Pro Apache Ant

Matthew Moodie

Apress®

Pro Apache Ant

Copyright © 2006 by Matthew Moodie

ISBN (pbk): 1-59059-559-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Carsten Ziegeler
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert
Proofreader: Kim Burton
Indexer: Carol Burbo
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 5

Building a Project

The first four chapters in this book dealt with setting up and installing Ant, as well as the basic building blocks of a project's build file. Now it's time to work with an example project to demonstrate some of the major Ant tasks. The example application will also serve as a template for organizing other projects. This is of course only one way of doing it. As long as your projects are organized sensibly, you can carry out the same project build steps.

Many project teams split their projects into pieces that logically belong together. For example, an application may have a GUI as well as a web interface. In this case, the project team would place the core functionality that deals with the database into one section, and they would place GUI code and web interface code in two other, separate sections. This allows the separation of functionality and effort. In other words, everyone knows exactly where the boundaries are in the code and where the boundaries are in responsibility.

Ant is particularly useful in this regard because, as I've said before, it is designed to model the project structure. You can easily separate project sections in an Ant build file. This kind of organization makes your projects easier to manage, and you'll find that you can also conceptualize them better.

This chapter will deal with the initial stages of the project where you take the raw building blocks of an application and turn them into a packaged application for distribution or immediate use. You'll see how a project is organized along the lines of functionality, how Java code is compiled, and how other files are added to a distributable package.

Introducing the Example Application

The example application is a database-backed application that users can access with a command-line Java client or a JSP/servlet web application. This will allow you to work with the core database-access code and other common functionality while using two separate front-end interfaces. You could quite easily implement a GUI for this application as well. The application also includes documentation that you have to package with the appropriate distribution.

The application is simple, but it uses a wide range of features so you can get used to adding many kinds of components to a project. For example, it uses a stand-alone Java class with a `main()` method as the command-line client, JSP tag files, servlets, plain HTML, Java property files, and third-party open-source software. The main instructive point is the separation of functionality.

Introducing the Shared Code

The command-line client and the web interface share a few classes. One shared class contains the database-access code that connects to the database and pulls data from it. This class then passes the data to whichever client class instantiated it. By doing this, you centralize any SQL statements and database connection code so that all releases of the application behave in the same way. Figure 5-1 shows this simple abstraction.

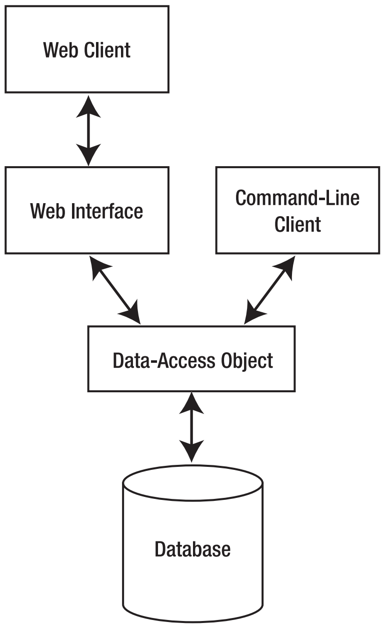

Figure 5-1. *Both interfaces use a common data-access object.*

You can see how the build process for both sides of the project will need to include the database layer shown here. In fact, Figure 5-1 describes the build process extremely well. The command-line client depends on the data-access code, so the target that builds the command-line client should depend on the target that builds the data-access object. The same goes for the web interface.

The two strands of the application also share a class that holds search choice constants. This means each client can offer the search options to users in an application-specific way, while using a common nomenclature underneath. For example, should a user want to order the results alphabetically, they would pass a command-line option to the command-line client, but would select a link or a drop-down box in the web interface. Once the application has divined which option the user has chosen, it sends the choice to the data-access layer, which also has access to the common choices. In other words, they all speak the same language. Figure 5-2 shows this new set of relationships.

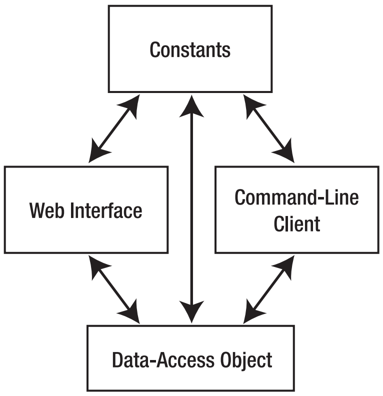

Figure 5-2. *Each component uses a set of constants to abstract the search choice.*

The final part of the shared code is a Java properties file that contains database connection information. The application uses JDBC to connect to the database, so providing the database driver and URL with a properties file is easy.

To start organizing the code, you need a `src` directory to store all your application's code. The first division you are going to use is, as you've just seen, shared code. Therefore, the `shared` child directory will contain the code that is common to all the incarnations of the application.

```
src/
    shared/
           conf/
                database.properties
           java/
                org/
                    mwrm/
                         shared Java classes
```

Introducing the Third-Party Libraries

The application uses third-party libraries from the Jakarta Project and MySQL. You can deal with third-party libraries in two ways: the first is to download a stable build manually and standardize the version across all those involved in a project. The second way of dealing with third-party libraries is to download the latest source files and compile them so that you have the latest, most up-to-date version of the software. This is an optional step, and you can easily factor it into the build process, as shown in Figure 5-3. I'll come back to this in the "Adding Third-Party Libraries to the Build" section.

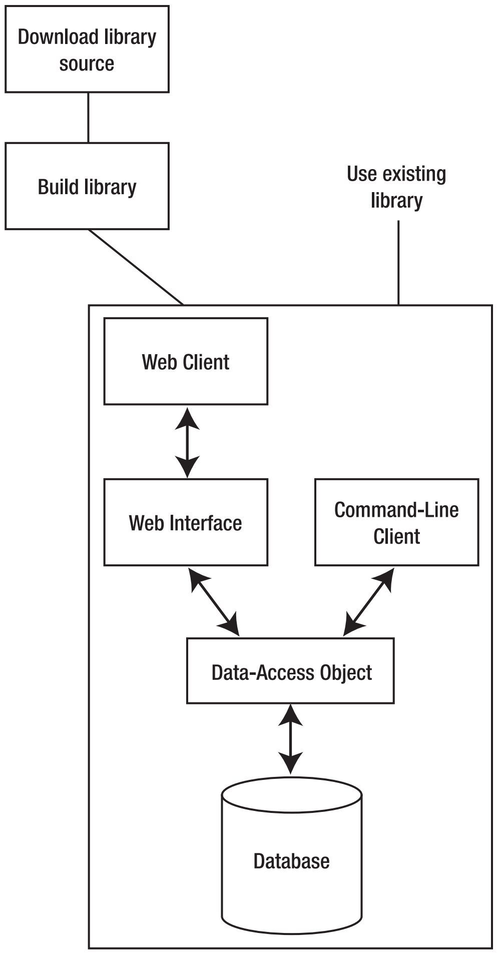

Figure 5-3. *Adding third-party libraries to a build*

Introducing the Stand-Alone Application

The stand-alone application is a command-line client that uses the data-access abstraction layer to connect to the database so that it can obtain data to display to the console. It takes a number of command-line options and displays the results according to the user's choice. It prints usage information if the user supplies invalid options.

