BEA WebLogic Server™

Deploying an Exploded J2EE Application

BEA WebLogic Server Version 6.1
Document Date: June 24, 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Introduction to BEA WebLogic Server

Part Number	Document Date	Software Version
N/A	June 24, 2002	BEA WebLogic Server Version 6.1SP3

Contents

1 Deploying an Exploded J2EE Application

If you have ever deployed a J2EE application, you know that the process varies among application servers. The good news is that on WebLogic Server™, the process is automated with tools for professional developers.

This tutorial chapter describes how to use the built-in tools to generate deployment descriptors, package a sample J2EE application, and deploy it on the server, using the sample banking application that you can download from the tutorials website. It aims to teach you enough that you can translate this process to your own applications.

As a quick reference, the steps to deploying are:

Choosing Exploded or Archive Form

WebLogic Server is sophisticated enough that you can deploy an application in either exploded or archive form. An application in exploded form has a full directory structure, with no JAR or other archive files. An application in archive form is a standard J2EE application EAR file that packages WAR and EJB JAR modules.

The server handles exploded and archived applications the same way, so it is really your choice which form to use. Deploy your application in exploded form if:

- You are still testing your application.
- You want to be able to modify static files (such as JSP or HTML files) while the application is running on the server without redeploying, using the `WebAppComponentRefreshTool`.
- You use WebLogic Server as an application server and another server that has a different document root (such as the Apache HTTP Server) as a Web server.
- You want to be able to change and recompile enterprise beans and redeploy them using the hot deploy feature.

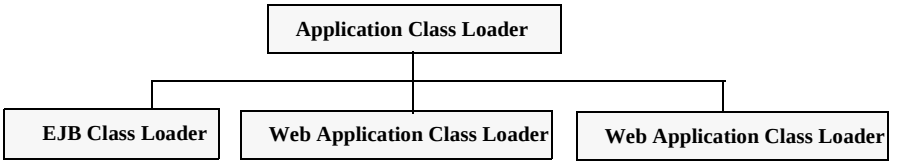
However, if your application is final and you want to package it into an EAR file for ease of deployment or to work as specified in the *J2EE Specification*, WebLogic Server fully supports you. The advantage of deploying an archived application are:

- Version control is much easier.
- It is also much easier to distribute the application to customers or other company sites.

If you want to package your application, create a single application EAR file and deploy it. The reason for this has to do with class loaders. In any application that has JSP files or servlets that access enterprise beans, the JSP files and servlets need to be able to load the classes they need to work with the beans (that means the interface and stub classes).

If you deploy a web application WAR file and an EJB JAR file separately, the server loads them in sibling class loaders, in which they don't have access to each other. This means that you need to include the EJB interface and stub classes in *both* archives.

Figure 1-1 The Class Loader Hierarchy for Your Application



But each time you deploy an application EAR file or an exploded application, the server creates a class loader specifically for that application. The application class loader is a *parent* class loader.

All EJB modules in your application are loaded by a single class loader that is a *child* of the application class loader. (Note that the terms *parent* and *child* do not imply a superclass/subclass relationship between the class loaders).

Each web module in your application is loaded by its own child class loader. Classes loaded by a child class loader can see any classes loaded by a parent class loader. This means that the web modules can see the interface files in the EJB module. What's more, the fact that each web module is loaded by its own class loader means that the web modules can contain files with the same name, such as `index.jsp`.

Whether you deploy an exploded application or an EAR file, the web application and enterprise bean modules are part of one application, with aligned class loading, better performance due to faster RMI calls and use of EJB local interfaces, and isolation from other applications.

To learn how to deploy an exploded application, stay right here, in this tutorial.

To learn how to package an application into an EAR file, read the chapter J2EE Packaging and Deployment from the book *Java Server Programming, J2EE 1.3 Edition* (Wrox Press, 2001).

Step 1: Set Up the Application Directory

The application directory should be your own directory, outside of the WebLogic Server directory, where you develop, build, and test before deploying on a production server. The application directory needs both a development area and a deployment area.

You should set up your application directory to look like Figure 1-2. This uses the standard J2EE application directory structure.

