element of the settings file in section 7.2. While this isn't required, it is common in many organizations as it eliminates the requirement for Internet access or proxy configuration. In addition, it provides faster performance (as most downloads to individual developers come from within their own network), and gives full control over the set of artifacts with which your software is built, by avoiding any reliance on Maven's relatively open central repository.

You can create a separate repository under the same server, using the following command:

```
C:\mvnbook\repository> mkdir central
```

This repository will be available at `http://localhost:8081/central/`.

To populate the repository you just created, there are a number of methods available:

- Manually add content as desired using `mvn` deploy:deploy-file
- Set up the Maven Repository Manager as a proxy to the central repository. This will download anything that is not already present, and keep a copy in your internal repository for others on your team to reuse.
- Use `rsync` to take a copy of the central repository and regularly update it. At the time of writing, the size of the Maven repository was 5.8G.

The *Maven Repository Manager* (MRM) is a new addition to the Maven build platform that is designed to administer your internal repository. It is deployed to your Jetty server (or any other servlet container) and provides remote repository proxies, as well as friendly repository browsing, searching, and reporting. The repository manager can be downloaded from http://maven.apache.org/repository-manager/.

When using this repository for your projects, there are two choices: use it as a mirror, or have it override the central repository. You would use it as a mirror if it is intended to be a copy of the central repository exclusively, and if it's acceptable to have developers configure this in their settings as demonstrated in section 7.2. Developers may choose to use a different mirror, or the original central repository directly without consequence to the outcome of the build.

On the other hand, if you want to prevent access to the central repository for greater control, to configure the repository from the project level instead of in each user's settings (with one exception that will be discussed next), or to include your own artifacts in the same repository, you should override the central repository.

To override the central repository with your internal repository, you must define a repository in a settings file and/or POM that uses the identifier central. Usually, this must be defined as both a regular repository and a plugin repository to ensure all access is consistent. For example:

```
<repositories>
 <repository>
  <id>central</id>
  <name>Internal Mirror of Central Repository</name>
  <url>http://localhost:8081/central/</url>
 </repository>
</repositories>
<pluginRepositories>
 <pluginRepository>
  <id>central</id>
  <name>Internal Mirror of Central Plugins Repository</name>
  <url>http://localhost:8081/central/</url>
 </pluginRepository>
</pluginRepositories>
```

You should not enter this into your project now, unless you have mirrored the central repository using one the techniques discussed previously, otherwise Maven will fail to download any dependencies that are not in your local repository.

Repositories such as the one above are configured in the POM usually, so that a project can add repositories itself for dependencies located out of those repositories configured initially. However, there is a problem – when a POM inherits from another POM that is not in the central repository, it must retrieve the parent from the repository. This makes it impossible to define the repository in the parent, and as a result, it would need to be declared in every POM. Not only is this very inconvenient, it would be a nightmare to change should the repository location change!

The solution is to declare your internal repository (or central replacement) in the shared `settings.xml` file, as shown in section 7.2. If you have multiple repositories, it is necessary to declare only those that contain an inherited POM.

It is still important to declare the repositories that will be used in the top-most POM itself, for a situation where a developer might not have configured their settings and instead manually installed the POM, or had it in their source code check out.

The next section discusses how to set up an “organization POM”, or hierarchy, that declares shared settings within an organization and its departments.

7.4. Creating an Organization POM

As previously mentioned in this chapter, consistency is important when setting up your build infrastructure. By declaring shared elements in a common parent POM, project inheritance can be used to assist in ensuring project consistency.

While project inheritance was limited by the extent of a developer's checkout in Maven 1.0 – that is, the current project – Maven 2 now retrieves parent projects from the repository, so it's possible to have one or more parents that define elements common to several projects. These parents (levels) may be used to define departments, or the organization as a whole.

As an example, consider the Maven project itself. It is a part of the Apache Software Foundation, and is a project that, itself, has a number of sub-projects (Maven, Maven SCM, Maven Continuum, etc.). As a result, there are three levels to consider when working with any individual module that makes up the Maven project. This project structure can be related to a company structure, wherein there's the organization, its departments, and then the teams within those departments. Any number of levels (parents) can be used, depending on the information that needs to be shared.

To continue the Maven example, consider the POM for Maven SCM:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <parent>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven-parent</artifactId>
  <version>1</version>
 </parent>
 <groupId>org.apache.maven.scm</groupId>
 <artifactId>maven-scm</artifactId>
 <url>http://maven.apache.org/maven-scm/</url>
...
 <modules>
  <module>maven-scm-api</module>
  <module>maven-scm-providers</module>
  ...
 </modules>
</project>
```

If you were to review the entire POM, you'd find that there is very little deployment or repository-related information, as this is consistent information, which is shared across all Maven projects through inheritance.

You may have noticed the unusual version declaration for the parent project. Since the version of the POM usually bears no resemblance to the software, the easiest way to version a POM is through sequential numbering. Future versions of Maven plan to automate the numbering of these types of parent projects to make this easier.

It is important to recall, from section 7.3, that if your inherited projects reside in an internal repository, then that repository will need to be added to the `settings.xml` file in the shared installation (or in each developer's home directory).

If you look at the Maven project's parent POM, you'd see it looks like the following:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <parent>
  <groupId>org.apache</groupId>
  <artifactId>apache</artifactId>
  <version>1</version>
 </parent>
 <groupId>org.apache.maven</groupId>
 <artifactId>maven-parent</artifactId>
 <version>5</version>
 <url>http://maven.apache.org/</url>
...
 <mailingLists>
  <mailingList>
    <name>Maven Announcements List</name>
     <post>announce@maven.apache.org</post>
  ...
  </mailingList>
 </mailingLists>

 <developers>
  <developer>
  ...
  </developer>
 </developers>
</project>
```

The Maven parent POM includes shared elements, such as the announcements mailing list and the list of developers that work across the whole project. Again, most of the elements are inherited from the organization-wide parent project, in this case the Apache Software Foundation:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.apache</groupId>
 <artifactId>apache</artifactId>
 <version>1</version>
 <organization>
  <name>Apache Software Foundation</name>
  <url>http://www.apache.org/</url>
 </organization>
 <url>http://www.apache.org/</url>
...
 <repositories>
  <repository>
   <id>apache.snapshots</id>
   <name>Apache Snapshot Repository</name>
   <url>http://svn.apache.org/maven-snapshot-repository</url>
   <releases>
    <enabled>false</enabled>
   </releases>
  </repository>
 </repositories>
...
 <distributionManagement>
  <repository>
  ...
  </repository>
  <snapshotRepository>
  ...
  </snapshotRepository>
 </distributionManagement>
</project>
```

The Maven project declares the elements that are common to all of its sub-projects – the snapshot repository (which will be discussed further in section 7.6), and the deployment locations.

An issue that can arise, when working with this type of hierarchy, is regarding the storage location of the source POM files. Source control management systems like CVS and SVN (with the traditional intervening trunk directory at the individual project level) do not make it easy to store and check out such a structure.

These parent POM files are likely to be updated on a different, and less frequent schedule than the projects themselves. For this reason, it is best to store the parent POM files in a separate area of the source control tree, where they can be checked out, modified, and deployed with their new version as appropriate. In fact, there is no best practice requirement to even store these files in your source control management system; you can retain the historical versions in the repository if it is backed up (in the future, the Maven Repository Manager will allow POM updates from a web interface).

7.5. Continuous Integration with Continuum

If you are not already familiar with it, continuous integration enables automated builds of your project on a regular interval, ensuring that conflicts are detected earlier in a project's release life cycle, rather than close to a release. More than just nightly builds, continuous integration can enable a better development culture where team members can make smaller, iterative changes that can more easily support concurrent development processes. As such, continuous integration is a key element of effective collaboration.

Continuum is Maven's continuous integration and build server. In this chapter, you will pick up the Proficio example from earlier in the book, and learn how to use Continuum to build this project on a regular basis. The examples discussed are based on Continuum 1.0.3, however newer versions should be similar. The examples also assumes you have Subversion installed, which you can obtain for your operating system from http://subversion.tigris.org/.

First, you will need to install Continuum. This is very simple – once you have downloaded it and unpacked it, you can run it using the following command:

```
C:\mvnbook\continuum-1.0.3> bin\win32\run
```

There are scripts for most major platforms, as well as the generic `bin/plexus.sh` for use on other Unix-based platforms.

You can verify the installation by viewing the web site at http://localhost:8080/continuum/.

The first screen to appear will be the one-time setup page shown in figure 7-1. The configuration on the screen is straight forward – all you should need to enter are the details of the administration account you'd like to use, and the company information for altering the logo in the top left of the screen.

For most installations this is all the configuration that's required, however, if you are running Continuum on your desktop and want to try the examples in this section, some additional steps are required. As of Continuum 1.0.3, these additional configuration requirements can be set only after the previous step has been completed, and you must stop the server to make the changes (to stop the server, press **Ctrl-C** in the window that is running Continuum).

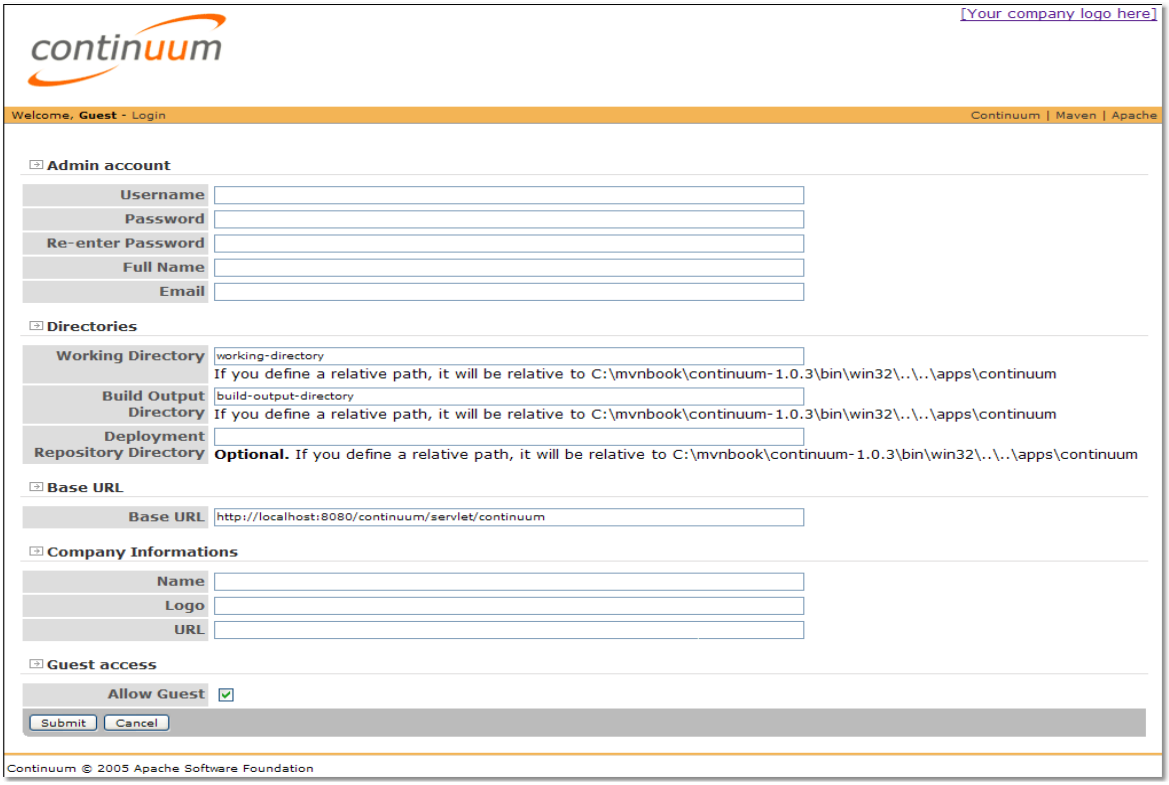


Figure 7-1: The Continuum setup screen

In the following examples, POM files will be read from the local hard disk where the server is running. By default, this is disabled as a security measure, since paths can be entered from the web interface. To enable this setting, edit `apps/continuum/conf/application.xml` and uncomment the following line:

```
...
 <implementation>
  org.codehaus.plexus.formica.validation.UrlValidator
 </implementation>
 <configuration>
  <allowedSchemes>
  ...
   <allowedScheme>file</allowedScheme>
  </allowedSchemes>
 </configuration>
...
```

To have Continuum send you e-mail notifications, you will also need an SMTP server to which to send the messages. The default is to use `localhost:25`. If you do not have this set up on your machine, edit the file above to change the `smtp-host` setting. For instructions, refer to http://maven.apache.org/continuum/guides/mini/guide-configuration.html.

After these steps are completed, you can start Continuum again.

The next step is to set up the Subversion repository for the examples. This requires obtaining the `Code_Ch07.zip` archive and unpacking it in your environment. You can then check out Proficio from that location, for example if it was unzipped in `C:\mvnbook\svn`:

```
C:\mvnbook> svn co file://localhost/C:/mvnbook/svn/proficio/trunk \
              proficio
```

The POM in this repository is not completely configured yet, since not all of the required details were known at the time of its creation. Edit `proficio/pom.xml` to correct the e-mail address to which notifications will be sent, and edit the location of the Subversion repository, by uncommenting and modifying the following lines:

```
...
<ciManagement>
 <system>continuum</system>
 <url>http://localhost:8080/continuum
 <notifiers>
  <notifier>
   <type>mail</type>
   <configuration>
    <address>youremail@yourdomain.com</address>
   </configuration>
  </notifier>
 </notifiers>
</ciManagement>
...
<scm>
 <connection>
  scm:svn:file://localhost/c:/mvnbook/svn/proficio/trunk
 </connection>
 <developerConnection>
  scm:svn:file://localhost/c:/mvnbook/svn/proficio/trunk
 </developerConnection>
</scm>
...
<distributionManagement>
 <site>
  <id>website</id>
  <url>
   file://localhost/c:/mvnbook/repository/sites/proficio
     /reference/${project.version}
  </url>
 </site>
</distributionManagement>
...
```

The `ciManagement` section can be used to configure a project for easier integration with Continuum.

The `distributionManagement` setting will be used in a later example to deploy the site from your continuous integration environment. This assumes that you are still running the repository Web server on `localhost:8081`, from the directory `C:\mvnbook\repository`. If you haven't done so already, refer to section 7.3 for information on how to set this up.

Once these settings have been edited to reflect your setup, commit the file with the following command:

```
C:\mvnbook\proficio> svn ci -m 'my settings' pom.xml
```

You should build all these modules to ensure everything is in order, with the following command:

