Developing Web Applications with Ant

By Richard Hightower, CTO, TriveraTech

Excerpt from Mastering TomCat, Rick Hightower, Co-Author

Trivera Technologies | www.triveratech.com

Collaborative Developer Education Services™
Training | Mentoring | Courseware | Consulting

educate.
collaborate.
accelerate.

Developing Web Applications with Ant

By Richard Hightower, CTO, Trivera Technologies

Excerpt from book Mastering Tomcat

Another Neat Tool (Ant) enables you to automate the build deploy process of your server-side Java Web components, such as custom tags, servlets, and JSPs. In this chapter, we show you how to write Ant build files, and explain how to automate the build and deploy process for your Web component development.

Ant's easy-to-use XML-based syntax overcomes many of the issues with make. Unlike using shell scripts and batch files to create build scripts, Ant works cross-platform and is geared toward Java build files. Ant gets its cross-platform support by relying on Java for file access, compilation, and other tasks.

Ant is extensible through its support for scripting (Jython and NetRexx, among others), Java custom tasks, and the ability to call OS commands, executables, and shell scripts via an Ant exec task (normally a last-resort measure). Ant makes continuous integration possible for server-side components with its automated build script and integration with JUnit, a unit-testing framework for Java.

Developing in the J2EE environment can be tricky, with its multitude of deployment descriptors and configuration files. In addition, these files often need to be configured and reconfigured for each deployment environment, and for each application the components will be deployed in and for each phase in the development process. After all, the advantage of using components is that they can be reused by many applications--and you are not going to deploy your application and components without going through the full development cycle. You may want to deploy to different servers; say you need to deploy to your development server (maybe a local Windows box), then your integration server (Solaris or Linux), then to the QA server--and with some good fortune one day, to your production server. Now, each of these application server instances will likely use different instances of your datastore (MySQL, SQL Server, Oracle, etc.). Then add the fact that you may be trying to deploy your components to different application servers, and your build process can quickly become too complex not to automate.

Ant comes to the rescue, allowing you to automate your build and deploy process. Ant lets you manage the complexities of component development and make continuous integration with J2EE development possible.

Note

Ant was developed by the Apache Software Foundation as part of its Jakarta project. Ant 1.5 is distributed with the Apache Software License version 1.1, and you can download it at http://jakarta.apache.org/ant/index.html.

This chapter starts out with a quick tour of Ant. For those of you who want an easier time of it, we present a step-by-step tutorial to using Ant with a limited set of Ant built-in tasks.

Setting Up Your Environment to Run Ant

If you are running Unix, install Ant in ~/tools/ant; if you are running Windows, install Ant in c:\tools\ant. You can set up the environment variables in Windows by using Control Panel. However, for your convenience, we created a Unix shell script (setenv.sh) and a Windows batch file (setenv.bat) that will set up the required environment variables for you.

Your Unix setenv.sh file should look something like this:

```
#
# Setup build environment variables using Bourne shell
#
export USR_ROOT=~
export JAVA_HOME=${USR_ROOT}/jdk1.4
export ANT_HOME=${USR_ROOT}/tools/ant
export PATH=${PATH}:${ANT_HOME}/bin
```

Your Windows setenv.bat file should look something like this:

```
:
: Setup build environment variables using DOS Batch
:
set USR_ROOT=c:
set JAVA_HOME=%USR_ROOT%\jdk1.4set
CLASSPATH=%USR_ROOT%\jdk1.4\lib\tools.jar;%CLASSPATH%
set ANT_HOME=%USR_ROOT%\tools\Ant
PATH=%PATH%;%ANT_HOME%\bin
```

Both of these setup files begin by setting JAVA_HOME to specify the location where you installed the JDK. This setting should reflect your local development environment--make adjustments accordingly. Then, the files set up the environment variable ANT_HOME, the location where you installed Ant.

Note

The examples in this chapter assume that you have installed Ant in c:\tools\ant on Windows and in ~/tools/ant on Unix.

What Does an Ant Build File Look Like?

Ant, which is XML based, looks a little like HTML. It has special tags called *tasks*, which give instructions to the Ant system. These instructions tell Ant how to manage files, compile files, jar files, create Web application archive files, and much more.

Listing 20.1 shows a simple Ant build file with two targets. Ant build scripts have a root element called project. The project element consists of subelements called *targets*. These elements in turn have *task* elements. Task elements do useful things, such as creating directories and compiling Java source into Java classes. All of the tasks for a given target are executed when the target is scheduled to execute. The project element has a default attribute, which specifies the target that should be executed for the project. In Listing 20.1, the compile target is the default. However, the compile target has a dependency specified by the depends attribute. The depends attribute can specify a comma-delimited list of dependencies for a target.


```
<project name="hello" default="compile">
  <target name="prepare">
      <mkdir dir="/tmp/classes" />
  </target>

  <target name="compile" depends="prepare">
      <javac srcdir="./src" destdir="/tmp/classes" />
  </target>


</project>
```

Listing 20.1

A simple Ant build file.

In Listing 20.1, the compile target depends on the prepare target; thus, the prepare target is executed before the compile target. The prepare target uses the mkdir task to create the directory /tmp/prepare. Then, the compile target executes the javac task to build the Java source files contained in the ./src directory.

Properties, File Sets, and Paths

Ant supports properties, file sets, and paths. Properties are to Ant build scripts as environment variables are to shell scripts and batch files. Properties allow you to define string constants that can be reused. The general rule is that if you use a string literal more than twice, you should probably create a property for it. Later when you need to modify the string constant, if you've defined it as a property you have to change it only once. In Listing 20.2 we've defined two properties: lib and outputdir.

```
<property name="lib" value="../lib"/>
<property name="outputdir" value="/tmp"/>

...
  <path id="myclasspath">
     <pathelement path=".;${lib}/log4j.jar"/>
     <fileset dir="${lib}">
        <include name="**/*.jar"/>
      </fileset>
  </path>
...
    <javac srcdir="./src" destdir="${outputdir}/classes" >
          <classpath refid="myclasspath"/>
    </javac>
```

Listing 20.2

Defining the properties lib and outputdir.

A property is defined as follows:

```
<property name="lib" value="../lib"/>
```

A property can be used inside another string literal as shown here:

```
<fileset dir="${lib}">
```


A file set is used to define a set of files. It allows you to group files based on patterns using one or more include subelements, as shown in Listing 20.2. A file set is defined as follows:

```
<fileset dir="${lib}">
   <include name="**/*.jar"/>
 </fileset>
```

A path allows you to define such things as classpaths (for compiling Java source and other tasks that need classpaths). A path consists of pathelements, which is a semicolon-delimited list of subdirectories or JAR files and file sets. Listing 20.2 uses the file set to add all the JAR files in the lib directory to the classpath defined with the ID myclasspath. Later in Listing 20.2 myclasspath is used by the javac task.

A path is defined as follows:

```
<path id="myclasspath">
   <pathelement path=".;${lib}/log4j.jar"/>
   <fileset dir="${lib}">
      <include name="**/*.jar"/>
    </fileset>
</path>
```

A path can be used by a task using the classpath subelement:

```
<javac srcdir="./src" destdir="${outputdir}/classes" >
      <classpath refid="myclasspath"/>
</javac>
```

Using Properties

Look at the number of environment variables being used in Listing 20.3: jdbc.jar, jdbc.driver, jdbc.userid, jdbc.password, jdbc.url, jdbc.jar, and build.sql. Defining properties can quickly become tedious, and often properties vary quite a bit from development environments to integration, QA, and production environments--so it makes sense to define properties in another file.

```
<project name="EJBQL" default="compile" >

    <property environment="env"/>

    <property file="build.properties" />

    <target name="createtables">
            <sql driver="${jdbc.driver}" url="${jdbc.url}"
                 userid="${jdbc.userid}" password="${jdbc.password}"
                 src="${build.sql}" print="yes">
                    <classpath>
                         <pathelement location="${jdbc.jar}"/>
                    </classpath>
            </sql>
    </target>
...
```


Listing 20.3

The Zen of property usage.

To import a whole file of properties, you use the property element:

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

Listing 20.3 defines the target createTables to create database tables using the sql task. Simply by changing the current build.properties file (shown in Listing 20.4) to the one shown in Listing 20.5, we can make the build script build database tables for MySQL instead of PointBase.

```
jdbc.driver=com.pointbase.jdbc.jdbcUniversalDriver
jdbc.userid=petstore
jdbc.password=petstore
jdbc.url=jdbc:pointbase:server://localhost/demo
jdbc.jar=${wlhome}/samples/server/eval/pb/lib/pbclient.jar
build.sql=build.sql
```

Listing 20.4

The PointBase build.properties.

```
jdbc.userid=trivera
jdbc.password=trivera7
jdbc.driver=org.gjt.mm.mysql.Driver
jdbc.url=jdbc:mysql://localhost/trivera
jdbc.jar=c:/mysql/mysql.jar
build.sql=mybuild.sql
```

Listing 20.5

The MySQL build.properties.