To separate it from the shared code, place it in the following directory structure:

```
src/
    stand-alone/
                java/
                     org/
                         mwrm/
                              stand-alone client
```

Introducing the Web Application

The web application is the most complicated of the three divisions. It uses plain HTML, JSP pages, servlets, and tags to provide a rich web interface to the database. Each of these components will be in separate locations, and you'll bring them together when you build the web application.

Simple HTML, JSP pages, and tags guide the user through the application, though a servlet carries out the work of processing the data from the database and providing it for the JSP pages. Essentially, it performs similar work to the stand-alone client, meaning it takes the choice made by the user and obtains the relevant data from the database. Instead of displaying the data, however, the servlet places it in the session so that the rest of the web application has access to it.

Here's the structure of the web application project:

```
src/
    web/
        conf/
             web.xml
        images/
        java/
             org/
                 mwrm/
                      servlet classes
        pages/
              HTML pages
              JSP pages
        tags/
             tag files
```

Introducing the Final Directory Structure

Now that you've separated the three sections of code, the final directory structure looks as follows:

```
src/
    shared/
           conf/
                database.properties
           java/
                org/
                    mwrm/
                         shared Java classes
    stand-alone/
                java/
                     org/
                         mwrm/
                              stand-alone client
```

```
web/
    conf/
         web.xml
    images/
    java/
         org/
             mwrm/
                  servlet classes
    pages/
          HTML pages
          JSP pages
    tags/
         tag files
```

In addition to these directories, you'll need other directories at the same level as `src`. These will help you organize the project when you run the build.

- `build`: This is a scratch directory where you will assemble all the code before running the final packaging steps.
- `dist`: When you have built and packaged the application, you will place it in here prior to distribution or deployment.
- `lib`: You will place third-party libraries in this folder so you can include them in a distribution.

Compiling Java Applications with Ant

Ant provides you with the `<javac>` task so that you can compile Java source code into Java classes. However, before you look into that, you should be aware of a number of preliminary considerations, the first of which is setting up your working environment. This involves setting global properties in a properties file and creating scratch directories so that you have space to work without potentially disrupting the source code.

Setting Up a Working Environment

To begin, you need to create a `build.properties` file in the root directory of the build (that is, the one that contains the `src` directory). You will import the properties from here into the build to ease the maintenance burden of the project. You need to set the names of your top-level directories. Every path in the build file will build on these directories and are in turn properties. By making every path in the build file a property, you centralize the most important parts of the build into one location where you can look after them. If one section of the path changes, then you will have to change only one property in the properties file.

To compile the web portion of the application, you'll need to use the Servlet API and so should include a reference to it in the properties file. This JAR file is included with Tomcat, but if you are using another servlet container, you will have to change this value or download it from `www.ibiblio.org/maven/servletapi/jars/`. I'll cover downloading this as part of the build later in the "Compiling the Source" section.

Listing 5-1 shows what you have so far. Note how each subdirectory is built from another property and how the name of the JAR and WAR files is set here too.

Listing 5-1. *The Names of the Directories in* `build.properties`

```
# The source directory that contains the code
src=src

# Subdirectory properties
src.shared.root=${src}/shared
src.shared.java=${src.shared.root}/java
src.shared.docs=${src.shared.root}/docs
src.shared.conf=${src.shared.root}/conf

src.stand-alone.root=${src}/stand-alone
src.stand-alone.java=${src.stand-alone.root}/java
src.stand-alone.docs=${src.stand-alone.root}/docs

src.web.root=${src}/web
src.web.java=${src.web.root}/java
src.web.docs=${src.web.root}/docs
src.web.pages=${src.web.root}/pages
src.web.tags=${src.web.root}/tags
src.web.conf=${src.web.root}/conf

# The scratch directory
build=build

build.stand-alone.root=${build}/stand-alone

build.web.root=${build}/web
build.web.web-inf=${build.web.root}/WEB-INF
build.web.classes=${build.web.web-inf}/classes
build.web.tags=${build.web.web-inf}/tags
build.web.lib=${build.web.web-inf}/lib

# The final destination of our project files
dist=dist

# The location of third-party JAR files
lib=lib

# This name will be appended to the JAR and WAR files
appName=antBook
appName.jar=${dist}/${appName}.jar
appName.war=${dist}/${appName}.war
```

```
# The Tomcat home directory
catalina.home=C:\\jakarta-tomcat-5.5.9
servlet24.jar=${catalina.home}/common/lib/servlet-api.jar
# Use the following line if using Ant to download the JAR
#servlet24.jar=${lib}/servlet-api.jar
```

Now that you have the properties ready, you can include them in your project's `build.xml` file, as shown in Listing 5-2.

Listing 5-2. *Including the Properties in `build.xml`*

```
<?xml version="1.0"?>

<project name="Example Application Build" default="build-both" basedir=".">

  <property file="build.properties"/>
  </project>
```

This should be familiar to you from Chapter 3. Now it's time to actually use the properties.

Creating Directories

The `src` directory is the only top-level directory you can assume exists, because you want to leave open the option of downloading fresh JARs for the `lib` directory. The others are necessarily absent from the first build, so the upshot is you want to create `build`, `dist`, and `lib`.

Ant's directory-creation task is called `<mkdir>`; Table 5-1 lists its single attribute.

Table 5-1. *The `<mkdir>` Task's Attribute*

Attribute	Description
`dir`	The name of the directory to create, which is either relative to the base directory of this build or an absolute path. This attribute is required.

Listing 5-3 shows the `dir` target that will create the necessary directories.

Listing 5-3. *The `<mkdir>` Task Creates the Directory Structure*

```
<!-- Create the working directories -->
<target name="dir" description="Create the working directories">
  <echo message="Creating the working directories"/>
  <mkdir dir="${build.stand-alone.root}"/>
  <mkdir dir="${build.web.classes}"/>
  <mkdir dir="${dist}"/>
  <mkdir dir="${lib}"/>
</target>
```

The first `<mkdir>` task creates the `build` directory as well as the `stand-alone` directory it contains. The recursive nature of the `<mkdir>` task saves you a lot of effort and allows you to create large directory structures in only a few steps. The second `<mkdir>` task shows this in action as well.

Compiling the Source

Now that you have the directory structure in place, it's time to build the application. Ant's Java compilation task is `<javac>`, and it's as flexible as the `javac` command is at the command line. You'll keep the stand-alone application in its own target because it allows you to compile just the stand-alone application, should you want. This follows the project structure shown in Figures 5-1 and 5-2 and means you can include it in more than one project build path. For example, you may sometimes want to build it for testing, but not for distribution.

