

Everything You Need to Know from Ideation to Deployment



Maven

The Definitive Guide

O'REILLY®

Sonatype

Maven: The Definitive Guide



Maven: The Definitive Guide serves as both an introduction and a comprehensive reference for Apache Maven, the tool that will transform the way your organization builds and manages application development. Written by members of Sonatype's engineering team—including Jason van Zyl, the creator of the Maven central repository—this book clearly explains why Maven is replacing Ant as the build tool of choice, not only for open source Java projects, but for applications in many other languages, including Scala, Ruby, and Groovy. It also provides the first in-depth overview of Maven 2.

The first half of this book introduces Maven by example, with a series of real-world, multimodule applications you can use as templates. The second half serves as a reference to a wide range of topics. You will learn to:

- Understand the Project Object Model (POM)
- Use the Nexus repository manager
- Integrate Maven with Eclipse, Spring, and Hibernate
- Write and use Maven plugins
- Generate a project website
- Customize a build with properties
- Use build profiles and profile activation
- Create and use Maven assemblies
- Develop with Maven archetypes

Written for the new user and veteran alike, *Maven: The Definitive Guide* is the perfect book to help you manage development projects for software, web applications, and enterprise applications.

“Today, with Maven on the forefront of all build tools, this invaluable guide by Sonatype is the first resource I reach for when training development teams on using Maven. It helps us handle Mojo curves nimbly, glide over rough sections of plugins, and accelerate through Java builds faster than competitors, allowing us to reach profitable product finish lines.”

—Matthew J. McCullough,
President, Denver Open
Source Users Group and
Managing Partner,
Ambient Ideas, LLC

The primary contributor to this book is Tim O'Brien, online editor of O'Reilly News and author of three other O'Reilly titles: *Harnessing Hibernate*, *Maven: A Developer's Notebook*, and *Jakarta Commons Cookbook*.

www.oreilly.com

US \$34.99

CAN \$34.99

ISBN: 978-0-596-51733-5



Safari 
Books Online

Free online edition
for 45 days with
purchase of this book.
Details on last page.

Maven: The Definitive Guide

Maven: The Definitive Guide

Sonatype

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Maven: The Definitive Guide

by Sonatype

Copyright © 2008 Sonatype. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Mike Loukides
Production Editor: Sarah Schneider
Production Services: Appingo, Inc.

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

August 2008: First Edition.

O'Reilly and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Maven: The Definitive Guide*, the image of a giant anteater, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-51733-5

[C]

1252360668

[9/09]

Table of Contents

Preface	xi
----------------------	-----------

Part I. Introduction

1. Introducing Apache Maven	3
Convention over Configuration	3
A Common Interface	4
Universal Reuse Through Maven Plugins	5
Conceptual Model of a “Project”	6
Is Maven an Alternative to XYZ?	6
Comparing Maven and Ant	7
Summary	11
2. Installing and Running Maven	13
Verify Your Java Installation	13
Downloading Maven	14
Installing Maven	14
Testing a Maven Installation	16
Maven Installation Details	16
Getting Help with Maven	17
Using the Maven Help Plugin	18
About the Apache Software License	21

Part II. Maven by Example

3. A Simple Maven Project	25
Introduction	25
Creating a Simple Project	26
Building a Simple Project	27
Simple Project Object Model	28

Core Concepts	30
Summary	41
4. Customizing a Maven Project	43
Introduction	43
Defining the Simple Weather Project	43
Creating the Simple Weather Project	44
Customize Project Information	45
Add New Dependencies	46
Simple Weather Source Code	48
Add Resources	53
Running the Simple Weather Program	54
Writing Unit Tests	58
Adding Test-Scoped Dependencies	60
Adding Unit Test Resources	61
Executing Unit Tests	62
Building a Packaged Command-Line Application	65
5. A Simple Web Application	67
Introduction	67
Defining the Simple Web Application	67
Creating the Simple Web Project	68
Configuring the Jetty Plugin	69
Adding a Simple Servlet	71
Adding J2EE Dependencies	73
Conclusion	75
6. A Multimodule Project	77
Introduction	77
The Simple Parent Project	77
The Simple Weather Module	79
The Simple Web Application Module	81
Building the Multimodule Project	83
Running the Web Application	85
7. Multimodule Enterprise Project	87
Introduction	87
The Simple Parent Project	90
The Simple Model Module	91
The Simple Weather Module	95
The Simple Persist Module	99
The Simple Web Application Module	105
Running the Web Application	116

The simple-command Module	117
Running simple-command	122
Conclusion	125

Part III. Maven Reference

8. Optimizing and Refactoring POMs	129
Introduction	129
POM Cleanup	130
Optimizing Dependencies	130
Optimizing Plugins	134
Optimizing with the Maven Dependency Plugin	136
Final POMs	139
Conclusion	147
9. The Project Object Model	149
Introduction	149
The POM	149
POM Syntax	156
Project Dependencies	159
Project Relationships	168
POM Best Practices	173
10. The Build Lifecycle	181
Introduction	181
Package-Specific Lifecycles	185
Common Lifecycle Goals	189
11. Build Profiles	197
What Are They For?	197
Portability Through Maven Profiles	200
Profile Activation	203
External Profiles	206
Settings Profiles	207
Listing Active Profiles	209
Tips and Tricks	209
Summary	215
12. Maven Assemblies	217
Introduction	217
Assembly Basics	218
Overview of the Assembly Descriptor	226

