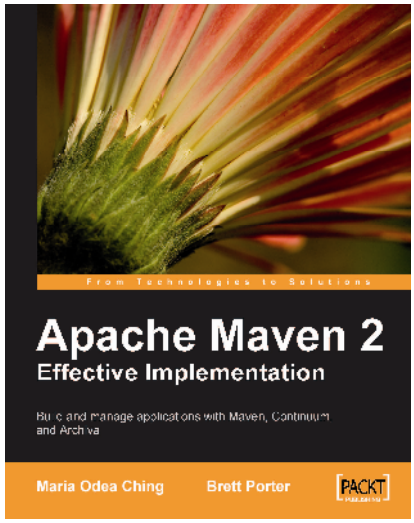
Apache Maven 2
Effective Implementation

Maria Odea Ching

Brett Porter



Chapter No. 6
"Useful Maven Plugins"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.6 "Useful Maven Plugins"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Maria Odea Ching grew up in Daet, a small town in the Philippines, then moved to the country's capital, Manila, when she went to college. She took up Computer Studies at De La Salle University, and graduated in 2004. She started using open source tools from her first job after graduating. From then on, she got interested in the open source philosophy. She was introduced to Apache Maven, Apache Continuum, and Apache Archiva early on in her career. She became a committer and a Project Management Committee member of Apache Maven. Eventually, she was elected as PMC Chair of Apache Archiva. She is also a member of the Apache Software Foundation.

Deng is currently a Senior Software Engineer and serves as the development lead for the Maestro project.

First, I'd like to thank Brett for the whirlwind endeavor which is this book. I'd also like to thank all our reviewers—Wendy, Emmanuel, Carsten and the Packt team, for taking the time to review and go through each chapter. You guys rock! And of course without the communities of Maven, Continuum, and Archiva, we wouldn't have anything to write about. So I'd like to thank each and everyone (committers/developers, contributors, buggers) in these respective communities. I'd also like to give special thanks to our Exist/G2iX family for their continuous support.

And last but definitely not the least, I'd like to thank my family and my boyfriend, Mike, for their unfaltering love and support and for being so patient and understanding when I have to run off to work on the book.

Brett Porter is a software developer from Sydney, Australia with a passion for development tooling and automation. Seeking a more standardized and reproducible solution to organize, build, and deploy a number of software projects across teams, he discovered an early beta of Maven 1.0 in 2003, and has been heavily involved in the development of the project since. He is a member of the Apache Maven Project Management Committee, and has conducted presentations and training on Maven and related tooling at several conferences and events. He founded the Archiva project in 2005. Brett is also a Director and Member of the Apache Software Foundation.

Brett is currently VP, Product Development at G2iX, in charge of the MaestroDev division. He and his team seek to make developers more efficient by offering support and services for development and automation tools including Apache Maven, Apache Continuum, Apache Archiva, and Selenium.

Brett was co-author of the book *Better Builds with Maven*, the first book to be written about the Maven 2.0 release in 2005, and has been involved in reviewing *Maven: A Developer's Notebook* and *Java Power Tools*.

I'd first like to thank my co-author and friend Deng for agreeing to participate in this book and lending her experience with Archiva and Continuum. I am grateful to all of the reviewers that volunteered their time to help make this the best that it can be. My great thanks go to all the members of the open source community that participate in these projects—the developers, as well as those that contribute patches, detailed bug reports, or answer questions on the user lists—not only do we build great software together, but I get the chance to work with truly remarkable individuals.

Finally, my love and thanks go to my wife Laura for sparing some more of our precious time so that I could complete this book, and for supporting me in everything I do.

Apache Maven 2
Effective Implementation

This book offers a comprehensive look at using Maven on a project, covering not only the build system itself, but how it is best used in concert with other development infrastructures such as source control, continuous integration and build servers, and an artifact repository. We cover this territory using Subversion, Apache Continuum, and Apache Archiva, respectively, though the concepts learned should apply to other comparable systems.

In many ways, this is the book we've always wanted to write about Maven, and it takes a different approach to the existing Maven titles. Rather than being a reference or documentation for the software, it takes the approach of walking through a single example application and associated infrastructure in the same way that you would develop your own projects. For this purpose, we have crafted the example application Centrepoint—a simple but functional web application composed of several modules that itself interacts with Maven, Continuum, and Archiva.

We believe this book will not only show you how to use Maven, but how to use it *effectively*, covering concepts and best practices that should endure beyond the current versions of Maven and apply to your development infrastructure and teams in general.

What This Book Covers

Chapter 1: Maven in a Nutshell is a quick overview of the fundamentals of Maven—from creating a simple Maven project to basic plugin configuration to generating sites and reports. These are demonstrated in an easy to follow step-by-step process. By the end of the chapter, you should be able to apply and use the skills that you have learned to your own project.

Chapter 2: Staying in Control with Archiva introduces you to Archiva and its role in building software. You will learn the basics of installing and configuring it for internal use. It also shows you how Archiva complements Maven and how they can be used together efficiently.

Chapter 3: Building an Application Using Maven delves into the details on how to accurately set up and build an application using Maven. The Centrepoint project is introduced in this chapter. This is the sample application that will be used for the hands-on demonstrations throughout the book. You will see how Maven enforces convention over configuration while building the Centrepoint project.

Chapter 4: Application Testing with Maven goes through the various types of automated tests that can be executed from Maven. This includes unit testing, integration testing, and testing web applications using Selenium to name a few. Instrumenting tests, implementing test coverage, and reporting of test results are also covered.

Chapter 5: Reporting and Checks shows how to configure Maven to generate project reports and incorporate them in the generated site. This chapter also tackles the basics on enforcing certain rules or checks on your code such as conforming to code styles and standards, and finding common bugs in the code while building your application.

Chapter 6: Useful Maven Plugins discusses some of the Maven plugins, both from Apache Maven and from the Codehaus Mojo project, that may be of great help in your Maven builds. The functionality of these plugins range from keeping track of the source revision number for the build to executing external applications as part of the build. You will learn to identify when to use each plugin and how to configure them properly to address your need.

Chapter 7: Maven Best Practices illustrates the effective usage of Maven. You will learn tips and tricks for setting up your development environment to managing your project dependencies to making your builds portable and reproducible. By the end of this chapter, you should be able to apply what you've learned to your next project, or even to your current one.

Chapter 8: Continuum: Ensuring the health of your source code highlights the importance of continuous integration in software development through Continuum. It covers basic installation and set up, adding projects to Continuum, and effective configuration and build scheduling, at the same time demonstrating how it works in accordance with Maven and also with Archiva.

Chapter 9: Continuum in Depth deals with releasing projects using both Maven and Continuum. The different phases involved in the release process will be covered along with a bit of troubleshooting on the side. You will also learn about building multiple projects simultaneously in Continuum through parallel and distributed builds.

