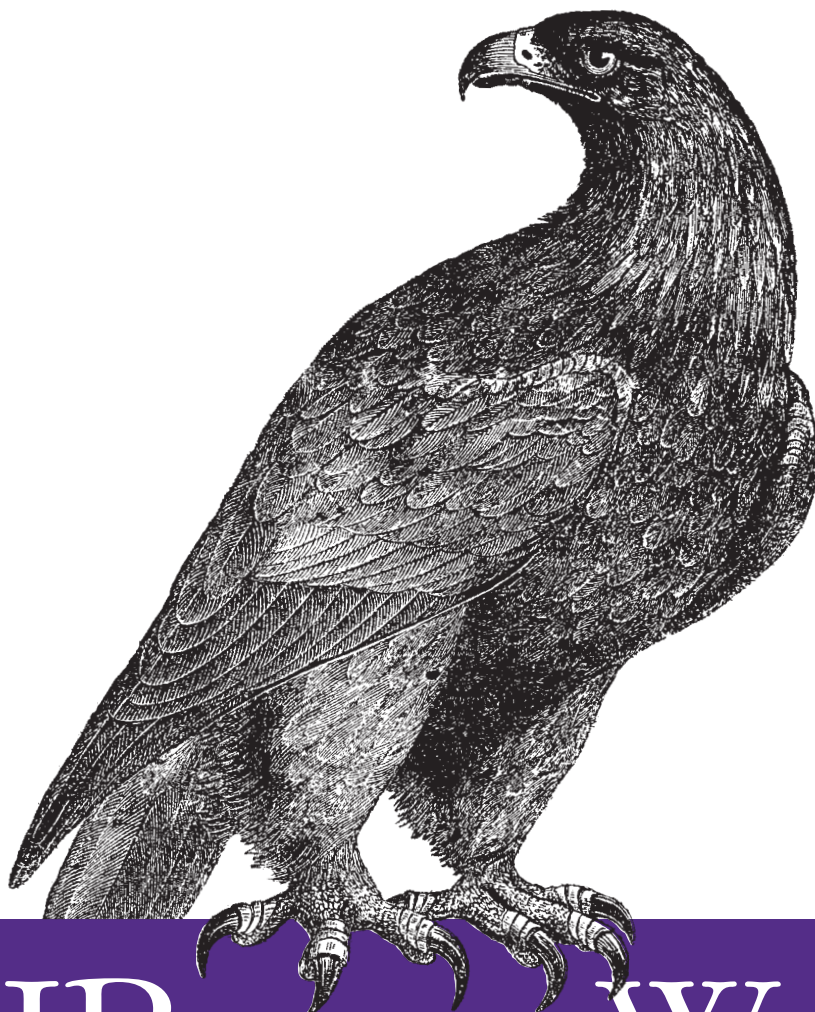


Building and Deploying J2EE in the Real World



JBoss *at* Work

A Practical Guide

O'REILLY®

Tom Marris & Scott Davis

Building and Deploying an EAR

In the last chapter we introduced you to web applications, but the web tier is just one part of the J2EE spectrum. In this chapter, we'll expand the JAW application from a simple WAR file into a full-fledged EAR.

We'll explore the different parts of an EAR file. We'll build a Common JAR containing classes that can be shared across all tiers of the application. Finally, we'll play with various Ant and XDoclet tasks to create our EAR and dynamically generate the deployment descriptors JBoss needs.

WARs Versus EARs

The WAR file is a convenient way to bundle up all pieces of a web application. All servlet containers know how to deploy a WAR file—they expand the bundle, look for the *WEB-INF* directory, and read the *web.xml* found there for further deployment instructions.

The EAR file provides the same type of functionality for a full-fledged J2EE application. JBoss expands the EAR, finds the required deployment descriptors, and proceeds from there.

An EAR is like a carton of eggs—it keeps everything organized. While the carton doesn't add any direct value to your omelet, it makes getting the eggs home from the store so easy that you wouldn't think about transporting eggs any other way.