The Assembly Descriptor	228
Controlling the Contents of an Assembly	229
Best Practices	252
Summary	259
13. Properties and Resource Filtering	261
Introduction	261
Maven Properties	261
Resource Filtering	266
14. Maven and Eclipse: m2eclipse	271
Introduction	271
m2eclipse	271
Installing the m2eclipse Plugin	272
Enabling the Maven Console	274
Creating a Maven Project	275
Create a Maven POM File	280
Importing Maven Projects	282
Running Maven Builds	285
Working with Maven Projects	286
Working with Maven Repositories	292
Using the Form-Based POM Editor	294
Analyzing Project Dependencies in m2eclipse	298
Maven Preferences	303
Summary	306
15. Site Generation	309
Introduction	309
Building a Project Site with Maven	310
Customizing the Site Descriptor	311
Site Directory Structure	314
Writing Project Documentation	315
Deploying Your Project Web Site	317
Customizing Site Appearance	319
Tips and Tricks	328
16. Repository Manager	333
Introduction	333
Installing Nexus	334
Using Nexus	341
Configuring Maven to Use Nexus Repositories	346
Configuring Nexus	354
Maintaining Repositories	374

Deploying Artifacts to Nexus	376
17. Writing Plugins	383
Introduction	383
Programming Maven	383
Plugin Descriptor	387
Writing a Custom Plugin	392
Mojo Parameters	400
Plugins and the Maven Lifecycle	406
18. Writing Plugins in Alternative Languages	411
Writing Plugins in Ant	411
Creating an Ant Plugin	412
Writing Plugins in JRuby	414
Writing Plugins in Groovy	420
<hr/>	
Part IV. Appendixes	
A. Settings Details	425
B. Sun Specification Alternatives	435
Index	439

Preface

Although there are a number of references for Maven online, there is no single, well-written narrative for introducing Maven that can serve as both an authoritative reference and an introduction. What we've tried to do with this effort is provide such a narrative coupled with useful reference material.

Maven... What Is It?

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

Here is a more formal definition of Apache Maven (<http://maven.apache.org>): Maven is a project management tool that encompasses a Project Object Model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle. When you use Maven, you describe your project using a well-defined Project Object Model, Maven can then apply cross-cutting logic from a set of shared (or custom) plugins.

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

Font Conventions

This book follows certain conventions for font usage. Understanding these conventions upfront makes it easier to use this book:

Italic

Used for filenames, file extensions, URLs, application names, emphasis, and new terms when they are first introduced.

Constant width

Used for Java™ class names, methods, variables, properties, data types, database elements, and snippets of code that appear in text.

Constant width bold

Used for commands you enter at the command line and to highlight new code inserted in a running example.

Constant width italic

Used to annotate output.

Maven Writing Conventions

The book follows certain conventions for naming and font usage in relation to Apache Maven. Understanding these conventions upfront makes it easier to read this book:

Compiler plugin

Maven plugins are capitalized.

`create goal`

Maven goal names are displayed in a constant width font.

plugin

Maven revolves around the heavy use of plugins, but you won't find *plugin* defined in the dictionary. This book uses “plugin” without a hyphen because it is easier to read and write and because it is a standard throughout the Maven community.

Maven Lifecycle, Maven Standard Directory Layout, Project Object Model

Core Maven concepts are capitalized whenever they are referenced in the text.

`goalParameter`

A Maven goal parameter is displayed in a constant width font.

`compile` phase

Lifecycle phases are displayed in a constant width font.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Maven: The Definitive Guide* by Sonatype. Copyright 2008 Sonatype, 978-0-596-51733-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international/local)
707-829-0104 (fax)

O'Reilly's web page for this book, where we list errata, examples, or any additional information. You can access this page at:

<http://www.oreilly.com/catalog/9780596517335>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

<http://www.oreilly.com>

Acknowledgments

Sonatype would like to thank the following contributors who have provided feedback that improved the quality of this book. Thanks to Chad Gorshing, Marcus Biel, Brian Dols, Mangalaganesh Balasubramanian, Marius Kruger, and Mark Stewart. Special thanks to Joel Costigliola for helping debug and correct the Spring web chapter. Stan Guillory was practically a contributing author given the number of corrections he posted to the book's Get Satisfaction page. Thank you, Stan. Special thanks to Richard Coasby of Bamboo for acting as the provisional grammar consultant.

Sarah Schneider and Marlowe Shaeffer of O'Reilly Media, and Mark Jewett of Appingo, should receive medals for the extreme patience they displayed as this book continued to miss schedule after schedule. Thank you, Sarah, and the entire production department for making this book a success. Thanks to Mike Loukides and Mike Hendrickson for providing the necessary editorial oversight to make sure that we were given enough time and feedback to publish a book that will remain relevant in the months and years to come.

Thanks to all of the authors, especially Bruce Snyder, Brian Fox, John Casey, Jason van Zyl, and Eugene Kuleshov. Everyone at Sonatype played a part in the content of this book, and everyone worked together to help us create a foundation for this documentation.

Thanks to all of our contributing authors, especially Eric Redmond.

Tim O'Brien would like to thank his perfect wife, Susan, and child, Josephine.

Introduction

The two chapters in this brief introduction describe Maven, explain how it stacks up to and improves on other build tools throughout time, and show you how to install and run it on all platforms. If you've already installed Maven and are familiar with the core concepts of the tool, you might want to skip ahead to Part II. Then again, even if you are already somewhat familiar with Maven, you might want to peruse some of the propaganda in this introduction so you're prepared when people start asking you why your organization or project should use Maven. After this section, you should have a better idea of what Maven is, what makes it different from some of the other options out there, and how to install it and learn more from the built-in help facilities.