Figure 1-2 Application Directory

```
banking\            .. or use your application directory name ..
  META-INF\
  dev\
    web\            .. jsp and html files, servlets, images, related files ..
    ejb\            .. enterprise bean source files ..
  deploy\
    web\            .. jsp, html, image files ..
      WEB-INF\
        classes\    .. compiled servlets ..
        lib\        .. third party libraries, tag libraries ..
    ejb\            .. compiled beans ..
      META-INF\
```

This structure shows you several things:

- Use the same directory structure and deployment descriptors, whether you deploy in exploded or archive form.
- Develop your application and store its source files in a directory outside the WebLogic Server directory.
- Use separate directories for the Web application and enterprise bean modules. This makes it easier to generate the descriptors using the tools and prepares the directory structure for creating the archive files.
- You can build your application and store the compiled files in the application directory, deploying them from there if you are testing on a server stored on the same computer.

Best Practice. Build the WAR, EJB JAR, and EAR packages in separate staging directories. This gives you separate packages that you can work with if you need to update and redeploy some classes.

The Deployment Descriptors

Some deployment descriptors you need are J2EE descriptors, others are specific to the WebLogic Server. These are the descriptors you need for the most common parts of a J2EE application:

Component	Descriptors	Type
Web application (or WAR file)	`web.xml` `weblogic.xml`	J2EE WebLogic

EJB component (or EJB JAR file)	`ejb-jar.xml`
	`weblogic-ejb-jar.xml`
	`weblogic-cmp-rdbms-jar.xml`
		J2EE
		WebLogic
		WebLogic

J2EE application (or EAR file)	`application.xml`	J2EE

If your application has a resource adapter or if you use a standalone Java client application, you need additional descriptors. Those descriptors, although beyond the scope of this tutorial, are `ra.xml` and `weblogic-ra.xml` for the resource adapter and `application-client.xml` and `client-application-runtime.xml` for the client application (get more info).

Compiling Your Application

After setting up the directories, you need to compile the enterprise beans and servlets to the right locations. The enterprise bean classes go into `deploy\ejb`, along with their helper, remote stub, and skeleton classes. (This is important for setting up the staging directory. Later, when you create an EJB JAR file, you can compile just the enterprise bean and helper classes and then use the `ejbc` utility to add the RMI stub and skeleton classes.)

For your Web application, you need to:

- *Compile* your servlets to *myapp*`\deploy\web\WEB-INF\classes` (don't simply move them there).
- *Move* the JSP files, HTML files, and images to the top level of `deploy\web`.

Then, compile your enterprise beans to *myapp*`\deploy\ejb`. It's very easy to compile a number of classes at once with a build script, storing the build script in the development directory, but compiling the classes to the deployment directory.

Listing 1-1 Compiling with a Build Script on Windows

```
@REM Compile ejb, servlet, rmi classes

javac -d ..\deploy\ejb ejb\Account.java ejb\AccountBean.java
  ejb\AccountHome.java ejb\RMILogger.java ejb\RMILoggerImpl.java
 ejb\BankConstants.java ejb\ProcessingErrorException.java
  ejb\Client.java

javac -d ..\deploy\web\WEB-INF\classes web\BankAppServlet.java
```

The `-d` option to `javac` specifies where to store the compiled classes.

Now look at the order in which the classes are compiled. The enterprise bean classes are compiled before the servlet class, and the servlet needs them for its own compilation. So, you need to add the location of the compiled enterprise beans to your `CLASSPATH` so that `BankAppServlet` can locate them while it is being compiled.

First, run the `setEnv` script to set the standard `CLASSPATH` for compiling applications for WebLogic Server:

```
cd WL_HOME\config\mydomain
setEnv
```

Then, add the relative path from the servlet *source* file (at `banking\dev\web`) to the *compiled* enterprise bean classes the servlet needs (at `banking\ejb\deploy`):

```
set CLASSPATH=..\deploy\ejb;%CLASSPATH%
```

After adjusting the `CLASSPATH`, you can run the build script to compile the application classes to the deployment directory.

Hands On: Set Up the Application Directory

Step 2: Generate the Descriptors

The next step is to generate the deployment descriptors. You may know how to write the J2EE descriptors, but not the WebLogic descriptors.