```
C:\mvnbook\proficio> mvn install
```

You are now ready to start using Continuum.

If you return to the `http://localhost:8080/continuum/` location that was set up previously, you will see an empty project list. Before you can add a project to the list, or perform other tasks, you must either log in with the administrator account you created during installation, or with another account you have since created with appropriate permissions. The login link is at the top-left of the screen, under the Continuum logo.

Once you have logged in, you can now select *Maven 2.0+ Project* from the *Add Project* menu. This will present the screen shown in figure 7-2. You have two options: you can provide the URL for a POM, or upload from your local drive. While uploading is a convenient way to configure from your existing check out, in Continuum 1.0.3 this does not work when the POM contains modules, as in the Proficio example. Instead, enter the `file:// URL` as shown. When you set up your own system later, you will enter either a HTTP URL to a POM in the repository, a ViewCVS installation, or a Subversion HTTP server.

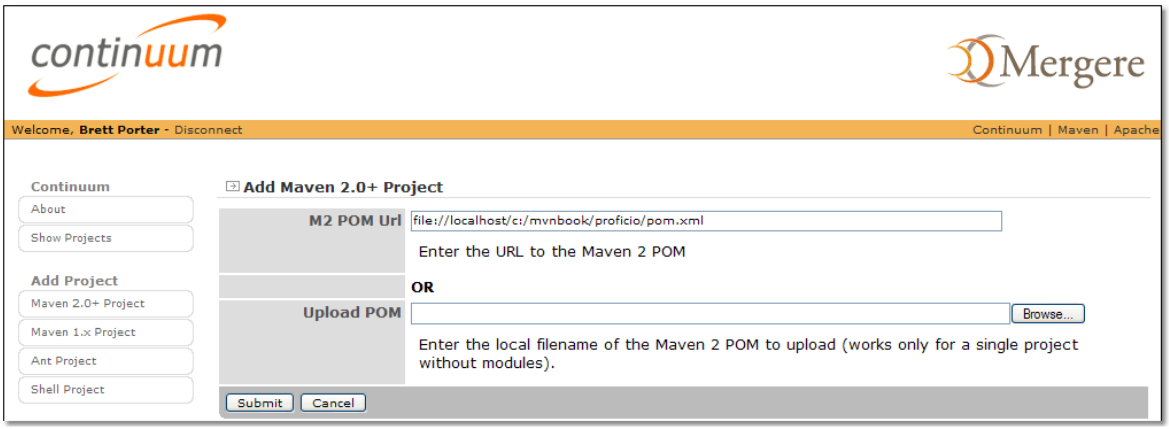


Figure 7-2: Add project screen shot

This is all that is required to add a Maven 2 project to Continuum. After submitting the URL, Continuum will return to the project summary page, and each of the modules will be added to the list of projects. Initially, the builds will be marked as *New* and their checkouts will be queued. The result is shown in figure 7-3.

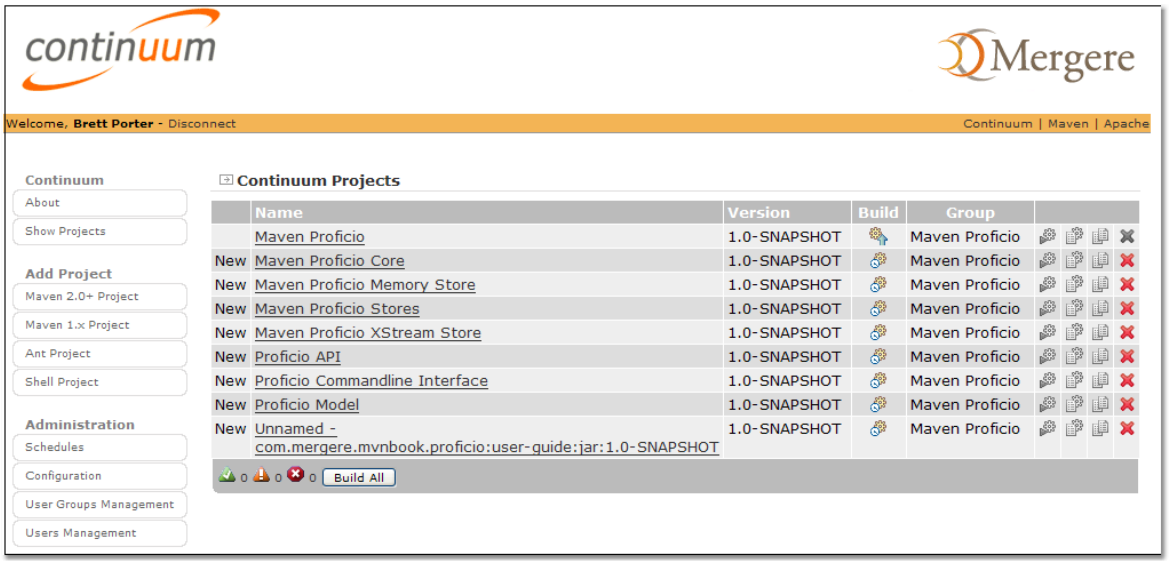


Figure 7-3: Summary page after projects have built

Continuum will now build the project hourly, and send an e-mail notification if there are any problems. If you want to put this to the test, go to your earlier checkout and introduce an error into Proficio.java, for example, remove the interface keyword:

```
[...]
public Proficio
[...]
```

Now, check the file in:

```
C:\mvnbook\proficio\proficio-api> svn ci -m 'introduce error' \
  src/main/java/com/mergere/mvnbook/proficio/Proficio.java
```

Finally, press *Build Now* on the Continuum web interface next to the *Proficio API* module. First, the build will show an “In progress” status, and then fail, marking the left column with an “!” to indicate a failed build (you will need to refresh the page using the *Show Projects* link in the navigation to see these changes). In addition, you should receive an e-mail at the address you configured earlier. The *Build History* link can be used to identify the failed build and to obtain a full output log.

To avoid receiving this error every hour, restore the file above to its previous state and commit it again. The build in Continuum will return to the successful state.

This chapter will not discuss all of the features available in Continuum, but you may wish to go ahead and try them. For example, you might want to set up a notification to your favorite instant messenger – IRC, Jabber, MSN and Google Talk are all supported.

Regardless of which continuous integration server you use, there are a few tips for getting the most out of the system:

- **Commit early, commit often**. Continuous integration is most effective when developers commit regularly. This doesn't mean committing incomplete code, but rather keeping changes small and well tested. This will make it much easier to detect the source of an error when the build does break.
- **Run builds as often as possible**. This will be constrained by the length of the build and the available resources on the build machine, but it is best to detect a failure as soon as possible, before the developer moves on or loses focus. Continuum can be configured to trigger a build whenever a commit occurs, if the source control repository supports post-commit hooks. This also means that builds should be fast – long integration and performance tests should be reserved for periodic builds.
- **Fix builds as soon as possible**. While this seems obvious, it is often ignored. Continuous integration will be pointless if developers repetitively ignore or delete broken build notifications, and your team will become desensitized to the notifications in the future.
- **Establish a stable environment**. Avoid customizing the JDK, or local settings, if it isn't something already in use in other development, test and production environments. When a failure occurs in the continuous integration environment, it is important that it can be isolated to the change that caused it, and independent of the environment being used.
- **Run clean builds**. While rapid, iterative builds are helpful in some situations, it is also important that failures don't occur due to old build state. Consider a regular, clean build. Continuum currently defaults to doing a clean build, and a future version will allow developers to request a fresh checkout, based on selected schedules.
- **Run comprehensive tests**. Continuous integration is most beneficial when tests are validating that the code is working as it always has, not just that the project still compiles after one or more changes occur. In addition, it is beneficial to test against all different versions of the JDK, operating system and other variables, periodically. Continuum has preliminary support for system profiles and distributed testing, enhancements that are planned for future versions.
- **Build all of a project's active branches**. If multiple branches are in development, the continuous integration environment should be set up for all of the active branches.
- **Run a copy of the application continuously**. If the application is a web application, for example, run a servlet container to which the application can be deployed from the continuous integration environment. This can be helpful for non-developers who need visibility into the state of the application, separate from QA and production releases.

In addition to the above best practices, there are two additional topics that deserve special attention: automated updates to the developer web site, and profile usage.

In Chapter 6, you learned how to create an effective site containing project information and reports about the project's health and vitality. For these reports to be of value, they need to be kept up-to-date. This is another way continuous integration can help with project collaboration and communication. Though it would be overkill to regenerate the site on ever commit, it is recommend that a separate, but regular schedule is established for site generation.

Verify that you are still logged into your Continuum instance. Next, from the *Administration* menu on the left-hand side, select *Schedules*. You will see that currently, only the default schedule is available. Click the *Add* button to add a new schedule, which will be configured to run every hour during business hours (8am – 4pm weekdays).

The appropriate configuration is shown in figure 7-4.

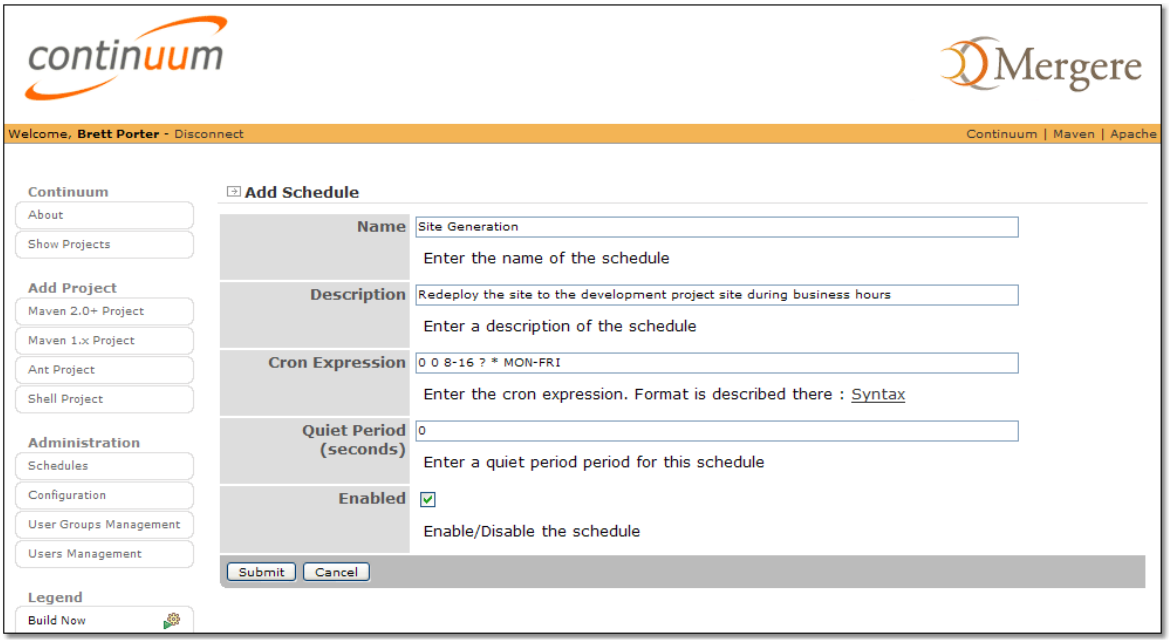


Figure 7-4: Schedule configuration

The cron expression entered here is much like the one entered for a Unix crontab and is further described at http://www.opensymphony.com/quartz/api/org/quartz/CronTrigger.html. The example above runs at 8:00:00, 9:00:00,..., 16:00:00 from Monday to Friday.

The “quiet period” is a setting that delays the build if there has been a commit in the defined number of seconds prior. This is useful when using CVS, since commits are not atomic and a developer might be committing midway through a update. It is not typically needed if using Subversion.

Once you add this schedule, return to the project list, and select the top-most project, *Maven Proficio*. The project information shows just one build on the default schedule that installs the parent POM, but does not recurse into the modules (the `-N or --non-recursive argument`). Since this is the root of the multi-module build – and it will also detect changes to any of the modules – this is the best place from which to build the site. In addition to building the sites for each module, it can aggregate changes into the top-level site as required.

The downside to this approach is that Continuum will build any unchanged modules, as well – if this is a concern, use the non-recursive mode instead, and add the same build definition to all of the modules.

In Continuum 1.0.3, there is no way to make bulk changes to build definitions, so you will need to add the definition to each module individually.

In this example you will add a new build definition to run the site deployment for the entirety of the multi-module build, on the business hours schedule. To add a new build definition, click the *Add* button below the default build definition. The *Add Build Definition* screen is shown in figure 7-5.

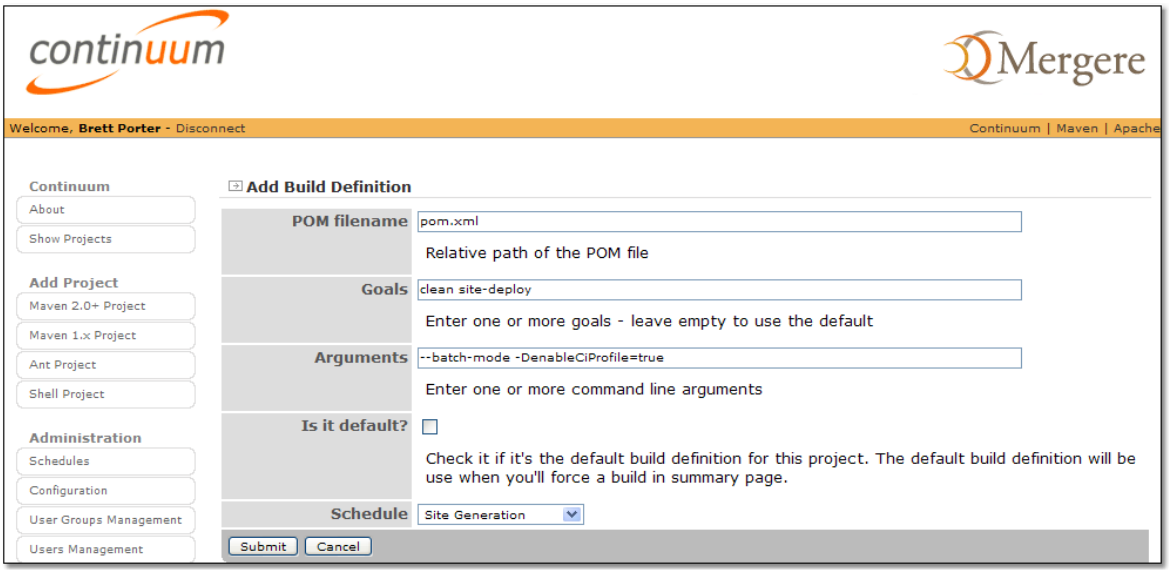


Figure 7-5: Adding a build definition for site deployment

The goals to run are clean and site-deploy. The site will be deployed to the file system location you specified in the POM, when you first set up the Subversion repository earlier in this chapter, which will be visible from `http://localhost:8081/sites/proficio/`.

The arguments provided are `--batch-mode`, which is essential for all builds to ensure they don't block for user input, and `-DenableCiProfile=true`, which sets the given system property. The meaning of this system property will be explained shortly. The `--non-recursive` option is omitted.

You can see also that the schedule is set to use the site generation schedule created earlier, and that it is not the default build definition, which means that *Build Now* from the project summary page will not trigger this build. However, each build definition on the project information page (to which you would have been returned after adding the build definition) has a *Build Now* icon. Click this for the site generation build definition, and view the generated site from `http://localhost:8081/sites/proficio/`.

It is rare that the site build will fail, since most reports continue under failure conditions. However, if you want to fail the build based on these checks as well, you can add the test, verify or integration-test goal to the list of goals, to ensure these checks are run.

Any of these test goals should be listed after the site-deploy goal, so that if the build fails because of a failed check, the generated site can be used as reference for what caused the failure.

In the previous example, a system property called `enableCiProfile` was set to true. In Chapter 6, a number of plugins were set up to fail the build if certain project health checks failed, such as the percentage of code covered in the unit tests dropping below a certain value. However, these checks delayed the build for all developers, which can be a discouragement to using them.

If you compare the example `proficio/pom.xml` file in your Subversion checkout to that used in Chapter 6, you'll see that these checks have now been moved to a profile. Profiles are a means for selectively enabling portions of the build. If you haven't previously encountered profiles, please refer to Chapter 3. In this particular case, the profile is enabled only when the `enableCiProfile` system property is set to true.