Introducing Apache Maven

Convention over Configuration

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

Maven incorporates the concept by providing sensible default behaviors for projects. Without customization, source code is assumed to be in `${basedir}/src/main/java` and resources are assumed to be in `${basedir}/src/main/resources`. Tests are assumed to be in `${basedir}/src/test`, and a project is assumed to produce a JAR (Java ARchive) file. Maven assumes that you want to compile byte code to `${basedir}/target/classes` and then create a distributable JAR file in `${basedir}/target`. Although this might seem trivial, consider the fact that most Ant-based builds have to define the locations of these directories in every subproject. Maven’s adoption of convention over configuration goes further than just simple directory locations; Maven’s core plugins apply a common set of conventions for compiling source code, packaging distributions, generating web sites, and many other processes. Maven’s strength comes from the fact that it is “opinionated.” It has a defined lifecycle and a set of common plugins that know how to build libraries and web applications. If you follow the convention, Maven will require almost zero effort—just put your source in the correct directory, and Maven will take care of the rest.

One side effect of using systems that follow “convention over configuration” is that end users might feel that they are forced to use a particular setup. While it is certainly

true that Maven has some central opinions that shouldn't be challenged, most of the defaults can be customized. For example, the location of a project's source code and resources can be customized, names of JAR files can be customized, and through the development of custom plugins, almost any behavior can be tailored to your specific environment's requirements. If you don't follow convention, Maven will allow you to customize defaults in order to adapt to your requirements.

A Common Interface

Before Maven provided a common interface for building software, every single project had someone dedicated to managing a completely custom build system, and developers had to take time away from developing software to learn about the idiosyncrasies of each new project they wanted to contribute to. In 2001, you'd take a completely different approach to building a project such as Apache Turbine (<http://turbine.apache.org/>) than you would to building a project such as Tomcat (<http://tomcat.apache.org>). If a new source analysis tool came out that would perform static analysis on source code, or if someone developed a new unit testing framework, everyone would have to drop what they were doing and figure out how to fit it into each project's custom build environment. How would you run unit tests? There were a thousand different answers. This environment was characterized by endless arguments about tools and build procedures. The age before Maven was an age of inefficiency—the age of the “Build Engineer.”

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

Although Maven provides an array of benefits, including dependency management and reuse of common build logic through plugins, the core reason it has succeeded is that it has defined a common interface for building software. When you see that a project such as Apache Wicket (<http://wicket.apache.org>) uses Maven, you can assume that you'll be able to check it out from source and build it with *mvn install* without much

hassle. You know where the ignition key goes, and you know that the gas pedal is on the right and the brake is on the left.

Universal Reuse Through Maven Plugins

The core of Maven is pretty dumb; it doesn't know how to do much beyond parsing a few XML documents and keeping track of a lifecycle and a few plugins. Maven has been designed to delegate most responsibility to a set of Maven plugins that can affect the Maven lifecycle and offer access to goals. Most of the action in Maven happens in plugin goals that take care of things like compiling source, packaging bytecode, publishing sites, and any other task that needs to happen in a build. The Maven you download from Apache doesn't know much about packaging a WAR file or running JUnit tests; most of Maven's intelligence is implemented in the plugins, and the plugins are retrieved from the Maven repository. In fact, the first time you run something like *mvn install* with a brand new Maven installation, it retrieves most of the core Maven plugins from the central Maven repository. This is more than just a trick to minimize the download size of the Maven distribution; this is behavior that allows you to upgrade a plugin to add capability to your project's build. The fact that Maven retrieves both dependencies and plugins from the remote repository allows for universal reuse of build logic.

The Maven Surefire plugin is responsible for running unit tests. At some point between version 1.0 and the version that is in wide use today, someone decided to add support for the TestNG unit testing framework in addition to the support for JUnit. This happened in a way that didn't break backward compatibility—if you were using the Surefire plugin to compile and execute JUnit 3 unit tests, and you upgraded to the most recent version of the Surefire plugin, your tests continued to execute without fail. You also gained new functionality, so if you wanted to execute unit tests in TestNG, you now had that ability, thanks to the efforts of the maintainers of the Surefire plugin. You also gained the ability to run annotated JUnit 4 unit tests. You gained all of these capabilities without having to upgrade your Maven installation or install new software. Most importantly, nothing about your project had to change aside from a version number for a plugin in a POM.

It is this mechanism that affects much more than the Surefire plugin: projects are compiled with a Compiler plugin, projects are turned into JAR files with a Jar plugin, and there are plugins for running reports, plugins for executing JRuby and Groovy code, as well as plugins to publish sites to remote servers. Maven has abstracted common build tasks into plugins that are maintained centrally and shared universally. If the state of the art changes in any area of the build, if some new unit testing framework is released or if some new tool is made available, you don't have to be the one to hack your project's custom build system to support it. You benefit from the fact that plugins are downloaded from a remote repository and maintained centrally. This is what is meant by universal reuse through Maven plugins.

Conceptual Model of a “Project”

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

Dependency management

A project is defined as unique coordinates that consists of a group identifier, artifact identifier, and version. Projects can now use these coordinates to declare dependencies.

Remote repositories

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

Universal reuse of build logic

Plugins are coded to work with the POM; they are not designed to operate on specific files in known locations. Everything is abstracted into the model—plugin configuration and customization happens in the model.

Tool portability and integration

Tools such as Eclipse, NetBeans, and IntelliJ now have a common place to find information about a project. Before the advent of Maven, every integrated development environment (IDE) had a different way to store what was essentially a custom POM. Maven has standardized this description, and although each IDE continues to maintain custom project files, they can be easily generated from the model.

Easy searching and filtering of project artifacts

Tools such as Nexus allow you to index and search the contents of a repository using the information stored in the POM.

Maven has provided a foundation for the beginnings of a consistent semantic description of a software project.

Is Maven an Alternative to XYZ?

