Developing Custom Maven Plugins

This chapter covers:

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

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

5.1. Introduction

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

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

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

5.2. A Review of Plugin Terminology

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

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

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

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

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

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

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

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

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

5.3. Bootstrapping into Plugin Development

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

5.3.1. The Plugin Framework

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

Participation in the build life cycle

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

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

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

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

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

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

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

Accessing build information

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

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

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

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

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

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

```
${patchDirectory}
```

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

The plugin descriptor

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

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

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

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

5.3.2. Plugin Development Tools

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

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

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

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

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

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

Here, the annotation identifies this field as a mojo parameter. This parameter annotation also specifies two attributes, expression and default-value. The first specifies that this parameter's default value should be set to false. The second specifies that this parameter can also be configured from the command line as follows:

-Dclean.verbose=false

Moreover, it specifies that this parameter can be configured from the POM using:

```
<configuration>
  <verbose>false</verbose>
</configuration>
```

You may notice that this configuration name isn't explicitly specified in the annotation; it's implicit when using the `@parameter` annotation.

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

```
private boolean verbose = false;
```

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

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

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

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

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

Choose your mojo implementation language

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

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

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

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

5.3.3. A Note on the Examples in this Chapter

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

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

```
mvn install
```

5.4. Developing Your First Mojo

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

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

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

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

Using the archetype plugin to generate a stub plugin project

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

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

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

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

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

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

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

The mojo

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

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

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

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

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

        Properties sysprops = System.getProperties();

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

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

                buildInfo.addSystemProperty( key, value );
            }
        }
```

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

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

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

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

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

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

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

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

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

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

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

The plugin POM

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

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

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

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

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

Binding to the life cycle

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

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

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

The output

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

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

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

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

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

```
[...]

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

[...]
```

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?><buildinfo>
  <systemProperties>
    <os.name>Linux</os.name>
    <java.version>1.4</java.version>
  </systemProperties>
</buildinfo>
```

While the name of the OS may differ, the output of of the generated build information is clear enough. Your mojo has captured the name of operating system being used to execute the build and the version of the jvm, and both of these properties can have profound effects on binary compatibility.

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

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

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

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

The Ant target

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

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

      <to>${listAddr}</to>

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

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

The mojo metadata file

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

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

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

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

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

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

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

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

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

Modifying the plugin POM for Ant mojos

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

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

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

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

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

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

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

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

Binding the notify mojo to the life cycle

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

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

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

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

Now, execute the following command:

```
> mvn deploy
```

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

5.5. Advanced Mojo Development

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

5.5.1. Accessing Project Dependencies

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

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

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

Injecting the project dependency set

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

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

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

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

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

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

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

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

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

Requiring dependency resolution

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

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

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

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

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

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

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

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

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

`BuildInfo` example: logging dependency versions

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

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

This will result in the addition of a new section in the `buildinfo` file, which enumerates all the dependencies used in the build, along with their versions – including those dependencies that are resolved transitively. Once you have access to the project dependency set, you will need to iterate through the set, adding the information for each individual dependency to your `buildinfo` object.

The code required is as follows:

```
if ( dependencies != null && !dependencies.isEmpty() )
{
    for ( Iterator it = dependencies.iterator(); it.hasNext(); )
    {
        Artifact artifact = (Artifact) it.next();
        ResolvedDependency rd = new ResolvedDependency();

        rd.setGroupId( artifact.getGroupId() );
        rd.setArtifactId( artifact.getArtifactId() );
        rd.setResolvedVersion( artifact.getVersion() );
        rd.setOptional( artifact.isOptional() );
        rd.setScope( artifact.getScope() );
        rd.setType( artifact.getType() );

        if ( artifact.getClassifier() != null )
        {
            rd.setClassifier( artifact.getClassifier() );
        }

        buildInfo.addResolvedDependency( rd );
    }
}
```