Each egg in your EAR carton is a specific piece of the J2EE puzzle. These eggs (or JARs) come in three basic varieties called “modules”:

Web module

A WAR file containing presentation tier components

EJB module

An EJB JAR file containing the middle-tier components (EJBs, MDBs, etc.)

Java module

A regular JAR file containing classes and libraries that are shared across the entire application. An application client JAR and a common JAR are two examples of Java modules.

An EAR can contain at least one of any of these modules. By the same token, any of them can be safely omitted if they aren't needed. Figure 3-1 shows the structure of an EAR file.

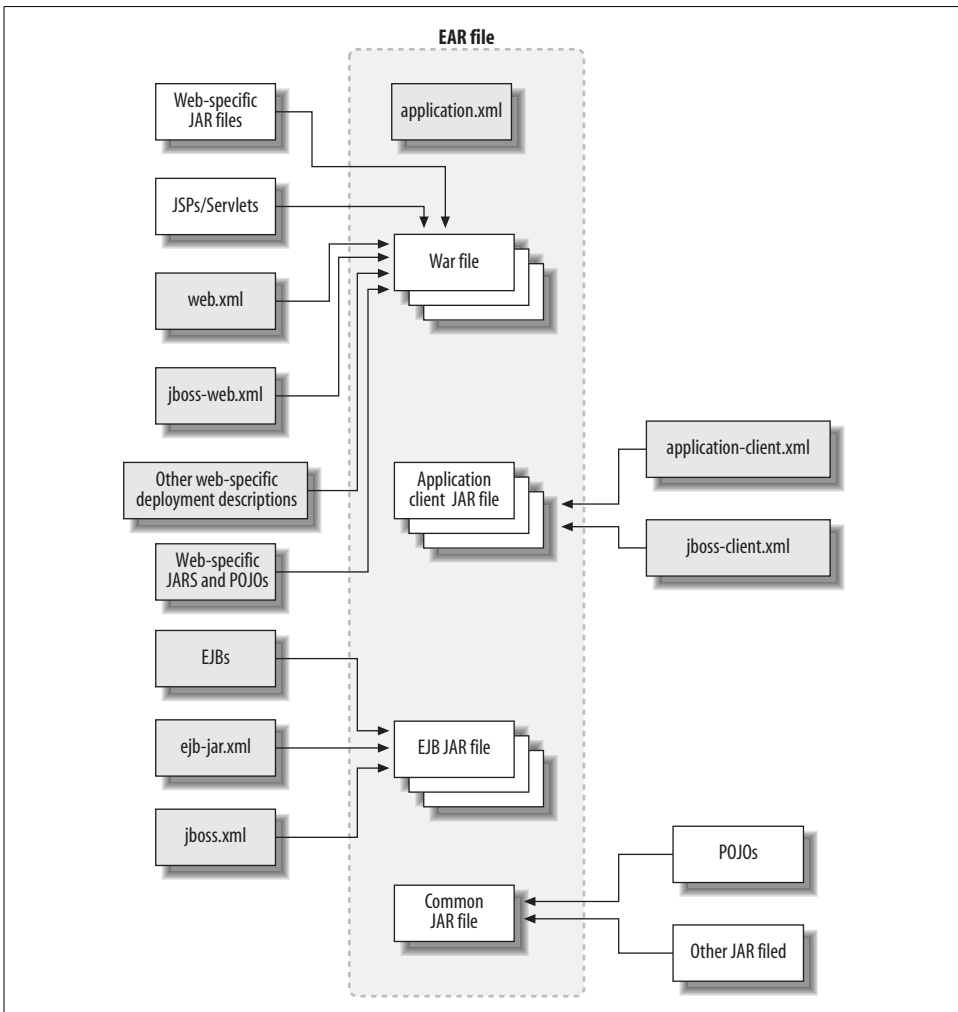


Figure 3-1. EAR file structure

Application.xml

Just as a WAR file contains a *web.xml* deployment descriptor, an EAR file contains a file named *application.xml*. It is essentially a packing list, telling the J2EE server exactly what files the EAR contains and where you can find the files relative to the root of the EAR. The EAR file's *META-INF* directory stores *application.xml*.