The `DDInit` tools generate all the descriptors for an application component, whether J2EE or WebLogic descriptors. For example, for the web application component, the tool generates `web.xml` and `weblogic.xml`. So even if you are very familiar with how to write the J2EE descriptors, you should probably either generate them all or write them all.

The tools that generate descriptors are:

- `war.DDInit`, for the `web.xml` and `weblogic.xml` deployment descriptors
- `ejb.DDInit`, for `ejb-jar.xml`, `weblogic-ejb-jar.xml`, and `weblogic-cmp-rdbms-jar.xml` for EJB 1.1 beans
- `ejb20.DDInit`, for `ejb-jar.xml`, `weblogic-ejb-jar.xml`, and `weblogic-cmp-rdbms-jar.xml` for EJB 2.0 beans

In this release (WebLogic Server 6.1), you cannot generate `application.xml`, the top-level descriptor for the J2EE application. So, you have two choices:

- Customize the `application.xml` descriptor that we provide (download `banking.zip` from Samples and Tutorials)
- Write `application.xml` yourself (more info).

To use the `DDInit` tools, just open a command window, move one directory level above the deployment directory (in our example, that's `banking\deploy`, above `ejb` and `web`). Then, enter either of these commands:

```
java weblogic.ant.taskdefs.war.DDInit directoryName
java weblogic.ant.taskdefs.ejb.DDInit directoryName
```

The generated descriptors will be stored in the correct `WEB-INF` and `META-INF` directories.

When to Regenerate the Descriptors

The `DDInit` tools *always* overwrite any existing descriptors. You should regenerate and re-edit descriptors whenever you add or change an enterprise bean, JSP file, or servlet in your application.

Otherwise, you should edit the existing descriptors if you change tables or columns in your database schema that your application uses. This would affect any application that uses those tables or columns. You should also edit descriptors to change values that affect how the server runs your application, such as HTTP session or JSP compilation parameters.

Hands On: Generate the Descriptors

Step 3: Edit the Web Descriptors

Once you generate the descriptors, you need to edit them with a validating XML editor (for example, the BEA XML Editor available at **http://dev2dev.bea.com/resourcelibrary/utilitiestools/index.jsp**). The first descriptors to edit are **`web.xml`** and **`weblogic.xml`**, in the `web\WEB-INF` directory.

The advantage of using an XML editor is that it tells you which XML elements are valid at a given point in the file, and then validates the file, telling you where you have incorrect elements or syntax. This way, you can make sure your descriptors are correct *before* you deploy your application.

Figure 1-3 Inserting a Valid XML Element with the BEA XML Editor

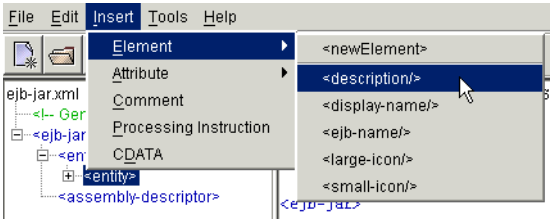
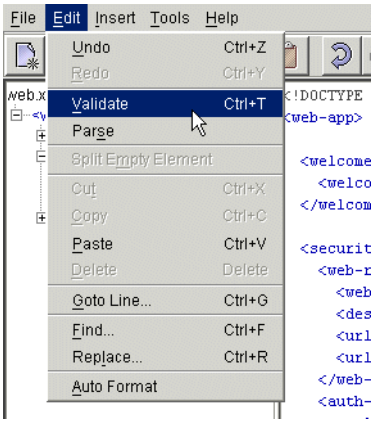


Figure 1-4 Validating an XML File



Best Practice. Edit deployment descriptors with a tool that parses and validates the XML. This catches XML syntax errors early and helps you deploy the application successfully.

Edit web.xml

Now the task is to look at and edit the generated `web.xml` and `weblogic.xml`. The `web.xml` schema is defined in the Java Servlet Specification 2.3 and in BEA documentation.

Listing 1-2 shows the generated `web.xml`.