When you re-build the plugin and re-run the Guinea Pig build, the extract mojo should produce the same `buildinfo` file, with an additional section called `resolvedDependencies` that looks similar to the following:

```
<resolvedDependencies>
  <resolvedDependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <resolvedVersion>3.8.1</resolvedVersion>
    <optional>false</optional>
    <type>jar</type>
    <scope>test</scope>
  </resolvedDependency>
  [...]
  <resolvedDependency>
    <groupId>com.mergere.mvnbook.guineapig</groupId>
    <artifactId>guinea-pig-api</artifactId>
    <resolvedVersion>1.0-20060210.094434-1</resolvedVersion>
    <optional>false</optional>
    <type>jar</type>
    <scope>compile</scope>
  </resolvedDependency>
  [...]
</resolvedDependencies>
```

The first dependency listed here, `junit`, has a static version of 3.8.1. This won't add much insight for debuggers looking for changes from build to build, but consider the next dependency: `guinea-pig-api`. This dependency is part of the example development effort, and is still listed with the version `1.0-alpha-SNAPSHOT` in the POM. The actual snapshot version used for this artifact in a previous build could yield tremendous insight into the reasons for a current build failure, particularly if the newest snapshot version is different.

If you were using a snapshot version from the local repository which has not been deployed, the `resolvedVersion` in the output above would be `1.0-alpha-SNAPSHOT`. This is because snapshot time-stamping happens on deployment only.

5.5.2. Accessing Project Sources and Resources

In certain cases, it's possible that a plugin may be introduced into the build process when a profile is activated. If this plugin adds resources like images, or new source code directories to the build, it can have dramatic effects on the resulting project artifact. For instance, when a project is built in a JDK 1.4 environment, it may be necessary to augment a project's code base with an additional source directory. Once this new source directory is in place, the compile mojo will require access to it, and other mojos may need to produce reports based on those same source directories. Therefore, it's important for mojos to be able to access and manipulate both the source directory list and the resource definition list for a project.

Adding a source directory to the build

Although the POM supports only a single `sourceDirectory` entry, Maven's concept of a project can accommodate a whole list of directories. This can be very useful when plugins generate source code, or simply need to augment the basic project code base. Maven's project API bridges this gap, allowing plugins to add new source directories as they execute. It requires access to the current `MavenProject` instance only, which can be injected into a mojo using the following code:

```
/**
 * Project instance, used to add new source directory to the build.
 * @parameter default-value="${project}"
 * @required
 * @readonly
 */
private MavenProject project;
```

This declaration identifies the project field as a required mojo parameter that will inject the current `MavenProject` instance into the mojo for use. As in the prior project dependencies discussion, this parameter also adds the `@readonly` annotation. This annotation tells Maven that users cannot modify this parameter, instead, it refers to a part of the build state that should always be present (a more in-depth discussion of this annotation is available in section 3.6, Chapter 3 of this book). The current project instance is a great example of this; any normal build will have a current project, and no other project contains current state information for this build.

It is possible that some builds won't have a current project, as in the case where the maven-archetype-plugin is used to create a stub of a new project. However, mojos require a current project instance to be available, unless declared otherwise. Maven will fail the build if it doesn't have a current project instance and it encounters a mojo that requires one. So, if you expect your mojo to be used in a context where there is no POM – as in the case of the archetype plugin – then simply add the class-level annotation: `@requiresProject` with a value of false, which tells Maven that it's OK to execute this mojo in the absence of a POM.

Once the current project instance is available to the mojo, it's a simple matter of adding a new source root to it, as in the following example:

```
project.addCompileSourceRoot( sourceDirectoryPath );
```

Mojos that augment the source-root list need to ensure that they execute ahead of the compile phase. The generally-accepted binding for this type of activity is in the generate-sources life-cycle phase. Further, when generating source code, the accepted default location for the generated source is in:

```
${project.build.directory}/generated-sources/<plugin-prefix>
```

While conforming with location standards like this is not required, it does improve the chances that your mojo will be compatible with other plugins bound to the same life cycle.