Chapter 10: Archiva in a Team gives you the more advanced features of Archiva and demonstrates how to configure it for use in a team. You will learn how to control access to a repository, how to take advantage of repository groups, how to make use of its reporting feature, and how to maintain your Archiva repositories.

Chapter 11: Archetypes covers Maven archetypes. It discusses some of the archetypes available—what their purpose is and what the generated project from each archetype looks like. You will also create a custom archetype specifically for the Centrepoint application, which will be used in the last chapter.

Chapter 12: Maven, Archiva, and Continuum in the Enterprise shows how to configure Archiva and Continuum effectively for use in the corporate environment. Tips on how to set up projects and repositories across multiple projects with respect to controlling who and what can be accessed by different teams covers the first half of the chapter. The second part demonstrates the web services feature of both applications by creating plugins for the Centrepoint application and using them to get information from Archiva and Continuum.

Appendix A: *Troubleshooting Maven* provides techniques for troubleshooting Maven. Incorrect POM or settings configuration, and dependency and download problems are a few of the usual suspects that will be covered here.

Appendix B: *Recent Maven Features* discusses the new features in Maven 2.1 and above. These features include password encryption, reactor project selection, and parallel downloads of dependencies.

Appendix C: *Migrating Archiva and Continuum Data* illustrates how to migrate data in Archiva and Continuum when upgrading to a higher version. How to switch to a different database from the built-in one is also discussed.

6
Useful Maven Plugins

Nobody can tell exactly how many Maven plugins exist today—since, like dependencies they can be retrieved from any specified remote repository, there are likely hundreds to choose from, and likely even more that have been custom written for use within the infrastructure of particular organizations.

A common practice for frameworks and tools that require build integration is to publish a Maven plugin to accomplish the task—and it is becoming increasingly common to encounter this as a standard part of the getting started section of a project you might hope to use. However, there are also a number of plugins that would be considered general purpose and handle some extended build cases in a wider variety of projects.

In this chapter, we will take a closer look at some of these plugins from two locations: those hosted as part of the **Apache Maven** project (`http://maven.apache.org/plugins/`), and a number of plugins from the **Codehaus Mojo** project (`http://mojo.codehaus.org/plugins.html`), which is oriented directly towards Maven plugin development. Some of these have been covered already in this book, so this will be an opportunity to examine their use in more depth, while others are new.

Where possible, we will apply the plugins to our example application to see how they can be used in practice, and then cover some of the other use cases and best practices for their use.

While this won't come close to covering all the plugins you are likely to encounter, with these common tools in your arsenal it will cover many of your Maven build needs, reducing the need for you to write your own plugins.

The Remote Resources plugin

Most projects will use the Resources plugin at some point, even if it isn't configured directly—it is standard in the default life cycle for any packaging that produces some type of artifact, bundling the resources found in `src/main/resources`.

However, what if you wanted to share those resources among multiple projects? The best approach to doing that is to store the resources in the repository and retrieve them for use in multiple builds—and that is where the Remote Resources plugin comes in.

First, we should note that this is not the only alternative for handling the scenario. The Dependency plugin's `unpack` goal is also quite capable of unpacking an artifact full of resources directly into the location that will be packaged.

However, the Remote Resources plugin offers several advantages:

1. Re-integration with the resources life cycle so that retrieved resources will automatically be processed in any goals in the `process-resources` phase.
2. The ability to perform additional processing on the resources (including the optional use of Velocity templates to generate the resources) before inclusion.
3. A specific bundle generation goal for creating the resource artifact in the first place.

These advantages can make the plugin very effective at dealing with some common scenarios. For example the inspiration for the creation of the plugin, and one of its more common uses, is to place aggregated license files within the final artifact.

There are other scenarios where the `dependency:unpack` goal remains more suitable—for example, the bundling of plugin configuration as seen in Chapter 5, *Reporting and Checks*. It is best to select the Remote Resources plugin when the files will be incorporated into the resources life cycle and the Dependency plugin when the files will be utilized independently.

Let's look at how to create a license file for our **Centrepoint** application. We will do this in two steps—the creation of the resource bundle that provides the generic resources for any project by the same organization, and the processing of the module resources.

Creating a Remote Resource bundle

Remote Resource bundles are regular JAR files packaged with additional information generated by the remote resource plugin's `bundle` goal. Creating a module follows the same process as with other JAR files.

In the example application, we will create the module outside of the Centrepoint multi-module hierarchy, so that it could (theoretically) be used by other projects from the same organization. This could be anywhere in source control, but we will assume it sits side-by-side with the `effectivemaven-parent` module in the workspace.

```
$ mvn archetype:generate -DartifactId=license-resources \
                         -DgroupId=com.effectivemaven
```

As this is not going to be a code project, the `src/main/java` and `src/test` directories can be removed from the generated content. We then continue to add the parent project to the POM, so the result 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>
  <parent>
    <groupId>com.effectivemaven</groupId>
    <artifactId>effectivemaven-parent</artifactId>
    <version>1-SNAPSHOT</version>
    <relativePath>../effectivemaven-parent/pom.xml</relativePath>
  </parent>
  <artifactId>license-resources</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>License Resource Bundle</name>
</project>
```

We will add the Remote Resources plugin shortly, but first let's create the resources that will be bundled. These are added to the `src/main/resources` like regular resources.

Consider the following Velocity template file, `src/main/resources/LICENSE.vm`:

```
## License Generator
#macro(showUrl $url)
  #if($url)
    ($url)
  #end
#end
```

```
This software is distributed under the following license(s):
#foreach ($l  in $project.licenses)
  - $l.name #showUrl ($l.url)
#end
#if (!$projectsSortedByOrganization.isEmpty())
The software relies on a number of dependencies. The individual
licenses are outlined below.
#set ($keys = $projectsSortedByOrganization.keySet())
#foreach ($o in $keys)
From: '$o.name' #showUrl($o.url)
#set ($projects = $projectsSortedByOrganization.get($o))
#foreach ($p in $projects)
  - $p.name #showUrl ($p.url)
    $p.artifact
#foreach ($l  in $p.licenses)
    License: $l.name #showUrl ($l.url)
#end
#end
#end
#end
```

For those not familiar with Velocity, the purpose of this is to first iterate through the project's licenses and list them, then secondly iterate through the project's dependencies (grouped by the organization they are from) and list their license. The `$projectsSortedByOrganization` variable is a special one added by the Remote Resources plugin to assist in this task.

Before we can move on to use the bundle, we need to add the plugin to the bundle project like so:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-remote-resources-plugin</artifactId>
      <version>1.0</version>
      <executions>
        <execution>
          <goals>
            <goal>bundle</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

This goal is required to generate a *bundle manifest*, the contents of which tell the plugin which resources to process when it is later called on to do so.

With this all in place, we can now install the bundle into the local repository, ready for use:

```
license-resources$ mvn install
```

If you were to inspect the contents of the generated JAR file, you would see both the `LICENSE.vm` file in the root, and the bundle manifest in `META-INF/maven/remote-resources.xml`. You would also find that the Velocity template is unmodified—the contents will be *executed* when the bundle is later processed in the target project, which we will proceed to look at now.

Processing Remote Resources in a project

Using the resource bundle we have created is now quite straightforward. We start by adding the folllowing to the `build` section of `modules/pom.xml` file of Centrepoint:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-remote-resources-plugin</artifactId>
  <version>1.0</version>
  <executions>
    <execution>
      <goals>
        <goal>process</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <resourceBundles>
      <resourceBundle>
        com.effectivemaven:license-resources:1.0-SNAPSHOT
      </resourceBundle>
    </resourceBundles>
  </configuration>
</plugin>
```

Here we have added a list of resource bundle artifacts to the configuration for the `process` goal, in the familiar shorthand artifact notation of `groupId:artifactId:version`. It has been added to the modules POM so that the license is included in the JAR files, but not included in the other non-code modules such as the documentation (which already generates a copy of the license from the reporting plugins).

Normally, you should use a released version of the license bundle, not a snapshot as we have here (as we have not yet covered the release process!). Since the bundle is configured directly and not through a dependency, the Release plugin will not detect this unresolved snapshot later.

Now, if we build a module such as `store-api`, we will see the license included in the root directory of the JAR file with the following content:

```
This software is distributed under the following license(s):
  - The Apache Software License, Version 2.0
      (http://www.apache.org/licenses/LICENSE-2.0.txt)

The software relies on a number of dependencies. The individual
licenses are outlined below.

From: 'Apache Maven 2: Effective Implementations Book'
      (http://www.effectivemaven.com/)

  - Centrepoint Data Model
    com.effectivemaven.centrepoint:model:jar:1.0-SNAPSHOT
    License: The Apache Software License, Version 2.0
      (http://www.apache.org/licenses/LICENSE-2.0.txt)

From: 'Google'
      (http://www.google.com/)

  - Guice
      (http://code.google.com/p/google-guice/)
    com.google.code.guice:guice:pom:1.0
    License: The Apache Software License, Version 2.0
      (http://www.apache.org/licenses/LICENSE-2.0.txt)
```

This is a good start, but we don't really need to include our own artifacts in the list, so we go back to the plugin declaration in `modules/pom.xml` and add another line of configuration:

```
<configuration>
  <excludeGroupIds>${project.groupId}</excludeGroupIds>
  <resourceBundles>
  ...
```

Regenerating the above artifact will alter the license to remove the dependencies from the project's group.

A different case is the final distribution. As this is not part of the `modules` hierarchy, first we need to include the plugin definition identical to the one added previously.

In the sample code for this chapter, you will notice that this has been taken a step further with the version and common configuration pushed into a `pluginManagement` section of the Centrepoint parent POM, and just the execution of the plugin goal remains in the `modules` and `distribution` POM files.

We can now build the assembly as usual:

```
distribution$ mvn clean install
```

Upon inspecting the generated assemblies, you will not see the license file included yet. This is because the Assembly plugin does not pick up Maven resources by default, as it does not participate in the normal life cycle.

To include the license file, we must alter the assembly descriptor `distribution/src/main/assembly/bin.xml` and add the following file set:

```
<fileSet>
  <directory>target/maven-shared-archive-resources</directory>
  <outputDirectory>/</outputDirectory>
</fileSet>
```

The directory given is the standard location in which the Remote Resources plugin stores the resources it has processed, so if you decide to configure that differently in your own projects you would need to change this to the corresponding location.

Upon building the assembly again we will see that the license has been generated, and that it includes the licenses of dependencies outside of the Centepoint application. As you can see, the distributed application depends on Jetty (also under the Apache License 2.0), which includes some portions of Glassfish (under the CDDL 1.0 License).

While the above technique can be very helpful in constructing some useful information about your project and its dependencies, it cannot be guaranteed to produce complete licensing information for a project. The method relies on accurate information in the POMs of your dependencies, and this can sometimes be inaccurate (particularly when using public repositories such as the Maven Central Repository). If you are redistributing your files, always confirm that you have correctly recorded any necessary licensing information that must accompany them!

The Remote Resources plugin is also capable of covering other scenarios that are particularly suited to license handling or more generally recording information about the project it is being processed for. These include:

1. The `supplementalDataModels` configuration option that allows you to fill in incomplete or incorrect metadata for a project dependency before the resources are processed (to avoid particular problems as described above).
2. The `appendedResourcesDirectory`, which allows you to store the above models in a separate file.
3. The `properties` configuration, which allows the injection of other build properties into the Velocity templates.

However, with this in mind, remember that the Remote Resources plugin is often just as suitable for any type of reusable resource, even if it is a static file.

The Build Number plugin

In Maven Mojos, the goals within a plugin are always designed to be simple tasks. Their aim is to do one thing, and do it well. A good example of this is the Build Number plugin. This simple plugin has one goal (`create`), with one purpose—to obtain a suitable build number and expose it to the build through properties or a file.

While the plugin focuses on exposing the current Subversion revision, it is capable of generating an incremented build number (stored in a specified properties file), and a representation of the current system date and time. This feature can be very useful in identifying the exact heritage of a particular build. The build number generated by the plugin is different to that used by Maven to identify snapshots or artifact versions. While it is possible that you might mark your version using the information it generates, this plugin is typically used to record information about a particular build—whether it is a snapshot, or a release—within the artifact itself as a permanent record.

Using the plugin is straightforward. By adding the goal to the project, the Subversion revision and a timestamp property will be exposed from the point that it is run onwards.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>buildnumber-maven-plugin</artifactId>
  <version>1.0-beta-1</version>
```

```
    <executions>
      <execution>
        <phase>generate-resources</phase>
        <goals>
          <goal>create</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

In this example, we execute the plugin in the `generate-resources` phase so that the properties are available to any resource processing. Note that the values could be used with the Remote Resources plugin that we have just seen.

There are two things to take into consideration with this configuration, however. Firstly, not all source builds will be Subversion checkouts, but the plugin does not verify that. To work around this potential problem, you can put the goal into a profile:

```
  <profile>
    <id>buildnumber</id>
    <activation>
      <file>
        <exists>.svn</exists>
      </file>
    </activation>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>buildnumber-maven-plugin</artifactId>
      ...
```

This particular activation check will cause the profile to be used within a Subversion checkout (that is, if `.svn` exists in the current directory), and to skip the plugin if not. In this case, the properties will not be set (or the file will not be created), so the code using these must take that into account.