The `<javac>` task has the attributes shown in Table 5-2 (as well as the attributes of a file set, as detailed in Table 4-6 in Chapter 4), and many of them will be familiar from the command line. You can specify most of the attributes that take paths as nested elements as well.

Table 5-2. *The `<mkdir>` Task's Attribute*

Attribute	Description
`bootclasspath`	The boot classpath to use for this compilation. The default is the system boot classpath.
`bootclasspathref`	A reference ID to a path that you have defined elsewhere in the build file.
`classpath`	The classpath for this compilation. As with all Java applications, the classpath is an important issue to understand. More details follow this table. The default is the system classpath plus Ant's classpath (`ANT_HOME/lib` and the `-lib` command-line option).
`classpathref`	A reference ID to a path that you have defined elsewhere in the build file.
`compiler`	The compiler implementation to use. If you do not set this attribute, Ant will use the value of the `build.compiler` property, if set.
`debug`	Sets the `javac -g` flag and can be used in combination with `debuglevel`. This attribute makes sense only if your compiler supports debugging. The default is `false`, which sends `-g:none` to the compiler.
`debuglevel`	Sets the keywords to be appended to the `-g` command, if it is sent. If `debug` is set to `false`, this attribute is ignored. You specify either `none` or a comma-separated list of `lines`, `vars`, and `source`. The default is an empty string (that is, nothing is appended to the `-g` option).
`depend`	Tells the compiler to track dependencies, if it supports this feature. The default is `false`.
`deprecation`	Tells Ant whether to compile with deprecation information. The default is `false`.
`destdir`	The directory where you want Ant to place the compiled class files. The default is the same location as the source files, just as it is for `javac` at the command line.
`encoding`	The encoding of the source files. The default depends on your system.

Table 5-2. *The `<mkdir>` Task's Attribute (Continued)*

Attribute	Description
`excludes`	The excludes list for this compilation, where each entry is separated from the next one with a space or a comma. You may use wildcards. The default is to omit nothing except the default excludes.
`excludesfile`	The name of the file that contains the exclude patterns. The default is not to use a file.
`executable`	The full path to the `javac` executable should you set `fork` to `true`. The default is the executable of the JVM that is running Ant.
`extdirs`	A list of the directories that contain installed extensions. The default is the system extension setting.
`failonerror`	Tells Ant whether to carry on with the build if there was a compilation error. The default is `true`.
`fork`	Tells Ant to compile the classes with an external JDK compiler. The default is `false`.
`includeAntRuntime`	Tells Ant to add its runtime libraries to the classpath. The default is `true`.
`includeJavaRuntime`	Tells Ant to add the Java run-time libraries to the classpath. The default is `false`.
`includes`	The includes list for this compilation, where each entry is separated from the next one with a space or a comma. You may use wildcards. The default is `*.java`.
`includesfile`	The name of the file that contains the include patterns. The default is to not use a file.
`listfiles`	Tells Ant to list the files that it is compiling, rather than just the number of files. The default is `false`.
`memoryInitialSize`	The initial size for the external JVM should you set `fork` to `true`. Ignored if you do not.
`memoryMaximumSize`	The maximum amount of memory to be used by the external JVM should you set `fork` to `true`. Ignored if you do not.
`nowarn`	Tells Ant whether to send the `-nowarn` option to the compiler. The default is `false`.
`optimize`	Tells Ant whether to compile the source with optimization. The default is `false`.
`source`	The value of the `-source` command-line option. The default value depends on your own VM, and some will ignore it. You will know best as to what the value of this attribute should be.
`sourcepath`	The source path for this compilation. The default is the value of `srcdir` or any nested `<src>` elements.
`sourcepathref`	A reference ID to a path that you have defined elsewhere in the build file.
`srcdir`	The base directory of the Java source code. All paths used in the `<javac>` task are relative to this directory. This attribute is required.

Table 5-2. *The <mkdir> Task's Attribute (Continued)*

Attribute	Description
tempdir	The location of a temporary directory that Ant should use in the build. Ant uses it only if you set fork to true and the length of the command-line arguments exceeds 4 kilobytes.
target	The version of Java for which these files will be compiled. The default value depends on your own VM. You will know best as to what the value of this attribute should be. If you are using JVM 1.4 or greater, you should note that your classes won't work in a 1.1 JVM.
verbose	Tells the compiler to be more verbose. The default is false.

Whether you set these attributes depends on your project and how you would have compiled it at the command line. If you have certain compiler and JVM version concerns for your project and want to know more, the best place to look is the <javac> task's documentation.

If you are using JDK 1.3 or greater on Windows, unforked compilation will lock files in the classpath, so you can't move or delete them later in the build. If this is part of your build plan, set fork to true.

The <javac> task can also have the following child elements, some of which can replace their corresponding attributes:

<bootclasspath>
<classpath>
<exclude>
<extdirs>
<include>
<patternset>
<sourcepath>
<src>

<bootclasspath>, <classpath>, <extdirs>, <patternset>, <sourcepath>, and <src> are pathlike structures (see Chapter 4 for details on this type of structure) and can take references to other paths defined elsewhere in the build file.

■ **Note** The <javac> task uses the system classpath, Ant's own classpath (the contents of ANT_HOME/lib and the values supplied with the -lib command-line option), and any custom classpath you supply as an argument or nested element.

This is a good point to add the master build classpath that contains all the JAR files for compiling and running the application. Listing 5-4 shows the <path> element that sets the master classpath. You'll see the jsp20.jar property when you compile the JSTL source code.

Listing 5-4. *Building the Master Classpath with a <path> Element*

```
<!-- ################################### -->
<!-- The master build classpath          -->
<!-- ################################### -->

<path id="build.classpath">
  <pathelement location="${servlet24.jar}"/>
  <pathelement location="${jsp20.jar}"/>
  <pathelement location="${mysql.jar}"/>
  <pathelement path="${appName.jar}"/>
</path>
```

Listing 5-5 shows the `<javac>` task that you will use to compile the stand-alone application. Note how the shared code is compiled first.

Listing 5-5. *The <javac> Task Compiles the Stand-Alone Application*

```
<!-- ########################### -->
<!-- The stand-alone application -->
<!-- ########################### -->

<!-- Compile the stand-alone application -->
<target name="compile-stand-alone" depends="dir"
        description="Compile stand-alone application">
  <echo message="Compiling the stand-alone application"/>
  <javac srcdir="${src.shared.java}" destdir="${build.stand-alone.root}"/>
  <javac srcdir="${src.stand-alone.java}"
           destdir="${build.stand-alone.root}"/>
  </target>
```

The sample web application has a similar compilation, though it requires the servlet classes to be in the classpath. As such, Listing 5-6 shows the variants of `<javac>` that you can use to compile the web application.