In addition, you can import environment variables as properties to make minor tweaks from one environment to another. To import environment variables from your OS as properties prepended with *env.*, you'd use the property element:

```
<property environment="env"/>
```

The environment variables can be used in a property file to define other properties:

```
src=${env.WS}/src
classes=${env.WS}/classes
jardir=${env.WS}/jars
```

This becomes useful if some members of your team use Linux while others use Windows. It appears Linux users are a bit picky about putting directories off the root directory--go figure. Windows users are generally not as picky. Specify the root for the application workspace with an environment variable and you can keep both developers happy.

Conditional Targets

Targets can be conditionally executed, depending on whether you set a property. For example, to optionally execute a target if the production property is set, you define a target as follows:

```
<target name="setupProduction" if="production">
    <property name="lib" value="/usr/home/production/lib"/>
    <property name="outputdir" value="/usr/home/production/classes"/>
</target>
```


To define a target that executes only if the production target is not set, use this:

```
<target name="setupDevelopment" unless="production">
      <property name="lib" value="c:/hello/lib"/>
      <property name="outputdir" value="c:/hello/classes"/>
  </target>
```

Conditional execution of targets is another way to support multiple deployment environments.

Using Filters

You can use filters to replace tokens in a configuration file with their proper values for the deployment environment (see Listing 20.6). Filters are another way to support multiple deployment environments. Let's look at a case where you have both a production database and a development database. When deploying, you want the production value (jdbc_url in Listing 20.6) to refer to the correct database.

```
<project name="hello" default="run">
  <target name="setupProduction" if="production">
        <filter token="jdbc_url" value="jdbc:rdbms:production"/>
  </target>

 <target name="setupDevelopment" unless="production">
        <filter token="jdbc_url" value="jdbc:rdbms:development"/>
  </target>

  <target name="setup" depends="setupProduction,setupDevelopment"/>

  <target name="run" depends="setup">
        <copy todir="/usr/home/production/properties" filtering="true">
              <fileset dir="/cvs/src/properties"/>
        </copy>
  </target>
</project>
```

Listing 20.6

An example that uses filters.

The setupProduction and setupDevelopment targets are executed conditionally based on the production property, then they set the filter to the proper JDBC driver:

```
  <target name="setupProduction" if="production">

     <filter token="jdbc_url" value="jdbc::production"/>

  </target>

  <target name="setupDevelopment" unless="production">

     <filter token="jdbc_url" value="jdbc::development"/>

  </target>
```

In our example, the filter in the setupProduction target sets jdbc_url to jdbc::production, while the filter in the setupDevelopment target sets jdbc_url to jdbc::development.

Later, when the script uses a copy task with filtering on, it applies the filter to all files in the file set specified by the copy. The copy task with filtering on replaces all occurrences of the string @jdbc_url@ with jdbc::production if the production property is set but to jdbc::development if the production property is not set.

Creating a Master Build File

Applications can include many components that can be reused by other applications. In addition, components can be written to target more than one application server. An application may consist of a large project that depends on many smaller projects, and these smaller projects also build components. Because Ant allows one Ant build file to call another, you may have an Ant file that calls a hierarchy of other Ant build files. Listing 20.7 shows a sample build script that calls other build scripts. The application project can direct the building of all its components by passing sub build files certain properties to customize the build for the current application.

```
<project name="main" default="build" >
...
 <target name="build" depends="prepare"
            description="build the model and application modules.">

         <ant dir="./model" target="package">
                <property name="outdir" value="/tmp/app" />
                <property name="setProps" value="true" />
         </ant>

        <ant dir="./application" target="package">
                <property name="outdir" value="/tmp/app" />
                <property name="setProps" value="true" />
        </ant>
        ...
    </target>
</project>
```

Listing 20.7

A build script that calls other build scripts.

The ant task runs a specified build file. You have the option of specifying the build file name or just the directory. (The application uses the file build.xml in the directory specified by the dir attribute.) You can also specify a target that you want to execute. If you do not specify a target, the application uses the default target. Any properties that you set in the called project are available to the nested build file.

Here are some examples. First, to call a build file from the current build file and pass a property:

```
<ant antfile="./hello/build.xml">
    <property name="production" value="true"/>
</ant>
```

Next, to call a build file from the current build file (since an Ant file is not specified, it uses ./hello/build.xml):

```
<ant dir="./hello"/>
```

Finally, to call a build file from the current build file and specify the run target that you want to execute (if the run target is not the default target, it will be executed anyway):


```
<ant antfile="./hello/build.xml"  target="run"/>
```

Using the War Task to Create a WAR File

Ant defines a special task for building Web Archive (WAR) files. The *war task* defines two special file sets for classes and library JAR files: classes and lib, respectively. Any other file set would be added to the document root of the WAR file (see Chapter 6 for more information).

Listing 20.8 uses the war task with three file sets. The war task will put the files specified by the fileset into the document root. As you can see in Listing 20.8, one file set includes the helloapplet.jar file, and the other two file sets contain all the files in the HTML and JSP directories. All three file sets will end up in the docroot of the Web application archive.

The war task body also specifies where Tomcat should locate the classes using the file set called classes (<classes dir="${build}" />). Files in the classes directory will be added to the WAR file at the docroot/WEB-INF/classes. Similarly, the lib element specifies a file set that will be added to the WAR file's docroot/WEB-INF/lib directory.

```
        <war warfile="${dist}/hello.war" webxml="${meta}/web.xml">
            <!--
                Include the html and jsp files.
                Put the classes from the build into the classes
directory of the war.
            /-->
            <fileset dir="./HTML" />
            <fileset dir="./JSP" />
            <!-- Include the applet. /-->
            <fileset dir="${lib}" includes="helloapplet.jar" />

            <classes dir="${build}" />

            <!-- Include all of the jar files except the ejbeans and
 applet. The other build files that create jars have to be run in
                 the correct order. This is covered later.
             /-->
            <lib dir="${lib}" >
                <exclude name="greet-ejbs.jar"/>
                <exclude name="helloapplet.jar"/>
            </lib>
        </war>
```

Listing 20.8

Using the war task.

The war task is a convenient method of building WAR files, but it does not help you write the deployment descriptors. XDoclet does help you write the deployment descriptors--and so much more to learn more about XDoclet go to http://xdoclet.sourceforge.net, read the Mastering Tomcat book or read the IBM tutorial that I wrote for XDoclet.

Running Ant for the First Time

To run the sample Ant build file, go to the directory that contains the project files. To run Ant, navigate to the examples/chap20/webdoclet directory and type:

```
ant deploy
```

As we stated earlier, Ant will find build.xml, which is the default name for the build file. (You may have to adjust your build.properties files.) For our example, here is the command-line output you should expect:

```
C:\tomcatbook\chap20\webdoclet>ant deploy
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\tmp\war

compile:
    [javac] Compiling 2 source files to C:\tomcatbook\webdoclet\WEB-
INF\classes


package:
      [war] Building war: C:\tmp\war\myapp.war

deploy:
     [copy] Copying 1 file to C:\tomcat4\webapps

BUILD SUCCESSFUL
Total time: 14 seconds
```

Notice that the targets and their associated tasks are displayed. That's it!

<SOME SECTIONS OMITTED>

Standard Targets

Steve Loughran wrote an Ant guide called *Ant in Anger* that is part of the Ant download. This guide explains many pitfalls and recommends ways to use Ant. Two very useful suggestions are creating a list of names for targets and learning how to divide build files.

The following are some of Steve's recommended names for Ant top-level targets:

test--Runs the junit tests

clean--Cleans the output directories

deploy--Ships the JARs, WARs, and so on to the execution system

publish--Outputs the source and binaries to any distribution site

fetch--Gets the latest source from the Concurrent Versions System (CVS) tree

docs/javadocs--Outputs the documentation

all--Performs clean, fetch, build, test, docs, and deploy

main--Performs the default build process (usually build or build and test)

The following are some recommended names for Ant internal targets:

init--Initializes properties and performs other initialization tasks; read in per-user property files

init-debug--Initializes debug properties

init-release--Initializes release properties

compile--Performs the actual compilation

link/jar--Makes the JARs or equivalent files

staging--Carries out any pre-deployment process in which the output is dropped off and then tested before being moved to the production site

We'll discuss some of the concepts from *Ant in Anger* in this chapter. We strongly suggest that you read *Ant in Anger*, because it contains excellent guidelines for using Ant. The guide is included with the Ant binary distribution under the docs directory.

We'll now begin examining the techniques involved in using Ant to build and deploy Java applications. During the rest of this chapter, we build a "Hello World" example. Because the emphasis is on how to *build* components with Ant--and not the mechanics of implementing these various components--the example is meant to be as simple as possible. We package the application and the common JAR, construct a build file that creates an applet, and construct a master build file that coordinates the entire build.