Secondly, how the values will be accessed needs to be given careful consideration. For example, it is unlikely that you want to create a convoluted build processing step to filter the value into a particular JSP file to appear in a web application. For the sake of keeping the build simple (and speedy), it is best to write the values into a single file that the application can then load from its classpath.

This can be achieved by creating a filtered resource that contains references to the values. The advantage of this method is that it is automatic if you have already configured filtered resources, and automatically ends up in the classpath of the application code that can load the file as a resource.

The plugin also supports a number of formatting options, in the event that you do not wish to use the raw build number and timestamp integers. These options are also used to trigger the manual build number increment feature of the plugin in the event that Subversion is not used.

As our example application is one such case, let's apply this technique ourselves. We will only need to include the build number plugin in one location—if it were to be run on every module, the number would end up being different for each. Some seemingly sensible options may not actually be appropriate here:

- The parent project will cause the build number to be included in every module, and executing it just in the parent may result in it not being executed at all.
- The final distribution might be the best place in some situations, but what if you needed to reference the build number from the application itself?

Considering these factors, the best location is the web application module, as this will be the code which will eventually display the build number. In another application where multiple modules were to require it, it might be best generated in a common dependency instead.

The `modules/webapp/pom.xml` file requires two modifications. First, to enable filtering, we must add *both* the new filtered resources location and the original resources location to the `build` section (as Maven inheritance overrides resources instead of adding to them). This will look like the following:

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
  </resource>
  <resource>
    <directory>src/main/filtered-resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

Depending on what resources you have in `src/main/resources`, you may choose to do away with it and include them in the `filtered-resources` directory instead. However, if some resources may be corrupted by filtering (such as images and other binary files), you will need to retain both.

Next, the addition of the build number plugin is needed with some particular configuration:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>buildnumber-maven-plugin</artifactId>
  <version>1.0-beta-1</version>
  <configuration>
    <format>Build: #{0} ({1,date})</format>
    <items>
      <item>buildNumber\d*</item>
      <item>timestamp</item>
    </items>
  </configuration>
  <executions>
    <execution>
      <phase>generate-resources</phase>
      <goals>
        <goal>create</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Note the formatting options being used. The first, `format`, specifies what the configured property (by default called `buildNumber`) will look like, using the Java `MessageFormat` syntax. The `items` configuration provides a list of variables to substitute into the message format for `{0}`, `{1}`, and so on. Here, the first configured variable is `buildNumber\d*`, which is used to trigger the automatic number generation. The second is the integer timestamp that represents the current system date and time.

You may also configure the location of the file to be generated that contains the incremented build number, with the default being `buildNumber.properties` in the current directory. It is important to place this in a location that will not be purged over time—for example, in the `target` directory, as this is regularly cleaned!

The exposed property is referenced from the resource files using the normal filtering syntax, so we can create `src/main/filtered-resources/build.properties` like so:

```
build.message=${buildNumber}
```

When we build the application, we will see the build number generated and incremented each time:

```
[INFO] Storing buildNumber: Build: #1 (27/03/2009) at timestamp:
1238074038389
```

All that is left to do is to access the build number from the application, which you can see in the `BuildNumber` class:

```
ResourceBundle bundle = ResourceBundle.getBundle( "build" );
msg = bundle.getString( "build.message" );
```

As you can see, if you are using Subversion it is likely a simpler option to use the revision number instead of creating a new build number, or perhaps passing the build number in from your build server using its own build numbering scheme. Using an incremented number as we have above can be inconsistent depending on the build order and build successes, with the additional requirement of maintaining the separate tracking file.

The Shade plugin

Maven's dependency-based nature promotes the practice of proper componentization of a build and producing a set of discrete artifacts that can be aggregated into an application. Throughout a dependency tree, dependencies may appear multiple times and can be mediated to the correct version so that a single version can be used. However, circumstances will occur where, rather than the discrete list of dependencies, it is necessary to merge, hide, or alter the parts of a number of dependencies into a single artifact. This is the purpose for which the Shade plugin was developed, and in this section we will look at two use cases in more detail.

Building a standalone artifact

In Chapter 3, *Building an Application Using Maven*, we learned how to build a suitable distribution for our example application. This was a suitable set up for a server where there are multiple configuration files, startup scripts, as well as the application itself. But for some Java applications, all that is needed is to run `static void main()`.

Distributing such applications if they have dependencies on other JAR files has always been problematic. Java has a mechanism to run a JAR on its own through the `Main-Class` manifest attribute, providing us with a simple command such as the following:

```
client$ java -jar centrepoint-client-1.0-SNAPSHOT.jar
```

However, this will generally fail as the dependencies required are not present on the classpath. From here, you might add the `Class-Path` element to the manifest. However, this requires that the JARs be placed in a particular location relative to the other files, and we are back to distributing a complete archive.

To make this easier, the Assembly plugin contains a pre-built descriptor called `jar-with-dependencies`. This simply collapses the JAR and all it's dependencies into one large JAR so that the above type of execution can work. In some cases, this configuration will work correctly, and no more work needs to be done.

One of the issues with the naïve approach of the assembly plugin is that many of the dependencies will house files with identical names, and it is given little choice but to pick just the last file that it encounters rather than dealing with the overlap properly.

This is where the Shade plugin can be useful, by providing hooks to transform these resources into a single resource as appropriate.

This is similar in intent to the Uberjar plugin that you may have used in Maven 1, however the technique is improved. In the assembly plugin and the shade plugin, the resulting artifact collapses the dependency JARs and corrects references instead of having to unpack and construct classloaders, making it a much faster alternative.

For example, consider the following configuration:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.2</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer implementation="org.apache.maven.plugins.shade.
                        resource.ComponentsXmlResourceTransformer" />
        </transformers>
        <artifactSet>
          <excludes>
            <exclude>xerces:xercesImpl</exclude>
```

```
            </excludes>
          </artifactSet>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

Here we have the `shade` goal bound to the `package` phase to produce the collapsed JAR file as we might expect. In the `transformers` section, a transformer is added that merges all encountered Plexus descriptor files under `META-INF/plexus/components.xml`. Finally, we have the ability to configure the transitive dependencies that are included and excluded from the final artifact using the `artifactSet` configuration.

As of Shade version 1.2, the following transformers are supplied with the plugin:

- `ApacheLicenseResourceTransformer` and `ApacheNoticeResourceTransformer`: Specific to Apache license files such as those we created with the remote resources plugin earlier
- `AppendingTransformer`: To concatenate plain text resources
- `ComponentsXmlResourceTransformer`: For merging Plexus descriptors
- `ManifestResourceTransformer`: For merging Java JAR file manifests
- `ServicesResourceTransformer`: For merging Java services metadata in `META-INF/services`
- `XmlAppendingTransformer`: To concatenate XML resource files with appropriate nesting

These transformers will cover many of your needs when using the Shade plugin. You may eventually need your own custom transformer if there are certain types of resources that must be merged between two artifacts that are not accommodated by the above. In that case, a transformer can be written in a separate artifact and added as a plugin dependency, then referenced directly from the configuration. This is similar to the technique illustrated for sharing other build resources such as the reporting configuration files in Chapter 5, *Reporting and Checks*.