Listing 5-6. *The <javac> Task Compiles the Web Application*

```
<!-- ########################### -->
<!-- The web application         -->
<!-- ########################### -->

<!-- Compile the web application -->
<target name="compile-web" depends="dir" description="Compile web application">
  <echo message="Compiling the web application"/>
  <javac destdir="${build.web.classes}">
    <src path="${src.shared.java}"/>
  </javac>
  <javac srcdir="${src.web.java}" destdir="${build.web.classes}">
    <classpath refid="build.classpath"/>
  </javac>
</target>
```

Recall that you built the `servlet24.jar` property from the `catalina.home` property in the property file because I assumed you have access to Tomcat. If you wanted to download the JAR file in the build, you could use the target in Listing 5-7 and add it to the `depends` attribute of the `compile-web` target. Remember to change the `servlet24.jar` property as well.

Listing 5-7. *Downloading the Servlet JAR File with the <get> Task*

```
<!-- ######################## -->
<!-- Download the servlet JAR -->
<!-- ######################## -->

<!-- Download the servlet JAR -->
<target name="download-servlet-jar" depends="dir"
        description="Download the servlet JAR">
  <echo message="Downloading the servlet JAR"/>

  <get src="http://www.ibiblio.org/maven/servletapi/jars/servletapi-2.4.jar"
       dest="${servlet24.jar}"
       verbose="true"/>
</target>
```

The `<get>` task is straightforward. The `src` attribute is the file you want to download, and the `dest` attribute is its name in your file system. These are the only two required attributes. The `verbose` attribute is set to `false` by default, though here you should see the details of the download for the sake of instruction. You can also take advantage of HTTP BASIC authentication with the `username` and `password` attributes, though you should set these only at the command line and not as properties or as hard-coded values in the file.

Now that you have compiled the code and placed the class files in the scratch directory, it's time to assemble the other parts of the project before you package them for distribution.

Adding Third-Party Libraries to the Build

If you are using third-party libraries in a build, you may want to build them at the same time as the main project, assuming the source code is available. However, building third-party libraries is not an important step when you are using a set, stable version of a third-party library to ensure standard behavior across a project team. You don't need to build the libraries from source during every run of the build, because you can set up a build path to do this, as the case may be.

Figure 5-4 shows a simple build path that allows you to choose between using the existing library in your base directory's `lib` directory and using a freshly downloaded source bundle.

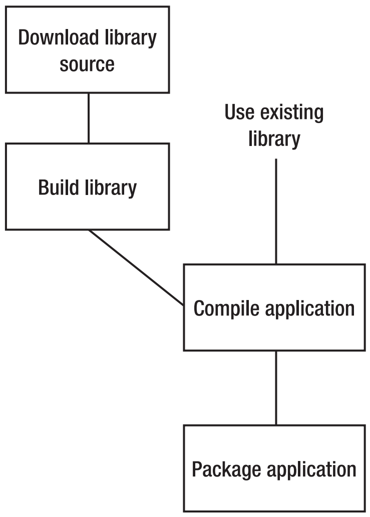

Figure 5-4. *Choosing between downloading source code and using preexisting binaries*

You can make this kind of choice in three ways: using properties, using the `<antcall>` task in a target, or using the `depends` attribute of `<target>`. This kind of choice is not unique to downloading third-party source code; you can apply it to many other situations. For example, you may want to exclude the documentation from some project builds, but not others.

The example project uses two third-party libraries: the MySQL JDBC connector and the JSTL tag library. You can obtain these easily as binary JAR files, but you can also get the latest CVS snapshot. The `<cvs>` task checks out source from a CVS repository and places it in your file system. Table 5-3 shows its attributes.

Table 5-3. *The `<cvs>` Task's Attributes*

Attribute	Description
`append`	Tells Ant whether it should append output messages to the file specified in `output` or `error`. The default is `false`.
`command`	The CVS command to execute, though you can add to this with, for example, the `package` attribute, as in Listing 5-9. The default is `checkout`.
`compression`	If you set this to `true`, it is equivalent to setting the `compressionlevel` to `3`. The default is `false`.
`compressionlevel`	Valid values are `1`–`9`. If you set it to anything else, it's equivalent to setting `compression` to `false`. Ignored if `compression` is set to `false`, and its default is `3` if `compression` is set to `true`.
`cvsRoot`	The root of the CVS repository you are querying. The default is `null`.
`cvsRsh`	The remote shell to use. The default is `null`.

Table 5-3. *The <cvs> Task's Attributes*

Attribute	Description
date	Tells Ant to check out the most recent files, as long as their modification times are not later than this date.
dest	The directory where you want to place the source code that is checked out of the repository. The default is your project's base directory.
error	The file where you want to direct error messages. The default is the Ant log (set at MSG_WARN).
failonerror	Tells Ant whether to carry on with the build if there was a CVS error. The default is false.
noexec	Tells Ant to make a report and not to change any files. The default is false.
output	The file where you want to direct output. The default is the Ant log (set at MSG_INFO).
package	The name of the package you want to check out of the CVS repository. The default is null.
passfile	The file where you have stored the CVS passwords, if any. The default is ~/.cvspass.
port	The port of the CVS server. The default is 2401.
quiet	Tells Ant to suppress information messages during the CVS process. The default is false.
reallyquiet	Tells Ant to not print any messages at all during the CVS process. The default is false.
tag	The tag of the package you want to check out of the CVS repository. The default is null.

All the following sections will require the property definitions in Listing 5-8.

Listing 5-8. *The CVS Homes of the Third-Party Libraries*

```
<!-- CVSROOT for the JSTL -->
<property name="cvsroot"
          value=":pserver:anoncvs@cvs.apache.org:/home/cvspublic" />

<!-- CVSROOT for the MySQL connector -->
<property name="mysql.cvsroot"
          value=":pserver:anonymous@cvs.sourceforge.net:/cvsroot/mmmysql" />
```

These are the login details for the CVS repositories of the third-party libraries. The actual targets for obtaining the source and compiling it won't change. Listing 5-9 shows the targets for obtaining the source code.

Listing 5-9. *The Targets for Obtaining the JSTL and the MySQL Connector Source*

```
<!-- Update or check out required sources from CVS for the JSTL -->
<target name="checkout-jstl" depends="dir"
        description="Update or check out required sources
                     from CVS for the JSTL">

  <echo message="Checking out the required JSTL sources from CVS"/>

  <cvs cvsroot="${cvsroot}" quiet="true"
       command="checkout -P ${jstl.build}"
       dest="${build}" compression="true" />

</target>
<!-- Update or check out required sources from CVS for the MySQL connector -->
<target name="checkout-mysql-connector" depends="dir"
        description="Update or check out required sources
        from CVS for the MySQL connector">

  <echo message="Checking out the required sources from CVS
               for the MySQL connector" />

  <cvs cvsroot="${mysql.cvsroot}" quiet="true"
       command="checkout" package="${mysql.build}"
       dest="${build}" compression="true" />

</target>
```

The task for the JSTL CVS contains the command `checkout -P ${jstl.build}`, and the MySQL CVS task builds the command using the `package` attribute.