```
...
<profiles>
 <profile>
  <id>ciProfile</id>
  <activation>
   <property>
    <name>enableCiProfile</name>
    <value>true</value>
   </property>
  </activation>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
    <executions>
    ...
```

You'll find that when you run the build from the command line (as was done in Continuum originally), none of the checks added in the previous chapter are executed. However, if you run `mvn -DenableCiProfile=true` verify, they will be.

There are two ways to ensure that all of the builds added in Continuum use this profile. The first is to adjust the default build definition for each module, by going to the module information page, and clicking *Edit* next to the default build definition. As you saw before, it is necessary to do this for each module individually, at least in the version of Continuum current at the time of writing.

The other alternative is to set this profile globally, for all projects in Continuum. As Maven 2 is still executed as normal, it reads the `${user.home}/.m2/settings.xml` file for the user under which it is running, as well as the settings in the Maven installation. To enable this profile by default from these settings, add the following configuration to the `settings.xml` file in `<maven home>/conf/settings.xml`:

```
...
<activeProfiles>
...
 <activeProfile>ciProfile</activeProfile>
</activeProfiles>
...
```

In this case the identifier of the profile itself, rather than the property used to enable it, indicates that the profile is always active when these settings are read.

How you configure your continuous integration depends on the culture of your development team and other environmental factors such as the size of your projects and the time it takes to build and test them. The guidelines discussed in this chapter will help point your team in the right direction, but the timing and configuration can be changed depending upon your circumstances. For example, if the additional checks take too much time for frequent continuous integration builds, it may be necessary to schedule them separately for each module, or for the entire multi-module project to run the additional checks after the site has been generated, the verify goal may need to be added to the site deployment build definition, as discussed previously.

7.6. Team Dependency Management Using Snapshots

Chapter 3 of this book discussed how to manage your dependencies in a multi-module build, and while dependency management is fundamental to any Maven build, the team dynamic makes it critical.

In this section, you will learn about using snapshots more effectively in a team environment, and how to enable this within your continuous integration environment.

So far in this book, snapshots have been used to refer to the development version of an individual module. The generated artifacts of the snapshot are stored in the local repository, and in contrast to regular dependencies, which are not changed, these artifacts will be updated frequently. Projects in Maven stay in the snapshot state until they are released, which is discussed in section 7.8 of this chapter.

Snapshots were designed to be used in a team environment as a means for sharing development versions of artifacts that have already been built. Usually, in an environment where a number of modules are undergoing concurrent development, the build involves checking out all of the dependent projects and building them yourself. Additionally, in some cases, where projects are closely related, you must build all of the modules simultaneously from a master build.

While building all of the modules from source can work well and is handled by Maven inherently, it can lead to a number of problems:

- It relies on manual updates from developers, which can be error-prone. This will result in local inconsistencies that can produce non-working builds
- There is no common baseline against which to measure progress
- Building can be slower as multiple dependencies must be rebuilt also
- Changes developed against outdated code can make integration more difficult

As you can see from these issues, building from source doesn't fit well with an environment that promotes continuous integration. Instead, using binary snapshots that have been built and tested already, is preferred.

In Maven, this is achieved by regularly deploying snapshots to a shared repository, such as the internal repository set up in section 7.3. Considering that example, you'll see that the repository was defined in `proficio/pom.xml`:

```
...
<distributionManagement>
 <repository>
  <id>internal</id>
  <url>file://localhost/c:/mvnbook/repository/internal</url>
 </repository>
...
</distributionManagement>
```

Now, deploy `proficio-api` to the repository with the following command:

```
C:\mvnbook\proficio\proficio-api> mvn deploy
```

You'll see that it is treated differently than when it was installed in the local repository. The filename that is used is similar to `proficio-api-1.0-20060211.131114-1.jar`. In this case, the version used is the time that it was deployed (in the UTC timezone) and the build number. If you were to deploy again, the time stamp would change and the build number would increment to 2.

This technique allows you to continue using the latest version by declaring a dependency on `1.0-SNAPSHOT`, or to lock down a stable version by declaring the dependency version to be the specific equivalent such as `1.0-20060211.131114-1`. While this is not usually the case, locking the version in this way may be important if there are recent changes to the repository that need to be ignored temporarily.

Currently, the Proficio project itself is not looking in the internal repository for dependencies, but rather relying on the other modules to be built first, though it may have been configured as part of your settings files. To add the internal repository to the list of repositories used by Proficio regardless of settings, add the following to `proficio/pom.xml`:

```
...
<repositories>
 <repository>
  <id>internal</id>
  <url>http://localhost:8081/internal</url>
 </repository>
</repositories>
...
```

If you are developing plugins, you may also want to add this as a `pluginRepository` element as well.

Now, to see the updated version downloaded, build `proficio-core` with the following command:

```
C:\mvnbook\proficio\proficio-core> mvn -U install
```

During the build, you will see that some of the dependencies are checked for updates, similar to the example below (note that this output has been abbreviated):

```
...
proficio-api:1.0-SNAPSHOT: checking for updates from internal
...
```

The `-U` argument in the prior command is required to force Maven to update all of the snapshots in the build. If it were omitted, by default, no update would be performed. This is because the default policy is to update snapshots daily – that is, to check for an update the first time that particular dependency is used after midnight local time.

You can always force the update using the `-U` command, but you can also change the interval by changing the repository configuration. To see this, add the following configuration to the repository configuration you defined above in `proficio/pom.xml`:

```
...
<repository>
 ...
 <snapshots>
  <updatePolicy>interval:60</updatePolicy>
 </snapshots>
</repository>
...
```

In this example, any snapshot dependencies will be checked once an hour to determine if there are updates in the remote repository. The settings that can be used for the update policy are never, always, daily (the default), and `interval:`*`minutes`*.

Whenever you use the `-U` argument, it updates both releases and snapshots. This causes many plugins to be checked for updates, as well as updating any version ranges.

This technique can ensure that developers get regular updates, without having to manually intervene, and without slowing down the build by checking on every access (as would be the case if the policy were set to always). However, the updates will still occur only as frequently as new versions are deployed to the repository.

It is possible to establish a policy where developers do an update from the source control management (SCM) system before committing, and then deploy the snapshot to share with the other team members. However, this introduces a risk that the snapshot will not be deployed at all, deployed with uncommitted code, or deployed without all the updates from the SCM, making it out-of-date. Several of the problems mentioned earlier still exist – so at this point, all that is being saved is some time, assuming that the other developers have remembered to follow the process.

A much better way to use snapshots is to automate their creation. Since the continuous integration server regularly rebuilds the code from a known state, it makes sense to have it build snapshots, as well.

How you implement this will depend on the continuous integration server that you use. To deploy from your server, you must ensure that the `distributionManagement` section of the POM is correctly configured. However, as you saw earlier, Continuum can be configured to deploy its builds to a Maven snapshot repository automatically. If there is a repository configured to which to deploy them, this feature is enabled by default in a build definition. In section 6.5 of this chapter, you were not asked to apply this setting, so let's go ahead and do it now. Log in as an administrator and go to the following Configuration screen, shown in figure 7-6.

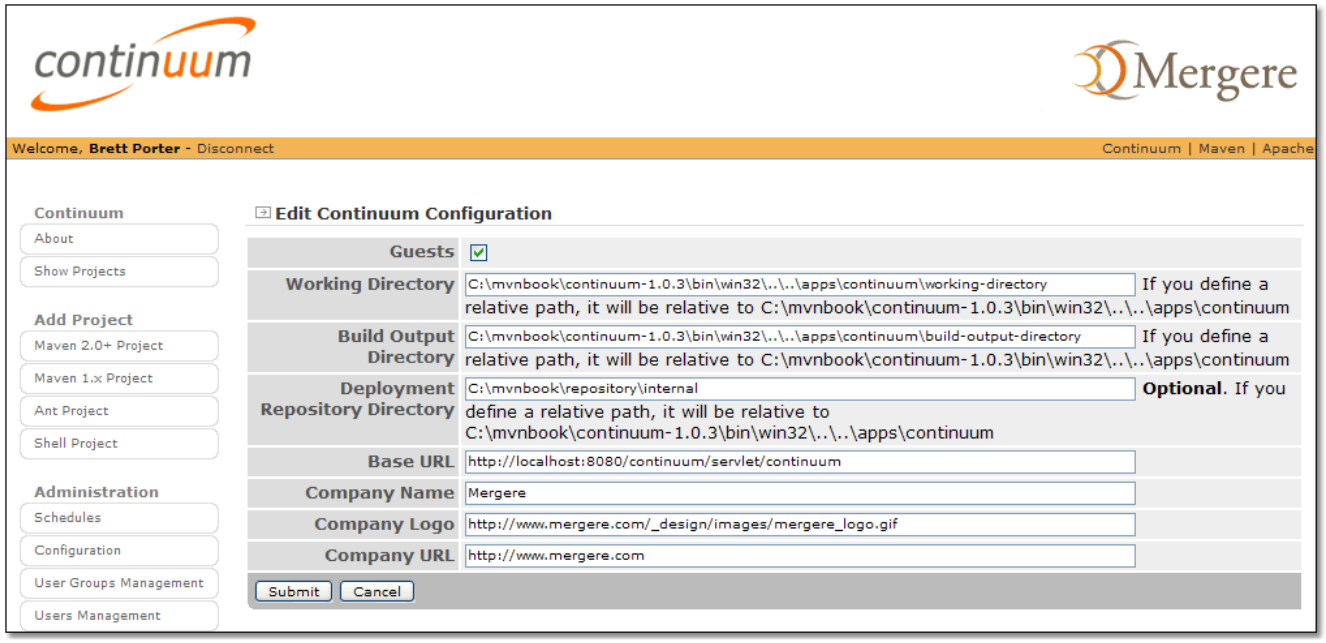


Figure 7-6: Continuum configuration

In Deployment Repository field, enter the location of your internal repository: `C:\mvnbook\repository\internal`.

> This relies on your internal repository and Continuum server being in the same location. If this is not the case, you can enter a full repository URL such as `scp://repositoryhost/www/repository/internal`.

To try this feature, follow the *Show Projects* link, and click *Build Now* on the *Proficio API* project. Once the build completes, return to your console and build `proficio-core` again using the following command:

```
C:\mvnbook\proficio\proficio-core> mvn -U install
```

You'll notice that a new version of `proficio-api` is downloaded, with an updated time stamp and build number.

With this setup, you can avoid all of the problems discussed previously. Better yet, while you get regular updates from published binary dependencies, when necessary, you can either lock a dependency to a particular build, or build from source.

Another point to note about snapshots is that it is possible to store them in a separate repository from the rest of your released artifacts. This can be useful if you need to clean up snapshots on a regular interval, but still keep a full archive of releases.

If you are using the regular deployment mechanism (instead of using Continuum), this separation is achieved by adding an additional repository to the `distributionManagement` section of your POM. For example, if you had a snapshot-only repository in `/www/repository/snapshots`, you would add the following:

```
...
<distributionManagement>
...
 <snapshotRepository>
  <id>internal.snapshots</id>
  <url>file://localhost/www/repository/snapshots</url>
 </snapshotRepository>
</distributionManagement>
...
```

This will deploy to that repository whenever the version contains SNAPSHOT, and deploy to the regular repository you listed earlier, when it doesn't.

Given this configuration, you can make the snapshot update process more efficient by not checking the repository that has only releases for updates. The replacement repository declarations in your POM would look like this:

```
...
<repositories>
 <repository>
  <id>internal</id>
  <url>http://localhost:8081/internal</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>internal.snapshots</id>
  <url>http://localhost:8081/snapshots</url>
  <snapshots>
   <updatePolicy>interval:60</updatePolicy>
  </snapshots>
 </repository>
</repositories>
...
```

7.7. Creating a Standard Project Archetype

Throughout this book, you have seen the archetypes that were introduced in Chapter 2 used to quickly lay down a project structure. While this is convenient, there is always some additional configuration required, either in adding or removing content from that generated by the archetypes. To avoid this, you can create one or more of your own archetypes.

Beyond the convenience of laying out a project structure instantly, archetypes give you the opportunity to start a project in the right way – that is, in a way that is consistent with other projects in your environment. As you saw in this chapter, the requirement of achieving consistency is a key issue facing teams.

Writing an archetype is quite like writing your own project, and replacing the specific values with parameters. There are two ways to create an archetype: one based on an existing project using `mvn archetype:create-from-project`, and the other, by hand, using an archetype. To get started with the archetype, run the following command:

```
C:\mvnbook\proficio> mvn archetype:create \
  -DgroupId=com.mergere.mvnbook \
  -DartifactId=proficio-archetype \
  -DarchetypeArtifactId=maven-archetype-archetype
```

The layout of the resulting archetype is shown in figure 7-7.

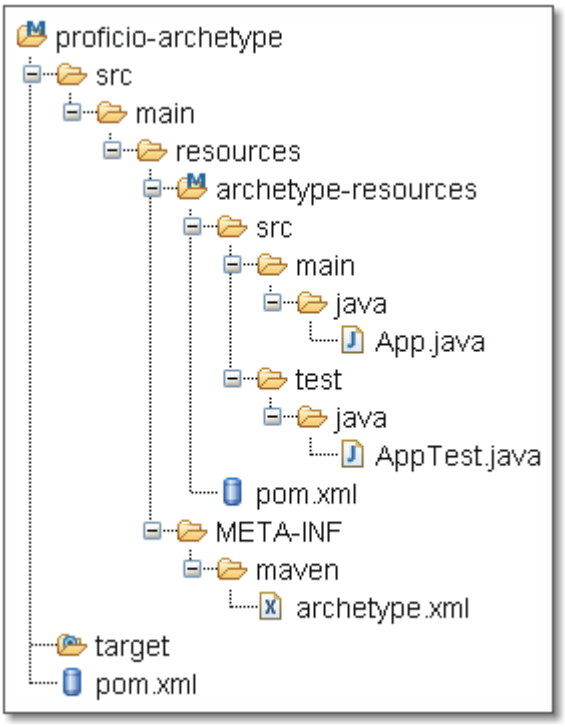


Figure 7-7: Archetype directory layout

If you look at `pom.xml` at the top level, you'll see that the archetype is just a normal JAR project – there is no special build configuration required. The JAR that is built is composed only of resources, so everything else is contained under `src/main/resources`. There are two pieces of information required: the archetype descriptor in `META-INF/maven/archetype.xml`, and the template project in archetype-resources.

The archetype descriptor describes how to construct a new project from the archetype-resources provided. The example descriptor looks like the following:

```
<archetype>
  <id>proficio-archetype</id>
  <sources>
    <source>src/main/java/App.java</source>
  </sources>
  <testSources>
    <source>src/test/java/AppTest.java</source>
  </testSources>
</archetype>
```

Each tag is a list of files to process and generate in the created project. The example above shows the sources and test sources, but it is also possible to specify files for resources, `testResources`, and `siteResources`.

The files within the archetype-resources section are Velocity templates. These files will be used to generate the template files when the archetype is run. For this example, the `pom.xml` file looks like the following:

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

As you can see, the `groupId`, `artifactId` and version elements are variables that will be substituted with the values provided by the developer running archetype:create.

From here, you need to populate the template with the content that you'd like to have applied consistently to new projects. For more information on creating an archetype, refer to the documentation on the Maven web site.

Once you have completed the content in the archetype, Maven will build, install and deploy it like any other JAR. Continuing from the example in section 7.3 of this chapter, you will use the “internal” repository. Since the archetype inherits the Proficio parent, it has the correct deployment settings already, so you can run the following command:

```
C:\mvnbook\proficio\proficio-archetype> mvn deploy
```

The archetype is now ready to be used. To do so, go to an empty directory and run the following command:

```
C:\mvnbook> mvn archetype:create -DgroupId=com.mergere.mvnbook \
  -DartifactId=proficio-example \
   -DarchetypeGroupId=com.mergere.mvnbook \
  -DarchetypeArtifactId=proficio-archetype \
  -DarchetypeVersion=1.0-SNAPSHOT
```

Normally, the `archetypeVersion` argument is not required at this point. However, since the archetype has not yet been released, if omitted, the required version would not be known (or if this was later development, a previous release would be used instead). Releasing a project is explained in section 7.9 of this chapter.

You now have the template project laid out in the `proficio-example` directory. It will look very similar to the content of the archetype-resources directory you created earlier, now however, the content of the files will be populated with the values that you provided on the command line.

7.8. Cutting a Release

Releasing software is difficult. It is usually tedious and error prone, full of manual steps that need to be completed in a particular order. Worse, it happens at the end of a long period of development when all everyone on the team wants to do is get it out there, which often leads to omissions or short cuts. Finally, once a release has been made, it is usually difficult or impossible to correct mistakes other than to make another, new release.

Once the definition for a release has been set by a team, releases should be consistent every time they are built, allowing them to be highly automated. Maven provides a release plugin that provides the basic functions of a standard release process. The release plugin takes care of a number of manual steps in updating the project POM, updating the source control management system to check and commit release related changes, and creating tags (or equivalent for your SCM).

The release plugin operates in two steps: *prepare* and *perform*. The prepare step is run once for a release, and does all of the project and source control manipulation that results in a tagged version. The perform step could potentially be run multiple times to rebuild a release from a clean checkout of the tagged version, and to perform standard tasks, such as deployment to the remote repository.

To demonstrate how the release plugin works, the Proficio example will be revisited, and released as 1.0. You can continue using the code that you have been working on in the previous sections, or check out the following:

```
C:\mvnbook> svn co \
  file://localhost/C:/mvnbook/svn/proficio/tags/proficio-final \
  proficio
```

To start the release process, run the following command:

```
c:\mvnbook\proficio> mvn release:prepare -DdryRun=true
```

This simulates a normal release preparation, without making any modifications to your project. You'll notice that each of the modules in the project is considered. As the command runs, you will be prompted for values. Accept the defaults in this instance (note that running Maven in “batch mode” avoids these prompts and will accept all of the defaults).

In this project, all of the dependencies being used are releases, or part of the project. However, if you are using a dependency that is a snapshot, an error will appear.

The prepare step ensures that there are no snapshots in the build, other than those that will be released as part of the process (that is, other modules). This is because the prepare step is attempting to guarantee that the build will be reproducible in the future, and snapshots are a transient build, not ready to be used as a part of a release.

In some cases, you may encounter a plugin snapshot, even if the plugin is not declared in the POM. This is because you are using a locally installed snapshot of a plugin (either built yourself, or obtained from the development repository of the Maven project) that is implied through the build life cycle. This can be corrected by adding the plugin definition to your POM, and setting the version to the latest release (But only after verifying that your project builds correctly with that version!).

To review the steps taken in this release process:

1. Check for correct version of the plugin and POM (for example, the appropriate SCM settings)
2. Check if there are any local modifications
3. Check for snapshots in dependency tree
4. Check for snapshots of plugins in the build
5. Modify all POM files in the build, as they will be committed to the tag
6. Run `mvn` clean integration-test to verify that the project will successfully build
7. Describe other preparation goals (none are configured by default, but this might include updating the metadata in your issue tracker, or creating and committing an announcement file)
8. Describe the SCM commit and tag operations
9. Modify all POM files in the build, as they will be committed for the next development iteration
10. Describe the SCM commit operation

You might like to review the POM files that are created for steps 4 and 6, named `pom.xml.tag` and `pom.xml.next` respectively in each module directory, to verify they are correct. You'll notice that the version is updated in both of these files, and is set based on the values for which you were prompted during the release process. The SCM information is also updated in the tag POM to reflect where it will reside once it is tagged, and this is reverted in the next POM.

However, these changes are not enough to guarantee a reproducible build – it is still possible that the plugin versions will vary, that resulting version ranges will be different, or that different profiles will be applied. For that reason, there is also a `release-pom.xml.tag` file written out to each module directory. This contains a resolved version of the POM that Maven will use to build from if it exists. In this POM, a number of changes are made:

- the explicit version of plugins and dependencies that were used are added
- any settings from `settings.xml` (both per-user and per-installation) are incorporated into the POM.
- any active profiles are explicitly activated, including profiles from `settings.xml` and `profiles.xml`

You may have expected that inheritance would have been resolved by incorporating any parent elements that are used, or that expressions would have been resolved. This is not the case however, as these can be established from the other settings already populated in the POM in a reproducible fashion.

When the final run is executed, this file will be `release-pom.xml` in the same directory as `pom.xml`. This is used by Maven, instead of the normal POM, when a build is run from this tag to ensure it matches the same circumstances as the release build.

Having run through this process you may have noticed that only the unit and integration tests were run as part of the test build. Recall from Chapter 6 that you learned how to configure a number of checks – so it is important to verify that they hold as part of the release. Also, recall that in section 7.5, you created a profile to enable those checks conditionally. To include these checks as part of the release process, you need to enable this profile during the verification step. To do so, use the following plugin configuration:

```
[...]
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <configuration>
   <prepareVerifyArgs>-DenableCiProfile=true</prepareVerifyArgs>
  </configuration>
 </plugin>
[...]
```

Try the dry run again:

```
C:\mvnbook\proficio> mvn release:prepare -DdryRun=true
```

Now that you've gone through the test run and are happy with the results, you can go for the real thing with the following command:

```
C:\mvnbook\proficio> mvn release:prepare
```

You'll notice that this time the operations on the SCM are actually performed, and the updated POM files are committed.

You won't be prompted for values as you were the first time – since by the default, the release plugin will resume a previous attempt by reading the `release.properties` file that was created at the end of the last run. If you need to start from the beginning, you can remove that file, or run `mvn -Dresume=false` release:prepare instead.

Once this is complete, you'll see in your SCM the new tag for the project (with the modified files), while locally, the version is now `1.1-SNAPSHOT`.

However, the release still hasn't been generated yet – for that, you need to deploy the build artifacts. This is achieved with the release:perform goal. This is run as follows:

```
C:\mvnbook\proficio> mvn release:perform
```

No special arguments are required, because the `release.properties` file still exists to tell the goal the version from which to release. To release from an older version, or if the `release.properties` file had been removed, you would run the following:

```
C:\mvnbook\proficio> mvn release:perform -DconnectionUrl=\
  scm:svn:file://localhost/c:/mvnbook/svn/proficio/tags/proficio-1.0
```

If you follow the output above, you'll see that a clean checkout was obtained from the created tag, before running Maven from that location with the goals deploy site-deploy. This is the default for the release plugin – to deploy all of the built artifacts, and to deploy a copy of the site.

If this is not what you want to run, you can change the goals used with the goals parameter:

```
C:\mvnbook\proficio> mvn release:perform -Dgoals="deploy"
```

However, this requires that you remember to add the parameter every time. Since the goal is for consistency, you want to avoid such problems. To do so, add the following goals to the POM:

```
[...]
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <configuration>
   <goals>deploy</goals>
  </configuration>
 </plugin>
[...]
```

You may also want to configure the release plugin to activate particular profiles, or to set certain properties. Refer to the plugin reference at http://maven.apache.org/plugins/maven-release-plugin/ for more information. It is important in these cases that you consider the settings you want, before you run the `release:prepare` goal, though. To ensure reproducibility, the release plugin will confirm that the checked out project has the same release plugin configuration as those being used (with the exception of `goals`).

When the release is performed, and the built artifacts are deployed, you can examine the files that are placed in the SCM repository. To do this, check out the tag:

```
C:\mvnbook> svn co \
  file://localhost/mvnbook/svn/proficio/tags/proficio-1.0
```

You'll notice that the contents of the POM match the `pom.xml` file, and not the `release-pom.xml` file. The reason for this is that the POM files in the repository are used as dependencies and the original information is more important than the release-time information – for example, it is necessary to know what version ranges are allowed for a dependency, rather than the specific version used for the release. For the same reason, both the original `pom.xml` file and the `release-pom.xml` files are included in the generated JAR file.

Also, during the process you will have noticed that Javadoc and source JAR files were produced and deployed into the repository for all the Java projects. These are configured by default in the Maven POM as part of a profile that is activated when the release is performed.

You can disable this profile by setting the `useReleaseProfile` parameter to `false`, as follows:

```
[...]
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <configuration>
   <useReleaseProfile>false</useReleaseProfile>
  </configuration>
 </plugin>
[...]
```

Instead, you may want to include additional actions in the profile, without having to declare and enable an additional profile. To do this, define a profile with the identifier release-profile, as follows:

```
[...]
 <profiles>
  <profile>
   <id>release-profile</id>

   <build>
    <!-- Extra plugin configuration would be inserted here -->
   </build>

  </profile>
 </profiles>
[...]
```

After the release process is complete, the only step left is to clean up after the plugin, removing `release.properties` and any POM files generated as a result of the dry run. Simply run the following command to clean up:

```
C:\mvnbook\proficio> mvn release:clean
```

7.9. Summary

As you've seen throughout this chapter, and indeed this entire book, Maven was designed to address issues that directly affect teams of developers. All of the features described in this chapter can be used by any development team. So, whether your team is large or small, Maven provides value by standardizing and automating the build process.

There are also strong team-related benefits in the preceding chapters – for example, the adoption of reusable plugins can capture and extend build knowledge throughout your entire organization, rather than creating silos of information around individual projects. The site and reports you've created can help a team communicate the status of a project and their work more effectively. And all of these features build on the essentials demonstrated in chapters 1 and 2 that facilitate consistent builds.

Lack of consistency is the source of many problems when working in a team, and while Maven focuses on delivering consistency in your build infrastructure through patterns, it can aid you in effectively using tools to achieve consistency in other areas of your development. This in turn can lead to and facilitate best practices for developing in a community-oriented, real-time engineering style, by making information about your projects visible and organized.

Migrating to Maven

This chapter explains how to migrate (convert) an existing build in Ant, to a build in Maven:

- Splitting existing sources and resources into modular Maven projects
- Taking advantage of Maven's inheritance and multi-project capabilities
- Compiling, testing and building jars with Maven, using both Java 1.4 and Java 5
- Using Ant tasks from within Maven
- Using Maven with your current directory structure

> This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.
>
> *- Morpheus. The Matrix*

8.1. Introduction

The purpose of this chapter is to show a migration path from an existing build in Ant to Maven.

The Maven migration example is based on the Spring Framework build, which uses an Ant script. This example will take you through the step-by-step process of migrating Spring to a modularized, component-based, Maven build.

You will learn how to start building with Maven, while still running your existing, Ant-based build system. This will allow you to evaluate Maven's technology, while enabling you to continue with your required work.

You will learn how to use an existing directory structure (though you will not be following the standard, recommended Maven directory structure), how to split your sources into modules or components, how to run Ant tasks from within Maven, and among other things, you will be introduced to the concept of dependencies.

8.1.1. Introducing the Spring Framework

The Spring Framework is one of today's most popular Java frameworks. For the purpose of this example, we will focus only on building version 2.0-m1 of Spring, which is the latest version at the time of writing.

The Spring release is composed of several modules, listed in build order:

- spring-core
- spring-beans
- spring-aop
- spring-context
- spring-dao
- spring-jdbc
- spring-support
- spring-web
- spring-webmvc
- spring-remoting
- spring-portlet
- spring-jdo
- spring-hibernate2
- spring-hibernate3
- spring-toplink
- spring-ojb
- spring-mock
- spring-aspects

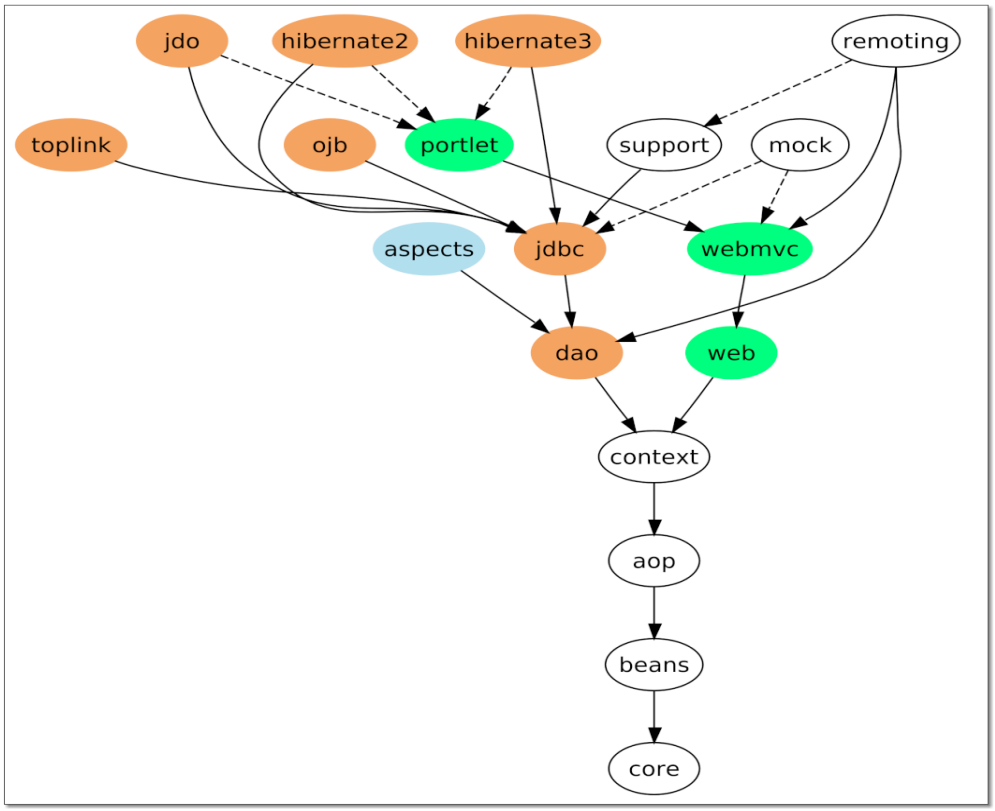


Figure 8-1: Dependency relationship between Spring modules

In figure 8-1. you can see graphically the dependencies between the modules. Optional dependencies are indicated by dotted lines.

Each of these modules corresponds, more or less, with the Java package structure, and each produces a JAR. These modules are built with an Ant script from the following source directories:

- `src` and `test`: contain JDK 1.4 compatible source code and JUnit tests respectively
- `tiger/src and tiger/test`: contain additional JDK 1.5 compatible source code and JUnit tests
- `mock`: contains the source code for the spring-mock module
- `aspectj/src and aspectj/test`: contain the source code for the spring-aspects module

Each of the source directories also include classpath resources (XML files, properties files, TLD files, etc.).

For Spring, the Ant script compiles each of these different source directories and then creates a JAR for each module, using inclusions and exclusions that are based on the Java packages of each class. The `src` and `tiger/src` directories are compiled to the same destination as the test and `tiger/test` directories, resulting in JARs that contain both 1.4 and 1.5 classes.

8.2. Where to Begin?

With Maven, the rule of thumb to use is to produce one artifact (JAR, WAR, etc.) per Maven project file. In the Spring example, that means you will need to have a Maven project (a POM) for each of the modules listed above.

To start, you will create a subdirectory called '`m2`' to keep all the necessary Maven changes clearly separated from the current build system. Inside the '`m2`' directory, you will need to create a directory for each of Spring's modules.

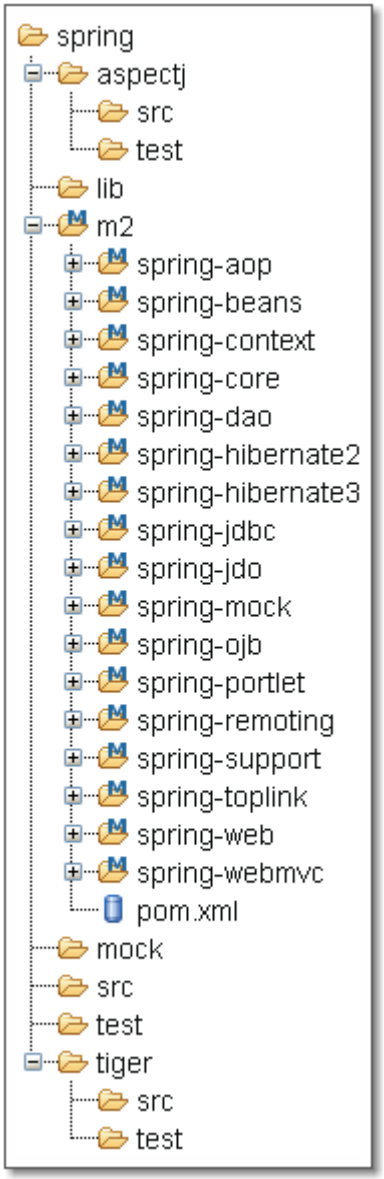


Figure 8-2: A sample spring module directory

In the `m2` directory, you will need to create a parent POM. You will use the parent POM to store the common configuration settings that apply to all of the modules. For example, each module will inherit the following values (settings) from the parent POM.

- `groupId`: this setting indicates your area of influence, company, department, project, etc., and it should mimic standard package naming conventions to avoid duplicate values. For this example, you will use `com.mergere.m2book.migrating`, as it is our 'unofficial' example version of Spring; however, the Spring team would use `org.springframework`
- `artifactId`: the setting specifies the name of this module (for example, `spring-parent`)
- `version`: this setting should always represent the next release version number appended with - *`SNAPSHOT`* – that is, the version you are developing in order to release. Recall from previous chapters that during the release process, Maven will convert to the definitive, non-snapshot version for a short period of time, in order to tag the release in your SCM.
- `packaging`: the `jar`, `war`, and `ear` values should be obvious to you (a `pom` value means that this project is used for metadata only)

The other values are not strictly required, primary used for documentation purposes.