Listing 1-2 The Generated web.xml

```
<!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>
  <servlet>
    <servlet-name>BankAppServlet</servlet-name>
    <servlet-class>examples.tutorials.migration.banking.
```

```
    BankAppServlet
    </servlet-class>
  </servlet>

  <servlet>
    <servlet-name>error</servlet-name>
    <jsp-file>error.jsp</jsp-file>
  </servlet>

  <servlet>
    <servlet-name>AccountDetail</servlet-name>
    <jsp-file>AccountDetail.jsp</jsp-file>
  </servlet>

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

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

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

  <welcome-file-list>
    <welcome-file>login.html</welcome-file>
  </welcome-file-list>

  <security-constraint>
    <display-name></display-name>
    <web-resource-collection>
      <web-resource-name>My secure resources</web-resource-name>
      <description>Resources to be placed under security
        control.</description>
      <url-pattern>/private/*.jsp</url-pattern>
      <url-pattern>/private/*.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>guest</role-name>
      </auth-constraint>
    </security-constraint>

  <security-role>
    <description>The role allowed to access our
```

```
    content</description>
  <role-name>guest</role-name>
  </security-role>
</web-app>
```

In `web.xml`, the elements to watch for are

- `<servlet>`
- `<servlet-mapping>`
- `<display-name>`

The `<servlet>` and `<servlet-mapping>` elements map a servlet class to a URL so that a user can access the servlet directly by its URL. If this is what you want, leave the `<servlet>` and `<servlet-mapping>` pair that describe a servlet. If not, delete it.

The `<display-name>` element is meant to display a Web application name in a tool, but typically does not have a value. If the `DOCTYPE` header shows that you are using a Web application version 2.2 DTD, the `<display-name>` element will cause the file not to validate. If that happens, just delete `<display-name>`.

Listing 1-3 The Edited web.xml for the Sample Banking Application

```
<!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>
  <welcome-file-list>
    <welcome-file>login.html</welcome-file>
  </welcome-file-list>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>My secure resources</web-resource-name>
      <description>Resources to be placed under security control.
      </description>
      <url-pattern>/private/*.jsp</url-pattern>
      <url-pattern>/private/*.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>guest</role-name>
    </auth-constraint>
  </security-constraint>
```

```
<security-role>
    <description>The role allowed to access our content
    </description>
    <role-name>guest</role-name>
</security-role>

</web-app>
```

Hands On: Edit web.xml

Edit weblogic.xml

The `weblogic.xml` descriptor specifies parameters (in name/value pairs) that describe information about how the server handles HTTP sessions and JSP compilation for your application (more info).

In this release, you need to replace the `DOCTYPE` statement in the generated `weblogic.xml` with this one:

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 6.1//EN" "http://www.bea.com/servers/wls610/dtd/
weblogic-web-jar.dtd">
```

This `DOCTYPE` statement uses the correct version of the DTD, so that your XML editor can validate the descriptor. The rest of the descriptor can stay the same, unless you need to edit values for your application.

You can also use the version of `weblogic.xml` distributed in `banking.zip` and edit it for your application.

Listing 1-4 The Corrected weblogic.xml

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 6.1//EN" "http://www.bea.com/servers/wls610/dtd/
weblogic-web-jar.dtd">

<weblogic-web-app>
  <session-descriptor>
    <session-param>
      <param-name>URLRewritingEnabled</param-name>
      <param-value>true</param-value>
    </session-param>
```

```
    <session-param>
      <param-name>InvalidationIntervalSecs</param-name>
      <param-value>60</param-value>
    </session-param>
    <session-param>
      <param-name>PersistentStoreType</param-name>
      <param-value>memory</param-value>
    </session-param>
    <session-param>
      <param-name>TimeoutSecs</param-name>
      <param-value>3600</param-value>
    </session-param>
  </session-descriptor>

  <jsp-descriptor>
    <jsp-param>
      <param-name>compileCommand</param-name>
      <param-value>javac</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>precompile</param-name>
      <param-value>false</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>workingDir</param-name>
      <param-value>C:\TEMP\</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>keepgenerated</param-name>
      <param-value>true</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>pageCheckSeconds</param-name>
      <param-value>5</param-value>
    </jsp-param>
  </jsp-descriptor>
</weblogic-web-app>
```

Step 4: Edit the EJB Descriptors

The generated EJB descriptors are stored in the `ejb\META-INF` directory. You should look at and verify `ejb-jar.xml`, but it probably won't require much change.