The source code for the Hello World example is divided into several directories. Each directory has its own Ant build file, which contains instructions for compiling the components and packaging the binaries. Each directory also includes the requisite configuration files (such as manifest files). The master Ant build file, located in the root directory, calls the other build files and coordinates the entire build.

This divide-and-conquer technique for organizing components into separate directories is quite common with Ant. It may seem like overkill on a simple project like this one, but suppose you were building a system with 50 components. Each component has its own set of deployment descriptors and configuration files, and each component is deployed in different containers. The divide-and-conquer technique becomes necessary to manage the complexity--it also makes it easier to reuse components.

Here is the directory structure for the Hello World project:

```
Model 2 Hello World root
+---Model
+---Application
+---Applet
+---WebApplication
```

The Model directory holds the common code (in this simple project, only the application will access the common code). The Application directory holds the Java application code, including the manifest file that marks the deployment JAR as executable. The Applet directory holds the applet code. The WebApplication directory holds the Web application code.

The Hello World Model Project

This section explains the basic structure for all the build files and directories you use in this example. We introduce the three Java class files: a GreetingBean class, a GreetingFactory class, and a Greeting interface. Then, we discuss how to build these files with Ant and break down the Ant build files' target execution step by step. We also explain how to use the Ant command-line utility to build the files with Ant.

Overview of Model Classes

You use the GreetingFactory class to create a Greeting object. Here is the code for GreetingFactory:

```
package xptoolkit.model;

public class GreetingFactory {
    private GreetingFactory(){}

    public Greeting getGreeting()throws Exception {
     String clazz = System.getProperty("Greeting.class",
                            "xptoolkit.model.GreetingBean");
        return (Greeting)Class.forName(clazz).newInstance();
    }

    public static GreetingFactory getGreetingFactory(){
        return new GreetingFactory();
    }

}
```

Next we have a Greeting interface that defines the *contract* of a Greeting object--that is, what type of behavior it supports. The Greeting interface looks like this:

```
package xptoolkit.model;

public interface Greeting extends java.io.Serializable{
    public String getGreeting();
}
```

Finally, the GreetingBean class implements the Greeting interface. GreetingBean is defined as follows:

```
package xptoolkit.model;

public class GreetingBean implements Greeting{

    public GreetingBean(){}

    public String getGreeting(){
      return "Hello World!";
    }
}
```

GreetingBean returns the message "Hello World!" . To create a Greeting instance, you use GreetingFactory. The default implementation of GreetingFactory gets the implementation class from a property and instantiates an instance of that class with the Class.forName().newInstance() method. It casts the created instance to the Greeting interface.

These two lines of code create the Greeting instance from GreetingFactory's getGreeting() method:

```
String clazz = System.getProperty("Greeting.class",
                        "xptoolkit.model.GreetingBean");
```



```
return (Greeting)Class.forName(clazz).newInstance();
```

Thus, any class that implements the Greeting interface can substitute for the Greeting.class system property. Then, when the class is instantiated with the factory's getGreeting() method, the application uses the new implementation of the Greeting interface.

In the book *Java Tools for Extreme Programming* (also published by Wiley), we use this technique to transparently add support for EJBs to the Web application. In that book we also create an Ant script that can deploy the same Web application to use either enterprise beans or another bean implementation just by setting an Ant property (and some Ant filter magic). We also map the Greeting interface with the use bean action of a JSP when we implement the Model 2 using servlets and JSP.

Creating a Project Directory Structure for Model

This part of the example application uses the smallest build file. Basically, we need to create a JAR file that acts as a common library. We don't need any special manifest file or deployment files. This is the most basic build file and directory structure you will see in this example. Here is the directory structure for the Model directory:

```
Root of Model
|   build.xml
|
+---src
    +---xptoolkit
        \---model
                GreetingFactory.java
                Greeting.java
                GreetingBean.java
```

Notice that there are only four files in the Model directory and subdirectories. Also notice that the name of the Ant file is build.xml. The build.xml file is the default build file; if Ant is run in this directory, it automatically finds build.xml without you having to specify it on the command line. Let's examine the model build file in greater detail.

Creating a Build File for a Shared Library

The model build file has six targets: init, clean, prepare, compile, package, and all. The build files in this example have similar targets:

init--Initializes all the other properties relative to the outputdir.

clean--Cleans up the output directories and the output JAR file.

prepare--Creates the output directories if they do not already exist.

compile--Compiles the Java source files for the model into the build directory defined in the init target.

package--Packages the compiled Java source into a JAR file.

all--Runs all the tags. It is the default target of this build project.

Analysis of the Model Project Build File

Listing 20.17 shows the entire build file for the model project. In this section, we provide a step-by-step analysis of how this build file executes. All the build files in the Hello World example are structured in a similar fashion, so understanding the model project build file is essential to understanding the others. A quick note on naming conventions: As you see from the first line of code in Listing 20.17, the project name for this build file is model. Therefore, we refer to this build file as the *model project build file*. This naming convention becomes essential once we begin dealing with the five other build files in this project.

```
<project name="model" default="all" >


    <property name="outdir" value="/tmp/app/" />

    <target name="init"
                          description="initialize the properties.">
        <property name="local_outdir" value="${outdir}/model" />
        <property name="build" value="${local_outdir}/classes" />
        <property name="lib" value="${outdir}/lib" />
        <property name="model_jar" value="${lib}/greetmodel.jar" />
    </target>

    <target name="clean" depends="init"
                     description="clean up the output directories
  and jar.">
        <delete dir="${local_outdir}" />
        <delete file="${model_jar}" />
    </target>

    <target name="prepare" depends="init"
                          description="prepare the output
  directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${lib}" />
    </target>

    <target name="compile" depends="prepare"
                          description="compile the Java source.">
        <javac srcdir="./src" destdir="${build}" />
    </target>

    <target name="package" depends="compile"
                    description="package the Java classes into a
  jar.">
        <jar jarfile="${model_jar}"
           basedir="${build}" />
    </target>

    <target name="all" depends="clean,package"
                    description="perform all targets."/>

</project>
```

Listing 20.17

The Hello World model project build file.

Let's go over the model project build file and each of its targets in the order they execute. First, the model project sets the all target as the default target:

```
<project name="model" default="all" >
```

The all target is executed by default, unless we specify another target as a command-line argument of Ant. The all target depends on the clean and package targets. The clean target depends on the init target; thus, the init target is executed first.

Next, the init target is executed. The init target is defined in build.xml as follows:

```
<target name="init"
                      description="initialize the properties.">

   <property name="local_outdir" value="${outdir}/model" />
   <property name="build" value="${local_outdir}/classes" />
   <property name="lib" value="${outdir}/lib" />
   <property name="model_jar" value="${lib}/greetmodel.jar" />
</target>
```

The init target defines several properties that refer to directories and files needed to compile and deploy the model project. Let's discuss the meaning of these properties because all the other build files for this example use the same or similar properties. The init target defines the following properties:

local_outdir--Defines the output directory of all the model project's intermediate files (Java class files).

build--Defines the output directory of the Java class files.

lib--Defines the directory that holds the common code libraries (JAR files) used for the whole Model 2 Hello World example application.

model_jar--Defines the output JAR file for this project.

Note

As a general rule, if you use the same literal twice, you should go ahead and define it in the init target. You don't know how many times we've shot ourselves in the foot by not following this rule. This build file is fairly simple, but the later ones are more complex. Please learn from our mistakes (and missing toes).

Now that all the clean target's dependencies have executed, the clean target can execute. The clean target deletes the intermediate files created by the compile and the output common JAR file, which is the output of this project. Here is the code for the clean target:

```
<target name="clean" depends="init"
         description="clean up the output directories and jar.">

  <delete dir="${local_outdir}" />
  <delete file="${model_jar}" />

</target>
```

Remember that the all target depends on the clean and package targets. The clean branch and all its dependencies have now executed, so it is time to execute the package target branch (a *branch* is a target and all its dependencies). The package target depends on the compile target, the compile target depends on the prepare target, and the prepare target depends on the init target, which has already been executed.

Note

> During development you would probably not use the all target; instead, you would use the compile or package target. However, as a general rule, the default target is preferred as the most complete target with the fewest side effects. You can always override this during development by passing the desired target on the command line when invoking Ant.