Example 3-1 shows the JAW Motors *application.xml* file.

Example 3-1. application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd"
  version="1.4">
  <display-name>JBossAtWorkEAR</display-name>

  <module>
    <web>
      <web-uri>webapp.war</web-uri>
      <context-root>jaw</context-root>
    </web>
  </module>

  <module>
    <java>common.jar</java>
  </module>
</application>
```

The elements in *application.xml* should be pretty self-explanatory. We are telling the application server the name of each JAR and what function it serves.

Notice that Web modules allow you to specify one other value—the `<context-root>`. Recall from the previous chapter that the context root is your web site's URL. If you deploy a simple WAR file, the name of the WAR will be used as the URL. When your WAR file is deployed inside an EAR, this element allows you to override the physical name of the WAR and use whatever URL you'd like.

Although not shown in this example, `<security-role>` is another important element in *application.xml*. The `<security-role>` element describes (what else?) the security roles used throughout a J2EE application for both web and EJB components. Defining security roles in *application.xml* provides a single place to set up J2EE declarative security without duplicating it in *web.xml* and *ejb-jar.xml*. The Security chapter describes `<security-role>` in greater detail.

Common JAR

In the previous chapter, we created a `CarBean` POJO to hold the various attributes of a car. We stored it in the WAR file because, well, you didn't have any other choice at that time. We should now reconsider the storage location for the `CarBean` to maximize its reuse.

By the end of this book, we will pull cars out of a database in the persistence tier, and hand them to objects in the business tier, which ultimately will pass them up to the presentation tier. An object that is shared across tiers is a perfect candidate for the Common JAR.

In addition to custom domain objects, the Common JAR is a great location to store common libraries such as Log4J or JDOM. While both WARs and EJB JARs have *lib* directories, they are best used for tier-specific libraries. For example, the JSTL JARs belong in the WAR—they have no other purpose than to support the creation of web pages. On the other hand, logging is something that happens throughout the codebase—it really belongs in a common JAR.

Let's factor our `CarBean` out of the WAR and into the Common JAR. In addition to moving directories, we're going to rename it to better describe its purpose in the application.

The suffix "Bean" is a bit overloaded: it includes JavaBeans, Enterprise Java Beans, Session Beans, Message-Driven Beans, JMX MBeans, and the list goes on. The design pattern that best describes the `CarBean`'s function is a Data Transfer Object (DTO), so when we move the bean, we'll also rename it `CarDTO`. The source code will remain the same, but the name will give us a better idea about the true purpose of the class.

Exploring the New Directory Structure

The previous chapter included only the *webapp* directory. If you change to the *ch03/03a-ear* directory, you'll see that we've expanded to a *webapp* directory and a *common* directory.

We also expanded from one to three *build.xml* files. Each subdirectory has its own *build.xml*, and the master build file lives in the top-level directory. The goal is to keep each portion of the application as autonomous as possible. Granted, most of the application will depend on the common sub-project, but by providing individual Ant scripts you have the opportunity to build each portion of the project separately.

The common sub-project

Take a moment to explore the *common* sub-project. It contains a single class—`CarDTO`. We have created a new package structure to store all DTOs—`com.jbossatwork.dto`.

You can build this sub-project by typing `ant` in the common directory. It compiles the `CarDTO` class and bundles it up into a JAR file. After you've built the sub-project, change to the *build/distribution* directory and type `jar tvf common.jar` to verify that the `CarDTO` class is indeed stored in *common.jar*.

The webapp sub-project

The only change to the webapp sub-project from the previous chapter is the removal of the `CarDTO` class. To accommodate this change, we now must import the `com.jbossatwork.dto` package at the top of `ControllerServlet`.

We also have to change our *build.xml* script to include the *common.jar* in our classpath. Notice that the definition of `common.jar.dir` uses a relative path to step up one level from the `basedir` of the webapp sub-project and down into the common sub-project's output directory in Example 3-2.

Example 3-2. webapp build.xml

```
<property name="lib.dir" value="lib"/>
<property name="compile.lib.dir" value="compile-lib"/>
<property name="common.jar.dir" value="../common/build/distribution"/>

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

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

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

Introducing cross project dependencies like this is not without risk. If you try to build the webapp sub-project before the common project is built, the build will fail because the dependent JAR won't be present. Of course, most other sub-projects will have dependencies on the common project *by design*—the common project is meant to hold objects that will be shared across all tiers.