Shading dependencies

Another interesting use case of the Shade plugin is to hide or alter the dependencies of an artifact. This can be helpful in several situations, such as:

- Distributing classes from JARs that are difficult to access as dependencies (as long as such redistribution is allowed). For example, you may need to use a patched version of a common library and not want to cause conflict for projects that might depend on both your library and the original version.
- Avoiding conflicts with other instances of the dependency within a classloader or to work around incompatible versions in a dependency tree.
- Reducing the amount of code shipped by selectively including code from a particular dependency.

These situations are all similar and boil down to including some or all of the classes from a set of dependencies into the current artifact and altering the transitive dependency tree to compensate.

Let's consider this example within the context of the example application. In the example code for this chapter, we have extended the model to use `commons-lang`, which is used to construct the `equals` and `hashCode` methods. Now, if we were to share the API and model with third party developers to create their own store implementations, it would raise the same issues:

- Requiring the dependency of `commons-lang` (about 260K in size) for just a couple of classes.
- Introducing a dependency with a potentially stricter requirement than the rest of the application may later require (for example, if an incompatible, Java 5 enabled, `commons-lang` 3.0 is made available), again for very few classes.

In this situation, *shading in* just the portions of `commons-lang` that are needed may be a viable alternative. It is important to note that we can only do this if the license of the dependency allows such a combination, which in this case is true.

An important part of this scenario is that the dependency is being shared outside of the current application. Inside an application, where you have full control of the dependency tree, you are less likely to benefit from shading dependencies for this purpose.

To start with, we will add the Shade plugin configuration to the `model/pom.xml` file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.2</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <artifactSet>
          <includes>
            <include>commons-lang:commons-lang</include>
          </includes>
        </artifactSet>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Running the `package` command with this in place will show the following:

```
[INFO] [jar:jar]
[INFO] Building jar: /Users/brett/code/06/centrepoint/modules/model/
target/model-1.0-SNAPSHOT.jar
[INFO] [shade:shade {execution: default}]
[INFO] Including commons-lang:commons-lang:jar:2.3 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing /Users/brett/code/06/centrepoint/modules/model/target/
model-1.0-SNAPSHOT.jar with /Users/brett/code/06/centrepoint/modules/
model/target/model-1.0-SNAPSHOT-shaded.jar
```

Multiple things are happening here. First, we can see that the normal JAR is produced as usual, and then the `shade` goal runs (consistent with the execution we added to the project). The plugin next shows that because of the `artifactSet` that we gave, `commons-lang` will be included in the JAR. If there were other dependencies, they would remain a regular dependency and not be included. Finally, we see that using the Shade plugin's default configuration, the original artifact is replaced with the shaded artifact so that this version will be installed or used in the reactor instead.

However, we still have some work to do. Notice the size and contents of the JAR for the model now—so far, we have added **all** of `commons-lang`. We would only like to include the builder classes, so for that purpose we add **artifact filters** to the Shade plugin configuration:

```
<filters>
  <filter>
    <artifact>commons-lang:commons-lang</artifact>
    <includes>
      <include>org/apache/commons/lang/builder/**</include>
    </includes>
  </filter>
</filters>
```

If we rebuild the project, we see that the JAR has reduced in size and upon inspection would find that only the files under the given path are included. This also ensures other files that might cause confusion (such as the `pom.properties` file under `META-INF` from `commons-lang`) are not included.

Of course, some caution is required here—these classes may well have required some of the now-excluded classes that are no longer present. In your own projects, ensure that your integration test cases adequately exercise the code being used; so that inadvertent runtime errors don't occur later.

Beware the catch here that unit tests won't exercise this change! Unit tests run before the shading process occurs, so the tests need to occur in the integration test phase or in a separate testing module.

The next thing that we might notice about the contents of the JAR file is that the classes are still in their original packages under `org/apache/commons/lang`. This poses a potential problem for other projects that use `commons-lang` and depend on this API. There will now be two copies of these classes on the classloader that contains both JARs, and depending on the order, either copy of the classes could be used. If the wrong version is silently picked up, confusion will ensue on the part of the developer that appears to be getting the right version of the dependency, but the wrong behavior.

To prevent this, the Shade plugin allows us to encapsulate the use of `commons-lang` entirely by using the **relocations** feature.

```
<relocations>
  <relocation>
    <pattern>org.apache.commons.lang</pattern>
  </relocation>
</relocations>
```

Now when we build the artifact, we will see that the class names have changed, for example:

```
hidden/org/apache/commons/lang/builder/EqualsBuilder.class
```

This is more than a simple renaming—the Shade plugin has also adjusted the bytecode of these classes and the references in our own Java classes to rename the package to include `hidden` at the start. This will now avoid any conflict with other instances of `commons-lang`!

Though this conflict is removed, you may still need to consider the converse case, if a class depends on only being instantiated once to operate correctly, or is accessed by a string using `Class.forName` for example, shading may not work.

One slight adjustment should be made here, as the default pattern of prepending `hidden` is not always a good choice. This can be confusing to see in stack traces, and if two different dependencies shade the same classes in the same way then the likelihood of conflict occurs again. Instead, we can choose to include them within the namespace of our own package instead by a slight adjustment to the relocation configuration:

```
<relocation>
  <pattern>org.apache.commons.lang</pattern>
  <shadedPattern>
    com.effectivemaven.centrepoint.model.shaded.lang
  </shadedPattern>
</relocation>
```

A final thing is worth noting about the configuration we have used so far. Initially, the shaded artifact replaces the original artifact and is installed in the local repository. In addition to replacing the original artifact, you may notice that the POM file installed in the local repository is also replaced with a different version that removes the `commons-lang` dependency. This is what we would expect as we have completely incorporated our `commons-lang` needs within the classes of the JAR through the shading process. This is thanks to the `createDependencyReducedPom` configuration option being enabled by default.

In some scenarios, you may not wish to replace the artifact, but instead create an additional artifact, with a different classifier, that contains the shaded alternative. This can be a good middle ground to give users the choice between an all-in-one artifact, or the default artifact that allows Maven to manage the dependencies normally. In this case, the `createDependencyReducedPom` option should be set to `false`, and the `shadedArtifactAttached` option set to `true`. Bear in mind, however, that the dependency representation for the classified artifact in this instance will be incorrect as the main artifact POM is used for classified artifacts as well.