You need to define some more properties in `build.properties`. The MySQL connector source comes with its own build file and uses a number of properties of its own. However, to finish the job you need three more, as shown in Listing 5-10. (You'll use `mysql.name` in the compilation process.) The JSTL build requires a few more properties, also shown in Listing 5-10. The JSP classes are taken from Tomcat, but if you have not installed Tomcat, you can get the JAR file from `www.ibiblio.org/maven/jspapi/jars/`.

Listing 5-10. *Properties for Obtaining and Building the Third-Party Libraries*

```
# Required for the JSTL build
jsp20.jar=${catalina.home}/common/lib/jsp-api.jar
# Use the following line if using Ant to download the JAR
#jsp20.jar=${lib}/jsp-api.jar
jstl.build=jakarta-taglibs/standard
library.src=src
examples.src=examples
doc.src=doc
build.library=${build}
```

```
# Required for the MySQL connector build
mysql.build=mm.mysql-2
mysql.name=mysql-connector
mysql.jar=${lib}/${mysql.name}-bin.jar
```

You can modify the `download-servlet-jar` target from Listing 5-7 to download the JSP JAR file if you want, as shown in Listing 5-11.

Listing 5-11. *Downloading the JSP JAR File with the <get> Task*

```
<!-- ######################## -->
<!-- Download the JSP JAR     -->
<!-- ######################## -->

<!-- Download the JSP JAR -->
<target name="download-jsp-jar" depends="dir"
        description="Download the JSP JAR">
  <echo message="Downloading the JSP JAR"/>

  <get src="http://www.ibiblio.org/maven/jspapi/jars/jsp-api-2.0.jar"
       dest="${jsp20.jar}"
       verbose="true"/>
</target>
```

You can now run these two targets and obtain the source code of the JSTL and the MySQL JDBC connector. Once you have the source, you can build them. Luckily, both are based on Java and come with their own Ant build files for seamless integration into your project.

To use another project's build file in your own project, you use the `<ant>` task. This task will run the default target of the target project, but it can also run a specific target if you want. In the case of this sample project, you want to pass all the properties to the new builds because you want to customize them to your own requirements. (The JSTL build also requires certain properties before it will build successfully.) You can disable property sharing like this if you are worried about naming clashes (by setting the `inheritAll` attribute to `false`). Naming clashes will occur because the properties from the calling project override those in the called project.

The `<ant>` task can have nested `<property>` elements, which Ant always passes to the called project, no matter what settings you have. These will override any properties in the called file just as if they were buildwide properties.

■ **Note** Properties passed to Ant at the command line are always passed to the called project. They will even overwrite the nested `<property>` elements.

Table 5-4 shows the attributes of the `<ant>` task.

Table 5-4. *The <ant> Task's Attributes*

Attribute	Description
`antfile`	The name of the build file to use, which is relative to the directory specified in the `dir` attribute. The default is `build.xml`.
`dir`	The base directory for the project you are calling. It should contain the file you specify in the `antfile` attribute. The default is the calling project's base directory.
`inheritAll`	Tells Ant whether to pass properties to the called project. The default is `true`.
`inheritRefs`	Tells Ant whether to pass references to the called project. The default is `false`.
`output`	The file where you want to direct output (set at `MSG_INFO`). The default is `null`.
`target`	The name of the target you want to call in the called project. The default is the called project's default target.

So, calling another project's build file is extremely easy. The hardest part of it is working out which properties you need to customize or supply for the called project to build. If you are calling one of your own projects, that shouldn't be a problem because they will mostly be in place to start. Third-party libraries require a bit more investigation (as shown by the properties the JSTL build requires, as listed in Table 5-4).

Listing 5-12 shows the two `<ant>` tasks that build the third-party libraries and the tasks that copy them to your `lib` directory. (I'll discuss the `<copy>` task in the "Assembling the Project" section.)

Listing 5-12. *The Targets for Building the JSTL and the MySQL Connector*

```
<!-- Build the JSTL from source -->
<target name="build-jstl" depends="checkout-jstl"
        description="Build the JSTL from source">
  <echo message="Building the JSTL from source"/>

  <ant antfile="build.xml" dir="${build}/${jstl.build}"/>

  <copy todir="${lib}">
    <fileset dir="${build}/${jstl.build}/${build}/lib">
      <include name="*.jar"/>
    </fileset>
  </copy>
</target>

<!-- Build the MySQL connector from source -->
<target name="build-mysql-connector" depends="checkout-mysql-connector"
        description="Build the MySQL connector from source">
  <echo message="Building the MySQL connector from source"/>
  <!-- The MySQL connector file needs this directory to exist -->
  <!-- Therefore we need to create it -->
  <mkdir dir="${build}/dist-mysql-jdbc"/>

  <ant antfile="build.xml" dir="${build}/${mysql.build}"/>

  <copy tofile="${mysql.jar}">
    <fileset dir="${build}/build-mysql-jdbc">
      <include name="mysql-connector*/*.jar"/>
    </fileset>
  </copy>
</target>
```

Both `<ant>` tasks call the appropriate `build.xml` file located in the directory you downloaded, as defined by the `jstl.build` and `mysql.build` properties. As noted, some research into the properties of the third-party libraries was required before you could run the build. Similar research was required before you could use the `<copy>` task to move the JAR files into the `lib` directory. As you can see, the location of the JARs is not common to both libraries, so you had to use different patterns to locate them.

The MySQL connector uses a version number in the names of its directories and JAR files, so you have to remove any dependencies on this naming convention. The wildcard characters are perfect for this. The JSTL isn't so complicated but has two binary JARs, both of which you must copy.

Using Properties to Decide

The `if` and `unless` attributes of `<target>` allow you to control whether a target will execute, depending on the presence or absence of a named property (see Chapter 3). Therefore, you can force Ant to skip steps in the build process by providing properties at the command line with the `-D` option. So, instead of the forked build path shown in Figure 5-3, you will have a linear build path that ignores some steps, depending on your choice of properties.

The `compile-stand-alone` target is the end of the stand-alone application's linear build path, so it must depend on the `build-mysql-connector` target, which in turn depends on the `checkout-mysql-connector` target. Therefore, you must change the `depends` attribute of `compile-stand-alone` as follows:

```
<target name="compile-stand-alone" depends="build-mysql-connector"
        description="Compile stand-alone application">
```

The same applies to the `compile-web` target, but it also needs the JSTL:

```
<target name="compile-web" depends="build-jstl, build-mysql-connector"
        description="Compile web application">
```

Now, when you run Ant on each of these targets, you will always run the download and build targets as well. To control this, you need to use properties at the command line. Listing 5-13 shows the `if` attributes of the targets that download and build the third-party libraries.