Having the webapp sub-project rebuild the common project every time could be an unnecessary step if the common project changes infrequently. If we do not couple the webapp build process to the common build process, webapp developers can informally baseline the common project by only rebuilding it when they make a conscious effort to do so.

Type `ant` in the `webapp` top-level directory to build the WAR file. Change to `build/distribution` and type `jar tvf webapp.war` to see the contents. Verify that `CarDTO` is no longer present in the WAR.

The master build

To ensure that each project gets built—in the proper order—we created a master build file. This file doesn't actually compile any code; rather, it calls the appropriate build files of each sub-project in the proper order and then EARs up the results. Any failure of any sub-build will fail the master build, so we can rest assured that a successful master build is predicated on each individual build completing successfully.

To invoke a build script in another directory, we use the `<ant>` task. Here is what our compile target looks like in the master `build.xml` file in Example 3-3.

Example 3-3. master build.xml

```
<target name="compile" description="Compiles all Java code">
  <echo message="##### Building common #####" />
  <ant dir="${common.dir}" target="all" >
    <property name="jar.name" value="${common.jar.name}"/>
  </ant>

  <echo message="##### Building webapp #####" />
  <ant dir="${webapp.dir}" target="all" >
    <property name="war.name" value="${webapp.war.name}"/>
    <property name="common.jar.dir" value="${basedir}/${common.jar.dir}"/>
  </ant>
</target>
```

Notice that we also can override properties in the child build process. In both instances, we override the name of the JAR or WAR file specified in the child build. In the case of the `webapp` build, we can no longer use the same relative path: your base directory is different now, so trying to move up a level and over doesn't work. We pass the `webapp` build file a fully qualified path to the common output directory.

Ant EAR task

The final step in the process is to EAR up the results of the `webapp` and `common` builds. Just as there is a WAR task, Ant also provides us with an EAR task, as in Example 3-4.

Example 3-4. master build.xml

```
<target name="ear" depends="compile">
  <ear destFile="${distribution.dir}/${ear.name}"
      appxml="${meta-inf.dir}/application.xml" >

    <!-- files to be included in / -->
    <fileset dir="${webapp.war.dir}" />
```

Example 3-4. *master build.xml* (continued)

```
        <fileset dir="${common.jar.dir}" />
    </ear>
</target>
```

Notice that the EAR task requires us to pass it a well-formed *application.xml* file. Example 3-5 shows what a simple one looks like.

Example 3-5. *application.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd"
  version="1.4">
  <display-name>JBossAtWorkEAR</display-name>

  <module>
    <web>
      <web-uri>webapp.war</web-uri>
      <context-root>jaw</context-root>
    </web>
  </module>

  <module>
    <java>common.jar</java>
  </module>

</application>
```

To verify the results of the master build, change to the *build/distribution* directory and type `jar tvf jaw.ear`. You should see *webapp.war*, *common.jar*, and *application.xml*. We are now ready to deploy the EAR file.

Deploying the EAR

Let's make sure that the WAR files from the previous chapters don't end up conflicting with our EAR file. Delete *jaw.war* from the *deploy* directory before moving on. Now we're ready to drop in our newly created EAR file.

Deploying an EAR by hand is no different than deploying a WAR by hand. Copy *jaw.ear* to `$JBOSS_HOME/server/default/deploy`. Your JBoss console should let you know that it was deployed successfully, as in Example 3-6.

Example 3-6. JBoss console output showing a successful EAR deployment