The Shade plugin configuration contains a number of other configuration options for altering inclusions and exclusions at each step of the process that we have seen here and in particular for altering the technique and naming of artifacts that are attached with a classifier. Refer to the Shade plugin documentation for more information: `http://maven.apache.org/plugins/maven-shade-plugin/shade-mojo.html`

Before we move on, you may have observed a potential side effect of this change; that there will now be *two* instances of each class in the application should another use `commons-lang`. If multiple dependencies shade in the same classes, there may be multiple copies so even though this is powerful it is worth being judicious about using the technique, particularly if you have good control over how your artifact will be used as a dependency. Maven's dependency mechanism and version management is there for a reason. If it is possible to continue using discreet artifacts as they are, then that may be the best and simplest alternative too!

The Build Helper plugin

Within Maven, there are a number of common tasks which plugins can perform to alter the current project for changes occurring during the build. We have seen the inclusion of new resources in the Remote Resources plugin, and the attachment of a new artifact from the Shade plugin. It is also possible to have a plugin generate new source code and include it for compilation, even though the directory is not included in the POM file.

The role of the Build Helper plugin is to provide a set of goals that can help achieve a collection of small but common tasks for which it would not be worth writing a custom plugin.

Adding source directories

Maven's inability to have multiple source directories in the project model has often been called into question. However, as time has progressed the request has died down as the idea of a standardized source structure took hold.

The Build Helper plugin offers the ability to add another source directory or test source directory to that configured in the POM. This is not necessarily to allow a workaround for the deliberate limitation in the project model, but rather to facilitate other use cases that require it. The most common need to use this technique is to assist with the migration of a project in an existing layout to Maven temporarily.

Even with this capability it is still recommended not to add multiple source directories without a particular reason—apart from breaking with convention, you may find that some tools that operate based on the values in the POM will not recognize the additional directories as containing source code.

The following example illustrates the addition of a source directory:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>add-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>src/main/more-java</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The need to use the Build Helper plugin for adding sources is now becoming more rare. Maven plugins that generate source code would be likely to add the extra directory to the project internally without the need for additional configuration. If some other means is used to generate the sources—for example, from a scripting plugin—it is common for the scripting plugin to have a way to add the source directory with fewer configurations than using the Build Helper plugin. However, if the need does arise, the Build Helper plugin will prove itself useful.

Attaching arbitrary artifacts

A similar scenario that can occur is the generation of additional artifacts that need to be *attached* to the build process. This means they use the same POM to define them, but are different types of related build artifacts, with their own classifier. The artifacts are installed and deployed to the repository alongside the original.

Typically, this will be in the form of another JAR file, possibly generated by one of the scripting plugins that did not attach the artifact itself.

However, it could be used for any number of files that need to be stored in the repository alongside the main artifact. Consider the example of deploying the license to the repository—if you were to run the install phase on the given project, you would be able to have the license installed into the local repository alongside the main artifact and its POM.

In reality, this particular configuration may be overkill, especially if the licenses are identical across many projects, or can be derived from the POM. However, depending on your deployment needs this possibility can be helpful in ensuring the repository contains the information about an artifact that you need, at the time it was deployed, in addition to any extra build artifacts that might be generated.

In our example application, we generated the license in two places—in all the Java modules, and the final distribution. Deploying it along with the final distribution makes some sense, so let's add it to the `distribution/pom.xml` file:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.1</version>
  <configuration>
    <artifacts>
      <artifact>
        <file>
          target/maven-shared-archive-resources/LICENSE
        </file>
        <type>txt</type>
        <classifier>license</classifier>
      </artifact>
    </artifacts>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>attach-artifact</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

This goal will execute after the packaging has occurred, but before installation so that it can be attached to the installation (and deployment) process. The file to attach is the license generated earlier by the Remote Resources plugin and is given an extension of `.txt` and classifier of `-license`. When running the `install` phase, we now see the file being processed:

```
[INFO] [build-helper:attach-artifact {execution: default}]
[INFO] [enforcer:enforce {execution: default}]
[INFO] [install:install]
[INFO] Installing /Users/brett/code/06/centrepoint/distribution/target/
pom-transformed.xml to /Users/brett/.m2/repository/com/effectivemaven/
centrepoint/distribution/1.0-SNAPSHOT/distribution-1.0-SNAPSHOT.pom
[INFO] Installing /Users/brett/code/06/centrepoint/distribution/target/
centrepoint-1.0-SNAPSHOT-bin.zip to /Users/brett/.m2/repository/com/
effectivemaven/centrepoint/distribution/1.0-SNAPSHOT/distribution-1.0-
SNAPSHOT-bin.zip
[INFO] Installing /Users/brett/code/06/centrepoint/distribution/target/
centrepoint-1.0-SNAPSHOT-bin.tar.gz to /Users/brett/.m2/repository/com/
effectivemaven/centrepoint/distribution/1.0-SNAPSHOT/distribution-1.0-
SNAPSHOT-bin.tar.gz
[INFO] Installing /Users/brett/code/06/centrepoint/distribution/target/
maven-shared-archive-resources/LICENSE to /Users/brett/.m2/repository/
com/effectivemaven/centrepoint/distribution/1.0-SNAPSHOT/distribution-
1.0-SNAPSHOT-license.txt
```

Other goals

The Build Helper plugin also contains some other goals in the latest release at the time of writing (v1.1) of more specific interest:

- `remove-project-artifact`: To clean the local repository of artifacts from the project being built to preserve space and remove outdated files. This may occur if the build no longer produces those files, or if it is necessary to remove older versions.
- `reserve-network-port`: Many networked applications may want to use a network port that doesn't conflict with other test cases. This goal can help reserve unique ports to use in the tests. This is useful for starting servers in integration tests and then referencing them in the test cases. However, note that it won't be available when running such tests in a non-Maven environment such as the IDE.

The goals available in the Build Helper plugin may increase over time, so if you have some small, common adjustments to make it is a good place to look to first for those utilities.

The AntRun plugin and scripting languages

Maven was designed to be extended through plugins. Because of the fact that this is so strongly encouraged, there are now many plugins available for a variety of tasks, and the need to write your own customizations, particularly for common tasks, is reduced. However, no two projects are the same, and in some projects, there are likely to be some customizations that will need to be made that are not covered by an existing plugin.

While it is virtuous to write a plugin for such cases so that it can be reused in multiple projects, it is also very reasonable to use some form of scripting for short, one off customizations.

One simple option is to use the AntRun plugin. Ant still contains the largest available set of build tasks to cover the types of customizations that you might need in your build, and through this plugin you can quickly string together some of these tasks within the Maven life cycle to achieve the outcome that you need.