However, you must always edit `weblogic-ejb-jar.xml` and `weblogic-cmp-rdbms-jar.xml`.

They always need values that are specific to your database schema or enterprise beans.

Edit ejb-jar.xml

The elements to look at in `ejb-jar.xml` (defined in the **Enterprise JavaBeans Specification**) are `<ejb-name>` and `<assembly-descriptor>`.

To illustrate, Listing 1-5 shows the generated `ejb-jar.xml` for the sample banking application.

Listing 1-5 The Generated ejb-jar.xml

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-
jar_1_1.dtd'>


<!-- Generated XML! -->
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>AccountBean</ejb-name>
      <home>examples.tutorials.migration.banking.AccountHome
      </home>
      <remote>examples.tutorials.migration.banking.Account
      </remote>
      <ejb-class>examples.tutorials.migration.banking.AccountBean
      </ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field>
```

```
      <field-name>accountId</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>balance</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>accountType</field-name>
    </cmp-field>
      <primkey-field>accountId</primkey-field>
  </entity>
</enterprise-beans>

<assembly-descriptor>
</assembly-descriptor>

</ejb-jar>
```

The value of `<ejb-name>` is a name for each enterprise bean in your application. The `<ejb-name>` value is used only within deployment descriptors, but must match in each descriptor. It is recommended that you accept the generated `<ejb-name>`. If you choose to change it, remember to change it in `weblogic-ejb-jar.xml` and `weblogic-cmp-rdbms-jar.xml` as well.

Notice also that the `<assembly-descriptor>` element is empty, meaning that it uses default values for the elements it contains. The empty `<assembly-descriptor>` element has this meaning:

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <description>container managed</description>
      <ejb-name>AccountBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>*</method-name>
    </method>
  <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

That is, the default value of `<assembly-descriptor>` specifies the transaction attribute `Required` for all methods in the bean. You only need to edit `<assembly-descriptor>` if you change the `<ejb-name>` in `<ejb-jar.xml>`, or if you want to specify different values for `<trans-attribute>` or `<method-name>`.

For example, you could write `<assembly-descriptor>` this way if you want to give the `withdraw` method a different transaction attribute:

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <description>container managed</description>
      <ejb-name>AccountBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>withdraw</method-name>
    </method>
  <trans-attribute>Mandatory</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

Hands On: Edit ejb-jar.xml

Edit weblogic-ejb-jar.xml

The `weblogic-ejb-jar.xml` descriptor (more info) describes the behavior of your enterprise beans that is specific to WebLogic Server. `weblogic-ejb-jar.xml` has many unique WebLogic elements.

For the sample banking application, and your own applications as well, you need to look for these elements:

- `<ejb-name>`, giving it the same value as in `ejb-jar.xml`
- `<type-version>`, using the correct version number for your enterprise beans (either 5.1.0 or 6.0)
- `<jndi-name>`, making sure to use the `<ejb-name>`

With the sample banking application, the `ejb.DDInit` tool generates the `weblogic-ejb-jar.xml` file shown in Listing 1-6.

Listing 1-6 The Generated weblogic-ejb-jar.xml

```
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD
WebLogic 6.0.0 EJB//EN' 'http://www.bea.com/servers/wls600/dtd/
weblogic-ejb-jar.dtd'>

<!-- Generated XML! -->
```

```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>AccountBean</ejb-name>
      <entity-descriptor>
        <persistence>
          <persistence-type>
            <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
            <type-version>5.1.0</type-version>
            <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml
            </type-storage>
          </persistence-type>
          <persistence-use>
            <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
            <type-version>5.1.0</type-version>
          </persistence-use>
        </persistence>
      </entity-descriptor>

      <jndi-name>examples.tutorials.migration.banking.AccountHome
      </jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

In this case, we have made no changes to the `<ejb-name>` in other descriptors, so that element is fine. The value of `<type-version>` can be `5.1.0` or `6.0` (in this case, 5.1.0 applies to the beans in the sample application). `<type-version>` cannot have a value of `6.1`, even if you use WebLogic Server Version 6.1, This makes a difference later, if you package your EJB component into an EJB JAR file and then use `ejbc` to verify the EJB JAR before deploying it on the server.