So, sure, Maven is an alternative to Ant, but Apache Ant (<http://ant.apache.org>) continues to be a great, widely used tool. It has been the reigning champion of Java builds for years, and you can integrate Ant build scripts with your project’s Maven build very easily. This is a common usage pattern for a Maven project. On the other hand, as more and more open source projects move to Maven as a project management platform,

working developers are starting to realize that Maven not only simplifies the task of build management, it is helping to encourage a common interface between developers and software projects. Maven is more of a platform than a tool. Although you can consider Maven an alternative to Ant, you are comparing apples to oranges. “Maven” includes more than just a build tool.

This is the central point that makes all of the Maven versus Ant, Maven versus Buildr, Maven versus Gradle arguments irrelevant. Maven isn’t totally defined by the mechanics of your build system. It isn’t about scripting the various tasks in your build as much as it is about encouraging a set of standards, a common interface, a lifecycle, a standard repository format, a standard directory layout, etc. It certainly isn’t about what format the POM happens to be in, i.e., XML versus YAML versus Ruby. Maven is much larger than that, and Maven refers to much more than the tool itself. When this book talks about Maven, it is referring to the constellation of software, systems, and standards that support it. Buildr, Ivy, Gradle—all of these tools interact with the repository format that Maven helped create, and you could just as easily use a tool such as Nexus to support a build written entirely in Buildr. Nexus is introduced in Chapter 16.

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

Comparing Maven and Ant

Although the previous section should convince you that the authors of this book have no interest in creating a feud between Apache Ant and Apache Maven, we are cognizant of the fact that most organizations have to make a decision between Ant and Maven. In this section, we compare and contrast the tools.

Ant excels at build process; it is a build system modeled after *make* with targets and dependencies. Each target consists of a set of instructions that are coded in XML. There is a `copy` task and a `javac` task as well as a `jar` task. When you use Ant, you supply it with specific instructions for compiling and packaging your output. Look at the simple *build.xml* file shown in Example 1-1.

Example 1-1. A simple Ant *build.xml* file

```
<project name="my-project" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist" location="target"/>

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

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

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

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

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

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

Contrast the previous Ant example with a Maven example. In Maven, to create a JAR file from some Java source, all you need to do is create a simple *pom.xml*, place your source code in `${basedir}/src/main/java`, and then run `mvn install` from the command

line. The example Maven *pom.xml* that achieves the same results as the simple Ant file listed in Example 1-1 is shown in Example 1-2.

Example 1-2. A simple Maven pom.xml

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

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

Admittedly, this is the simplest possible example project: a project that contains only source code and produces a JAR; a project that follows Maven conventions and doesn't require any dependencies or customization. If we want to start customizing the behavior, our *pom.xml* is going to grow in size, and in the largest of projects, you can see collections of very complex Maven POMs that contain a great deal of plugin customization and dependency declarations. But even when your project's POM files become more substantial, they hold an entirely different kind of information from the build file of a similarly sized project using Ant. Maven POMs contain declarations: "This is a JAR project," and "The source code is in *src/main/java*." Ant build files contain explicit instructions: "This is project," "The source is in *src/main/java*," "Run *javac* against this directory," "Put the results in *target/classes*," "Create a JAR from the", etc. Where Ant has to be explicit about the process, there is something "built-in" to Maven that just knows where the source code is and how it should be processed.

The differences between Ant and Maven in this example are:

Apache Ant

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

Apache Maven

- Maven has conventions: in the example, it already knew where your source code was because you followed the convention. It put the bytecode in *target/classes*, and it produced a JAR file in *target*.
- Maven is declarative; all you had to do was create a *pom.xml* file and put your source in the default directory. Maven took care of the rest.
- Maven has a lifecycle, which you invoked when you executed *mvn install*. This command told Maven to execute a series of sequence steps until it reached the lifecycle. As a side effect of this journey through the lifecycle, Maven executed a number of default plugin goals that did things such as compile and create a JAR.

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

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

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

Summary

This introduction has been kept purposefully short. We have covered a basic outline of what Maven is and how it stacks up to and improves on other build tools throughout time. The next chapter will explain how to install and run Maven, and Chapter 3 will dive into a simple project and show how Maven can perform phenomenal tasks with the smallest amount of configuration.

Installing and Running Maven

This chapter contains very detailed instructions for installing Maven on a number of different platforms. Instead of assuming a level of familiarity with installing software and setting environment variables, we've opted to be as thorough as possible to minimize any problems that might arise due to a partial installation. The only thing this chapter assumes is that you've already installed a suitable Java Development Kit (JDK). If you are just interested in installation, you can move on to the rest of the book after reading through the “Downloading Maven” and “Installing Maven” sections. If you are interested in the details of your Maven installation, this entire chapter will give you an overview of what you've installed and the Apache Software License.

Verify Your Java Installation

Although Maven can run on Java 1.4, this book assumes that you are running at least Java 5. Go with the most recent stable JDK available for your operating system. Either Java 5 or Java 6 will work with all of the examples in this book:

```
% java -version
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing)
```

Maven works with all certified Java-compatible development kits, and a few noncertified implementations of Java. The examples in this book were written and tested against the official Java Development Kit releases downloaded from the Sun Microsystems web site. If you're working with a Linux distribution, you may need to download Sun's JDK yourself and make sure it's the version you're invoking (by running *java -version*, as shown earlier). Now that Sun has open sourced Java, this will hopefully improve in the future, and we'll get the Sun Java Runtime Environment (JRE) and JDK by default even in purist distributions. Until that day, you may need to do some of your own downloading.

Downloading Maven

You can download Maven from the Apache Maven project web site by going to <http://maven.apache.org/download.html>.

When downloading Maven, make sure you choose the latest version of Apache Maven from the web site. The latest version of Maven at the time of this writing is Maven 2.0.9. If you are not familiar with the Apache Software License, you should get acquainted with the terms of the license before you start using the product. More information on the Apache Software License can be found in “About the Apache Software License,” later in this chapter.

Installing Maven

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

Installing Maven on Mac OS X

You can download a binary release of Maven from <http://maven.apache.org/download.html>. Download the current release of Maven in a format that is convenient for you to work with. Pick an appropriate place for it to live, and expand the archive there. If you expanded the archive into the directory `/usr/local/maven-2.0.9`, you may want to create a symbolic link to make it easier to work with and to avoid the need to change any environment configuration when you upgrade to a newer version:

```
/usr/local % ln -s maven-2.0.9 maven
/usr/local % export M2_HOME=/usr/local/maven
/usr/local % export PATH=${M2_HOME}/bin:${PATH}
```

Once Maven is installed, you need to do a couple of things to make it work correctly. You need to add its `bin` directory in the distribution (in this example, `/usr/local/maven/bin`) to your command path. You also need to set the environment variable `M2_HOME` to the top-level directory you installed (in this example, `/usr/local/maven`).



Installation instructions are the same for both OS X Tiger and Leopard. It has been reported that Maven 2.0.6 is shipping with a preview release of Xcode. If you have installed XCode, run `mvn` from the command line to check availability. XCode installs Maven in `/usr/share/maven`. We recommend installing the most recent version of Maven 2.0.9, as there have been a number of bug fixes and improvements since Maven 2.0.9 was released.

You'll need to add both `M2_HOME` and `PATH` to a script that will run every time you log in. To do this, add the following lines to *.bash_login*:

```
export M2_HOME=/usr/local/maven
export PATH=${M2_HOME}/bin:${PATH}
```

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



These installation instructions assume that you are running *bash*.

Installing Maven on Microsoft Windows

Installing Maven on Windows is very similar to installing Maven on Mac OS X, the main differences being the installation location and the setting of an environment variable. This book assumes a Maven installation directory located at *c:\Program Files\maven-2.0.9*, but it won't make a difference if you install Maven in another directory as long as you configure the proper environment variables. Once you've unpacked Maven in the installation directory, you will need to set two environment variables—`PATH` and `M2_HOME`. To set these environment variables from the command line, type in the following commands:

```
C:\Users\tobrien > set M2_HOME=c:\Program Files\maven-2.0.9
C:\Users\tobrien > set PATH=%PATH%;%M2_HOME%\bin
```

Setting these environment variables on the command line will allow you to run Maven in your current session, but unless you add them to the system environment variables through the control panel, you'll have to execute these two lines every time you log into your system. Set both `M2_HOME` and `PATH` to point to your Maven installation.

Installing Maven on Linux

To install Maven on a Linux machine, follow the exact procedure outlined in “Installing Maven on Mac OS X,” earlier in this chapter.

Installing Maven on FreeBSD or OpenBSD

To install Maven on a FreeBSD or OpenBSD machine, follow the exact procedure outlined in “Installing Maven on Mac OS X,” earlier in this chapter.

Testing a Maven Installation

Once Maven is installed, you can see if it is installed properly by running *mvn -v* from the command line. If Maven has been installed, you should see something resembling the following output:

```
~/examples $ mvn -v
Maven 2.0.9
```

If you see this output, you know that Maven has been successfully installed. If you do not see this output and your operating system cannot find the *mvn* command, make sure that your `PATH` and `M2_HOME` environment variables have been properly set.

Maven Installation Details

Maven's download measures in at roughly 1.5 MiB.* It has attained such a slim download size because the core of Maven has been designed to retrieve plugins and dependencies from a remote repository on demand. When you start using Maven, it will start to download plugins to a local repository as described in the section “User-Specific Configuration and Repository,” later in this chapter. In case you are curious, let's take a quick look at what is in Maven's installation directory:

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

LICENSE.txt contains the software license for Apache Maven. This license is described in some detail later in the section “About the Apache Software License.” *NOTICE.txt* contains some notices and attributions required by libraries that Maven depends on. *README.txt* contains some installation instructions. *bin/* contains the *mvn* script that executes Maven. *boot/* contains a JAR file (*classworlds-1.1.jar*) that is responsible for creating the Class Loader in which Maven executes. *conf/* contains a global *settings.xml* that can be used to customize the behavior of your Maven installation. If you need to customize Maven, it is customary to override any settings in a *settings.xml* file stored in *~/.m2*. *lib/* contains a single JAR file (*maven-core-2.0.9-uber.jar*) that contains the core of Maven.

* Ever purchased a 200 GB hard drive only to realize that it showed up as less than 200 GiB when you installed it? Computers understand Gibibytes, but retailers sell products using Gigabytes. MiB stands for Mebibyte, which is defined as 2^{20} or 1024^2 . These binary prefix standards are endorsed by the Institute of Electrical and Electronics Engineers (IEEE), the International Committee for Weights and Measures (CIPM), and the International Electrotechnical Commission (IEC). For more information about Kibibytes, Mebibytes, Gibibytes, and Tebibytes, see <http://en.wikipedia.org/wiki/Mebibyte>.

User-Specific Configuration and Repository

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

settings.xml

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

repository/

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



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

Upgrading a Maven Installation

If you've installed Maven on a Mac OS X or Unix machine according to the details given in "Installing Maven on Mac OS X" and "Installing Maven on Linux," it should be easy to upgrade to newer versions of Maven when they become available. Simply install the newer version of Maven (`/usr/local/maven-2.future`) next to the existing version of Maven (`/usr/local/maven-2.0.9`). Then, switch the symbolic link `/usr/local/maven` from `/usr/local/maven-2.0.9` to `/usr/local/maven-2.future`. Since you've already set your `M2_HOME` variable to point to `/usr/local/maven`, you won't need to change any environment variables.

If you've installed Maven on a Windows machine, simply unpack Maven to `c:\Program Files\maven-2.future` and update your `M2_HOME` variable.

Getting Help with Maven

Although this book aims to be a comprehensive reference, there are going to be topics we miss and special situations and tips that are not covered. The core of Maven is very simple, but the real work in Maven happens in the plugins, and there are too many plugins available to cover them all in one book. When you encounter problems and features that are not covered in this book, we suggest searching for answers at the following locations:

<http://maven.apache.org>

This is the first place you should look; the Maven web site contains a wealth of information and documentation. Every plugin has a few pages of documentation, and it provides a series of “quick start” documents that will be helpful in addition to the contents of this book. Although the Maven site contains plenty of information, it can also be a frustrating, confusing, and overwhelming. A custom Google search box on the main Maven page will search known Maven sites for information. This provides better results than a generic Google search.

Maven user mailing list

The Maven user mailing list is the place for users to ask questions. Before you ask a question on the user mailing list, you will want to search for any previous discussion that might relate to your question. It is bad form to ask a question that has already been asked without first checking to see whether an answer already exists in the archives. There are a number of useful mailing list archive browsers; we’ve found Nabble to be the most useful. You can browse the user mailing list archives here: <http://www.nabble.com/Maven---Users-f178.html>. You can join the user mailing list by following the instructions available here: <http://maven.apache.org/maillists.html>.

<http://www.sonatype.com>

Sonatype maintains an online copy of this book and other tutorials related to Apache Maven.



Despite the best efforts of some very dedicated Maven contributors, the Maven web site is poorly organized and full of incomplete (and sometimes misleading) snippets of documentation. Throughout the Maven community there is a lack of a common standards for plugin documentation. Some plugins are heavily documented, whereas others lack even the most basic instructions for usage. Often your best bet is to search for a solution in the archives of the user mailing list. If you really want to help, submit a patch to the Maven site (or this book).

Using the Maven Help Plugin

Throughout the book, we will be introducing Maven plugins and talking about Maven Project Object Model (POM) files, settings files, and profiles. There are going to be times when you need a tool to help you make sense of some of the models that Maven is using and what goals are available on a specific plugin. The Maven Help plugin allows you to list active Maven profiles, display an effective POM, print the effective settings, or list the attributes of a Maven plugin.



For a conceptual overview of the POM and plugins, see Chapter 3.

The Maven Help plugin has four goals. The first three goals—**active-profiles**, **effective-pom**, and **effective-settings**—describe a particular project and must be run in the base directory of a project. The last goal—**describe**—is slightly more complex, showing you information about a plugin or a plugin goal. The following commands provide some general information about the four goals:

help:active-profiles

Lists the profiles (project, user, global) that are active for the build.

help:effective-pom

Displays the effective POM for the current build, with the active profiles factored in.

help:effective-settings

Prints out the calculated settings for the project, given any profile enhancement and the inheritance of the global settings into the user-level settings.

help:describe

Describes the attributes of a plugin. This need not run under an existing project directory. You must give at least the **groupId** and **artifactId** of the plugin you wish to describe.

Describing a Maven Plugin

Once you start using Maven, you'll spend most of your time trying to get more information about plugins. You'll want to know: How do plugins work? What are the configuration parameters? What are the goals? The **help:describe** goal is something you'll use very frequently to retrieve this information. With the **plugin** parameter, you can specify a plugin you wish to investigate, passing in either the plugin prefix (e.g., **maven-help-plugin** as **help**) or the **groupId:artifact[:version]**, where **version** is optional. For example, the following command uses the Help plugin's **describe** goal to print out information about the Maven Help plugin:

```
$ mvn help:describe -Dplugin=help
...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:    2.0.1
Goal Prefix: help
Description:
```

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

```
usage information.  
...
```

Executing the `describe` goal with the `plugin` parameter prints out the Maven coordinates for the plugin, the goal prefix, and a brief description of the plugin. Although this information is helpful, you'll usually be looking for more detail than this. If you want the `Help` plugin to print a full list of goals with parameters, execute the `help:describe` goal with the parameter `full` as follows:

```
$ mvn help:describe -Dplugin=help -Dfull  
...  
Group Id:  org.apache.maven.plugins  
Artifact Id: maven-help-plugin  
Version:    2.0.1  
Goal Prefix: help  
Description:
```

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

Mojos:

```
=====
```

Goal: 'active-profiles'

```
=====
```

Description:

Lists the profiles which are currently active for this build.

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

Parameters:

```
-----
```

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

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

```
-----
```

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

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

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

... remove the other goals ...

This option is great for discovering all of a plugin's goals as well as their parameters. But sometimes this gives you far more information than you need. To get information about a single goal, set the `mojo` parameter as well as the `plugin` parameter. The following command lists all of the information about the Compiler plugin's `compile` goal:

```
$ mvn help:describe -Dplugin=compiler -Dmojo=compile -Dfull
```



What's a Mojo? In Maven, a plugin goal is known as a Mojo.

About the Apache Software License

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

Chances are good that, if you are reading this book, you are not a lawyer. If you are wondering what the Apache License, version 2.0 means, the Apache Software Foundation has assembled a very helpful Frequently Asked Questions (FAQ) page about the license, available here: <http://www.apache.org/foundation/licence-FAQ.html>. Here's the answer to the frequently asked question "I am not a lawyer. What does it all mean?":

[This license] allows you to:

- Freely download and use Apache software, in whole or in part, for personal, company internal, or commercial purposes;
- Use Apache software in packages or distributions that you create.

It forbids you to:

- Redistribute any piece of Apache-originated software without proper attribution;
- Use any marks owned by the Apache Software Foundation in any way that might state or imply that the Foundation endorses your distribution;
- Use any marks owned by the Apache Software Foundation in any way that might state or imply that you created the Apache software in question.

It requires you to:

- Include a copy of the license in any redistribution you may make that includes Apache software;

- Provide clear attribution to the Apache Software Foundation for any distributions that include Apache software.

It does not require you to:

- Include the source of the Apache software itself, or of any modifications you may have made to it, in any redistribution you may assemble that includes it;
- Submit changes that you make to the software back to the Apache Software Foundation (though such feedback is encouraged).

This ends the installation information. The next part of the book contains Maven examples.

Maven by Example

The first Maven book was *Maven: A Developer's Notebook* (O'Reilly). That book introduced Maven in a series of steps via a conversation between you and a colleague who already knew how to use Maven. The idea behind the (now-retired) Developer's Notebook series was that developers learn best when they are sitting next to other developers and going through the same thought processes, learning to code by doing and experimenting. Although the series was successful, the Notebook format had limitations. Notebooks were designed to be “goal-focused” books that take you through a series of steps to achieve very specific goals. By contrast, larger reference books provide comprehensive material that covers the entirety of the topic.

If you read *Maven: A Developer's Notebook*, you'll learn how to create a simple project or a project that creates a WAR from a set of source files. But if you want to find out the specifics of something like the Assembly plugin, you'll hit an impasse. Because there is currently no well-written reference material for Maven, you have to hunt through plugin documentation on the Maven web site or cull from a series of mailing lists. Once you really dig into Maven, you end up reading through thousands of HTML pages on the Maven site written by hundreds of developers, each with a different idea of what it means to document a plugin. Despite the best efforts of well-meaning volunteers, reading through plugin documentation on the Maven site is frustrating at best, and at worst, it's a reason to abandon Maven. Quite often, Maven users get stuck because they just can't find an answer.

This lack of an authoritative (or definitive) reference manual has held Maven back for a few years, and it has been something of a dampening force on Maven adoption. With *Maven: The Definitive Guide*, we intend to change that situation by providing a comprehensive reference in Part III. In Part II, we're preserving the narrative progression of a Developer's Notebook; it is valuable material that helps people learn Maven by example. Thus, here we “introduce by doing,” and in Part III, we fill in the blanks and dig into the details. Where Part III might use a reference table and a program listing detached from an example project, Part II is motivated by real examples.

After reading this part, you should have everything you need to start using Maven. You might need to refer to Part III only when you start customizing Maven by writing custom plugins or when you want more detail about specific plugins.

A Simple Maven Project

Introduction

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

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

Downloading This Chapter's Example

This chapter develops a very simple example that will be used to explore core concepts of Maven. If you follow the steps as described, you shouldn't need to download the examples to recreate the code produced by Maven. We will be using the Maven Archetype plugin to create this simple project, and in this chapter we won't modify the project in any way. If you would prefer to read this chapter with the final example source code, the example project may be downloaded with the book's example code at <http://www.sonatype.com/book/mvn-examples-1.0.zip> or <http://www.sonatype.com/book/mvn-examples-1.0.tar.gz>. Unzip this archive in any directory, and then go to the *ch03/* directory. There you will see a directory named *simple/*, which contains the source code for this chapter. If you wish to follow along with the example code in a web

browser, go to <http://www.sonatype.com/book/examples-1.0> and click on the *ch03/* directory.

Creating a Simple Project

To start a new Maven project, use the Maven Archetype plugin from the command line:

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 \
                        -DartifactId=simple \
                        -DpackageName=org.sonatype.mavenbook

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] artifact org.apache.maven.plugins:maven-archetype-plugin: checking for \
updates from central
[INFO] -----
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] -----
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
checking for updates from central
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch03
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: /Users/tobrien/svnw/sonatype/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: simple
[INFO] * End of debug info from resources from generated POM *
[INFO] Archetype created in dir: /Users/tobrien/svnw/sonatype/examples/simple
```

mvn is the Maven 2 command. *archetype:create* is called a Maven goal. If you are familiar with Apache Ant, a Maven goal is analogous to an Ant target; both describe a unit of work to be completed in a build. The *-Dname=value* pairs are arguments that are passed to the goal and take the form of *-D* properties, similar to the system property options you might pass to the Java Virtual Machine via the command line. The purpose of the *archetype:create* goal is to quickly create a project from an archetype. In this context, an archetype is defined as “an original model or type after which other similar things are patterned; a prototype.”* A number of archetypes are available in Maven for anything from a simple Swing application to a complex web application. In this chapter, we are going to use the most basic archetype to create a simple skeleton starter project. The plugin is the prefix *archetype*, and the goal is *create*.

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

```
simple/❶
simple/pom.xml❷
  /src/
    /src/main/❸
```

* *The American Heritage Dictionary of the English Language*

```
    /main/java
/src/test/❹
    /test/java
```

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

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

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

```
package org.sonatype.mavenbook;

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

The simplest Maven archetype generates the simplest possible program: a program that prints “Hello World!” to standard output.

Building a Simple Project

Once you have created the project with the Maven Archetype plugin by following the directions from the previous section (“Creating a Simple Project”) you will want to build and package the application. To do so, run `mvn install` from the directory that contains the `pom.xml`:

```

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

-----
T E S T S
-----

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

Results :

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

[INFO] [jar:jar]
[INFO] Building jar: /simple/target/simple-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing /simple/target/simple-1.0-SNAPSHOT.jar to \
~/.m2/repository/org/sonatype/mavenbook/ch03/simple/1.0-SNAPSHOT/ \
simple-1.0-SNAPSHOT.jar

```

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

```

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

```

Simple Project Object Model

When Maven executes, it looks to the Project Object Model for information about the project. The POM answers such questions as: What type of project is this? What is the project's name? Are there any build customizations for this project? Example 3-1 shows the default *pom.xml* file created by the Maven Archetype plugin's `create goal`.

Example 3-1. Simple project's pom.xml file

```

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

```

```

        http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.sonatype.mavenbook.ch03</groupId>
<artifactId>simple</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>simple</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

This *pom.xml* file is the most basic POM you will ever deal with for a Maven project. Usually a POM file is considerably more complex, defining multiple dependencies and customizing plugin behavior. The first few elements—`groupId`, `artifactId`, `packaging`, `version`—are known as the Maven coordinates, which uniquely identify a project. `name` and `url` are descriptive elements of the POM, providing a human-readable name and associating the project with a project web site. Lastly, the `dependencies` element defines a single, `test`-scoped dependency on a unit testing framework called JUnit. These topics will be further introduced in the next section, “Core Concepts,” and in Chapter 9. All you need to know at this point is that the *pom.xml* is the file that makes Maven go.