Running simple tasks

We have already used the AntRun plugin in the distribution module of the example application. This snippet was used to copy some configuration files into place and create a `logs` directory, ready for the Assembly plugin to create the archive from:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <id>config</id>
      <phase>process-resources</phase>
      <configuration>
        <tasks>
          <copy todir="${project.build.directory}/generated-
                       resources/appassembler/jsw/centrepoint/conf">
            <fileset dir="src/main/conf" />
          </copy>
          <mkdir dir="${project.build.directory}/generated-
                      resources/appassembler/jsw/centrepoint/logs" />
        </tasks>
      </configuration>
      <goals>
```

```
            <goal>run</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

This shows how quick and useful the AntRun plugin can be for simple tasks. However, it also contains a number of other features that can be of benefit to the build for more significant tasks.

Interacting with the Maven project

As we mentioned in the section, *The Build Helper plugin*, you can tell the plugin to map some directories to new source directories. This functionality is identical to that of the Build Helper plugin, but is more conveniently located when the directories are being generated by Ant tasks.

This can be useful because even though tools are increasingly supplying native Maven plugins in addition to Ant tasks, you might come across a source generation tool that only has an Ant task. In this scenario, you can use the AntRun plugin to run the tool, generate the source code, and use the `sourceRoot` parameter to have that directory added back into the build life cycle.

In addition to injecting source directories back into the life cycle, the AntRun plugin also injects Maven project information into Ant's context. Probably the most important of these is the availability of the project's and plugin's dependencies as Ant path references:

- `maven.compile.classpath`: The dependencies in the compile scope (this syntax will look familiar to those that used Maven 1's built in Ant-based files)
- `maven.runtime.classpath`: The dependencies in the runtime scope (including the above)
- `maven.test.classpath`: The dependencies in the test scope (including both of the above)
- `maven.plugin.classpath`: The dependencies of the AntRun plugin itself, including any added via the POM

Though we have not needed it in the example application, to illustrate how these two options would work, consider if you needed to use the XJC Ant task from JAXB to generate some sources.

JAXB is a **Java-to-XML** binding framework that can be used to generate Java source code from XML schema (among many other things), using its XJC tool. Though it serves as a suitable example here, you would not be faced with this issue with JAXB itself, as it now offers a Maven plugin.

In this example, you might add the following configuration to an AntRun execution in a POM file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.3</version>
  <executions>
    <execution>
      <id>xjc</id>
      <phase>generate-sources</phase>
      <configuration>
        <tasks>
          <taskdef name="xjc"
              classname="com.sun.tools.xjc.XJCTask"
              classpathref="maven.plugin.classpath" />
          <xjc destdir="${project.build.directory}/xjc"
              schema="src/main/jaxb/schema.xsd">
              <classpath refid="maven.compile.classpath" />
          </xjc>
        </tasks>
        <sourceRoot>${project.build.directory}/xjc</sourceRoot>
      </configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>com.sun.xml.bind</groupId>
      <artifactId>jaxb-xjc</artifactId>
      <version>2.1.9</version>
    </dependency>
  </dependencies>
</plugin>
```

We see here that the XJC Task is defined using the plugin classpath to locate the task and its dependencies (and that task's artifact is added as a plugin dependency to accommodate this). Additional built-in Ant tasks would also be added as plugin dependencies (such as `ant-nodeps`).

AntRun and Ant versions

While in some cases they might be compatible, generally you should use the same version of the Ant optional tasks as the version of Ant itself. The version of Ant used by the plugin is predetermined by what it has been built against. In AntRun v1.3, that is Ant 1.7.1. To use a different version of Ant, consider a different version of the AntRun plugin.

Next, the task is run—being passed the project's dependencies and schema to generate the source code from. The source code is output to `target/xjc`, which is also added as a source directory by the AntRun plugin because of the configuration specified. As the task runs in the `generate-sources` phase, it is available for compilation in the same way as any other source code.

Again, the configuration of AntRun here has been relatively simple, and is completely integrated with the Maven artifact handling and build life cycle such that it would not likely be needed to write a plugin to wrap the tool completely if you were faced with this decision in your environment.

Converting Ant or Maven 1 builds

The AntRun plugin can be most useful when it comes to converting an existing build from Ant or Maven 1 (if it used custom Ant-based plugins or `maven.xml` heavily).

The approach to converting such a build varies depending on the project. For some projects, it is easier to start over on the build and map in modules one by one, interacting via the repository. For others, it might be a matter of gradually turning the existing script into a POM file and using AntRun to execute the existing code for the unconverted parts, keeping the flow wrapped in the new Maven life cycle.

The topic of build conversion is covered in more depth in the Better Builds with Maven section in Chapter 8, available at `http://www.maestrodev.com/better-build-maven`.

Let's look at another example. The following execution might be added to the AntRun plugin, with the same pattern repeated for other such examples in different parts of the life cycle:

```
<execution>
  <id>gen-src</id>
  <phase>generate-sources</phase>
  <configuration>
    <tasks>
      <ant antfile="build-migration.xml" target="gen-src" />
    </tasks>
  </configuration>
  <sourceRoot>${project.build.directory}/gen-src</sourceRoot>
  <goals>
    <goal>run</goal>
  </goals>
</execution>
```

In this source generation example, the previous build file is executed to perform a certain task. As before, the resultant directory is added back into the Maven project so that Maven can continue to control the build.

You will notice that the original script (perhaps broken up) is retained and called rather than pasting the script fragment into the POM. While either is acceptable (as long as the script does not contain multiple targets), it is a good idea to keep the POM as trim as possible. Unless the Ant script is just one or two lines long, it is better to put those commands into an external build file to execute.

This advice applies even if not performing a build conversion. If your Maven build contains script fragments that are longer than a few lines, consider placing them outside of the POM file. However, bear in mind that in doing so it will not be available when reading the POM from the repository later.

In some conversions, a particular task may be reusable in multiple places or projects (particularly if migrating a Maven 1 plugin). In this case, instead of the AntRun plugin, you may consider writing your own plugin for the task. Luckily, to reduce the work involved Maven also offers the ability to run plugins written in Ant.

Maven plugins written in Ant

While the AntRun plugin offers a convenient way to string some simple tasks together, as with any part of the build process in Maven it is worth taking into consideration a simple rule of thumb: if you might use it twice, consider writing a plugin.

Of course, if you do take that step, it is not required to write the plugin in Ant. However, having the option available is useful as:

- It will be easier to use for converting from a previous Ant/Maven 1 build
- Ant tasks may be more familiar to your team than writing Maven plugins in another language
- It can be easier to put together a set of Ant tasks for certain procedures than to write the corresponding code in another language such as Java

The process for creating Ant plugins involves the following steps:

1. Create a Maven project of type `maven-plugin`.
2. Add the `maven-plugin-plugin` to the `build` section, including a plugin dependency on `maven-plugin-tools-ant`.
3. Add an Ant build script for each goal to `src/main/scripts` directory under the name `goalName.build.xml`. This should be a completely executable Ant script containing one target definition.
4. Add a Mojo definition for each goal to `src/main/scripts` directory under the name `goalName.mojo.xml`. This defines the mapping of Maven information to the Ant target.

Plugin authoring (in Ant or other languages) is not something we will go into detail on in this chapter. For more information on writing plugins in Ant, see the documentation: `http://maven.apache.org/guides/plugin/guide-ant-plugin-development.html`.

Other scripting languages

While this section has focused on Ant as one of the most common scripting tools for builds, it is worth noting that other scripting languages can be used both for executable fragments (such as the fragments that AntRun is used for) and whole plugins (such as the Ant plugin tools are used for).

If you have expertise in a particular scripting language, and would like to use that for your own plugins or to add fragments to the build, you might consider one or more of the following projects:

- **GMaven**: This is a mature solution for writing plugins in Groovy, and running Groovy scripts from the POM. For more information, see `http://groovy.codehaus.org/GMaven`.

- JRuby Maven Plugin: This is an early but functional tool for writing plugins in JRuby, or running Ruby scripts from the POM. For more information, see `http://mojo.codehaus.org/jruby-maven-plugin/howto.html`.
- Script Maven Plugin: This uses **BSF** (**Bean Scripting Framework**) for running scripts in other languages directly from the POM. Supports a larger number of languages, but does not allow the creation of native Maven plugins from the source. For more information, see `http://mojo.codehaus.org/script-maven-plugin`.

At the time of writing, the Script Maven Plugin has not had an official release and needs to be built from source.

The Exec plugin

In some build situations, the best (or only!) way to get something done might be to run an external application, or a piece of Java code. It was for this purpose that the Exec plugin was created.

The plugin contains two goals: `exec:exec` and `exec:java`. They are similar in purpose, however the `java` goal sets up an easier way to pass Java configuration and execution parameters to a forked JVM instance, whereas the `exec` goal simply runs any executable available on the system.

To use the `exec` goal, you pass the path of the executable program in the `executable` configuration option, and optionally can add the `environmentVariables` or `arguments` configuration to run the command as desired.

Portability

Keep in mind the question of portability when using the `exec` goal. Some executables may not be available on all platforms that the build will run on. You may need to clearly state the requirement in the build's documentation, gracefully degrade the functionality, or use a profile to run a different executable on a different platform.

In contrast, the `java` goal automatically locates the JVM executable to run, and instead you provide the `mainClass` configuration option. The `arguments` configuration can again be given, but can also include a `classpath` option that helps construct Java `-classpath` arguments from the Maven build (though it is also possible to pass this to the `exec` goal if you happen to be running a particular `java` executable with it). You may also configure `systemProperties` for the plugin to pass to the forked JVM instance.

By default, the `java` goal will construct its base classpath from the current project's dependencies, though it is also possible to have it pass the plugin's dependencies as well or instead of the project dependencies. For more information on configuring either of the plugin goals, refer to the plugin website at: `http://mojo.codehaus.org/exec-maven-plugin`.

Both of these goals provide a useful way to run external processes, depending on what type of application it is. However, the context that it will be used in is also important. For execution there are often two scenarios—integrating the external process into the build life cycle (much like the examples we have seen with AntRun previously), or pre-configuring the plugin to be able to be run from the command line for a given project.

Adding the Exec plugin to the Build life cycle

In the first scenario, there may be an external tool or Java application that needs to be run at a certain point in the build for which the plugin should be configured. This occurs in the same way as for any other plugin, using a particular execution.

In *The AntRun plugin and scripting languages*, we looked at running the XJC tool from Ant. Another alternative could be to run the tool from the command line (assuming it had been pre-installed in the path):

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.1.1</version>
  <executions>
    <execution>
      <id>xjc</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>exec</goal>
      </goals>
      <configuration>
        <executable>xjc</executable>
```

```
          <arguments>
            <argument>-d</argument>
            <argument>${project.build.directory}/xjc</argument>
            <argument>src/main/jaxb/schema.xsd</argument>
          </arguments>
          <sourceRoot>${project.build.directory}/xjc</sourceRoot>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

For this particular scenario of source generation, the Exec plugin also contains a `sourceRoot` and `testSourceRoot` parameter for adding any generated sources into the build for later—exactly like the AntRun plugin.

Notice in the example that the configuration is inside the execution element, and therefore will not be able to be run from the command line. This is usually the best course of action when binding to the life cycle so that multiple instances are possible.

Running the Exec plugin standalone

There are also a number of uses for running the Exec plugin directly from the command line. This might be used with a particular project, or run independently.

A common reason to use this approach is if a particular Maven project produces an executable JAR—we looked at such an example with the Shade plugin earlier. By pre-configuring the Exec plugin with the necessary information, the JAR can be easily run by Maven itself. The following POM snippet from the Archiva XMLRPC Client illustrates this:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.1.1</version>
  <configuration>
    <mainClass>
      org.apache.archiva.web.xmlrpc.client.SampleClient
    </mainClass>
    <arguments>
      <argument>http://127.0.0.1:8081/xmlrpc</argument>
      <argument>admin</argument>
      <argument>${password}</argument>
    </arguments>
  </configuration>
</plugin>
```

Here, the project dependencies are included and the given arguments passed to the `main` function of the class specified. Notice that in contrast to the previous example, the configuration is not in an execution, both because it is not bound to the life cycle, but also so that it can be run from the command line.

Executing this is now a matter of running the following command:

```
$ mvn exec:java -Dpassword=ADMIN_PASSWORD
```

Clearly, this is much easier to remember than the full set of arguments!

In similar scenarios, the Exec plugin can become very useful in being able to give a simple demonstration or quick use of an application while in development. In some ways, it can be compared to `jetty:run` for web applications as bringing the same functionality to executable standalone applications.

Summary

As we learned early in the first chapter, once you have the right framework in place with Maven, it is a matter of finding and selecting the right combination of plugins to assemble your build from there on. Here, we have seen a selection of such plugins to help achieve some common builds goals.

The list doesn't stop here though. If you are looking to integrate a particular tool or framework into your build, see if a plugin is already available for it. A number of plugins such as the Dependency plugin, Enforcer plugin, Assembly and App Assembler plugins, discussed elsewhere in the book, have more goals and configurations that are worth investigating as well. If you have some other needs that might apply to multiple builds, search for a plugin to serve that case, or beyond that write one yourself in a scripting language, as a set of Ant tasks, or your own Maven plugin. Online Maven repository search engines can be helpful in finding other plugins to use.

In the first half of this book, we have covered all the pieces of the build puzzle needed to build an application with Maven—from the basics to dependency management, multi-module applications, distribution, reporting, and now a variety of plugins to augment build functionality.

With this in place, in the next chapter we are going to review what we have learned about Maven so far and lay out some best practices to keep in mind as you write or review your own builds.

Where to buy this book

You can buy Apache Maven 2 Effective Implementation from the Packt Publishing website: `http://www.packtpub.com/apache-maven-2-effective-implementations/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com