Then, adjust the value of `<jndi-name>` to use the `<ejb-name>` instead of the package name:

```
<jndi-name>AccountBean.AccountHome</jndi-name>
```

So, once edited, the `weblogic-ejb-jar.xml` file looks like Listing 1-7.

Listing 1-7 The Edited weblogic-ejb-jar.xml for the Sample Banking Application

```
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD
WebLogic 6.0 EJB//EN' 'http://www.bea.com/servers/wls600/dtd/
weblogic-ejb-jar.dtd'>

<weblogic-ejb-jar>
<weblogic-enterprise-bean>
```

```
  <ejb-name>AccountBean</ejb-name>
  <entity-descriptor>
    <persistence>
      <persistence-type>
          <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
          <type-version>5.1.0</type-version>
          <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml
          </type-storage>
      </persistence-type>
      <persistence-use>
        <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
        <type-version>5.1.0</type-version>
      </persistence-use>
    </persistence>
  </entity-descriptor>

  <jndi-name>AccountBean.AccountHome</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Hands On: Edit weblogic-ejb-jar.xml

Edit weblogic-cmp-rdbms-jar.xml

The `weblogic-cmp-rdbms-jar.xml` descriptor (more info) describes how an entity bean accesses database tables and columns. This descriptor is generated only for entity beans, not for session beans.

The main element you should look for in this descriptor is `finder` (see Listing 1-8). The `finder` element describes the finder methods used in the bean and need to be updated with database queries.

Listing 1-8 The Generated weblogic-cmp-rdbms-jar.xml

```
<!DOCTYPE weblogic-rdbms-jar PUBLIC  '-//BEA Systems, Inc.//DTD
WebLogic 6.0.0 EJB 1.1 RDBMS Persistence//EN' 'http://www.bea.com/
servers/wls600/dtd/weblogic-rdbms11-persistence-600.dtd'>

<!-- Generated XML! -->

<weblogic-rdbms-jar>
  <weblogic-rdbms-bean>
    <ejb-name>AccountBean</ejb-name>
```

```
    <pool-name>AccountBeanPool</pool-name>
    <table-name>AccountBeanTable</table-name>
    <field-map>
      <cmp-field>accountId</cmp-field>
      <dbms-column>accountIdColumn</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>balance</cmp-field>
      <dbms-column>balanceColumn</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>accountType</cmp-field>
      <dbms-column>accountTypeColumn</dbms-column>
    </field-map>
    <finder>
      <finder-name>findByPrimaryKey</finder-name>
      <finder-param>java.lang.String</finder-param>
      <finder-query><![CDATA[(= 1 1)]]></finder-query>
    </finder>
    <finder>
      <finder-name>findAccount</finder-name
      <finder-param>double</finder-param>
      <finder-query><![CDATA[(= 1 1)]]></finder-query>
    </finder>
    <finder>
      <finder-name>findBigAccounts</finder-name>
      <finder-param>double</finder-param>
      <finder-query><![CDATA[(= 1 1)]]></finder-query>
    </finder>
    <finder>
      <finder-name>findNullAccounts</finder-name>
      <finder-query><![CDATA[(= 1 1)]]></finder-query>
    </finder>
    </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

A `<finder>` element associates a finder method signature in an EJB home interface with a database query that retrieves the data.

The `<finder>` elements have `CDATA` attributes that mark blocks of text that would otherwise be interpreted as markup. Each `CDATA` attribute needs a database query in WebLogic Query Language that is specific to your database.

But, if you have a `<finder>` element for a method named `<findByPrimaryKey>`, you can delete it, because the server generates a default query that you should accept in most cases.

