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