Thus, the next target that executes is prepare, because all its dependencies have already executed. The prepare target creates the build output directory, which ensures that the lib directory is created. The prepare target is defined as follows:

```
<target name="prepare" depends="init"
                       description="prepare the output directory.">

    <mkdir dir="${build}" />
    <mkdir dir="${lib}" />
</target>
```

The next target in the package target branch that executes is the compile target--another dependency of the package target. The compile target compiles the code in the src directory to the build directory, which was defined by the build property in the init target. The compile target is defined in this way:

```
 <target name="compile" depends="prepare"
                       description="compile the Java source.">

    <javac srcdir="./src" destdir="${build}"/>
 </target>
```

Now that all the target dependencies of the package target have been executed, we can run the package target. The package target packages the Java classes created in the compile target into a JAR file in the common lib directory. The package target is defined as follows:

```
<target name="package" depends="compile"
               description="package the Java classes into a jar.">

    <jar jarfile="${model_jar}"
       basedir="${build}" />
</target>
```

Running an Ant Build File

In this section, we discuss how to run the Hello World model project build file. There are three steps to running this file:

1. Set up the environment.
2. Go to the directory that contains the build.xml file for the model.
3. Run Ant.

Successfully running the build script gives us the following output:

```
Buildfile: build.xml

init:
```

```
clean:

prepare:
    [mkdir] Created dir: C:\tmp\app\model\classes

compile:
    [javac] Compiling 3 source files to C:\tmp\app\model\classes

package:
      [jar] Building jar: C:\tmp\app\lib\greetmodel.jar

all:

BUILD SUCCESSFUL

Total time: 3 seconds
```

If you do not get this output, check that the properties defined in the init target make sense for your environment. If you are on a Unix platform and the build file is not working, make sure that the /tmp directory exists and that you have the rights to access it. Alternatively, you could run the previous script by entering this on the command line:

```
$ ant -Doutdir=/usr/rick/tmp/app
```

Basically, you want to output to a directory that you have access to, just in case you are not the administrator of your own box. If for some reason Ant still does not run, make sure you set up the Ant environment variables.

After you successfully run Ant, the output directory for the model project looks like this:

```
Root of output directory
\---app
    +---lib
    |       greetmodel.jar
    |
    \---model
        \---classes
            \---xptoolkit
                \---model
                        GreetingFactory.class
                        Greeting.class
                        GreetingBean.class
```

Notice that all the intermediate files to build the JAR file are in the model subdirectory. The output from this project is the greetmodel.jar file, which is in ${outdir}/app/lib. The next project, the application project, needs this JAR file in order to compile. In the next section, we discuss how to use Ant to build a stand-alone Java application that uses the JAR file (greetmodel.jar) from the model project.

The Hello World Application Project

The goal of the Hello World application project is to create a stand-alone Java application that uses greetmodel.jar to get the greeting message. The application project build file is nearly identical to the model project build file, so we focus our discussion on the differences between the two. We also explain how to make a JAR file an executable JAR file.

Overview of Application Java Classes

The Java source code for this application is as simple as it gets for the Hello World Model 2 examples. Here is the Java application:

```
package xptoolkit;

import xptoolkit.model.GreetingFactory;
import xptoolkit.model.Greeting;

public class HelloWorld{

    public static void main(String []args)throws Exception{
       Greeting greet = (Greeting)
         GreetingFactory.getGreetingFactory().getGreeting();

       System.out.println(greet.getGreeting());

    }
}
```

As you can see, this application imports the GreetingFactory class and the Greeting interface from the model project. It uses GreetingFactory to get an instance of the Greeting interface, and then uses the instance of the Greeting interface to print the greeting to the console.

Creating a Project Directory Structure for the Application

The directory structure of the Hello World Java application is as follows:

```
Hello World Application root
|   build.xml
|
+---src
|   |
|   +---xptoolkit
|           HelloWorld.java
|
\---META-INF
        MANIFEST.MF
```

Notice the addition of the META-INF directory, which holds the name of the manifest file we will use to make the application's JAR file executable. The only other file that this project needs is not shown; the file is greetmodel.jar, which is created by the model project (the reason for this will become obvious in the following sections).

Creating a Manifest File for a Stand-alone Application

The goal of this application is for it to work as a stand-alone JAR file. To do this, we need to modify the manifest file that the application JAR file uses to include the main class and the dependency on greetmodel.jar. The manifest entries that this application needs look something like this:

```
Manifest-Version: 1.0
Created-By: Rick Hightower
Main-Class: xptoolkit.HelloWorld
```



```
Class-Path: greetmodel.jar
```

The JAR file that holds the Hello World Java application needs to run a certain JAR file (in our case, greetmodel.jar), as specified by the Class-Path manifest entry The Main-Class manifest entry specifies the main class of the JAR file--that is, the class with the main() method that is run when the executable JAR file executes.

Creating an Ant Build File for a Stand-alone Application

Listing 20.18 shows the application project build file; you'll notice that it is very similar to the model project build file. It is divided into the same targets: init, clean, delete, prepare, mkdir, compile, package, and all. The application project build file defines the properties differently, but even the property names are almost identical (compare with the model project build file in Listing 20.17).

```
<project name="application" default="all" >


    <property name="outdir" value="/tmp/app" />

    <target name="init"
                          description="initialize the properties.">
      <property name="local_outdir" value="${outdir}/java_app" />
      <property name="build" value="${local_outdir}/classes" />
      <property name="lib" value="${outdir}/lib" />
      <property name="app_jar" value="${lib}/greetapp.jar" />
    </target>

    <target name="clean" depends="init"
                          description="clean up the output
directories.">
         <delete dir="${build}" />
         <delete file="${app_jar}" />
    </target>

    <target name="prepare" depends="init"
                          description="prepare the output directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${lib}" />
    </target>

    <target name="compile" depends="prepare"
                          description="compile the Java source.">

        <javac srcdir="./src" destdir="${build}">
            <classpath >

                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>

            </classpath>

        </javac>

    </target>
```



```
    <target name="package" depends="compile"
                   description="package the Java classes into a jar.">
        <jar jarfile="${app_jar}"
             manifest="./META-INF/MANIFEST.MF"
           basedir="${build}" />
    </target>

    <target name="all" depends="clean,package"
                         description="perform all targets."/>

</project>
```

Listing 20.18

The Hello World application project build file.

One of the differences in the application project build file is the way that it compiles the Java source:

```
    <target name="compile" depends="prepare"
                         description="compile the Java source.">

        <javac srcdir="./src" destdir="${build}">
            <classpath >

                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>

            </classpath>

        </javac>

    </target>
```

Notice that the compile target specifies all the JAR files in the common lib directory (<include name="**/*.jar"/>). The greetmodel.jar file is in the common lib directory, so it is included when the javac task compiles the source. Another difference is the way the application project build file packages the Ant source:

```
    <target name="package" depends="compile"
                   description="package the Java classes into a jar.">
        <jar jarfile="${app_jar}"
             manifest="./META-INF/MANIFEST.MF"
           basedir="${build}" />
    </target>
```

Notice that the package target uses the jar task as before, but the jar task's manifest is set to the manifest file described earlier. This is unlike the model project build file, which did not specify a manifest file; the model used the default manifest file. The application project build file's manifest file has the entries that allow us to execute the JAR file from the command line.

In order to run the Hello World Java application, after we run the application project build file we go to the output common lib directory (tmp/app/lib) and run Java from the command line with the -jar command-line argument:

```
$ java -jar greetapp.jar
Hello World!
```


You may wonder how the application loaded the Greeting interface and the GreetingFactory class. This is possible because the manifest entry Class-Path causes the JVM to search for any directory or JAR file that is specified (refer to the JAR file specification included with the Java Platform documentation for more detail). The list of items (directory or JAR files) specified on the Class-Path manifest entry is a relative URI list. Because the greetmodel.jar file is in the same directory (such as /tmp/app/lib) and it is specified on the Class-Path manifest, the JVM finds the classes in greetmodel.jar.

One issue with the application project is its dependence on the model project. The model project must be executed before the application project. How can we manage this? The next section proposes one way to manage the situation with an Ant build file.

The Hello World Main Project

The Hello World Java application depends on the existence of the Hello World model common library file. If we try to compile the application before the model, we get an error. The application requires the model, so we need a way to call the model project build file and the application project build file in the correct order.

Creating a Master Build File

We can control the execution of two build files by using a master build file. The master build file shown in Listing 20.19 is located in the root directory of the Model 2 Hello World Example of the main project. This build file treats the model and application build files as subprojects (the model and application projects are the first of many subprojects that we want to fit into a larger project).

```
<project name="main" default="build" >


    <property name="outdir" value="/tmp/app" />


    <target name="init"
                          description="initialize the properties.">
          <property name="lib" value="${outdir}/lib" />
    </target>

    <target name="clean" depends="init"
                          description="clean up the output
directories.">
        <ant dir="./Model" target="clean">
            <property name="outdir" value="${outdir}" />
        </ant>

        <ant dir="./Application" target="clean">
            <property name="outdir" value="${outdir}" />
        </ant>

        <delete dir="${outdir}" />

    </target>
```



```
    <target name="prepare" depends="init"
                          description="prepare the output directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${lib}" />
    </target>


    <target name="build" depends="prepare"
            description="build the model and application modules.">

        <ant dir="./model" target="package">
            <property name="outdir" value="${outdir}" />
        </ant>

        <ant dir="./application" target="package">
            <property name="outdir" value="${outdir}" />
        </ant>
    </target>

</project>
```

Listing 20.19

The Hello World master build file.