```
<groupId>com.mergere.m2book.migrating</groupId>
<artifactId>spring-parent</artifactId>
<version>2.0-m1-SNAPSHOT</version>
<name>Spring parent</name>
<packaging>pom</packaging>
<description>Spring Framework</description>
<inceptionYear>2002</inceptionYear>
<url>http://www.springframework.org</url>
<organization>
 <name>The Spring Framework Project</name>
</organization>
```

In this parent POM we can also add dependencies such as JUnit, which will be used for testing in every module, thereby eliminating the requirement to specify the dependency repeatedly across multiple modules.

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

As explained previously, in Spring, the main source and test directories are `src` and `test`, respectively. Let's begin with these directories.

Using the following code snippet from Spring's Ant build script, in the `buildmain` target, you can retrieve some of the configuration parameters for the compiler.

```
<javac destdir="${target.classes.dir}" source="1.3" target="1.3" debug="${debug}"
    deprecation="false" optimize="false" failonerror="true">
 <src path="${src.dir}"/>
 <!-- Include Commons Attributes generated Java sources -->
 <src path="${commons.attributes.tempdir.src}"/>
 <classpath refid="all-libs"/>
</javac>
```

As you can see these include the source and target compatibility (**1.3**), deprecation and optimize (**false**), and `failonerror` (**true**) values. These last three properties use Maven's default values, so there is no need for you to add the configuration parameters.

For the debug attribute, Spring's Ant script uses a parameter, To specify the required debug function in Maven, you will need to append `-Dmaven.compiler.debug=false` to the `mvn` command (by default this is set to true). For now, you don't have to worry about the commons-attributes generated sources mentioned in the snippet, as you will learn about that later in this chapter. Recall from Chapter 2, that Maven automatically manages the classpath from its list of dependencies.

At this point, your build section will look like this:

```
<build>
 <sourceDirectory>../../src</sourceDirectory>
 <testSourceDirectory>../../test</testSourceDirectory>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
    <source>1.3</source>
    <target>1.3</target>
   </configuration>
  </plugin>
 </plugins>
</build>
```

The other configuration that will be shared is related to the JUnit tests. From the `tests` target in the Ant script:

```
<junit forkmode="perBatch" printsummary="yes" haltonfailure="yes"
haltonerror="yes">

 <jvmarg line="-Djava.awt.headless=true -XX:MaxPermSize=128m -Xmx128m"/>

 <!-- Must go first to ensure any jndi.properties files etc take precedence -->
 <classpath location="${target.testclasses.dir}"/>
 <classpath location="${target.mockclasses.dir}"/>
 <classpath location="${target.classes.dir}"/>

 <!-- Need files loaded as resources -->
 <classpath location="${test.dir}"/>

 <classpath refid="all-libs"/>

 <formatter type="plain" usefile="false"/>
 <formatter type="xml"/>

 <batchtest fork="yes" todir="${reports.dir}">
  <fileset dir="${target.testclasses.dir}" includes="${test.includes}"
excludes="${test.excludes}"/>
 </batchtest>
</junit>
```

You can extract some configuration information from the previous code:

- `forkMode`=”`perBatch`” matches with Maven's `forkMode` parameter with a value of once, since the concept of a batch for testing does not exist.
- You will not need any `printsummary`, `haltonfailure` and `haltonerror` settings, as Maven prints the test summary and stops for any test error or failure, by default.
- `formatter` elements are not required as Maven generates both plain `text` and `xml` reports.
- The nested element `jvmarg` is mapped to the configuration parameter `argLine`
- As previously noted, `classpath` is automatically managed by Maven from the list of dependencies.
- Maven sets the reports destination directory (`todir`) to `target/surefire-reports`, by default, and this doesn't need to be changed.
- You will need to specify the value of the properties `test.includes` and `test.excludes` from the nested `fileset`; this value is read from the `project.properties` file loaded from the Ant script (refer to the code snippet below for details).
- Maven uses the default value from the compiler plugin, so you will not need to locate the test classes directory (`dir`).

```
# Wildcards to be matched by JUnit tests.
# Convention is that our JUnit test classes have XXXTests-style names.
test.includes=**/*Tests.class
#
# Wildcards to exclude among JUnit tests.
# Second exclude needs to be used for JDK 1.3, due to Hibernate 3.1
# being compiled with target JDK 1.4.
test.excludes=**/Abstract*
#test.excludes=**/Abstract* org/springframework/orm/hibernate3/**
```

The includes and excludes referenced above, translate directly into the include/exclude elements of the POM's plugin configuration.

> Since Maven requires JDK 1.4 to run you do not need to exclude `hibernate3` tests. Note that it is possible to use another lower JVM to run tests if you wish – refer to the Surefire plugin reference documentation for more information.

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-plugin</artifactId>
 <configuration>
  <forkMode>once</forkMode>
  <childDelegation>false</childDelegation>
  <argLine>
   -Djava.awt.headless=true -XX:MaxPermSize=128m -Xmx128m
  </argLine>
  <includes>
   <include>**/*Tests.class</include>
  </includes>
  <excludes>
   <exclude>**/Abstract*</exclude>
  </excludes>
 </configuration>
</plugin>
```

The `childDelegation` option is required to prevent conflicts when running under Java 5 between the XML parser provided by the JDK and the one included in the dependencies in some modules, mandatory when building in JDK 1.4. It makes tests run using the standard `classloader` delegation instead of the default Maven isolated `classloader`. When building only on Java 5 you could remove that option and the XML parser (`Xerces`) and APIs (`xml-apis`) dependencies.

Spring's Ant build script also makes use of the commons-attributes compiler in its `compileattr` and `compiletestattr` targets, which are processed prior to the compilation. The commons-attributes compiler processes javadoc style annotations – it was created before Java supported annotations in the core language on JDK 1.5 - and generates sources from them that have to be compiled with the normal Java compiler.

From `compileattr`:

```
<!-- Compile to a temp directory: Commons Attributes will place Java Source here.
-->
<attribute-compiler destdir="${commons.attributes.tempdir.src}">
 <!--
 Only the PathMap attribute in the org.springframework.web.servlet.handler.metadata
 package currently needs to be shipped with an attribute, to support indexing.
 -->
 <fileset dir="${src.dir}" includes="**/metadata/*.java"/>
</attribute-compiler>
```

From `compiletestattr`:

```
<!-- Compile to a temp directory: Commons Attributes will place Java Source here.
-->
<attribute-compiler destdir="${commons.attributes.tempdir.test}">
 <fileset dir="${test.dir}" includes="org/springframework/aop/**/*.java"/>
 <fileset dir="${test.dir}" includes="org/springframework/jmx/**/*.java"/>
</attribute-compiler>
```

In Maven, this same function can be accomplished by adding the commons-attributes plugin to the build section in the POM. Maven handles the source and destination directories automatically, so you will only need to add the inclusions for the main source and test source compilation.

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>commons-attributes-maven-plugin</artifactId>
 <executions>
  <execution>
   <configuration>
    <includes>
     <include>**/metadata/*.java</include>
    </includes>
    <testIncludes>
     <include>org/springframework/aop/**/*.java</include>
     <include>org/springframework/jmx/**/*.java</include>
    </testIncludes>
   </configuration>
   <goals>
    <goal>compile</goal>
    <goal>test-compile</goal>
    </goals>
  </execution>
 </executions>
</plugin>
```

Later in this chapter you will need to modify these test configurations.

8.3. Creating POM files

Now that you have the basic configuration shared by all modules (project information, compiler configuration, JUnit test configuration, etc.), you need to create the POM files for each of Spring's modules. In each subdirectory, you will need to create a POM that extends the parent POM.

The following is the POM for the spring-core module. This module is the best to begin with because all of the other modules depend on it.

```
<parent>
 <groupId>com.mergere.m2book.migrating</groupId>
 <artifactId>spring-parent</artifactId>
 <version>2.0-m1-SNAPSHOT</version>
</parent>
<artifactId>spring-core</artifactId>
<name>Spring core</name>
```

Again, you won't need to specify the version or `groupId` elements of the current module, as those values are inherited from parent POM, which centralizes and maintains information common to the project.

8.4. Compiling

In this section, you will start to compile the main Spring source; tests will be dealt with later in the chapter. To begin, review the following code snippet from Spring's Ant script, where `spring-core` JAR is created:

```
<jar jarfile="${dist.dir}/modules/spring-core.jar">
  <fileset dir="${target.classes.dir}">
    <include name="org/springframework/core/**"/>
    <include name="org/springframework/util/**"/>
  </fileset>
  <manifest>
    <attribute name="Implementation-Title" value="${spring-title}"/>
  <attribute name="Implementation-Version" value="${spring-version}"/>
    <attribute name="Spring-Version" value="${spring-version}"/>
  </manifest>
</jar>
```

From the previous code snippet, you can determine which classes are included in the JAR and what attributes are written into the JAR's manifest. Maven will automatically set manifest attributes such as name, version, description, and organization name to the values in the POM. While manifest entries can also be customized with additional configuration to the JAR plugin, in this case the defaults are sufficient. However, you will need to tell Maven to pick the correct classes and resources from the `core` and `util` packages.

For the resources, you will need to add a `resources` element in the build section, setting the files you want to include (by default Maven will pick everything from the resource directory). As you saw before, since the sources and resources are in the same directory in the current Spring build, you will need to exclude the `*.java` files from the resources, or they will get included in the JAR.

For the classes, you will need to configure the compiler plugin to include only those in the `core` and `util` packages, because as with resources, Maven will by default compile everything from the source directory, which is inherited from the parent POM.

```
<build>
 <resources>
  <resource>
   <directory>../../src</directory>
   <includes>
     <include>org/springframework/core/**</include>
     <include>org/springframework/util/**</include>
   </includes>
   <excludes>
     <exclude>**/*.java</exclude>
   </excludes>
  </resource>
 </resources>
 <plugins>
 <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
    <includes>
     <include>org/springframework/core/**</include>
     <include>org/springframework/util/**</include>
    </includes>
   </configuration>
  </plugin>
 </plugins>
</build>
```

To compile your Spring build, you can now run `mvn compile`. You will see a long list of compilation failures, beginning with the following:

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\core\io\support\PathMatchingResourcePatternResol
ver.java:[30,34] package org.apache.commons.logging does not exist

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\core\io\support\PathMatchingResourcePatternResol
ver.java:[31,34] package org.apache.commons.logging does not exist

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\core\io\support\PathMatchingResourcePatternResol
ver.java:[107,24] cannot find symbol
symbol : class Log
location: class
org.springframework.core.io.support.PathMatchingResourcePatternResolver

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\util\xml\SimpleSaxErrorHandler.java:[19,34]
package org.apache.commons.logging does not exist
```

These are typical compiler messages, caused by the required classes not being on the classpath.

From the previous output, you now know that you need the Apache Commons Logging library (`commons-logging`) to be added to the dependencies section in the POM. But, what `groupid`, `artifactId` and version should we use?

For the `groupId` and `artifactId`, you need to check the central repository at ibiblio. Typically, the convention is to use a `groupId` that mirrors the package name, changing dots to slashes. For example, `commons-logging` `groupId` would become `org.apache.commons.logging`, located in the `org/apache/commons/logging` directory in the repository.

However, for historical reasons some `groupId` values don't follow this convention and use only the name of the project. In the case of commons-logging, the actual `groupId` is commons-logging.

Regarding the `artifactId`, it's usually the JAR name without a version (in this case commons-logging). If you check the repository, you will find all the available versions of commons-logging under http://www.ibiblio.org/maven2/commons-logging/commons-logging/.

As an alternative, you can search the repository using Google. Specify `site:www.ibiblio.org/maven2` commons logging, and then choose from the search results, the option that is closest to what is required by your project.

With regard the version, you will find that there is documentation for all of Spring's dependencies in `readme.txt` in the lib directory of the Spring source. You can use this as a reference to determine the versions of each of the dependencies. However, you have to be careful as the documentation may contain mistakes and/or inaccuracies. For instance, during the process of migrating Spring to Maven, we discovered that the commons-beanutils version stated in the documentation is wrong and that some required dependencies are missing from the documentation.

```
<dependencies>
 <dependency>
  <groupId>commons-logging</groupId>
   <artifactId>commons-logging</artifactId>
  <version>1.0.4</version>
 </dependency>
</dependencies>
```

Usually you will convert your own project, so you will have first hand knowledge about the dependencies and versions used. When needed, there are some other options to try to determine the appropriate versions for the dependencies included in your build:

- Check if the JAR has the version in the file name
- Open the JAR file and look in the manifest file `META-INF/MANIFEST.MF`
- For advanced users, search the ibiblio repository through Google by calculating the MD5 checksum of the JAR file with a program such as md5sum, and then search in Google pre-pending `site:www.ibiblio.org/maven2` to the query. For example, for the `hibernate3.jar` provided with Spring under lib/hibernate, you could search with: site:www.ibiblio.org/maven2 78d5c38f1415efc64f7498f828d8069a

  The search will return: www.ibiblio.org/maven2/org/hibernate/hibernate/3.1/hibernate-3.1.jar.md5

You can see that the last directory is the version (3.1), the previous directory is the `artifactId` (hibernate) and the other directories compose the `groupId`, with the slashes changed to dots (`org.hibernate`)

An easier way to search for dependencies, using a web interface, is under development (refer to the Maven Repository Manager project for details).

While adding dependencies can be the most painful part of migrating to Maven, explicit dependency management is one of the biggest benefits of Maven once you have invested the effort upfront. So, although you could simply follow the same behavior used in Ant (by adding all the dependencies in the parent POM so that, through inheritance, all sub-modules would use the same dependencies), we strongly encourage and recommend that you invest the time at the outset of your migration, to make explicit the dependencies and interrelationships of your projects. Doing so will result in cleaner, component-oriented, modular projects that are easier to maintain in the long term.

Running again `mvn` compile and repeating the process previously outlined for commons-logging, you will notice that you also need Apache Commons Collections (aka commons-collections) and `log4j`.

```
<dependency>
 <groupId>commons-collections</groupId>
 <artifactId>commons-collections</artifactId>
 <version>3.1</version>
</dependency>
<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.9</version>
 <optional>true</optional>
</dependency>
```

Notice that `log4j` is marked as optional. Optional dependencies are not included transitively, so log4j will not be included in other projects that depend on this. This is because in other projects, you may decide to use another log implementation, and it is just for the convenience of the users. Using the optional tag does not affect the current project.

Now, run `mvn compile` again - this time all of the sources for spring-core will compile.

8.5. Testing

Now you're ready to compile and run the tests. For the first step, you will repeat the previous procedure for the main classes, setting which test resources to use, and setting the JUnit test sources to compile. After compiling the tests, we will cover how to run the tests.