Adding a resource to the build

Another common practice is for a mojo to generate some sort of non-code resource, which will be packaged up in the same jar as the project classes. This could be a descriptor for binding the project artifact into an application framework, as in the case of Maven itself and the components.xml file found in all maven artifacts. Many different mojo's package resources with their generated artifacts such as web.xml files for servlet engines, or wsdl files for web services.

Whatever the purpose of the mojo, the process of adding a new resource directory to the current build is straightforward and requires access to the MavenProject and MavenProjectHelper:

```
/**
 * Project instance, used to add new source directory to the build.
 * @parameter default-value="${project}"
 * @required
 * @readonly
 */
private MavenProject project;
```

This declaration will inject the current project instance into the mojo, as discussed previously. However, to simplify adding resources to a project, the mojo also needs access to the `MavenProjectHelper` component. This component is part of the Maven application, which means it's always present; so your mojo simply needs to ask for it. The project helper component can be injected as follows:

```
/**
 * project-helper instance, used to make addition of resources
 * simpler.
 * @component
 */
private MavenProjectHelper helper;
```

Right away, you should notice something very different about this parameter. Namely, that it's not a parameter at all! In fact, this is what Maven calls a *component requirement* (it's a dependency on an internal component of the running Maven application). To be clear, the project helper is not a build state; it is a utility.

Component requirements are simple to declare; in most cases, the unadorned `@component` annotation – like the above code snippet – is adequate. Component requirements are not available for configuration by users.

Normally, the Maven application itself is well-hidden from the mojo developer. However, in some special cases, Maven components can make it much simpler to interact with the build process. For example, the `MavenProjectHelper` is provided to standardize the process of augmenting the project instance, and abstract the associated complexities away from the mojo developer. It provides methods for attaching artifacts and adding new resource definitions to the current project.

A complete discussion of Maven's architecture – and the components available – is beyond the scope of this chapter; however, the `MavenProjectHelper` component is worth mentioning here, as it is particularly useful to mojo developers.

With these two objects at your disposal, adding a new resource couldn't be easier. Simply define the resources directory to add, along with inclusion and exclusion patterns for resources within that directory, and then call a utility method on the project helper. The code should look similar to the following:

```
String directory = "relative/path/to/some/directory";
List includes = Collections.singletonList("**/*");
List excludes = null;

helper.addResource(project, directory, includes, excludes);
```

The prior example instantiates the resource's directory, inclusion patterns, and exclusion patterns as local variables, for the sake of brevity. In a typical case, these values would come from other mojo parameters, which may or may not be directly configurable.

Again, it's important to understand where resources should be added during the build life cycle. Resources are copied to the classes directory of the build during the process-resources phase. If your mojo is meant to add resources to the eventual project artifact, it will need to execute ahead of this phase. The most common place for such activities is in the generate-resources life-cycle phase. Again, conforming with these standards improves the compatibility of your plugin with other plugins in the build.

Accessing the source-root list

Just as some mojos add new source directories to the build, others must read the list of active source directories, in order to perform some operation on the source code. The classic example is the compile mojo in the maven-compiler-plugin, which actually compiles the source code contained in these root directories into classes in the project output directory. Other examples include javadoc mojo in the maven-javadoc-plugin, and the jar mojo in the maven-source-plugin. Gaining access to the list of source root directories for a project is easy; all you have to do is declare a single parameter to inject them, as in the following example:

```
/**
 * List of source roots containing non-test code.
 * @parameter default-value="${project.compileSourceRoots}"
 * @required
 * @readonly
 */
private List sourceRoots;
```

Similar to the parameter declarations from previous sections, this parameter declaration states that Maven does not allow users to configure this parameter directly; instead, they have to modify the `sourceDirectory` element in the POM, or else bind a mojo to the life-cycle phase that will add an additional source directory to the build. The parameter is also required for this mojo to execute; if it's missing, the entire build will fail.