Analysis of the Master Build File

Notice that the main project build file simply delegates to the application and model subproject and ensures that the subprojects' build files are called in the correct order. For example, when the clean target is executed, the main project build file uses the ant task to call the model project's clean target. Then, the main project calls the application project's clean target using the ant task again, as shown here:

```
    <target name="clean" depends="init"
                          description="clean up the output
directories.">
        <ant dir="./Model" target="clean">
            <property name="outdir" value="${outdir}" />
        </ant>

        <ant dir="./Application" target="clean">
            <property name="outdir" value="${outdir}" />
        </ant>

        <delete dir="${outdir}" />

    </target>
```

A similar strategy is used with the main project's build target. The build target calls the package target on both the model and application subprojects:

```
    <target name="build" depends="prepare"
            description="build the model and application modules.">

        <ant dir="./model" target="package">
            <property name="outdir" value="${outdir}" />
```

```
        </ant>

        <ant dir="./application" target="package">
            <property name="outdir" value="${outdir}" />
        </ant>

    </target>
```

Thus, we can build both the application and model projects by running the main project. This may not seem like a big deal, but imagine a project with hundreds of subprojects that build thousands of components. Without a build file, such a project could become unmanageable. In fact, a project with just 10 to 20 components can benefit greatly from using nested build files. The master build file orchestrates the correct running order for all the subprojects. We could revisit this main project after we finish each additional subproject and update it. In the next section, we discuss the applet build file.

The Applet Project

The applet project is a simple applet that reads the output of the HelloWorldServlet and shows it in a JLabel. The dependency on the Web application is at runtime; there are no compile-time dependencies to the Web application. Let's take a look at the applet next.

Overview of the Applet Class

The meat of the applet implementation is in the init() method, as shown here:

```
    public void init(){
        URL uGreeting;
        String sGreeting="Bye Bye";

        getAppletContext()
           .showStatus("Getting hello message from server.");

        try{
            uGreeting = new URL(
                   getDocumentBase(),
                   "HelloWorldServlet");

            sGreeting = getGreeting(uGreeting);
        }
        catch(Exception e){
            getAppletContext()
                .showStatus("Unable to communicate with server.");
            e.printStackTrace();
        }
        text.setText(sGreeting);

    }
```

The init() method gets the document base URL (the URL from which the applet's page was loaded) from the applet context. Then, the init() method uses the document base and the URI identifying HelloWorldServlet to create a URL that has the output of HelloWorldServlet:

```
uGreeting = new URL( getDocumentBase(), "HelloWorldServlet");
```


The init() method uses a helper method called getGreeting() to parse the output of HelloWorldServlet. It then displays the greeting in the applet's JLabel (text.setText(sGreeting);). The helper method looks like this:

```
    private String getGreeting(URL uGreeting)throws Exception{
         String line;
         int endTagIndex;
         BufferedReader reader=null;
        . . .

           reader = new BufferedReader(
                    new InputStreamReader (
                            uGreeting.openStream()));

            while((line=reader.readLine())!=null){
                System.out.println(line);

                if (line.startsWith("<h1>")){
                    getAppletContext().showStatus("Parsing message.");
                    endTagIndex=line.indexOf("</h1>");
                    line=line.substring(4,endTagIndex);
                    break;
                }
            }
          ...
         return line;
    }
```

Basically, the method gets the output stream from the URL (uGreeting.openStream()) and goes through the stream line by line looking for a line that begins with <h1>. Then, it pulls the text out of the <h1> tag.

The output of HelloServlet looks like this:

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World!</h1>
</body>
```

The code that the helper method retrieves appears in bold. Listing 20.20 shows the complete code for HelloWorldApplet.

```
package xptoolkit.applet;
import javax.swing.JApplet;
import javax.swing.JLabel;
import java.awt.Font;
import java.awt.BorderLayout;
import java.applet.AppletContext;
import java.net.URL;
import java.io.InputStreamReader;
import java.io.BufferedReader;

public class HelloWorldApplet extends javax.swing.JApplet {

    JLabel text;
```



```
    public HelloWorldApplet() {
        this.getContentPane().setLayout(new BorderLayout());
        text = new JLabel("Bye Bye");
        text.setAlignmentX(JLabel.CENTER_ALIGNMENT);
        text.setAlignmentY(JLabel.CENTER_ALIGNMENT);
        Font f = new Font("Arial", Font.BOLD, 20);
        text.setFont(f);
        getContentPane().add(text,BorderLayout.CENTER);
    }


    public void init(){
        URL uGreeting;
        String sGreeting="Bye Bye";

        this.doLayout();
        getAppletContext()
.showStatus("Getting hello message from server.");

        try{
            uGreeting = new URL(
                   this.getDocumentBase(),
                   "HelloWorldServlet");

            sGreeting = getGreeting(uGreeting);
        }
        catch(Exception e){
            getAppletContext()
.showStatus("Unable to communicate with server.");
            e.printStackTrace();
        }
        text.setText(sGreeting);

    }

    private String getGreeting(URL uGreeting)throws Exception{
         String line;
         int endTagIndex;
         BufferedReader reader=null;


         try{
            reader = new BufferedReader(
                    new InputStreamReader (
                            uGreeting.openStream()));
            while((line=reader.readLine())!=null){
                System.out.println(line);
                if (line.startsWith("<h1>")){
                    getAppletContext().showStatus("Parsing message.");
                    endTagIndex=line.indexOf("</h1>");
                    line=line.substring(4,endTagIndex);
                    break;
                }
```

```
            }
         }
         finally{
            if (reader!=null)reader.close();
         }
         return line;
    }

}
```

Listing 20.20

The HelloWorldApplet that communicates with HelloWorldServlet.

Creating a Build File for the Applet

The applet project build file is quite simple, as shown in Listing 20.21. It is structured much like the application project build file.

```
<project name="applet" default="all" >


    <property name="outdir" value="/tmp/app" />


    <target name="init"
                          description="initialize the properties.">

        <property name="local_outdir" value="${outdir}/applet" />
        <property name="build" value="${local_outdir}/classes" />
        <property name="lib" value="${outdir}/lib" />
        <property name="jar" value="${lib}/helloapplet.jar" />
    </target>

    <target name="clean" depends="init"
                          description="clean up the output
directories.">
          <delete dir="${build}" />
          <delete dir="${jar}" />
    </target>


    <target name="prepare" depends="init"
                          description="prepare the output directory.">
          <mkdir dir="${build}" />
          <mkdir dir="${lib}" />
    </target>

    <target name="compile" depends="prepare"
                          description="compile the Java source.">
        <javac srcdir="./src" destdir="${build}" />
    </target>

    <target name="package" depends="compile"
             description="package the Java classes into a jar.">
```

```
        <jar jarfile="${jar} "
           basedir="${build}" />
    </target>

    <target name="all" depends="clean,package"
                         description="perform all targets."/>

</project>
```

Listing 20.21

The applet project build file.

Building the Applet with Ant

To build the applet, we need to navigate to the Applet directory, set up the environment, and then run Ant at the command line. First we build the applet:

```
C:\CVS\...\MVCHelloWorld\Applet>ant
Buildfile: build.xml


init:

clean:
   [delete] Deleting directory C:\tmp\app\lib

prepare:
    [mkdir] Created dir: C:\tmp\app\applet\classes
    [mkdir] Created dir: C:\tmp\app\lib

compile:
    [javac] Compiling 1 source file to C:\tmp\app\applet\classes

package:
      [jar] Building jar: C:\tmp\app\lib\helloapplet.jar

all:

BUILD SUCCESSFUL

Total time: 4 seconds
```

Now we clean the applet:

```
C:\CVS\...\MVCHelloWorld\Applet>ant clean
Buildfile: build.xml

init:

clean:
   [delete] Deleting directory C:\tmp\app\applet\classes
   [delete] Deleting directory C:\tmp\app\lib

BUILD SUCCESSFUL
```



```
Total time: 0 seconds
```

Hello World Recap

It's important to recap what we have done. We created a common Java library called model.jar. This model.jar file is used by a Web application and a stand-alone executable Java application in an executable JAR file. We created an applet that can communicate with the Web application we create in the next section. Once the applet loads in the browser, the applet communicates over HTTP to the Web application's HelloWorldServlet.

Hello World Model 2 and J2EE

The following section explains techniques for using Ant to build and deploy J2EE applications. The Hello World example includes an applet, a Web application, an application, support libraries, and other components. This may be the only Hello World example that has an applet, servlet, and JSP and that attempts to be Model 2.

You're probably thinking, "Why should I implement the most complex Hello World application in the world?" This example--although a bit complex to just say "Hello World"--is as simple as possible while demonstrating how to build and deploy a J2EE application and its components with Ant. By working through this example, you will understand how to use Ant to build these different types of components and applications, and how to combine them by nesting build files.

The Model 2 HelloWorld example for this chapter is the simplest example of Model 2 architecture--also known as Model-View-Controller (MVC)--for JSP servlets. In this example, the applet and JSP are the View; the servlet is the Controller; and the object Model is a Java class.

The Web application build file is set up so that if we add one property, it can talk to the local implementation in the original model library (common code) defined earlier. The WebApplication directory holds HTML files, deployment descriptors, JSP files, and servlet Java source.

Because each component has its own set of deployment descriptors and configuration files, it makes sense to separate components into their own directories; this practice also makes it easier to reuse the components in other projects and applications. Each directory has its own Ant build file, which knows how to compile the components, package the binaries, and include the requisite configuration files (deployment descriptors and manifest files).

In the next section, we cover the Web application build file—the heart of this example.

The Web Application Project

The Web application is another subproject of the Model 2 Hello World application; it consists of a servlet, two JSPs, an HTML file, and a deployment descriptor. This section describes how to build a WAR file with a deployment descriptor. We also explain how to map servlets and JSPs to servlet elements in the deployment descriptor and how to map the servlet elements to URLs. In addition, this section breaks down the Web application project build file step by step, and shows how to use the build file to build and deploy the Web application.

The Web Application Project Directory Structure

We build these files into a Web application:

```
Web application root directory
```



```
|   build.xml
|
+---JSP
|      HelloWorld.jsp
|      HelloApplet.jsp
|
+---src
|   \---xptoolkit
|      \---web
|             HelloWorldServlet.java
|
+---HTML
|       index.html
|
\---meta-data
       web.xml
```

Notice that the Web application project includes only six files. There are four subdirectories: JSP, src, HTML, and meta-data, and the root directory holds the build.xml file. The JSP directory contains two JSPs: HelloWorld.jsp and HelloApplet.jsp. Under the src directory is the Java source for the servlet xptoolkit.web.HelloWorldServlet.java. The web.xml file under the meta-data directory holds the deployment file for this Web application.

HelloWorldServlet.java

The servlet is contained in the class xptoolkit.web.HelloWorldServlet (see Listing 20.22). Like the Java application, it uses the Greeting interface and the GreetingFactory class that are packaged in greetmodel.jar, the output of the model project.

```
package xptoolkit.web;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.RequestDispatcher;

/* import the classes to create a greeting object or type greeting */
import xptoolkit.model.GreetingFactory;
import xptoolkit.model.Greeting;



public class HelloWorldServlet extends  HttpServlet{

    public void init(ServletConfig config) throws ServletException{
        super.init(config);
           /* Read in the greeting type that the factory should create */
        String clazz = config.getInitParameter("Greeting.class") ;
        if(clazz!=null)System.setProperty("Greeting.class",clazz);

    }
```



```
    public void doGet(HttpServletRequest request,
                                  HttpServletResponse response)
                                          throws ServletException{
        RequestDispatcher dispatch;
        ServletContext context;
       /*Get the session, create a greeting bean, map the greeting
         bean in the session, and redirect to the Hello World JSP.
       */
        try {


                /* Create the greeting bean and map it to the
session. */
          HttpSession session = request.getSession(true);
          Greeting greet = (Greeting)
               GreetingFactory.getGreetingFactory().getGreeting();
          session.setAttribute("greeting", greet);


                /* Redirect to the HelloWorld.jsp */
           context = getServletContext();
           dispatch = context.getRequestDispatcher("/HelloWorldJSP");
           dispatch.forward(request, response);
        }catch(Exception e){
            throw new ServletException(e);
        }
    }

     /* Just call the doGet method */
    public void doPost(HttpServletRequest request,
                                   HttpServletResponse response)
                                          throws ServletException{
        doGet(request, response);
    }

}
```

Listing 20.22

xptoolkit.web.HelloWorldServlet.

Analyzing HelloWorldServlet

HelloWorldServlet is a simple servlet; it reads in the servlet initialization parameters in the init() method, as follows:

```
String clazz = config.getInitParameter("Greeting.class") ;
```

It uses the value of the initialization parameter "Greeting.class" to set the System property "Greeting.class", as follows:

```
System.setProperty("Greeting.class",clazz);
```

You will recall from earlier that GreetingFactory uses the system property "Greeting.class" to decide which implementation of the Greeting interface to load. Now let's get to the real action: the doGet() and doPost() methods.

When the doGet() or doPost() method of HelloWorldServlet is called, the servlet uses GreetingFactory to create a greeting:

```
Greeting greet = (Greeting)
     GreetingFactory.getGreetingFactory().getGreeting();
```

HelloWorldServlet then maps the greeting object into the current session (javax.servlet.http.HttpSession) under the name greeting:

```
session.setAttribute("greeting", greet);
```

Finally, HelloWorldServlet forwards processing of this request to the JSP file HelloWorld.jsp by getting the request dispatcher for HelloWorldServlet from the servlet's context:

```
     /* Redirect to the HelloWorld.jsp */
context = getServletContext();
dispatch = context.getRequestDispatcher("/HelloWorldJSP");
dispatch.forward(request, response);
```

You may notice that the context.getRequestDispatcher call looks a little strange. This is because HelloWorld.jsp is mapped to /HelloWorldJSP in the deployment descriptor for the servlet. Next, let's examine HelloWorld.jsp.

HelloWorld.jsp

HelloWorld.jsp exists to display the message it gets from the Greeting reference that HelloWorldServlet mapped into that session. The HelloWorld.jsp code looks like this:

```
<jsp:useBean id="greeting" type="xptoolkit.model.Greeting"
                                              scope="session"/>
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1><%=greeting.getGreeting()%></h1>
</body>
```

If you are a Web designer at heart, we understand if you are shocked and horrified by this HTML code. But for a moment, let's focus on the following two lines of code from the JSP:

```
<jsp:useBean id="greeting" type="xptoolkit.model.Greeting"
```



```
                                                scope="session"/>

<h1><%=greeting.getGreeting()%></h1>
```

Notice that the jsp:useBean action grabs the Greeting reference that we put into the session with HelloWorldServlet. Then, we print out the greeting with the JSP scriptlet expression <%=greeting.getGreeting()%>.

This sums up what the Model 2 Hello World Web application does. We discuss the other JSP, HelloApplet.jsp, after we examine the applet subproject. For now, the next section explains why the servlet could forward HelloWorldJSP to the JSP HelloWorld.jsp.

The Deployment Descriptor for the Hello World Web Application

In order to configure the JSPs and servlets, we need a deployment descriptor. The following code defines a simple deployment descriptor that assigns names and mappings to the JSPs and servlet. Please note that the deployment descriptor goes in the web.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <error-page>
        <error-code>404</error-code>
        <location>/HelloWorldServlet</location>
    </error-page>

    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>
        <servlet-class>xptoolkit.web.HelloWorldServlet</servlet-class>
        <init-param>
          <param-name>Greeting.class</param-name>
          <param-value>@Greeting.class@</param-value>
        </init-param>
    </servlet>

    <servlet>
        <servlet-name>HelloWorldJSP</servlet-name>
        <jsp-file>HelloWorld.jsp</jsp-file>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/HelloWorldServlet</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>HelloWorldJSP</servlet-name>
        <url-pattern>/HelloWorldJSP</url-pattern>
    </servlet-mapping>
```



```
</web-app>
```

The deployment descriptor defines two servlet elements: one for HelloWorldServlet and one for HelloWorldJSP. If you are wondering why there is a servlet element for HelloWorldJSP, remember that HelloWorld.jsp is compiled to a servlet before it is used for the first time. The HelloWorldServlet servlet element maps to the servlet (<servlet-class>xptoolkit.web.HelloWorldServlet</servlet-class>). The HelloWorldJSP element maps to the JSP file HelloWorld.jsp (<jsp-file>HelloWorld.jsp</jsp-file>). Then, the servlet mapping elements map the servlet element to specific URL patterns.

Thus HelloWorldServlet maps to /HelloWorldServlet (<url-pattern>/HelloWorldServlet</url-pattern>); this is relative to the Web application location from the root of the server. And the HelloWorldJSP servlet element is mapped to the /HelloWorldJSP URL pattern (<url-pattern>/HelloWorldJSP</url-pattern>).

The build file must deploy the descriptor to a place where the application server can find it. It does this by packaging the HTML files, JSP files, Java servlet, and deployment descriptor in a WAR file. The next section describes the build file for this project.

The Build File for the Hello World Web Application

This project has many more components than the other subprojects. As you would expect, the Web application project build file (see Listing 20.23) is much more complex, but it builds on the foundation set by the model project--that is, the Web application project build file has the same base targets with the same meanings: init, clean, delete, prepare, mkdir, compile, package, and all.