```
22:37:55,659 INFO [EARDeployer] Init J2EE application:
  file:/Library/jboss-4.0.1/server/default/deploy/jaw.ear
22:37:55,853 INFO [TomcatDeployer] deploy, ctxPath=/jaw,
  warUrl=file:/Library/jboss-4.0.1/server/default/tmp/deploy/
```

Example 3-6. JBoss console output showing a successful EAR deployment (continued)

```
tmp25111jaw-ear.ear-contents/webapp.war/  
22:37:56,159 INFO [EARDeployer] Started J2EE application:  
file:/Library/jboss-4.0.1/server/default/deploy/jaw-ear.ear
```

Automated Deployments Using Ant

Ant allows you to do far more than simply compile and bundle up your Java application. It has tasks that let you create and delete directories and copy files to a local sub-directory or a remote server by using ftp or scp. Using tasks like exec, rexec, or sshexec, you can even remotely start and stop JBoss.

It's not hard to imagine completely automating the deployment process with an Ant script. But just because you *can* do something doesn't always mean you *should*.

Automating deployment to a test server certainly will help speed up your development iterations. But deployments to a production server should be taken a bit more seriously. Upgrading a production application is something that should be done deliberately, and in our opinion, should be done by hand.

If you provide an Ant task to deploy your application to a production server, you almost certainly guarantee that you will invoke it accidentally at the most inopportune time.

Since we point at a test server for this book, we've provided a convenient couple of Ant targets to deploy your EAR. To do a hot deploy, make sure that you have the \$JBOSS_HOME environment variable set, and then simply type ant deploy. The Ant task will copy the EAR file to the correct location for you.

If you want to do a cold deploy, shut down JBoss, type Ant cleandeploy, and then start JBoss back up again. The cleandeploy target will delete the existing EAR file and several temporary directories. Running cleandeploy against a running JBoss instance will cause bad things to happen, so make sure that JBoss is not running before invoking it.

Visit <http://localhost:8080/jaw> to confirm that the application was indeed deployed and still works as expected. Yes, this application doesn't look or behave any differently than the one in the Web chapter. But we added hundreds of lines of new code—isn't that the true measure of a successful J2EE project? (Only kidding....)

In all seriousness, we haven't added any new functionality that the user would notice, but we have set the stage for easy future growth and maximum flexibility. Knowing that the other tiers are coming up soon, these changes will allow you to incorporate the new technology with minimal effort.

Adding a DAO

In that spirit, let's add another component that will pay dividends in future flexibility. Right now, your servlet is creating the car list each time a request comes in. This really isn't optimal. Servlets should deal with the mechanics of the HTTP request/response cycle. They shouldn't perform persistence tier tasks.

We aren't quite ready to install a database (that happens in the next chapter), but we can lay the groundwork by creating a Data Access Object (DAO). A DAO is a layer of abstraction—it hides the actual persistence specifics behind a common interface.

The DAO we create in this chapter still stores the DTO objects in a simple `ArrayList`. In the next chapter, the DAO will pull car data from a database that uses JDBC. In the chapter after that, it will use Hibernate (an Object/Relational Mapper) to do the same thing. By getting the DAO in place now, however, we'll be able to make these implementation changes without affecting presentation-tier code. Loose coupling and high cohesion comes to the rescue again.

The `CarDAO` provides a `findAll()` method that returns a `List` of `CarDTOs`. The source code in Example 3-7 can be found in the `common` directory in `ch03b-dao`.

Example 3-7. CarDAO.java

```
package com.jbossatwork.dao;

import java.util.*;
import com.jbossatwork.dto.CarDTO;

public class CarDAO
{
    private List carList;

    public CarDAO()
    {
        carList = new ArrayList();

        carList.add(new CarDTO("Toyota", "Camry", "2005"));
        carList.add(new CarDTO("Toyota", "Corolla", "1999"));
        carList.add(new CarDTO("Ford", "Explorer", "2005"));
    }

    public List findAll()
    {
        return carList;
    }
}
```

The corresponding change in the `ControllerServlet` calls the newly created DAO in Example 3-8.

Example 3-8. ControllerServlet.java