Listing 5-13. *The `if` Attribute Determines Whether a Target Runs*

```
<target name="checkout-jstl" depends="dir" if="jstl"
        description="Update or check out required sources
        from CVS for the JSTL">
</target>

<target name="build-jstl" depends="checkout-jstl" if="jstl"
        description="Build the JSTL from source">
</target>

<target name="checkout-mysql-connector" depends="dir" if="mysql"
        description="Update or check out required sources
        from CVS for the MySQL connector">
</target>

<target name="build-mysql-connector" depends="checkout-mysql-connector"
        if="mysql" description="Build the MySQL connector from source">
</target>
```

Now, if you set the `mysql` property at the command line, the MySQL-specific targets will run. The same goes for the `jstl` property.

```
> ant -Djstl=true -Dmysql=true compile-web
```

If you don't set them, Ant will not run the targets.

Using the <antcall> Task

The `<antcall>` task is similar to the `<ant>` task, except that it calls a target in the current project's build file. This is a useful technique when you have a forked build process. You cannot use `<antcall>` outside a target, though you won't have reason to do so. It has the attributes shown in Table 5-5.

Table 5-5. *The `<antcall>` Task's Attributes*

Attribute	Description
`inheritAll`	Tells Ant whether to pass properties to the called project. The default is `true`.
`inheritRefs`	Tells Ant whether to pass references to the called project. The default is `false`.
`target`	The name of the target you want Ant to run. This attribute is required.

To use `<antcall>` to control the build, place an `<antcall>` task for each target you want to call in a master target, as shown in Listing 5-14.

Listing 5-14. *Using `<antcall>` to Control a Project*

```
<target name="stand-alone-complete"
        description="Compile stand-alone application,
        using CVS version of the MySQL connector">
  <echo message="Compiling stand-alone application,
                 using CVS versions of the MySQL connector"/>
  <antcall target="build-mysql-connector"/>
  <antcall target="package-stand-alone"/>
</target>

<target name="web-complete"
        description="Compile web application,
                     using CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiling web application,
                 using CVS versions of the MySQL connector and the JSTL"/>
  <antcall target="build-mysql-connector"/>
  <antcall target="build-jstl"/>
  <antcall target="package-web"/>
</target>
```

To build the third-party libraries as well as the application, you just need to run the following:

```
> ant stand-alone-complete
> ant web-complete
```

If you want to build just the application, run the following:

```
> ant package-stand-alone
> ant package-web
```

To increase the build functionality, you can add a master target that will run these commands for you if you want to build the stand-alone application at the same time as the web application. Listing 5-15 shows how to do this.

Listing 5-15. *Master Targets for Building Both Applications*

```
<!-- ######################################## -->
<!-- Targets that work with both applications -->
<!-- ######################################## -->

<target name="build-both"
        description="Compile both applications,
                     without CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiling both applications,
                 without CVS versions of the MySQL connector and the JSTL"/>
  <antcall target="package-stand-alone"/>
  <antcall target="package-web"/>
</target>
```

```
<target name="build-all"
        description="Compile both applications,
                     using CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiling both applications,
                 using CVS versions of the MySQL connector and the JSTL"/>
  <antcall target="stand-alone-complete"/>
  <antcall target="web-complete"/>
</target>
```

You can quite easily extend this structure of `<antcall>` tasks to cover as many permutations as you like.

Using Dependencies

The third method for choosing which targets to run is the `depends` attribute of the `<target>` element. The principle behind this technique is similar to that of the `<antcall>` task. However, instead of grouping `<antcall>` elements, you specify target names in a master target's `depends` attribute.

In the example build file, this means you will replace every `<antcall>` with a setting in the target's `depends` attribute. Compare Listing 5-16 with Listings 5-14 and 5-15.

Listing 5-16. *Using the depends Attribute to Control a Build*

```
<target name="stand-alone-complete"
        depends="build-mysql-connector, package-stand-alone"
        description="Compile stand-alone application,
        using CVS version of the MySQL connector">
  <echo message="Compiling stand-alone application,
                 using CVS versions of the MySQL connector"/>
</target>

<target name="web-complete"
        depends="build-mysql-connector, build-jstl, package-web"
        description="Compile web application,
                     using CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiled web application,
                   using CVS versions of the MySQL connector and the JSTL"/>
</target>

<!-- ######################################## -->
<!-- Targets that work with both applications -->
<!-- ######################################## -->
```

```
<target name="build-both"
        depends="package-stand-alone, package-web"
        description="Compile both applications,
                     without CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiled both applications,
                 without CVS versions of the MySQL connector and the JSTL"/>
</target>

<target name="build-all"
        depends="stand-alone-complete, web-complete"
        description="Compile both applications,
                     using CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiled both applications,
                 using CVS versions of the MySQL connector and the JSTL"/>
</target>
```

So, when you run the build-all target, Ant calls stand-alone-complete, which calls build-mysql-connector, which calls checkout-mysql-connector, which calls dir. If the last three targets complete their tasks, stand-alone-complete calls package-stand-alone, which calls compile-stand-alone. The dir target has already completed successfully, so the compile-stand-alone target runs. If it and package-stand-alone complete successfully, half the targets in the build-all target's depends attribute have completed successfully. Ant then calls web-complete, and the process goes much like the one described for stand-alone-complete.

Choosing Which Technique to Use

The depends attribute is the usual method for controlling the flow of a project. Its advantages include performance, with dependencies being up to six times faster than <antcall> tasks. Another advantage of depends attributes is that they group all the dependencies in one place right at the top of a target in a location that is useful to a casual reader. If you use properties, it may not be clear that you are using them to control the whole build unless you document thoroughly. Even then, users may not even read the build file and will try to run the build without setting any properties.

You should use depends attributes as much as possible if your project is confined to a single build file. It is a more maintainable technique because of its centralized nature and unambiguous meaning. If your build is split between files (as described in Chapter 9), then you have no choice but to use <antcall> tasks, though you should still try to minimize them.

Assembling the Project

Many, many ways of assembling a project for distribution exist, and Ant covers each one. The first step is to collect every piece of the project. Once you have done that, you can then choose which method of packaging, if any, you are going to use. For example, you may have source distributions as tarballs or zip files, binary distributions as JAR files, and local test copies as unpackaged directories. Your source distributions may be released daily, while the binary distributions may go out only when there is a major revision, so you want to build this flexibility into your build process.

Manipulating File Location

Ant has the full range of directory- and file-manipulation tasks that you would expect of an operating system, so you can do anything in a build process that you can do at the command line. These tasks take full advantage of Ant's pattern-matching capabilities, as well as heavily used pathlike structures.

You have already seen the `<copy>` task when you used it to move the third-party libraries to your `lib` directory. Table 5-6 shows the attributes of this task. It can also take nested `<fileset>`, `<mapper>`, `<filterset>`, and `<filterchain>` elements.