8.5.1. Compiling Tests

Setting the test resources is identical to setting the main resources, with the exception of changing the location from which the element name and directory are pulled. In addition, you will need to add the log4j.properties file required for logging configuration.

```
<testResources>
 <testResource>
  <directory>../../test</directory>
  <includes>
    <include>log4j.properties</include>
    <include>org/springframework/core/**</include>
    <include>org/springframework/util/**</include>
  </includes>
  <excludes>
    <exclude>**/*.java</exclude>
  </excludes>
 </testResource>
</testResources>
```

Setting the test sources for compilation follows the same procedure, as well. Inside the maven-compiler-plugin configuration, you will need to add the testIncludes element.

```
<testIncludes>
 <include>org/springframework/core/**</include>
 <include>org/springframework/util/**</include>
</testIncludes>
```

Now, if you try to compile the test classes by running mvn test-compile, as before, you will get compilation errors, but this time there is a special case where the compiler complains because some of the classes from the org.springframework.mock, org.springframework.web and org.springframework.beans packages are missing. It may appear initially that spring-core depends on spring-mock, spring-web and spring-beans modules, but if you try to compile those other modules, you will see that their main classes, not tests, depend on classes from spring-core. As a result, we cannot add a dependency from spring-core without creating a circular dependency. In other words, if spring-core depends on spring-beans and spring-beans depends on spring-core, which one do we build first? Impossible to know.

So, the key here is to understand that some of the test classes are not actually unit tests for spring-core, but rather require other modules to be present. Therefore, it makes sense to exclude all the test classes that reference other modules from this one and include them elsewhere.

To exclude test classes in Maven, add the testExcludes element to the compiler configuration as follows.

```
<testExcludes>
 <exclude>org/springframework/util/comparator/ComparatorTests.java</exclude>
 <exclude>org/springframework/util/ClassUtilsTests.java</exclude>
 <exclude>org/springframework/util/ObjectUtilsTests.java</exclude>
 <exclude>org/springframework/util/ReflectionUtilsTests.java</exclude>
 <exclude>org/springframework/util/SerializationTestUtils.java</exclude>
 <exclude>org/springframework/core/io/ResourceTests.java</exclude>
</testExcludes>
```

Now, when you run mvn test-compile, you will see the following error:

```
package javax.servlet does not exist
```

This means that the following dependency must be added to the POM, in order to compile the tests:

```
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>servlet-api</artifactId>
 <version>2.4</version>
 <scope>test</scope>
</dependency>
```

The scope is set to test, as this is not needed for the main sources.

If you run mvn test-compile again you will have a successful build, as all the test classes compile correctly now.

8.5.2. Running Tests

Running the tests in Maven, simply requires running `mvn test`. However, when you run this command, you will get the following error report:

```
Results :
[surefire] Tests run: 113, Failures: 1, Errors: 1

[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] There are test failures.
[INFO] ------------------------------------------------------------------------
```

Upon closer examination of the report output, you will find the following:

```
[surefire] Running
org.springframework.core.io.support.PathMatchingResourcePatternResolverTests
[surefire] Tests run: 5, Failures: 1, Errors: 1, Time elapsed: 0.015 sec <<<<<<<<
FAILURE !!
```

This output means that this test has logged a JUnit failure and error. To debug the problem, you will need to check the test logs under target/surefire-reports, for the test class that is failing `org.springframework.core.io.support.PathMatchingResourcePatternResolverTests.txt`. Within this file, there is a section for each failed test called `stacktrace`.

The first section starts with `java.io.FileNotFoundException: class path resource [org/aopalliance/]` cannot be resolved to URL because it does not exist.

This indicates that there is something missing in the classpath that is required to run the tests. The `org.aopalliance` package is inside the `aopallience` JAR, so to resolve the problem add the following to your POM

```
<dependency>
 <groupId>aopalliance</groupId>
 <artifactId>aopalliance</artifactId>
 <version>1.0</version>
 <scope>test</scope>
</dependency>
```

Now run `mvn test` again. You will get the following wonderful report:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

The last step in migrating this module (spring-core) from Ant to Maven, is to run `mvn install` to make the resulting JAR available to other projects in your local Maven repository. This command can be used instead most of the time, as it will process all of the previous phases of the build life cycle (generate sources, compile, compile tests, run tests, etc.)

8.6. Other Modules

Now that you have one module working it is time to move on to the other modules. If you follow the order of the modules described at the beginning of the chapter you will be fine, otherwise you will find that the main classes from some of the modules reference classes from modules that have not yet been built. See figure 8-1 to get the overall picture of the interdependencies between the Spring modules.

8.6.1. Avoiding Duplication

As soon as you begin migrating the second module, you will find that you are repeating yourself. For instance, you will be adding the Surefire plugin configuration settings repeatedly for each module that you convert. To avoid duplication, move these configuration settings to the parent POM instead. That way, each of the modules will be able to inherit the required Surefire configuration.

In the same way, instead of repeatedly adding the same dependency version information to each module, use the parent POM's `dependencyManagement` section to specify this information once, and remove the versions from the individual modules (see Chapter 3 for more information).

Using the parent POM to centralize this information makes it possible to upgrade a dependency version across all sub-projects from a single location.

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>commons-logging</groupId>
   <artifactId>commons-logging</artifactId>
   <version>1.0.4</version>
  </dependency>
</dependencyManagement>
```

The following are some variables that may also be helpful to reduce duplication:

- `${project.version}`: version of the current POM being built
- `${project.groupId}`: `groupId` of the current POM being built

For example, you can refer to `spring-core` from `spring-beans` with the following, since they have the same `groupId` and `version`:

```
<dependency>
 <groupId>${project.groupId}</groupId>
 <artifactId>spring-core</artifactId>
 <version>${project.version}</version>
</dependency>
```

8.6.2. Referring to Test Classes from Other Modules

If you have tests from one component that refer to tests from other modules, there is a procedure you can use. Although it is typically not recommended, in this case it is necessary to avoid refactoring the test source code. First, make sure that when you run `mvn install`, that a JAR that contains the test classes is also installed in the repository:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 <executions>
  <execution>
   <goals>
    <goal>test-jar</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

Once that JAR is installed, you can use it as a dependency for other components, by specifying the `test-jar` type. However, be sure to put that JAR in the test scope as follows:

```
<dependency>
 <groupId>${project.groupId}</groupId>
 <artifactId>spring-beans</artifactId>
 <version>${project.version}</version>
 <type>test-jar</type>
 <scope>test</scope>
</dependency>
```

A final note on referring to test classes from other modules: if you have all of Spring's mock classes inside the same module, this can cause previously-described cyclic dependencies problem. To eliminate this problem, you can split Spring's mock classes into `spring-context-mock`, with only those classes related to `spring-context` module, and `spring-web-mock`, with only those classes related to `spring-web`. Generally with Maven, it's easier to deal with small modules, particularly in light of transitive dependencies.

8.6.3. Building Java 5 Classes

Some of Spring's modules include Java 5 classes from the tiger folder. As the compiler plugin was earlier configured to compile with Java 1.3 compatibility, how can the Java 1.5 sources be added? To do this with Maven, you need to create a new module with only Java 5 classes instead of adding them to the same module and mixing classes with different requirements. So, you will need to create a new `spring-beans-tiger` module.

Consider that if you include some classes compiled for Java 1.3 and some compiled for Java 5 in the same JAR, any users, attempting to use one of the Java 5 classes under Java 1.3 or 1.4, would experience runtime errors. By splitting them into different modules, users will know that if they depend on the module composed of Java 5 classes, they will need to run them under Java 5.

As with the other modules that have been covered, the Java 5 modules will share a common configuration for the compiler. The best way to split them is to create a tiger folder with the Java 5 parent POM, and then a directory for each one of the individual tiger modules, as follows:

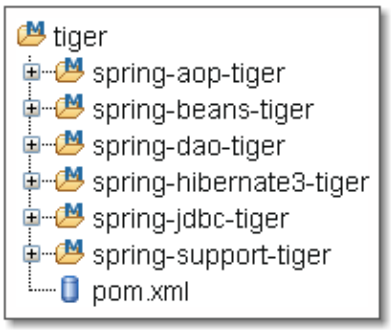


Figure 8-3: A tiger module directory

The final directory structure should appear as follows:

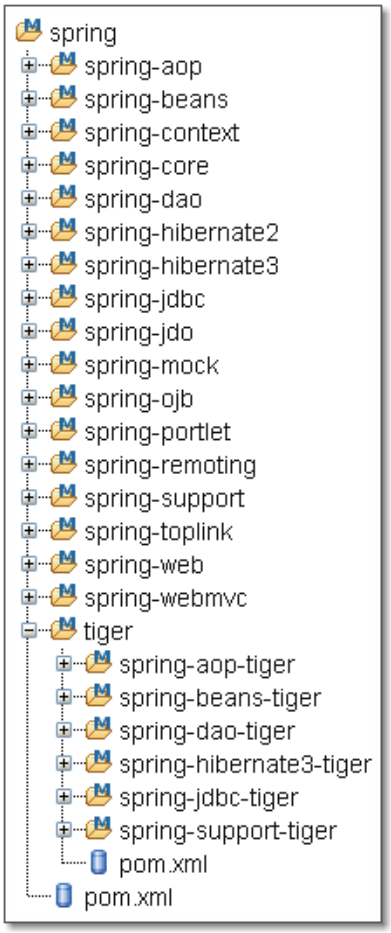


Figure 8-4: The final directory structure

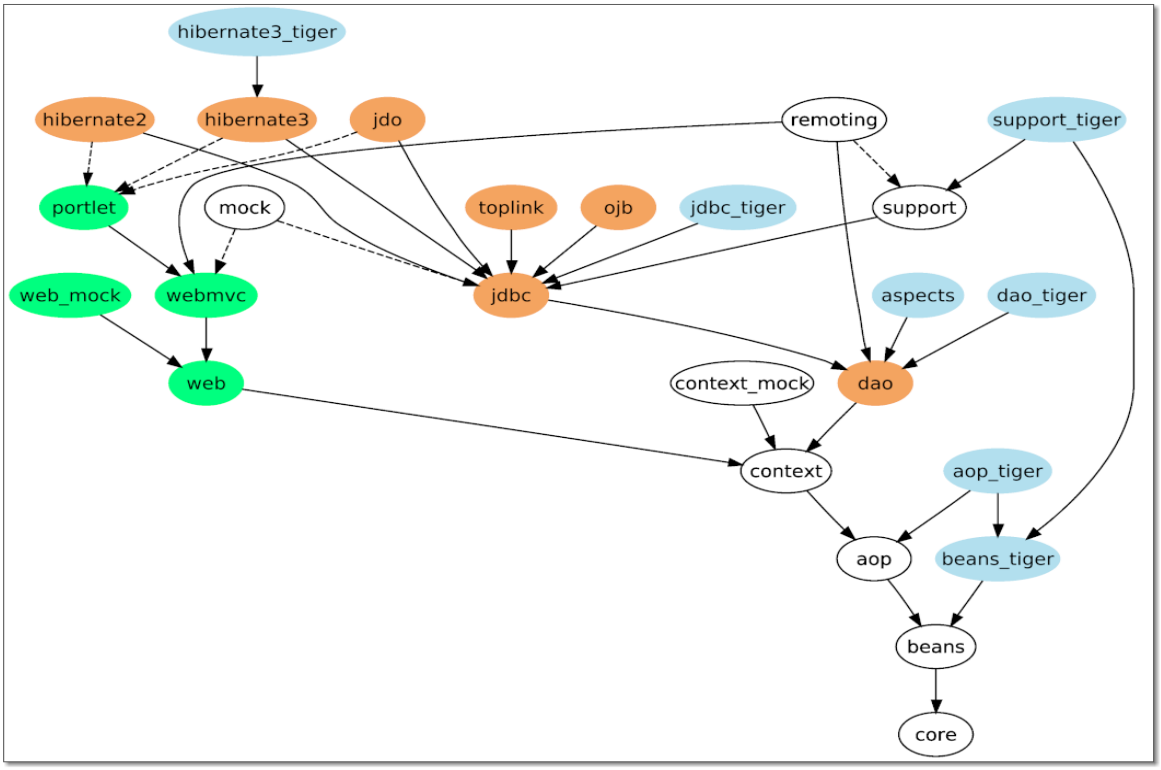


Figure 8-5: Dependency relationship, with all modules

In the tiger POM, you will need to add a module entry for each of the directories, as well as build sections with source folders and compiler options:

```
<modules>
 <module>spring-beans-tiger</module>
 <module>spring-aop-tiger</module>
 <module>spring-dao-tiger</module>
 <module>spring-jdbc-tiger</module>
 <module>spring-support-tiger</module>
 <module>spring-hibernate3-tiger</module>
 <module>spring-aspects</module>
</modules>
<build>
 <sourceDirectory>../../../tiger/src</sourceDirectory>
 <testSourceDirectory>../../../tiger/test</testSourceDirectory>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
    <source>1.5</source>
    <target>1.5</target>
   </configuration>
  </plugin>
 </plugins>
</build>
```

In the parent POM, you just need a new module entry for the tiger folder, but to still be able to build the other modules when using Java 1.4 you will add that module in a profile that will be triggered only when using 1.5 JDK.

```
<profiles>
 <profile>
  <id>jdk1.5</id>
  <activation>
   <jdk>1.5</jdk>
  </activation>
  <modules>
   <module>tiger</module>
  </modules>
 </profile>
</profiles>
```

8.6.4. Using Ant Tasks From Inside Maven

In certain migration cases, you may find that Maven does not have a plugin for a particular task or an Ant target is so small that it may not be worth creating a new plugin. In this case, Maven can call Ant tasks directly from a POM using the `maven-antrun-plugin`.

For example, with the Spring migration, you need to use the Ant task in the `spring-remoting` module to use the RMI compiler.

From Ant, this is:

```
<rmic base="${target.classes.dir}"
  classname="org.springframework.remoting.rmi.RmiInvocationWrapper"/>

<rmic base="${target.classes.dir}"
  classname="org.springframework.remoting.rmi.RmiInvocationWrapper" iiop="true">
  <classpath refid="all-libs"/>
</rmic>
```

To include this in Maven build, add:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
   <execution>
    <phase>process-classes</phase>
    <configuration>
     <tasks>
      <echo>Running rmic</echo>
      <rmic base="${project.build.directory}/classes"
            classname="org.springframework.remoting.rmi.RmiInvocationWrapper"/>
      <rmic base="${project.build.directory}/classes"
            classname="org.springframework.remoting.rmi.RmiInvocationWrapper"
            iiop="true">
       <classpath refid="maven.compile.classpath"/>
      </rmic>
     </tasks>
    </configuration>
    <goals>
     <goal>run</goal>
    </goals>
   </execution>
  </executions>
  <dependencies>
   <dependency>
    <groupId>com.sun</groupId>
    <artifactId>tools</artifactId>
    <scope>system</scope>
    <version>1.4</version>
    <systemPath>${java.home}/../lib/tools.jar</systemPath>
   </dependency>
  </dependencies>
 </plugin>
```

As shown in the code snippet above, there are some references available already, such as `${project.build.directory}` and `maven.compile.classpath`, which is a classpath reference constructed from all of the dependencies in the compile scope or lower. There are also references for anything that was added to the plugin's dependencies section, which applies to that plugin only, such as the reference to the `tools.jar` above, which is bundled with the JDK, and required by the RMI task.

To complete the configuration, you will need to determine when Maven should run the Ant task. In this case, the `rmic` task, will take the compiled classes and generate the `rmi` skeleton, stub and tie classes from them. So, the most appropriate phase in which to run this Ant task is in the process-classes phase.