Maven always executes against an effective POM, a combination of settings from this project’s *pom.xml*, all parent POMs, a Super POM defined within Maven, user-defined settings, and active profiles. All projects ultimately extend the Super POM, which defines a set of sensible default configuration settings and which is fully explained in Chapter 9. Although your project might have a relatively minimal *pom.xml*, the contents of your project’s POM are interpolated with the contents of all parent POMs, user settings, and any active profiles. To see this “effective” POM, run the following command in the simple project’s base directory:

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

When you run this, you should see a much larger POM that exposes the default settings of Maven. This goal can come in handy if you are trying to debug a build and want to see how all of the current project’s ancestor POMs are contributing to the effective POM. For more information about the Maven Help plugin, see “Using the Maven Help Plugin” in Chapter 2.

Core Concepts

Now that we’ve just run Maven for the first time, this is a good point to introduce a few of the core concepts of Maven. In Example 3-1, you generated a project that consisted of a POM and some code assembled in the Maven Standard Directory Layout. You then executed Maven with a lifecycle phase as an argument that prompted Maven to execute a series of Maven plugin goals. Lastly, you installed a Maven artifact into your local repository. Wait—what is a “lifecycle”? What is a “local repository”? The following section defines some of Maven’s central concepts.

Maven Plugins and Goals

In the previous section, we ran Maven with two different types of command-line arguments. The first command was a single plugin goal, the `create` goal of the `Arche`-`type` plugin. The second execution of Maven was a lifecycle phase, `install`. To execute a single Maven plugin goal, we used the syntax `mvn archetype:create`, where `archetype` is the identifier of a plugin and `create` is the identifier of a goal. When Maven executes a plugin goal, it prints out the plugin identifier and goal identifier to standard output:

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 \
                        -DartifactId=simple \
                        -DpackageName=org.sonatype.mavenbook
...
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
        checking for updates from central
...
```

A Maven plugin is a collection of one or more goals (see Figure 3-1). Examples of Maven plugins can be simple core plugins such as the `Jar` plugin that contains goals for creating JAR files, the `Compiler` plugin that contains goals for compiling source code and unit tests, or the `Surefire` plugin that contains goals for executing unit tests and generating reports. Other, more specialized Maven plugins include the `Hibernate3` plugin, for integration with the popular persistence library `Hibernate`, and the `JRuby` plugin, which allows you to execute Ruby as part of a Maven build or to write Maven plugins in Ruby. Maven also provides you with the ability to define custom plugins. A custom plugin can be written in any number of languages, including Java, Ant, Groovy, BeanShell, and, as previously mentioned, Ruby.

A goal is a specific task that may be executed as a standalone goal or along with other goals as part of a larger build. A goal is a “unit of work” in Maven. Examples of goals include the `compile` goal in the `Compiler` plugin, which compiles all of the source code for a project, or the `test` goal of the `Surefire` plugin, which can execute unit tests. Goals are configured via configuration properties that can be used to customize behavior. For example, the `compile` goal of the `Compiler` plugin defines a set of configuration parameters that allow you to specify the target JDK version or whether to use the

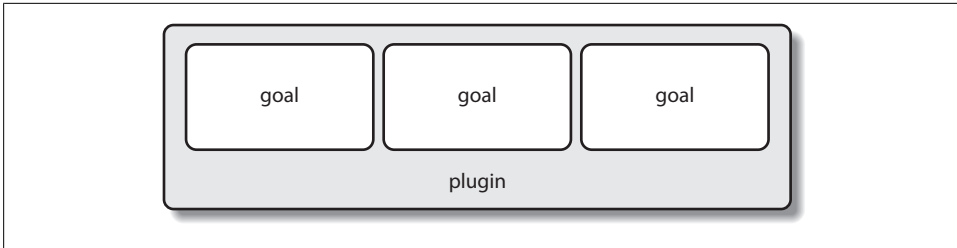


Figure 3-1. A plugin contains goals

compiler optimizations. In the previous example, we passed in the configuration parameters `groupId` and `artifactId` to the `create` goal of the Archetype plugin via the command-line parameters `-DgroupId=org.sonatype.mavenbook.ch03` and `-DartifactId=simple`. We also passed the `packageName` parameter to the `create` goal as `org.sonatype.mavenbook`. If we had omitted the `packageName` parameter, the package name would have defaulted to `org.sonatype.mavenbook.ch03`.



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

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

The core of Maven has little to do with the specific tasks involved in your project's build. By itself, Maven doesn't know how to compile your code or even how to make a JAR file. It delegates all of this work to Maven plugins like the Compiler plugin and the Jar plugin, which are downloaded on an as-needed basis and periodically updated from the central Maven repository. When you download Maven, you are getting the core of Maven, which consists of a very basic shell that knows only how to parse the command line, manage a classpath, parse a POM file, and download Maven plugins as needed. By keeping the Compiler plugin separate from Maven's core and providing for an update mechanism, Maven makes it easier for users to have access to the latest options in the compiler. In this way, Maven plugins allow for universal reusability of