Now that the mojo has access to the list of project source roots, it can iterate through them, applying whatever processing is necessary. Returning to the `buildinfo` example, it could be critically important to track the list of source directories used in a particular build, for eventual debugging purposes. If a certain profile injects a supplemental source directory into the build (most likely by way of a special mojo binding), then this profile would dramatically alter the resulting project artifact when activated. Therefore, in order to incorporate list of source directories to the `buildinfo` object, you need to add the following code:

```
for ( Iterator it = sourceRoots.iterator(); it.hasNext(); )
{
    String sourceRoot = (String) it.next();

    buildInfo.addSourceRoot( makeRelative( sourceRoot ) );
}
```

One thing to note about this code snippet is the `makeRelative()` method. By the time the mojo gains access to them, source roots are expressed as absolute file-system paths. In order to make this information more generally applicable, any reference to the path of the project directory in the local file system should be removed. This involves subtracting `${basedir}` from the source-root paths. To be clear, the `${basedir}` expression refers to the location of the project directory in the local file system.

When you add this code to the extract mojo in the `maven-buildinfo-plugin`, it will add a corresponding section to the `buildinfo` file that looks like the following:

```
<sourceRoots>
  <sourceRoot>src/main/java</sourceRoot>
  <sourceRoot>some/custom/srcDir</sourceRoot>
</sourceRoots>
```

Since a mojo using this code to access project source-roots does not actually modify the build state in any way, it can be bound to any phase in the life cycle. However, as in the case of the extract mojo, it's better to bind it to a later phase like package if capturing a complete picture of the project is important. Remember, binding this mojo to an early phase of the life cycle increases the risk of another mojo adding a new source root in a later phase. In this case however, binding to any phase later than compile should be acceptable, since compile is the phase where source files are converted into classes.

Accessing the resource list

Non-code resources complete the picture of the raw materials processed by a Maven build. You've already learned that mojos can modify the list of resources included in the project artifact; now, let's learn about how a mojo can access the list of resources used in a build. This is the mechanism used by the resources mojo in the maven-resources-plugin, which copies all non-code resources to the output directory for inclusion in the project artifact.

Much like the source-root list, the resources list is easy to inject as a mojo parameter. The parameter appears as follows:

```
/**
 * List of Resource objects for the current build, containing
 * directory, includes, and excludes.
 * @parameter default-value="${project.resources}"
 * @required
 * @readonly
 */
private List resources;
```

Just like the source-root injection parameter, this parameter is declared as required for mojo execution and cannot be edited by the user. In this case, the user has the option of modifying the value of the list by configuring the resources section of the POM.

As noted before with the dependencies parameter, allowing direct configuration of this parameter could easily produce results that are inconsistent with other resource-consuming mojos. It's also important to note that this list consists of Resource objects, which in fact contain information about a resource root, along with some matching rules for the resource files it contains.

Since the resources list is an instance of `java.util.List`, and Maven mojos must be able to execute in a JDK 1.4 environment that doesn't support Java generics, mojos must be smart enough to cast list elements as `org.apache.maven.model.Resource` instances.

Since mojos can add new resources to the build programmatically, capturing the list of resources used to produce a project artifact can yield information that is vital for debugging purposes. For instance, if an activated profile introduces a mojo that generates some sort of supplemental framework descriptor, it can mean the difference between an artifact that can be deployed into a server environment and an artifact that cannot. Therefore, it is important that the `buildinfo` file capture the resource root directories used in the build for future reference. It's a simple task to add this capability, and can be accomplished through the following code snippet:

```
if ( resources != null && !resources.isEmpty() )
{
    for ( Iterator it = resources.iterator(); it.hasNext(); )
    {
        Resource resource = (Resource) it.next();
        String resourceRoot = resource.getDirectory();

        buildInfo.addResourceRoot( makeRelative( resourceRoot ) );
    }
}
```

As with the prior source-root example, you'll notice the `makeRelative()` method. This method converts the absolute path of the resource directory into a relative path, by trimming the `${basedir}` prefix. All POM paths injected into mojos are converted to their absolute form first, to avoid any ambiguity. It's necessary to revert resource directories to relative locations for the purposes of the `buildinfo` plugin, since the `${basedir}` path won't have meaning outside the context of the local file system.