To the base targets, the Web application project build file adds the prepare_metadata and deploy targets. The prepare_metadata target sets up the Ant filtering for the deployment descriptor. The deploy target adds the ability to deploy to both Tomcat and Resin Web application servers. The remaining details of this build file are covered in the applet and the enterprise beans sections later in this chapter.

```
<project name="webapplication" default="all" >

    <property name="outdir" value="/tmp/app" />


    <target name="init"
                         description="initialize the properties.">
      <property name="local_outdir" value="${outdir}/webapps" />
      <property name="lib" value="${outdir}/lib" />
      <property name="dist" value="${outdir}/dist" />

      <property name="build" value="${local_outdir}/webclasses" />
      <property name="meta" value="${local_outdir}/meta" />

      <property name="deploy_resin" value="/resin/webapps" />
      <property name="deploy_tomcat" value="/tomcat/webapps" />

      <property name="build_lib" value="./../lib" />
      <property name="jsdk_lib" value="/resin/lib" />
    </target>

    <target name="clean_deploy" >

        <delete file="${deploy_resin}/hello.war" />
```



```
        <delete dir="${deploy_resin}/hello" />
        <delete file="${deploy_tomcat}/hello.war" />
        <delete dir="${deploy_tomcat}/hello" />
    </target>

    <target name="clean" depends="init,clean_deploy"
                         description="clean up the output
directories.">
        <delete dir="${local_outdir}" />
        <delete file="${dist}/hello.war" />
    </target>


    <target name="prepare" depends="init"
                        description="prepare the output directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${dist}" />
        <mkdir dir="${build_lib}" />
    </target>



    <target name="compile" depends="prepare"
                        description="compile the Java source.">
        <javac srcdir="./src" destdir="${build}">
            <classpath >

                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>

                <fileset dir="${jsdk_lib}">
                    <include name="**/*.jar"/>
                </fileset>

                <fileset dir="${build_lib}">
                    <include name="**/*.jar"/>
                </fileset>

            </classpath>
        </javac>

    </target>

    <target name="prepare_meta_ejb" if="ejb">
        <filter token="Greeting.class"
               value="xptoolkit.model.GreetingShadow"/>
    </target>

    <target name="prepare_meta_noejb" unless="ejb">
        <filter token="Greeting.class"
                value="xptoolkit.model.GreetingBean"/>
    </target>

    <target name="prepare_meta"
```



```
            depends="prepare_meta_ejb, prepare_meta_noejb">
        <copy todir="${meta}" filtering="true">
            <fileset dir="./meta-data"/>
        </copy>
    </target>

    <target name="package" depends="compile">

        <mkdir dir="${meta}" />

        <antcall target="prepare_meta" />

        <war warfile="${dist}/hello.war" webxml="${meta}/web.xml">
            <!--
                Include the html and jsp files.
                Put the classes from the build into the classes
directory
                of the war.
            /-->
        <fileset dir="./HTML" />
        <fileset dir="./JSP" />
          <classes dir="${build}" />

            <!-- Include the applet. /-->
          <fileset dir="${lib}" includes="helloapplet.jar" />

       <!-- Include all of the jar files except the ejbeans and
applet. The other build files that create jars have to be run
in the correct order. This is covered later.
        /-->
        <lib dir="${lib}" >
            <exclude name="greet-ejbs.jar"/>
            <exclude name="helloapplet.jar"/>
          </lib>
        </war>

   </target>

    <target name="deploy" depends="package">
        <copy file="${dist}/hello.war" todir="${deploy_resin}" />

        <copy file="${dist}/hello.war" todir="${deploy_tomcat}" />


    </target>

    <target name="all" depends="clean,package"
                          description="perform all targets."/>

</project>
```


Listing 20.23

The Hello World Web application project build file.

The final output of the Web application project is a single WAR file. The WAR file is built (not surprisingly) by the package target. Here is the code for the package target:

```
    <target name="package" depends="compile">

        <mkdir dir="${meta}" />
        <antcall target="prepare_meta" />

        <war warfile="${dist}/hello.war" webxml="${meta}/web.xml">
            <!--
                Include the html and jsp files.
                Put the classes from the build into the classes
                directory
                of the war.
            /-->
           <fileset dir="./HTML" />
           <fileset dir="./JSP" />
           <classes dir="${build}" />

            <!-- Include the applet. /-->
            <fileset dir="${lib}" includes="helloapplet.jar" />

            <!-- Include all of the jar files except the ejbeans
                and applet.
            /-->
            <lib dir="${lib}" />
        </war>

   </target>
```

As you can see, this package target is much larger than the other two we've discussed (model and application). For now we'll offer a detailed discussion of the second and third lines of code:

```
        <mkdir dir="${meta}" />
        <antcall target="prepare_meta" />
```

These lines do some processing on the web.xml deployment descriptor file and put the file in the directory defined by the ${meta} directory (note that the meta property is set in the init target). Next, the package target calls the war task:

```
        <war warfile="${dist}/hello.war" webxml="${meta}/web.xml">
           <fileset dir="./HTML" />
           <fileset dir="./JSP" />
           <classes dir="${build}" />
           <fileset dir="${lib}" includes="helloapplet.jar" />
           <lib dir="${lib}" />
        </war>
```

The WAR file hello.war is put in the distribution directory (dist), which is specified by the war task's warfile attribute (warfile="${dist}/hello.war"). The dist directory is another common directory that is used by the main project build file later to build an enterprise archive (EAR) file; the dist property is defined in the init target. The webxml attribute of the war task defines the deployment descriptor to use; it's the one we processed at the beginning of the package target. The web.xml file is put in the WAR file's WEB-INF/ directory.

In addition, the war task body specifies three file sets. One file set includes the helloapplet.jar file (which we discuss in the section "HelloWorld.jsp Applet Delivery" later in this chapter) and all the files in the HTML and JSP directories. The war task body also specifies where to locate the classes using

```
<classes dir="${build}" />
```

This command puts the classes in the WEB-INF/classes directory.

The Web application project build file defines a slightly more complex compile target:

```
    <target name="compile" depends="prepare"
                        description="compile the Java source.">
        <javac srcdir="./src" destdir="${build}">
            <classpath >

                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>

                <fileset dir="${build_lib}">
                    <include name="**/*.jar"/>
                </fileset>

            </classpath>
        </javac>

    </target>
```

Notice that this compile target defines two file sets. One file set (<fileset dir="${build_lib}">) is used to include the classes needed for servlets (such as import javax.servlet.*). The other file set (<fileset dir="${lib}">) is used to include the Greeting interfaces and the GreetingFactory class. The only real difference from the application compile target is the inclusion of the JAR file for servlets. The build_lib property is defined in the Web application project's init target, as shown here:

```
<property name="build_lib" value="./../lib" />
```

The good thing about this approach is that if we need additional JAR files, we can put them in build_lib. The second file set (<fileset dir="${build_lib}">) grabs all the JAR files in the ./../lib directory.

The Web application project build file adds a few convenience targets geared toward Web applications. The deploy target copies the WAR file that this build file generates to the webapps directory of Tomcat and Resin. (Resin is an easy-to-use Java application server that supports JSPs, EJBs, J2EE container specification, XSL, and so on. We show it here to show that the build script would work with more than one app server since it genretates J2EE compliant war files) Without further ado, here is the deploy target:

```
<target name="deploy" depends="package">
        <copy file="${dist}/hello.war" todir="${deploy_resin}" />
        <copy file="${dist}/hello.war" todir="${deploy_tomcat}" />
</target>
```

Both Tomcat and Resin pick up the WAR files automatically, in the interest of doing no harm and cleaning up after ourselves. The Web application project build file adds an extra clean_deploy target that deletes the WAR file it deployed and cleans up the generated directory:

```
    <target name="clean_deploy" >
        <delete file="${deploy_resin}/hello.war" />
        <delete dir="${deploy_resin}/hello" />
```



```
        <delete file="${deploy_tomcat}/hello.war" />
        <delete dir="${deploy_tomcat}/hello" />
    </target>
```

"Great," you say. "But what if the application server I am deploying to is on another server that is halfway around the world?" No problem; you could use the following FTP task:

```
<ftp server="ftp.texas.austin.building7.eblox.org"
       remotedir="/deploy/resin/webapps"
       userid="kingJon"
       password="killMyLandLord"
       depends="yes"
       binary="yes"
>


    <fileset dir="${dist}">
      <include name="**/*.war"/>
    </fileset>

</ftp>
```

The key lesson here is that Ant is powerful and extensible and if you can think of a whiz-bang task that would be great, there is a good chance it may already exist. Be sure to read the Ant online docs--they are a great reference.

Building and Deploying the Web Application

This section explains how to build and deploy the Web application. The build file assumes that you have Tomcat (or Resin) installed in the root of your drive. You may need to make adjustments to the build file if you installed Tomcat in another directory or if you are using another J2EE-compliant Web application server.