8.6.5. Non-redistributable Jars

You will find that some of the modules in the Spring build depend on JARs that are not available in the Maven central repository. For example, Sun's Activation Framework and JavaMail are not redistributable from the repository due to constraints in their licenses. You may need to download them yourself from the Sun site or get them from the lib directory in the example code for this chapter. You can then install them in your local repository with the following command.

```
mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id>
-DartifactId=<artifact-id> -Dversion=<version> -Dpackaging=<packaging>
```

For instance, to install JavaMail:

```
mvn install:install-file -Dfile=mail.jar -DgroupId=javax.mail
-DartifactId=mail -Dversion=1.3.2 -Dpackaging=jar
```

You will only need to do this process once for all of your projects or you may use a corporate repository to share them across your organization. For more information on dealing with this issue, see http://maven.apache.org/guides/mini/guide-coping-with-sun-jars.html.

8.6.6. Some Special Cases

In addition to the procedures outlined previously for migrating Spring to Maven, there are two additional, special cases that must be handled. These issues were shared with the Spring developer community and are listed below:

- Moving one test class, `NamespaceHandlerUtilsTests`, from test directory to `tiger/test_` directory – as it depends on a tiger class
- Fixing toplink tests that don't compile against the oracle toplink jar (Spring developers use a different version than the official one from Oracle)

In this example it is necessary to comment out two unit tests, which used relative paths in `Log4JConfigurerTests` class, as these test cases will not work in both Maven and Ant. Using classpath resources is recommended over using file system resources.

> There is some additional configuration required for some modules, such as `spring-aspects`, which uses AspectJ for weaving the classes. These can be viewed in the example code.

8.7. Restructuring the Code

If you finally decide to stick with Maven, you are encouraged to use the default directory structure to take advantage of

For example, for the `spring-core` module, you would move all Java files under `org/springframework/core` and `org/springframework/util` from the original `src` folder to the module's folder `src/main/java`.

All of the other files under those two packages would go to `src/main/resources`. The same for tests, these would move from the original test folder to `src/test/java` and `src/test/resources` respectively for Java sources and other files - just remember not to move the excluded tests (`ComparatorTests`, `ClassUtilsTests`, `ObjectUtilsTests`, `ReflectionUtilsTests`, `SerializationTestUtils` and `ResourceTests`).

By adopting Maven's standard directory structure, you can simplify the POM significantly, reducing its size by two-thirds!

8.8. Summary

By following and completing this chapter, you will be able to take an existing Ant-based build, split it into modular components (if needed), compile and test the code, create JARs, and install those JARs in your local repository using Maven. At the same time, you will be able to keep your current build working. Once you decide to switch completely to Maven, you will be able to take advantage of the benefits of adopting Maven's standard directory structure. By doing this, you would eliminate the need to include and exclude sources and resources “by hand” in the POM files as shown in this chapter.

Once you have spent this initial setup time Maven, you can realize Maven' other benefits - advantages such as built-in project documentation generation, reports, and quality metrics.

Finally, in addition to the improvements to your build life cycle, Maven can eliminate the requirement of storing jars in a source code management system. In the case of the Spring example, as Maven downloads everything it needs and shares it across all your Maven projects automatically - you can delete that 80 MB lib folder.

Now that you have seen how to do this for Spring, you can apply similar concepts to your own Ant based build.

Appendix A: Resources for Plugin Developers

In this appendix you will find:

- Mavens Life Cycles
- Mojo Parameter Expressions
- Plugin Metadata

Scotty: She's all yours, sir. All systems automated and ready. A chimpanzee and two trainees could run her!

Kirk: Thank you, Mr. Scott, I'll try not to take that personally.

- Star Trek

A.1. Maven's Life Cycles

Below is a discussion of Maven's three life cycles and their default mappings. It begins by listing the phases in each life cycles, along with a short description for the mojos which should be bound to each. It continues by describing the mojos bound to the default life cycle for both the jar and maven-plugin packagings. Finally, this section will describe the mojos bound by default to the clean and site life cycles.

A.1.1. The `default` Life Cycle

Maven provides three life cycles, corresponding to the three major activities performed by Maven: building a project from source, cleaning a project of the files generated by a build, and generating a project web site.

For the default life cycle, mojo-binding defaults are specified in a packaging-specific manner. This is necessary to accommodate the inevitable variability of requirements for building different types of projects. This section contains a listing of the phases in the default life cycle, along with a summary of bindings for the jar and maven-plugin packagings.

Life-cycle phases

The default life cycle is executed in order to perform a traditional build. In other words, it takes care of compiling the project's code, performing any associated tests, archiving it into a jar, and distributing it into the Maven repository system. It contains the following phases:

1. `validate` – verify that the configuration of Maven, and the content of the current set of POMs to be built is valid.
2. `initialize` – perform any initialization steps required before the main part of the build can start.
3. `generate-sources` – generate compilable code from other source formats.
4. `process-sources` – perform any source modification processes necessary to prepare the code for compilation. For example, a mojo may apply source code patches here.
5. `generate-resources` – generate non-code resources (such as configuration files, etc.) from other source formats.
6. `process-resources` – perform any modification of non-code resources necessary. This may include copying these resources into the target classpath directory in a Java build.
7. `compile` – compile source code into binary form, in the target output location.
8. `process-classes` – perform any post-processing of the binaries produced in the preceding step, such as instrumentation or offline code-weaving, as when using Aspect-Oriented Programming techniques.
9. `generate-test-sources` – generate compilable unit test code from other source formats.

10. `process-test-sources` – perform any source modification processes necessary to prepare the unit test code for compilation. For example, a mojo may apply source code patches here.
11. `generate-test-resources` – generate non-code testing resources (such as configuration files, etc.) from other source formats.
12. `process-test-resources` – perform any modification of non-code testing resources necessary. This may include copying these resources into the testing target classpath location in a Java build.
13. `test-compile` – compile unit test source code into binary form, in the testing target output location.
14. `test` – execute unit tests on the application compiled and assembled up to step 8 above.
15. `package` – assemble the tested application code and resources into a distributable archive.
16. `preintegration-test` – setup the integration testing environment for this project. This may involve installing the archive from the preceding step into some sort of application server.
17. `integration-test` – execute any integration tests defined for this project, using the environment configured in the preceding step.
18. `post-integration-test` – return the environment to its baseline form after executing the integration tests in the preceding step. This could involve removing the archive produced in step 15 from the application server used to test it.
19. `verify` – verify the contents of the distributable archive, before it is available or installation or deployment.
20. `install` – install the distributable archive into the local Maven repository.
21. `deploy` – deploy the distributable archive into the remove Maven repository configured in the `distributionManagement` section of the POM.

Bindings for the `jar` packaging

Below are the default life-cycle bindings for the jar packaging. Alongside each, you will find a short description of what that mojo does.

Table A-1: The default life-cycle bindings for the jar packaging

Phase	Mojo	Plugin	Description
process-resources	resources	maven-resources-plugin	Copy non-source-code resources to the staging directory for jar creation. Filter variables if necessary.
compile	compile	maven-compiler-plugin	Compile project source code to the staging directory for jar creation.
process-test-resources	testResources	maven-resources-plugin	Copy non-source-code test resources to the test output directory for unit-test compilation.
test-compile	testCompile	maven-compiler-plugin	Compile unit-test source code to the test output directory.
test	test	maven-surefire-plugin	Execute project unit tests.
package	jar	maven-jar-plugin	Create a jar archive from the staging directory.
install	install	maven-install-plugin	Install the jar archive into the local Maven repository.
deploy	deploy	maven-deploy-plugin	Deploy the jar archive to a remote Maven repository, specified in the POM distribution Management section.

Bindings for the `maven-plugin` packaging

The maven-plugin project packaging behaves in almost the same way as the more common jar packaging. Indeed, maven-plugin artifacts are in fact jar files. As such, they undergo the same basic processes of marshaling non-source-code resources, compiling source code, testing, packaging, and the rest. However, the maven-plugin packaging also introduces a few new mojo bindings, to extract and format the metadata for the mojos within. Below is a summary of the *additional* mojo bindings provided by the maven-plugin packaging:

Table A-2: A summary of the additional mojo bindings

Phase	Mojo	Plugin	Description
`generate-resources`	`descriptor`	`maven-plugin-plugin`	Extract mojo metadata (from javadoc annotations, for example), and generate a plugin descriptor.
`package`	`addPluginArtifactMetadata`	`maven-plugin-plugin`	Integrate current plugin information with plugin search metadata, and metadata references to latest plugin version.
`install`	`updateRegistry`	`maven-plugin-plugin`	Update the plugin registry, if one exists, to reflect the new plugin installed in the local repository.

A.1.2. The `clean` Life Cycle

This life cycle is executed in order to restore a project back to some baseline state – usually, the state of the project before it was built. Maven provides a set of default mojo bindings for this life cycle, which perform the most common tasks involved in cleaning a project. Below is a listing of the phases in the clean life cycle, along with a summary of the default bindings, effective for all POM packagings.

Life-cycle phases

The clean life-cycle phase contains the following phases:

1. `pre-clean` – execute any setup or initialization procedures to prepare the project for cleaning
2. `clean` – remove all files that were generated during another build process
3. `post-clean` – finalize the cleaning process.

Default life-cycle bindings

Below are the clean life-cycle bindings for the jar packaging. Alongside each, you will find a short description of what that mojo does.

Table A-3: The clean life-cycle bindings for the jar packaging

Phase	Mojo	Plugin	Description
`clean`	`clean`	`maven-clean-plugin`	Remove the project build directory, along with any additional directories configured in the POM.

A.1.3. The `site` Life Cycle

This life cycle is executed in order to generate a web site for your project. It will run any reports that are associated with your project, render your documentation source files into HTML, and even deploy the resulting web site to your server. Maven provides a set of default mojo bindings for this life cycle, which perform the most common tasks involved in generating the web site for a project. Below is a listing of the phases in the site life cycle, along with a summary of the default bindings, effective for all POM packagings.

Life-cycle phases

The site life cycle contains the following phases:

1. `pre-site` – execute any setup or initialization steps to prepare the project for site generation
2. `site` – run all associated project reports, and render documentation into HTML
3. `post-site` – execute any actions required to finalize the site generation process, and prepare the generated web site for potential deployment
4. `site-deploy` – use the `distributionManagement` configuration in the project's POM to deploy the generated web site files to the web server.

Default Life Cycle Bindings

Below are the site life-cycle bindings for the jar packaging. Alongside each, you will find a short description of what that mojo does.

Table A-4: The site life-cycle bindings for the jar packaging

Phase	Mojo	Plugin	Description
`site`	`site`	`maven-site-plugin`	Generate all configured project reports, and render documentation source files into HTML.
`site-deploy`	`deploy`	`maven-site-plugin`	Deploy the generated web site to the web server path specified in the POM distribution Management section.

A.2. Mojo Parameter Expressions

Mojo parameter values are resolved by way of parameter expressions when a mojo is initialized. These expressions allow a mojo to traverse complex build state, and extract only the information it requires. This reduces the complexity of the code contained in the mojo, and often eliminates dependencies on Maven itself beyond the plugin API.

This section discusses the expression language used by Maven to inject build state and plugin configuration into mojos. It will summarize the root objects of the build state which are available for mojo expressions. Finally, it will describe the algorithm used to resolve complex parameter expressions.

Using the discussion below, along with the published Maven API documentation, mojo developers should have everything they need to extract the build state they require.

A.2.1. Simple Expressions

Maven's plugin parameter injector supports several primitive expressions, which act as a shorthand for referencing commonly used build state objects. They are summarized below:

Table A-5: Primitive expressions supported by Maven's plugin parameter

Expression	Type	Description
`${localRepository}`	`org.apache.maven.artifact.repository.ArtifactRepository`	This is a reference to the local repository used to cache artifacts during a Maven build.
`${session}`	`org.apache.maven.execution.MavenSession`	The current build session. This contains methods for accessing information about how Maven was called, in addition to providing a mechanism for looking up Maven components on-demand.
`${reactorProjects}`	`java.util.List<org.apache.maven.project.MavenProject>`	List of project instances which will be processed as part of the current build.
`${reports}`	`java.util.List<org.apache.maven.reporting.MavenReport>`	List of reports to be generated when the site life cycle executes.
`${executedProject}`	`org.apache.maven.project.MavenProject`	This is a cloned instance of the project instance currently being built. It is used for bridging results from forked life cycles back to the main line of execution.

A.2.2. Complex Expression Roots

In addition to the simple expressions above, Maven supports more complex expressions that traverse the object graph starting at some root object that contains build state. The valid root objects for plugin parameter expressions are summarized below:

Table A-6: A summary of the valid root objects for plugin parameter expressions

Expression Root	Type	Description
`${basedir}`	`java.io.File`	The current project's root directory.
`${project}`	`org.apache.maven.project.MavenProject`	Project instance which is currently being built.
`${settings}`	`org.apache.maven.settings.Settings`	The Maven settings, merged from `conf/settings.xml` in the maven application directory and from `.m2/settings.xml` in the user's home directory, unless specified otherwise.
`${plugin}`	`org.apache.maven.plugin.descriptor.PluginDescriptor`	The descriptor instance for the current plugin, including its dependency artifacts.

A.2.2. The Expression Resolution Algorithm

Plugin parameter expressions are resolved using a straightforward algorithm. First, if the expression matches one of the primitive expressions (mentioned above) **exactly**, then the value mapped to that expression is returned. No advanced navigation can take place using is such expressions.

Otherwise, the expression is split at each '.' character, rendering an array of navigational directions. The first is the root object, and must correspond to one of the roots mentioned above. This root object is retrieved from the running application using a hard-wired mapping, much like a primitive expression would. From there, the next expression part is used as a basis for reflectively traversing that object' state. During this process, an expression part named 'child' translates into a call to the `getChild()` method on that object, following standard JavaBeans naming conventions. The resulting value then becomes the new 'root' object for the next round of traversal, if there is one. Repeating this, successive expression parts will extract values from deeper and deeper inside the build state.

When there are no more expression parts, the value that was resolved last will be returned as the expression's value. If at some point the referenced object doesn't contain a property that matches the next expression part, this reflective lookup process is aborted.

If at this point Maven still has not been able to resolve a value for the parameter expression, it will attempt to find a value in one of two remaining places, resolved in this order:

1. **The POM properties**. If a user has specified a property mapping this expression to a specific value in the current POM, an ancestor POM, or an active profile, it will be resolved as the parameter value at this point.
2. **The system properties**. If the value is still empty, Maven will consult the current system properties. This includes properties specified on the command line using the `-D` command-line option.

If the parameter is still empty after these two lookups, then the string literal of the expression itself is used as the resolved value. Currently, Maven plugin parameter expressions do not support collection lookups, array index references, or method invocations that don't conform to standard JavaBean naming conventions.

Plugin metadata

Below is a review of the mechanisms used to specify metadata for plugins. It includes summaries of the essential plugin descriptor, as well the metadata formats which are translated into plugin descriptors from Java- and Ant-specific mojo source files.