Adding this code snippet to the extract mojo in the `maven-buildinfo-plugin` will result in a `resourceRoots` section being added to the `buildinfo` file. That section should appear as follows:

```
<resourceRoots>
  <resourceRoot>src/main/resources</resourceRoot>
  <resourceRoot>target/generated-resources/xdoclet</resourceRoot>
</resourceRoots>
```

Once more, it's worthwhile to discuss the proper place for this type of activity within the build life cycle. Since all project resources are collected and copied to the project output directory in the process-resources phase, any mojo seeking to catalog the resources used in the build should execute at least as late as the process-resources phase. This ensures that any resource modifications introduced by mojos in the build process have been completed. Like the vast majority of activities, which may be executed during the build process, collecting the list of project resources has an appropriate place in the life cycle.

Note on testing source-roots and resources

All of the examples in this advanced development discussion have focused on the handling of source code and resources, which must be processed and included in the final project artifact. It's important to note however, that for every activity examined that relates to source-root directories or resource definitions, a corresponding activity can be written to work with their test-time counterparts.

This chapter does not discuss test-time and compile-time source roots and resources as separate topics; instead, due to the similarities, the key differences are summarized in the table below. The concepts are the same; only the parameter expressions and method names are different.

Table 5-2: Key differences between compile-time and test-time mojo activities

Activity	Change This	To This
Add testing source root	`project.addCompileSourceRoot()`	`project.addTestSourceRoot()`
Get testing source roots	`${project.compileSourceRoots}`	`${project.testSourceRoots}`
Add testing resource	`helper.addResource()`	`helper.addTestResource()`
Get testing resources	`${project.resources}`	`${project.testResources}`

5.5.3. Attaching Artifacts for Installation and Deployment

Occasionally, mojos produce new artifacts that should be distributed alongside the main project artifact in the Maven repository system. These artifacts are typically a derivative action or side effect of the main build process. Maven treats these *derivative* artifacts as attachments to the main project artifact, in that they are never distributed without the project artifact being distributed. Classic examples of attached artifacts are source archives, javadoc bundles, and even the `buildinfo` file produced in the examples throughout this chapter.

Once an artifact attachment is deposited in the Maven repository, it can be referenced like any other artifact. Usually, an artifact attachment will have a `classifier`, like `sources` or `javadoc`, which sets it apart from the main project artifact in the repository. Therefore, this classifier must also be specified when declaring the dependency for such an artifact, by using the `classifier` element for that dependency section within the POM.

When a mojo, or set of mojos, produces a derivative artifact, an extra piece of code must be executed in order to attach that artifact to the project artifact. Doing so guarantees that attachment will be distributed when the install or deploy phases are run. This extra step, which is still missing from the `maven-buildinfo-plugin` example, can provide valuable information to the development team, since it provides information about how each snapshot of the project came into existence.

While an e-mail describing the build environment is transient, and only serves to describe the latest build, the distribution of the `buildinfo` file via Maven's repository will provide a more permanent record of the build for each snapshot in the repository, for historical reference.

Including an artifact attachment involves adding two parameters and one line of code to your mojo. First, you'll need a parameter that references the current project instance as follows:

```
/**
 * Project instance, to which we want to add an attached artifact.
 * @parameter default-value="${project}"
 * @required
 * @readonly
 */
private MavenProject project;
```

The MavenProject instance is the object with which your plugin will register the attachment with for use in later phases of the lifecycle. For convenience you should also inject the following reference to `MavenProjectHelper`, which will make the process of attaching the `buildinfo` artifact a little easier:

```
/**
 * This helper class makes adding an artifact attachment simpler.
 * @component
 */
private MavenProjectHelper helper;
```

> See Section 5.5.2 for a discussion about `MavenProjectHelper` and component requirements.