Table 5-6. *The <copy> Task's Attributes*

Attribute	Description
`enablemultiplemappings`	If you have specified a `<mapper>` nested element, this attribute tells Ant to process all the possible mappings for the source path; otherwise it will process only the first file or directory. The default is `false`.
`encoding`	The encoding of the source files. The default is the JVM's default encoding.
`failonerror`	Tells Ant whether to carry on with the build if there was a copy error. The default is `true`.
`file`	The name of the file to copy. This attribute is required unless you nest `<fileset>` elements.
`filtering`	Tells Ant whether to use the project's global filters. It will always use nested `<filterset>` elements, regardless of its setting. The default is `false`.
`flatten`	Tells Ant whether to copy all the files into the directory specified by `todir`, ignoring the source directory hierarchy. The default is `false`.
`granularity`	The number of milliseconds that Ant should allow either way when it is deciding whether a file is out-of-date. The default is `2000` on DOS-based operating systems. It is `0` on all other systems.
`includeEmptyDirs`	Tells Ant whether to include empty directories in the copy. The default is `true`.
`outputencoding`	The encoding that Ant should use for the copied files. The default is the value of `encoding` if you have set it or the JVM's default if not.
`overwrite`	Tells Ant whether to overwrite existing files at the destination, even if they are newer than the files you are copying. The default is `false`.
`preservelastmodified`	Tells Ant to maintain the last modified time of the files you are copying. The default is `false`.
`todir`	The name of the directory to which you are copying. You must specify one of `todir` and `tofile`. I discuss the rules governing the two after this table.

Table 5-6. *The <copy> Task's Attributes*

Attribute	Description
tofile	The name of the file to which you are copying. You must specify one of todir and tofile. I discuss the rules governing the two after this table.
verbose	Tells Ant to list the files as it copies them. The default is false.

The general rule with the todir and tofile attributes is that if more than one file is to be copied, you must use the todir attribute. For example, more than one file will be copied if you use the file attribute and a <fileset> nested element or if a file set contains more than one file (if the file set contains a single file, you may use tofile). If only one file is to be copied, you can use whichever attribute you want. The tofile attribute will rename the file, while todir won't.

The stand-alone application is almost all in place by this point, but it still requires the Java properties file. Listing 5-17 shows the <copy> task that copies it into the working directory. There's more to this target, but that will wait until the "Creating JAR Files" section.

Listing 5-17. *Copying the Java Properties File Using the <copy> Task*

```
<!-- Package the stand-alone application -->
<target name="package-stand-alone" depends="compile-stand-alone"
        description="Package the stand-alone application">
  ...
  <copy file="${database.properties}" todir="${build.stand-alone.root}"/>
  ...
</target>
```

The web application has more files to work with, so a few more <copy> tasks exist, as shown in Listing 5-18.

Listing 5-18. *Copying the Web Application's Web Pages and Configuration Files*

```
<!-- Copy the web pages and configuration files -->
<target name="copy-web" depends="compile-web" description="Copy the web files">
  <echo message="Copying the web pages and configuration files"/>
  <copy todir="${build.web.root}">
    <fileset dir="${src.web.pages}"/>
  </copy>
  <!-- Copy the tags -->
  <copy todir="${build.web.tags}">
    <fileset dir="${src.web.tags}"/>
  </copy>
  <copy todir="${build.web.web-inf}">
    <fileset dir="${src.web.conf}">
      <include name="*.tld"/>
    </fileset>
  </copy>
  <!-- Copy the JAR files -->
  <copy todir="${build.web.lib}">
    <fileset dir="${lib}"/>
  </copy>
  <!-- Copy the properties file -->
  <copy file="${database.properties}" todir="${build.web.classes}"/>
  <!-- No need to copy web.xml, as the WAR task does this for us -->
</target>
```

These tasks are straightforward, though the final comment is worth discussing. When you create the WAR file of this web application in a moment, you will use the `<war>` task, which will pick up the `web.xml` file and place it in the WAR for you. If you wanted to use the expanded web application, then you would have to remember to copy the `web.xml` file into the expanded directory structure. It is also possible to assemble the entire WAR file in the `<war>` task, though you'll need zip file sets for this. You'll see zip file sets in the next chapter, so I'll defer this version of the `<war>` task until then.

One target that all Ant projects should have is a `clean` target. This will typically remove the working directories and remove any other unnecessary files. To do this, it will use the `<delete>` task, the attributes of which are shown in Table 5-7. (The deprecated attributes are not included, because they are replaced by nested file sets.) You can nest file sets in this task as well, and if you do, empty directories will be ignored by default (see the `includeemptydirs` attribute).

Table 5-7. *The `<delete>` Task's Attributes*

Attribute	Description
`deleteonexit`	Tells Ant to use the `File.deleteOnExit()` method to delete the file when the JVM terminates. The default is `false`.
`dir`	The name of the directory to delete. All its subdirectories are deleted as well. You must specify one of `dir` or `file` or supply a nested file set.
`failonerror`	Tells Ant whether to carry on with the build if there was an error. Is not used when `quiet` is set to `true`. The default is `true`.
`file`	The name of the file to delete. You must specify one of `dir` or `file` or supply a nested file set.
`includeemptydirs`	Tells Ant to delete empty directories if they match the pattern specified in a nested file set. The default is `false`.
`quiet`	This attribute is not quite the same as the `quiet` attribute of other tasks, and if set to `true`, it sets `failonerror` to `false`. If you set this to `true`, Ant does not display any error messages if a file or directory does not exist or can't be deleted, and the task continues processing. However, the `-verbose` and `-debug` command-line options override this attribute. The default is `false`.
`verbose`	Tells Ant to list the files as it deletes them. The default is `false`.

The example project uses a `clean` target, as shown in Listing 5-19.

Listing 5-19. *The clean Target Removes the Working Directories*

```
<target name="clean" description="Clean up the working directories">
  <echo message="Cleaning up"/>
  <delete dir="${build}"/>
</target>
```

Creating the JAR Files

Once you have assembled all the Java files you want to package into a JAR, you are ready to use the `<jar>` task. It has all the functionality of the `jar` command at the command line, so you will probably be familiar with what it does. Table 5-8 shows its attributes. You can use nested `<metainf>`, `<manifest>`, and `<indexjars>` elements.

Table 5-8. *The <jar> Task's Attributes*

Attribute	Description
`basedir`	The directory that will form the root of the resultant JAR file. The default is the base directory of the project.
`compress`	Tells Ant to compress the files as it adds them to the JAR file. If `keepcompression` is set to `true`, this applies to the entire archive, not just to the files you are adding. The default is `true`.
`defaultexcludes`	Tells Ant to use the default excludes (see Chapter 4). The default is `true`.
`destfile`	The name of the JAR file you want to create. This attribute is required.
`duplicate`	Tells Ant what to do if duplicate files are found. You can specify `add`, `preserve`, or `fail`. The default is `add`.
`encoding`	The encoding to use for filenames in the archive. The default is `UTF8`.
`excludes`	The excludes list for this task, where each entry is separated from the next one with a space or a comma. You may use wildcards. The default is to omit nothing except the default excludes.
`excludesfile`	The name of the file that contains the exclude patterns. The default is not to use a file.
`filesetmanifest`	Tells Ant how to react when it encounters a manifest file in a nested file set. `skip` ignores the file, `merge` tells Ant to merge the manifests, and `mergewithoutmain` merges the files without the main sections. The default is `skip`.
`filesonly`	Tells Ant to store only file entries. The default is `false`.
`includes`	The includes list for this task, where each entry is separated from the next one with a space or a comma. You may use wildcards. The default is all files.
`includesfile`	The name of the file that contains the include patterns. The default is not to use a file.
`index`	Tells Ant to create an index list to speed class loading (JDK 1.3 and greater). Only this JAR will be included in the list, unless you add nested `<indexjars>` elements. The default is `false`.