```
// perform action
    if(VIEW_CAR_LIST_ACTION.equals(actionName))
    {
        CarDAO carDAO = new CarDAO();
        request.setAttribute("carList", carDAO.findAll());

        destinationPage = "/carList.jsp";
    }
```

Not only does this change simplify the code in the servlet, it *feels* more correct as well. The servlet concerns itself solely with web mechanics and delegates the database tasks to a dedicated class. Another pleasant side effect of this is reuse—your data access code can now be called outside of the web tier. If a business tier object needs access to this data, it can make the same call that we make.

Build and deploy the code to verify that we haven't broken your application with this change.

Using XDoclet

The last thing we'll do in this chapter is one more tiny bit of automation.

One of the most important parts of the WAR file is the deployment descriptor—*web.xml*. It lists each servlet and tells the servlet container how to deploy them. By maintaining this file by hand, you almost certainly guarantee that you will forget to update it when you add new components to your application. By using XDoclet in your build process, we can generate it automatically by using nothing but your source code and some well-placed comments.

XDoclet is a collection of custom Ant tasks that generate code during the build process. Sometimes XDoclet can generate the necessary code or deployment descriptor using just the source code. At other times, you'll need to add custom comments to nudge XDoclet along in the right direction.

We have only one servlet right now—ControllerServlet. Example 3-9 shows what your *web.xml* must contain to deploy this servlet correctly.

Example 3-9. web.xml

```
<!-- servlet definition -->
<servlet>
    <servlet-name>Controller</servlet-name>
    <servlet-class>com.jbossatwork.ControllerServlet</servlet-class>
</servlet>

<!-- servlet mapping -->
<servlet-mapping>
    <servlet-name>Controller</servlet-name>
```

Example 3-9. *web.xml* (continued)

```
<url-pattern>/controller/*</url-pattern>
</servlet-mapping>
```

Look in *ch03c-xdoclet/src* for the new `ControllerServlet` source code in Example 3-10. Notice the XDoclet comments we've added.

Example 3-10. *ControllerServlet.java*

```
/**
 * @web.servlet
 *   name="Controller"
 *
 * @web.servlet-mapping
 *   url-pattern="/controller/*"
 */
public class ControllerServlet extends HttpServlet
```

See how they correspond to the *web.xml* elements? Now let's look at *build.xml* to see the newly added XDoclet Ant tasks. We first need to create a couple of new variables, as in Example 3-11.

Example 3-11. *Defining XDoclet variables in build.xml*

```
<property name="xdoclet.lib.dir" value="${env.XDOCLET_HOME} /lib"/>
<property name="gen.source.dir" value="${build.dir}/gensrc"/>
```

`xdoclet.lib.dir` points to the XDoclet jars. If we have a defined `$XDOCLET_HOME` environment variable, *build.xml* should be automatically pointed in the right direction. (Recall that we installed XDoclet in Chapter 1.)

The second variable defines a new location for our dynamically generated code. By keeping our compiled code and generated source code in the same location (`build.dir`), we can easily delete and recreate it each time we run the Ant tasks. It also gently reminds us that we shouldn't store generated artifacts in source control.

Next, we need to set up a new classpath that includes the XDoclet libraries. We keep XDoclet stuff separate from our regular build process, as in Example 3-12. This ensures that we don't inadvertently create dependencies in our deployed code.

Example 3-12. *XDoclet Library Path in build.xml*

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

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

Next we create the generate-web target in Example 3-13.

Example 3-13. generate-web target in build.xml

```
<!-- ===== -->
  <target name="generate-web" description="Generate web.xml">
    <taskdef name="webdoclet"
            classname="xdoclet.modules.web.WebDocletTask"
            classpathref="xdoclet.lib.path" />

    <mkdir dir="{gen.source.dir}" />

    <webdoclet destdir="{gen.source.dir}">
      <fileset dir="{source.dir}">
        <include name="**/*Servlet.java" />
      </fileset>