Once you include these two fields in the extract mojo within the maven-buildinfo-plugin, the process of attaching the generated buildinfo file to the main project artifact can be accomplished by adding the following code snippet:

```
helper.attachArtifact( project, "xml",
                               "buildinfo", outputFile );
```

From the prior examples, the meaning and requirement of project and outputFile references should be clear. However, there are also two somewhat cryptic string values being passed in: "xml" and "buildinfo". These values represent the artifact extension and classifier, respectively.

By specifying an extension of "xml", you're telling Maven that the file in the repository should be named using a.xml extension. By specifying the "buildinfo" classifier, you're telling Maven that this artifact should be distinguished from other project artifacts by using this value in the classifier element of the dependency declaration. It identifies the file as being produced by the the maven-buildinfo-plugin, as opposed to another plugin in the build process which might produce another XML file with different meaning. This serves to attach meaning beyond simply saying, "This is an XML file".

Now that you've added code to distribute the buildinfo file, you can test it by re-building the plugin, then running Maven to the install life-cycle phase on our test project. If you build the Guinea Pig project using this modified version of the maven-buildinfo-plugin, you should see the buildinfo file appear in the local repository alongside the project jar, as follows:

```
> mvn install
> cd C:\Documents and Settings\jdcasey\.m2\repository
> cd org\codehaus\guineapig\guinea-pig-core\1.0-SNAPSHOT
> dir

guinea-pig-core-1.0-SNAPSHOT-buildinfo.xml
guinea-pig-core-1.0-SNAPSHOT.jar
guinea-pig-core-1.0-SNAPSHOT.pom
```

Now, the maven-buildinfo-plugin is ready for action. It can extract relevant details from a running build and generate a buildinfo file based on these details. From there, it can attach the buildinfo file to the main project artifact so that it's distributed whenever Maven installs or deploys the project.

Finally, when the project is deployed, the maven-buildinfo-plugin can also generate an e-mail that contains the buildinfo file contents, and route that message to other development team members on the project development mailing list.

5.6. Summary

In its unadorned state, Maven represents an implementation of the 80/20 rule. Using the default life-cycle mapping, Maven can build a basic project with little or no modification – thus covering the 80% case. However, in certain circumstances, a project requires special tasks in order to build successfully. Whether they be code-generation, reporting, or verification steps, Maven can integrate these custom tasks into the build process through its extensible plugin framework. Since the build process for a project is defined by the plugins – or more accurately, the mojos – that are bound to the build life cycle, there is a standardized way to inject new behavior into the build by binding new mojos at different life-cycle phases.

In this chapter, you've learned that it's a relatively simple to create a mojo, which can extract relevant parts of the build state in order to perform a custom build-process task – even to the point of altering the set of source-code directories used to build the project. Working with project dependencies and resources is equally as simple. Finally, you've also learned how a plugin generated file can be distributed alongside the project artifact in Maven's repository system, enabling you to attach custom artifacts for installation or deployment.

Many plugins already exist for Maven use, only a tiny fraction of which are a part of the default life-cycle mapping. If your project requires special handling, chances are good that you can find a plugin to address this need at the Apache Maven project, the Codehaus Mojo project, or the project web site of the tools with which your project's build must to integrate. If not, developing a custom Maven plugin is an easy next step.

Mojo development can be as simple or as complex (to the point of embedding nested Maven processes within the build) as you need it to be. Using the plugin mechanisms described in this chapter, you can integrate almost any tool into the build process.

However, remember that whatever problem your custom-developed plugin solves, it's unlikely to be a requirement unique to your project. So, if you have the means, please consider contributing back to the Maven community by providing access to your new plugin. It is in great part due to the re-usable nature of its plugins that Maven can offer such a powerful build platform.