To build the Web application, follow these steps:

1. Navigate to the WebApplication directory, set up the environment, and then enter the following at the command line:

```
C:\CVS\...\MVCHelloWorld\WebApplication>ant
```

You will get the following output:

```
Buildfile: build.xml

setProps:

init:

clean_deploy:
   [delete] Could not find file C:\resin\webapps\hello.war to delete.
   [delete] Could not find file C:\tomcat\webapps\hello.war to delete

clean:
   [delete] Could not find file C:\tmp\app\dist\hello.war to delete.

prepare:
```



```
    [mkdir] Created dir: C:\tmp\app\webapps\webclasses
    [mkdir] Created dir: C:\tmp\app\dist

compile:
    [javac] Compiling 1 source file to C:\tmp\app\webapps\webclasses

package:
    [mkdir] Created dir: C:\tmp\app\webapps\meta

prepare_meta_ejb:

prepare_meta_noejb:

prepare_meta:
     [copy] Copying 1 file to C:\tmp\app\webapps\meta
      [war] Building war: C:\tmp\app\dist\hello.war

all:

BUILD SUCCESSFUL
```

2. Deploy the WAR files to the application server. If you install Tomcat off the root directory, then you can run the deploy target. Otherwise, modify the appropriate deploy properties defined in the init target. To deploy the application with Ant, do the following:

```
C:\CVS\?.\MVCHelloWorld\WebApplication>ant deploy
Buildfile: build.xml

...
...

deploy:
     [copy] Copying 1 file to C:\resin\webapps
     [copy] Copying 1 file to C:\tomcat\webapps

BUILD SUCCESSFUL

Total time: 0 seconds
```

3. After we run the application, we start Tomcat, and then hit the site with our browser. We can also clean out the directories when we are ready to deploy a new version:

```
C:\CVS\...\MVCHelloWorld\WebApplication>ant clean
```

The output looks like this:

```
Buildfile: build.xml

setProps:

init:

clean_deploy:
   [delete] Deleting: C:\resin\webapps\hello.war
   [delete] Deleting: C:\tomcat\webapps\hello.war
```



```
clean:
   [delete] Deleting directory C:\tmp\app\webapps
   [delete] Deleting: C:\tmp\app\dist\hello.war

BUILD SUCCESSFUL

Total time: 0 seconds
```

Notice that we delete the WAR files and the deployment directories. This is just good housecleaning for when we do a build and deploy. In the next section, we run the Web application project.

Running the Web Application

Now that we've built and deployed the Web application project, let's run it. We start our servlet engine and then open the site in our browser--for example, http://localhost/hello/HelloWorldServlet. (Tomcat's default setup is port 8080, so you may have to adjust the URL.)

You may notice a couple of things. The application URL is defined in a directory called hello (http://localhost/hello/HelloWorldServlet). By default, Tomcat unjars our WAR file in a directory called <War file File Name>.

The HelloWorldServlet part of the application's URL is defined by a mapping in the deployment descriptor:

```
    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>
        <servlet-class>xptoolkit.web.HelloWorldServlet</servlet-class>
        <init-param>
          <param-name>Greeting.class</param-name>
          <param-value>@Greeting.class@</param-value>
        </init-param>
    </servlet>


    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/HelloWorldServlet</url-pattern>
    </servlet-mapping>
```

The servlet tag declares the servlet and gives it a name. The servlet mapping assigns HelloWorldServlet the URL pattern /HelloWorldServlet. We could change the URL pattern to /PeanutButter, and the URL http://localhost/hello/PeanutButter would work.

Actually, we mapped the 404 error to HelloWorldServlet as well, so the server sends any URL it does not recognize to HelloWorldServlet to process (for the benefit of people with fumble fingers...not a good idea for a production system).

The next section describes a simple applet project that integrates with the Web application project.

HelloWorld.jsp Applet Delivery

Now the applets JAR we built earlier becomes part of the Web application. It is put in the Web application where the browser can find it. The Web application has a JSP page, HelloApplet.jsp, which has a jsp:plugin tag that delivers the applet to the browser. The HelloApplet.jsp with the jsp:plugin action looks like this:

```
<html>
<head><title>Hello World Applet</title></head>
<body>
<jsp:plugin type="applet"
            code="xptoolkit.applet.HelloWorldApplet"
            archive="helloapplet.jar"
            height="200"
            width="200"
            align="center">
    <jsp:fallback>
    <!-- This fallback message will display if the plugin does not work.
/-->
        <p> Java is cool. Get a browser that supports the plugin. </ br>
            Or we will hunt you down and melt your computer!
        </p>
    </jsp:fallback>
</jsp:plugin>

</body>
</html>
```

This shows how the applet is delivered to the browser. How is the applet included in the Web application's WAR file in the first place? We explain in the next section.

Including an Applet in a WAR File

If you look at the Web application project build file, you will note that the war task in the package target does the following:

```
     <war warfile="${dist}/hello.war" webxml="${meta}/web.xml">
           <fileset dir="./HTML" />
           <fileset dir="./JSP" />
           <classes dir="${build}" />
           <fileset dir="${lib}" includes="helloapplet.jar" />
           <lib dir="${lib}" />
     </war>
```

The fileset directive

```
<fileset dir="${lib}" includes="helloapplet.jar" />
```

tells the war task to include only the helloapplet.jar file from the lib directory. Because this file set is not a classes- or lib-type file set, helloapplet.jar goes to the root of the WAR file. In contrast, the special lib and classes file sets put their files in WEB-INF/lib and WEB-INF/classes, respectively. The end effect is that the browser is able to get the applet.

After we build the applet, we go back and rebuild the Web application and then deploy it. We run Ant in the root of both the projects' home directories (if you get a compile error, be sure you have built the model,

because the Web application depends on it). After everything compiles and the appropriate JAR and WAR files are built, we deploy the Web application project by using the deploy target of the Web application build file.

Let's review what we've done so far. We've created a common Java library called model.jar. This model.jar file is used by a Web application and a regular Java application that is a stand-alone executable Java application in an executable JAR file. We've also created an applet that is loaded with a JSP page that has the jsp:plugin action. Once the applet loads into the browser, the applet communicates over HTTP to the Web application's HelloWorldServlet. HelloWorldServlet calls the getGreeting() method on the GreetingFactory, which is contained in the model.jar file.

Note

In the book *Java Tools for Extreme Programming*, we set up GreetingFactory so that it talks to an enterprise session bean, which in turn talks to an enterprise entity bean. The Web application build file is set up so that after we add one property file, it can talk to either the enterprise beans or to the local implementation in the model library. We achieve this magic by using Ant filters.

After we define the ejb property, the Web application project has the option of deploying/configuring whether the Web application that is deployed uses enterprise beans. Notice that the prepare_meta target, which is a dependency of the prepare target, has two dependencies: prepare_meta_ejb and prepare_meta_noejb, as shown here:

```
<target name="prepare_meta"
        depends="prepare_meta_ejb, prepare_meta_noejb">
    <copy todir="${meta}" filtering="true">
        <fileset dir="./meta-data"/>
    </copy>
</target>
```

The prepare_meta_ejb target is executed only if the ejb property is set as follows:

```
<target name="prepare_meta_ejb" if="ejb">
    <filter token="Greeting.class"
            value="xptoolkit.model.GreetingShadow"/>
</target>
```

If the ejb property is set, then the target creates a filter token called Greeting.class. Here, we set the value of Greeting.class to GreetingShadow. Conversely, the prepare_meta_ejb target is executed only if the ejb property is not set, as follows:

```
<target name="prepare_meta_noejb" unless="ejb">
    <filter token="Greeting.class"
            value="xptoolkit.model.GreetingBean"/>
</target>
```

Here, we set GreetingBean as Greeting.class. But how is this used by the application? You may recall that HelloWorldServlet uses the servlet parameter Greeting.class to set the system property Greeting.class (which is used by GreetingFactory to create an instance of Greeting). We put an Ant filter key in the Web application project deployment descriptor, as follows:

```
<servlet>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>xptoolkit.web.HelloWorldServlet</servlet-class>
    <init-param>
      <param-name>Greeting.class</param-name>
      <param-value>@Greeting.class@</param-value>
    </init-param>
```



```
</servlet>
```

If we copy this file using the filter command after the filter token Greeting.class has been set, then *@Greeting.class@* is replaced with the value of our token Greeting.class, which is set to xptoolkit.model.GreetingShadow in the prepare_meta_ejb target and to xptoolkit.model.GreetingBean in the prepare_meta_noejb target. Notice that the prepare_meta target copies the deployment descriptor with filtering turned on, as follows:

```
<copy todir="${meta}" filtering="true">
    <fileset dir="./meta-data"/>
</copy>
```

Now that is a cool trick. We hope this step-by-step Ant example will help you visualize how you go about using Ant.

Summary

In this chapter, we took a very complex project with a few components and subsystems, albeit a simple implementation, and built it in an orchestrated fashion. We showed how to create a Java application in an executable JAR and a Java applet in a JAR. We also demonstrated how to package a set of classes that is shared by more than one application in a JAR.

This chapter covered the basics of using Ant and the concepts of build files, projects, targets, conditional targets, tasks, file sets, filters, nested build files, and properties. Our discussion included the basics styles and naming conventions for Ant build files.

The chapter also explained the importance of Ant in its relationship to continuous integration.

For More Information

For additional information please contact Training@triveratech.com or call us at 609-953-1515.

Trivera Technologies
Global Developer Education Services

135 Meeshaway Trail
Medford Lakes NJ 08055
609-953-1515 phone
609-953-6886 fax

www.triveratech.com
Info@triveratech.com

Educate. Collaborate. Accelerate!