Plugin descriptor syntax

The following is a sample plugin descriptor. Its syntax has been annotated to provide descriptions of the elements.

```
<plugin>
  <!-- The description element of the plugin's POM. -->
  <description>Sample Maven Plugin</description>

  <!-- These are the identity elements (groupId/artifactId/version)
   | from the plugin POM.
   |->
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-myplugin-plugin</artifactId>
<version>2.0-SNAPSHOT</version>

<!-- This element provides the shorthand reference for this plugin. For
 | instance, this plugin could be referred to from the command line using
 | the 'myplugin:' prefix.
 |->
<goalPrefix>myplugin</goalPrefix>

<!-- Tells Maven that the this plugin's configuration should be inherted from
 | a parent POM by default, unless the user specifies
 | <inherit>false</inherit>.
 |->
<inheritedByDefault>true</inheritedByDefault>

<!-- This is a list of the mojos contained within this plugin. -->
<mojos>
  <mojo>

    <!-- The name of the mojo. Combined with the 'goalPrefix' element above,
     | this name allows the user to invoke this mojo from the command line
     | using 'myplugin:do-something'.
     |->
    <goal>do-something</goal>

    <!-- Description of what this mojo does. -->
    <description>Do something cool.</description>

    <!-- This tells Maven to create a clone of the current project and
     | life cycle, then execute that life cycle up to the specified phase.
     | This is useful when the user will be invoking this mojo directly from
     | the command line, but the mojo itself has certain life-cycle
     | prerequisites.
     |->
    <executePhase>process-resources</executePhase>

    <!-- This is optionally used in conjunction with the executePhase element,
     | and specifies a custom life-cycle overlay that should be added to the
     | cloned life cycle before the specified phase is executed. This is
     | useful to inject specialized behavior in cases where the main life
     | cycle should remain unchanged.
     |->
    <executeLifecycle>myLifecycle</executeLifecycle>

    <!-- Ensure that this other mojo within the same plugin executes before
     | this one. It's restricted to this plugin to avoid creating inter-plugin
     | dependencies.
     |->
    <executeGoal>do-something-first</executeGoal>

    <!-- Which phase of the life cycle this mojo will bind to by default.
     | This allows the user to specify that this mojo be executed (via the
     | <execution> section of the plugin configuration in the POM), without
     | also having to specify which phase is appropriate for the mojo's
     | execution. It is a good idea to provide this, to give users a hint
     | at where this task should run.
     |->
    <phase>compile</phase>

    <!-- Tells Maven that this mojo can ONLY be invoked directly, via the
     | command line.
     |->
```

```
<requiresDirectInvocation>false</requiresDirectInvocation>

<!-- Tells Maven that a valid project instance must be present for this
 | mojo to execute.
 |->
<requiresProject>true</requiresProject>

<!-- Tells Maven that a valid list of reports for the current project are
 | required before this plugin can execute.
 |->
<requiresReports>false</requiresReports>

<!-- Determines how Maven will execute this mojo in the context of a
 | multimodule build. If a mojo is marked as an aggregator, it will only
 | execute once, regardless of the number of project instances in the
 | current build. Mojos that are marked as aggregators should use the
 | ${reactorProjects} expression to retrieve a list of the project
 | instances in the current build. If the mojo is not marked as an
 | aggregator, it will be executed once for each project instance in the
 | current build.
 |->
<aggregator>false</aggregator>

<!-- Some mojos cannot execute if they don't have access to a network
 | connection. If Maven is operating in offline mode, such mojos will
 | cause the build to fail. This flag controls whether the mojo requires
 | Maven to be online.
 |->
<requiresOnline>false</requiresOnline>

<!-- Tells Maven that the this plugin's configuration should be inherted
 | from a parent POM by default, unless the user specifies
 | <inherit>false</inherit>.
 |->
<inheritedByDefault>true</inheritedByDefault>

<!-- The class or script path (within the plugin's jar) for this mojo's
 | implementation.
 |->
<implementation>org.apache.maven.plugins.site.SiteDeployMojo</implementation>

<!-- The implementation language for this mojo. -->
<language>java</language>

<!-- This is a list of the parameters used by this mojo. -->
<parameters>
  <parameter>
    <!-- The parameter's name. In Java mojos, this will often reflect the
     | parameter field name in the mojo class.
     |->
    <name>inputDirectory</name>

    <!-- This is an optional alternate parameter name for this parameter.
     | It will be used as a backup for retrieving the parameter value.
     |->
    <alias>outputDirectory</alias>

    <!-- The Java type for this parameter. -->
    <type>java.io.File</type>

    <!-- Whether this parameter is required to have a value. If true, the
     | mojo (and the build) will fail when this parameter doesn't have a
     | value.
```

```
          |->
          <required>true</required>

          <!-- Whether this parameter's value can be directly specified by the
           | user, either via command-line or POM configuration. If set to
           | false, this parameter must be configured via some other section of
           | the POM, as in the case of the list of project dependencies.
           |->
          <editable>true</editable>

          <!-- Description for this parameter, specified in the javadoc comment
           | for the parameter field in Java mojo implementations.
           |->
          <description>This parameter does something important.</description>
        </parameter>
      </parameters>

      <!-- This is the operational specification of this mojo's parameters, as
       | compared to the descriptive specification above. Each parameter must
       | have an entry here that describes the parameter name, parameter type,
       | and the primary expression used to extract the parameter's value.
       |
       | The general form is:
       |    <param-name implementation="param-type">param-expr</param-name>
       |
       |->
      <configuration>
        <!-- For example, this parameter is named "inputDirectory", and it
         | expects a type of java.io.File. The expression used to extract the
         | parameter value is ${project.reporting.outputDirectory}.
         |->
        <inputDirectory
implementation="java.io.File">${project.reporting.outputDirectory}</inputDirectory>
      </configuration>

      <!-- This is the list of non-parameter component references used by this
       | mojo. Components are specified by their interface class name (role),
       | along with an optional classifier for the specific component instance
       | to be used (role-hint). Finally, the requirement specification tells
       | Maven which mojo-field should receive the component instance.
       |->
      <requirements>
        <requirement>
          <!-- Use a component of type:
                  org.apache.maven.artifact.manager.WagonManager
          |->
          <role>org.apache.maven.artifact.manager.WagonManager</role>

          <!-- Inject the component instance into the "wagonManager" field of
           | this mojo.
           |->
          <field-name>wagonManager</field-name>
        </requirement>
      </requirements>
    </mojo>
  </mojos>
</plugin>
```

A.2.4. Java Mojo Metadata: Supported Javadoc Annotations

The Javadoc annotations used to supply metadata about a particular mojo come in two types. Class-level annotations correspond to mojo-level metadata elements, and field-level annotations correspond to parameter-level metadata elements.

Class-level annotations

The table below summarizes the class-level javadoc annotations which translate into specific elements of the mojo section in the plugin descriptor.

Table A-7: A summary of class-level javadoc annotations

Descriptor Element	Javadoc Annotation	Values	Required?
`aggregator`	`@aggregator`	`true` or `false` (default is `false`)	No
`description`	N/A (class comment)	Anything	No (recommended)
`executePhase`, `executeLifecycle`, `executeGoal`	`@execute goal="mojo" phase="phase" lifecycle="lifecycle"`	Any valid mojo, phase, life cycle name.	No
`goal`	`@goal`	Alphanumeric, with dash ('-')	**Yes**
`phase`	`@phase`	Any valid phase name	No
`requiresDirectInvocation`	`@requiresDirectInvocation`	`true` or `false` (default is `false`)	No
`requiresProject`	`@requiresProject`	`true` or `false` (default is `true`)	No
`requiresReports`	`@requiresReports`	`true` or `false` (default is `false`)	No
`requiresOnline`	`@requiresOnline`	`true` or `false` (default is `false`)	No

Field-level annotations

The table below summarizes the field-level annotations which supply metadata about mojo parameters. These metadata translate into elements within the parameter, configuration, and requirements sections of a mojo's specification in the plugin descriptor.

Table A-8: Field-level annotations

Descriptor Element	Javadoc Annotation	Values	Required?
`alias, parameter-configuration section`	`@parameter expression="${expr}" alias="alias" default-value="val"`	Anything	**Yes**
`Requirements section`	`@component roleHint="someHint"`	`roleHint` is optional, and usually left blank	No
`required`	`@required`	None	No
`editable`	`@readonly`	None	No
`description`	`N/A (field comment)`	Anything	No (recommended)
`deprecated`	`@deprecated`	Alternative parameter	No

A.2.5. Ant Metadata Syntax

The following is a sample Ant-based mojo metadata file. Its syntax has been annotated to provide descriptions of the elements.

```
<pluginMetadata>
  <!-- Contains the list of mojos described by this metadata file. NOTE:
   | multiple mojos are allowed here, corresponding to the ability to map
   | multiple mojos into a single build script.
   |->
  <mojos>
    <mojo>
      <!-- The name for this mojo -->
      <goal>myGoal</goal>

      <!-- The default life-cycle phase binding for this mojo -->
      <phase>compile</phase>

      <!-- The dependency scope required for this mojo; Maven will resolve
       | the dependencies in this scope before this mojo executes.
       |->
      <requiresDependencyResolution>compile</requiresDependencyResolution>

      <!-- Whether this mojo requires a current project instance -->
      <requiresProject>true</requiresProject>

      <!-- Whether this mojo requires access to project reports -->
      <requiresReports>true</requiresReports>
```

```
<!-- Whether this mojo requires Maven to execute in online mode -->
<requiresOnline>true</requiresOnline>

<!-- Whether the configuration for this mojo should be inherited
 | from parent to child POMs by default.
 |->
<inheritByDefault>true</inheritByDefault>

<!-- Whether this mojo must be invoked directly from the command
 | line.
 |->
<requiresDirectInvocation>true</requiresDirectInvocation>

<!-- Whether this mojo operates as an aggregator -->
<aggregator>true</aggregator>

<!-- This describes the mechanism for forking a new life cycle to be
 | executed prior to this mojo executing.
 |->
<execute>
  <!-- The phase of the forked life cycle to execute -->
  <phase>initialize</phase>

  <!-- A named overlay to augment the cloned life cycle for this fork
   | only
   |->
  <lifecycle>mine</lifecycle>

  <!-- Another mojo within this plugin to execute before this mojo
   | executes.
   |->
  <goal>goal</goal>
</execute>

<!-- List of non-parameter application components used in this mojo -->
<components>
  <component>
    <!-- This is the type for the component to be injected. -->
    <role>org.apache.maven.artifact.resolver.ArtifactResolver</role>

    <!-- This is an optional classifier for which instance of a particular
     | component type should be used.
     |->
    <hint>custom</hint>
  </component>
</components>

<!-- The list of parameters this mojo uses -->
<parameters>
  <parameter>
    <!-- The parameter name. -->
    <name>nom</name>

    <!-- The property name used by Ant tasks to reference this parameter
     | value.
     |->
    <property>prop</property>

    <!-- Whether this parameter is required for mojo execution -->
    <required>true</required>
```

```
        <!-- Whether the user can edit this parameter directly in the POM
         | configuration or the command line
         |->
        <readonly>true</readonly>

        <!-- The expression used to extract this parameter's value -->
        <expression>${my.property}</expression>

        <!-- The default value provided when the expression won't resolve -->
        <defaultValue>${project.artifactId}</defaultValue>

        <!-- The Java type of this mojo parameter -->
        <type>org.apache.maven.project.MavenProject</type>

        <!-- An alternative configuration name for this parameter -->
        <alias>otherProp</alias>

        <!-- The description of this parameter -->
        <description>Test parameter</description>

        <!-- When this is specified, this element will provide advice for an
         | alternative parameter to use instead.
         |->
        <deprecated>Use something else</deprecated>
      </parameter>
    </parameters>

    <!-- The description of what the mojo is meant to accomplish -->
    <description>
      This is a test.
    </description>

    <!-- If this is specified, it provides advice on which alternative mojo
     | to use.
    |->
    <deprecated>Use another mojo</deprecated>
  </mojo>
 </mojos>
</pluginMetadata>
```

Appendix B: Standard Conventions

In this appendix you will find:

- Standard Directory Structure for Maven Project Content
- Maven's Super POM
- Maven's Default Build Life Cycle

Kirk: Do you want to know something?
Everybody's human.

Spock: I find that remark insulting.

- Star Trek

B.1. Standard Directory Structure

Table B-1: Standard directory layout for maven project content

Standard Location	Description
`pom.xml`	Maven's POM, which is always at the top-level of a project.
`LICENSE.txt`	A license file is encouraged for easy identification by users and is optional.
`README.txt`	A simple note which might help first time users and is optional.
`target/`	Directory for all generated output. This would include compiled classes, generated sources that may be compiled, the generated site or anything else that might be generated as part of your build.
`target/generated-sources/<plugin-id>`	Standard location for generated sources. For example, you may generate some sources from a JavaCC grammar.
`src/main/java/`	Standard location for application sources.
`src/main/resources/`	Standard location for application resources.
`src/main/filters/`	Standard location for resource filters.
`src/main/assembly/`	Standard location for assembly descriptors.
`src/main/config/`	Standard location for application configuration files.
`src/test/java/`	Standard location for test sources.
`src/test/resources/`	Standard location for test resources.
`src/test/filters/`	Standard location for test resource filters.

B.2. Maven's Super POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>
  <!-- Repository Conventions -->
  <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>
  <!-- Plugin Repository Conventions -->
  <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 Conventions -->
  <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 Conventions -->
  <reporting>
    <outputDirectory>target/site</outputDirectory>
  </reporting>
 ...
</project>
```

B.3. Maven's Default Build Life Cycle

Table B-2: Phases in Maven's life cycle

Phase	Description
validate	Validate the project is correct and all necessary information is available.
initialize	Initialize the build process.
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 byte code 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.
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.

Bibliography

Online Books

des Rivieres, Jim. *Evolving Java-based APIs.* June 8, 2001 - http://www.eclipse.org/eclipse/development/java-api-evolution.html

Bloch, Joshua. *Effective Java*. Sun Developer Network - http://java.sun.com/docs/books/effective/

Web Sites

Axis Building Java Classes from WSDL- http://ws.apache.org/axis/java/userguide.html#WSDL2JavaBuildingStubsSkeletonsAndDataTypesFromWSDL

Axis Tool Plugin - http://ws.apache.org/axis/java/

AxisTools Reference Documentation - http://mojo.codehaus.org/axistools-maven-plugin/

Cargo Containers Reference - http://cargo.codehaus.org/Containers

Cargo Container Deployments - http://cargo.codehaus.org/Deploying+to+a+running+container

Cargo Plugin Configuration Options - http://cargo.codehaus.org/Maven2+plugin

Cargo Merging War Files Plugin - http://cargo.codehaus.org/Merging+WAR+files

Cargo Reference Documentation - http://cargo.codehaus.org/

Checkstyle - http://checkstyle.sf.net/config.html

Checkstyle Available Checks - http://checkstyle.sf.net/availablechecks.html

Cobertura - http://cobertura.sf.net/

Clirr - http://clirr.sf.net/

Clover Plugin - http://maven.apache.org/plugins/maven-clover-plugin/

DBUnit Java API - http://dbunit.sourceforge.net/

EJB Plugin Documentation - http://maven.apache.org/plugins/maven-ejb-plugin/

ibiblio- www.ibiblio.com

Introduction to Archetypes - http://maven.apache.org/guides/introduction/introduction-to-archetypes.html

Introduction to the Build Life Cycle – Maven - http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Jdiff - http://mojo.codehaus.org/jdiff-maven-plugin

Jetty 6 Plugin Documentation - http://jetty.mortbay.org/jetty6/maven-plugin/index.html

Jester - http://jester.sf.net

J2EE Specification - http://java.sun.com/j2ee/reference/api/

Maven 2 Wiki - www.apache.maven.org

Maven Downloads - http://maven.apache.org/download.html

Maven Plugins - http://maven.apache.org/plugins/

Mojo - http://mojo.codehaus.org/

PMD Best Practices - http://pmd.sf.net/bestpractices.html

PMD Rulesets - http://pmd.sf.net/howtomakearuleset.html

POM Reference - http://maven.apache.org/maven-model/maven.html

Ruby on Rails - http://www.rubyonrails.org/

Simian - http://www.redhillconsulting.com.au/products/simian/

Tomcat Manager Web Application - http://tomcat.apache.org/tomcat-5.0-doc/manager-howto.html

Xdoclet - http://xdoclet.sourceforge.net/

XDoclet EjbDocletTask - http://xdoclet.sourceforge.net/xdoclet/ant/xdoclet/modules/ejb/EjbDocletTask.html

XDoclet Maven Plugin - http://mojo.codehaus.org/xdoclet-maven-plugin/

XDoclet Reference Documentation - http://xdoclet.sourceforge.net/xdoclet/ant/xdoclet/modules/ejb/EjbDocletTask.html

Xdoclet2 - http://xdoclet.codehaus.org

XDoclet2 Maven Plugin - http://xdoclet.codehaus.org/Maven2+Plugin