      <deploymentdescriptor destdir="{gen.source.dir}"
                          distributable="false"
                          servletspec="2.4" />

    </webdoclet>
  </target>
```

Let's step through this:

- To start, we need to define the WebDoclet task. We don't need to use the `<taskdef>` tag for core Ant tasks, but for third-party tasks, this directive shows Ant where to find the implementation code.
- Next, we create the destination directory for our generated code.
- Finally, we call the `<webdoclet>` task. As you might guess, it generates web-tier code. In later chapters, we'll see other XDoclet tasks such as `<hibernatedoclet>` and `<ejbdoclet>`.
- The `<filesetdir>` tags tells the `<webdoclet>` task to look for the source code.
- The `<deploymentdescriptor>` tag tells `<webdoclet>` to generate *web.xml*.
- We use the `distributable="false"` attribute since we are not in a clustered JBoss environment. It adjusts *web.xml* appropriately.

To make sure that our new target gets called at the appropriate time, we add it as a dependency to the war target in Example 3-14.

Example 3-14. war target in build.xml

```
<!-- ===== -->
  <target name="war" depends="generate-web,compile"
        description="Packages the Web files into a WAR file">
    <mkdir dir="{distribution.dir}" />

    <war destFile="{distribution.dir}/{war.name}"
        webxml="{gen.source.dir}/web.xml">
```

Example 3-14. *war* target in *build.xml* (continued)

```
<!-- files to be included in / -->
<fileset dir="${web.dir}" exclude="WEB-INF/web.xml" />

<!-- files to be included in /WEB-INF/classes -->
<classes dir="${classes.dir}" />

<!-- files to be included in /WEB-INF/lib -->
<lib dir="${lib.dir}" />

<!-- files to be included in /WEB-INF -->
<webinf dir="${web.inf.dir}" excludes="web.xml" />
</war>
</target>
```

Notice that we changed the *webxml* attribute in the *war* task to point to the newly generated *web.xml* file.

Type *ant* in the *webapp* directory to rebuilt the subproject. You should see console output that looks like this:

```
Buildfile: build.xml

clean:
  [delete] Deleting directory /Users/sdavis/Desktop/jbossatwork/ch03/
           03c-webdoclet/webapp/build

generate-web:
  [mkdir] Created dir: /Users/sdavis/Desktop/jbossatwork/ch03/
           03c-webdoclet/webapp/build/gensrc
  [webdoclet] (XDocletMain.start          47 )
              Running <deploymentdescriptor/>
  [webdoclet] Generating web.xml.

compile:
  [mkdir] Created dir: /Users/sdavis/Desktop/jbossatwork/ch03/
           03c-webdoclet/webapp/build/classes
  [javac] Compiling 1 source file to /Users/sdavis/Desktop/jbossatwork/ch03/
           03c-webdoclet/webapp/build/classes

war:
  [mkdir] Created dir: /Users/sdavis/Desktop/jbossatwork/ch03/
           03c-webdoclet/webapp/build/distribution
  [war] Building war: /Users/sdavis/Desktop/jbossatwork/ch03/
           03c-webdoclet/webapp/build/distribution/jaw.war

all:

BUILD SUCCESSFUL
Total time: 6 seconds
```

Look at the newly generated *web.xml* in *build/gensrc*. In addition to the required servlet elements, the comments show the various other parts of *web.xml* that can be

generated as required by our application. Clearly, we have just scratched the surface of what the `<webdoclet>` task brings to the table.

Deploy *jaw.ear* one last time to make sure our application still behaves as expected.

Looking Ahead...

We've covered a lot of ground in these first few chapters. We now have a single EAR file that encapsulates many moving parts of our J2EE application. We have automated the build process where it is appropriate, and set the stage for future growth.

In the next several chapters, we'll leave the web tier and move on to the persistence tier. We've stored our DTOs in an `ArrayList` for long enough—let's tackle saving them in a true database.