Table 5-8. *The <jar> Task's Attributes (Continued)*

Attribute	Description
keepcompression	Tells Ant to keep the original compression of the files you are adding. The default is false.
manifest	The name of the manifest file to use. It can be a manifest file in the file system or the name of a JAR file that contains the manifest you want to use. This JAR file must be specified in a nested file set and should contain a manifest at META-INF/MANIFEST.MF. The default is null.
manifestencoding	The encoding to use when reading the manifest. The default is the operating system's default.
roundup	Tells Ant to round up file modification times to the next even number of seconds. If you don't do this, the times will be rounded down in the JAR file. This means the JAR file will seem out-of-date when you run the target again. The default is true.
update	Tells Ant to overwrite files in the JAR file. The default is false.
whenempty	Tells Ant what to do if no files match. You can specify fail, create, or skip. The default is skip.

Listing 5-20 shows the full version of the package-stand-alone target.

Listing 5-20. *The package-stand-alone Target Creates the Stand-Alone Application's JAR File*

```
<!-- Package the stand-alone application -->
<target name="package-stand-alone" depends="compile-stand-alone"
        description="Package the stand-alone application">
  <echo message="Creating the stand-alone JAR file"/>
  <copy file="${database.properties}" todir="${build.stand-alone.root}"/>
  <jar destfile="${appName.jar}" basedir="${build.stand-alone.root}"/>
</target>
```

Creating WAR Files

Creating WAR files is usually the same as creating JAR files, because they share everything except the file extension. However, the <war> task has some unique attributes and nested elements. Table 5-9 shows the attributes.

Table 5-9. *The <war> Task's Attributes*

Attribute	Description
basedir	The directory that will form the root of the resultant WAR file. The default is the base directory of the project.
compress	Tells Ant to compress the files as it adds them to the WAR file. If keepcompression is set to true, this applies to the entire archive, not just to the files you are adding. The default is true.

Table 5-9. *The <war> Task's Attributes*

Attribute	Description
defaultexcludes	Tells Ant to use the default excludes (see Chapter 4). The default is true.
destfile	The name of the WAR file you want to create. This attribute is required.
duplicate	Tells Ant what to do if duplicate files are found. You can specify add, preserve, or fail. The default is add.
encoding	The encoding to use for filenames in the archive. The default is UTF8.
excludes	The excludes list for this task, where each entry is separated from the next one with a space or a comma. You may use wildcards. The default is to omit nothing except the default excludes.
excludesfile	The name of the file that contains the exclude patterns. The default is to not use a file.
filesonly	Tells Ant to store only file entries. The default is false.
includes	The includes list for this task, where each entry is separated from the next one with a space or a comma. You may use wildcards. The default is all files.
includesfile	The name of the file that contains the include patterns. The default is to not use a file.
keepcompression	Tells Ant to keep the original compression of the files you are adding. The default is false.
manifest	The name of the manifest file to use. The default is null.
roundup	Tells Ant to round up file modification times to the next even number of seconds. If you don't do this, the times will be rounded down in the WAR file. This means that the WAR file will seem out-of-date when you run the target again. If you do round up, you will have problems precompiling JSP pages, because they will always seem newer than the precompiled versions. The default is true.
update	Tells Ant to overwrite files in the WAR file. The default is false.
webxml	The location of the web.xml file for this web application. This attribute is required, unless you set update to true.

You can nest <classes>, <lib>, <metainf>, and <webinf> directories, which specify a file set that represents the files to be added to the WEB-INF/classes, WEB-INF/lib, META-INF, and WEB-INF directories, respectively. This means you can build the WAR file from disparate sources in the project's directory hierarchy, though the <webinf> element ignores any web.xml files contained in its file set. You'll see more of these nested elements in the next chapter once you have learned about zip file sets, which means you can build the WAR in one step without having to copy any files.

In the example application, however, most of the files already exist in the correct locations. Listing 5-21 shows how you build the WAR file for this application. Note the webxml attribute, which picks the web.xml file out and places it in the WAR.

Listing 5-21. *The `<war>` Task Assembles the Web Application's WAR File*

```
<!-- Build the WAR file -->
<target name="package-web" depends="copy-web" description="Build the WAR">
  <echo message="Building the WAR file"/>
  <war destfile="${appName.war}" basedir="${build.web.root}"
       webxml="${src.web.conf}/web.xml"/>
</target>
```

Building the Example Application

The final step in building both parts of the application is to link the packaging steps with `depends` attributes (or properties or `<antcall>` tasks, as per your preferences). The `package-stand-alone` and `package-web` targets don't download and build the third-party libraries, so you need to provide targets that do every step in the project build. Listing 5-22 shows the updated versions of `stand-alone-complete` and `web-complete`.

Listing 5-22. *The Updated Versions of `stand-alone-complete` and `web-complete`*

```
<target name="stand-alone-complete"
        depends="build-mysql-connector, package-stand-alone"
        description="Compile stand-alone application,
                     using CVS version of the MySQL connector">
  <echo message="Compiling stand-alone application,
                 using CVS versions of the MySQL connector"/>
</target>

<target name="web-complete"
        depends="build-mysql-connector, build-jstl, package-web"
        description="Compile web application,
                     using CVS versions of the MySQL connector and the JSTL">
  <echo message="Compiling web application,
                 using CVS versions of the MySQL connector and the JSTL"/>
</target>
```

Summary

In this chapter, you went through the processes that make up the main project build. You considered different techniques of structuring a build process and how to control which parts of the process run and which don't. This included using properties to selectively run targets, as well as the `<antcall>` task and the `depends` attribute of `<target>`, both of which are used in the same way.

You set up the example application's directory structure so that you could see these techniques on a project that has multiple applications within it. In this example, you have shared database-access code, a stand-alone application, and a web interface. These interfaces use third-party libraries for accessing the MySQL database and using the JSTL tag library. You saw how to check out the most recent version of these libraries from a CVS repository and the issues to watch for when working with third-party Ant builds.

This chapter did not include deploying or distributing the application, which I will cover in the next chapter, but did include creating JAR and WAR files of the binaries. You will also create packages of the source and include documentation in larger distributions in the next chapter.