Listing 1-9 The Edited weblogic-cmp-rdbms-jar.xml

```
<!DOCTYPE weblogic-rdbms-jar PUBLIC  '-//BEA Systems, Inc.//DTD
WebLogic 6.0.0 EJB 1.1 RDBMS Persistence//EN' 'http://www.bea.com/
servers/wls600/dtd/weblogic-rdbms11-persistence-600.dtd'>

<weblogic-rdbms-jar>
  <weblogic-rdbms-bean>
    <ejb-name>AccountBean</ejb-name>
    <pool-name>AccountBeanPool</pool-name>
    <table-name>AccountBeanTable</table-name>
    <field-map>
      <cmp-field>accountId</cmp-field>
      <dbms-column>accountIdColumn</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>balance</cmp-field>
      <dbms-column>balanceColumn</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>accountType</cmp-field>
      <dbms-column>accountTypeColumn</dbms-column>
    </field-map>
    <finder>
      <finder-name>findAccount</finder-name>
      <finder-param>double</finder-param>
      <finder-query><![CDATA[(= balance $0)]]></finder-query>
    </finder>
    <finder>
      <finder-name>findBigAccounts</finder-name>
      <finder-param>double</finder-param>
      <finder-query><![CDATA[(> balance $0)]]></finder-query>
    </finder>
    <finder>
      <finder-name>findNullAccounts</finder-name>
      <finder-query><![CDATA[(isNull accountType)]]></finder-query>
    </finder>
  </weblogic-rdbms-bean>
</weblogic-rdbms-jar
```

Hands On: Edit weblogic-cmp-rdbms-jar.xml

Step 5: Write application.xml

The last descriptor you need is `application.xml`, which describes the J2EE application as a whole. With WebLogic Server 6.1SP1, you cannot generate `application.xml`. You can, however, use the sample `application.xml` descriptor provided with this tutorial and edit it for your application. Editing it is very simple and straightforward.

The main purpose of `application.xml` is to specify where to locate the web and EJB modules of your J2EE application. If you are using J2EE 1.2, `application.xml` must contain a `DOCTYPE` definition like this one (all on one line):

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN" "http://java.sun.com/j2ee/dtds/
application_1_2.dtd">
```

Or, with J2EE 1.3, like this one:

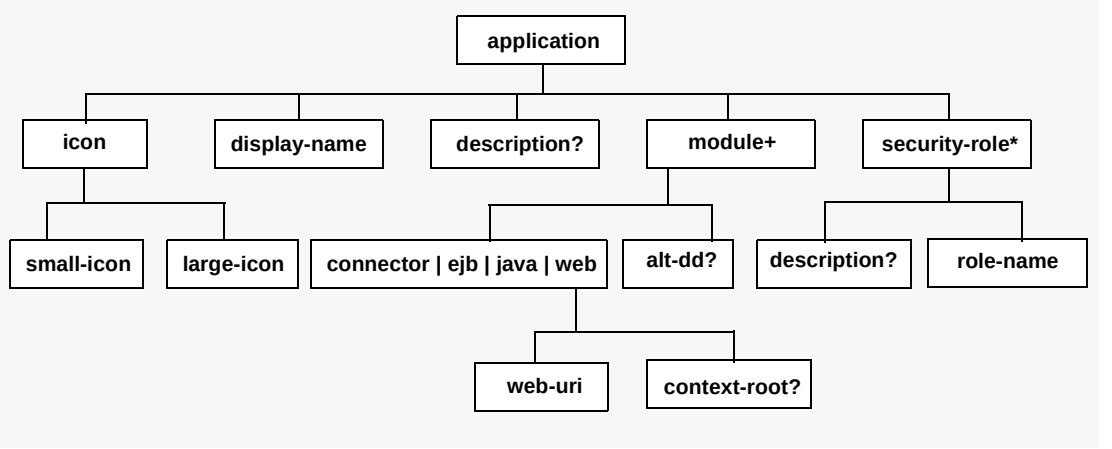
```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN" "http://java.sun.com/dtd/
application_1_3.dtd">
```

Following the `DOCTYPE`, you need just a few elements:

- An <application> element that contains everything else
- Within <application>, <icon>, <display-name>, and <description> elements for use by tools
- A <module> element that contains <ejb>, <web>, <connector>, and <java> elements for the modules in your application

To help you if you need to add elements or add values, the schema for `application.xml` is shown in Figure 1-5.

Figure 1-5 The Structure of application.xml



If you are using the directory structure shown in Figure 1-2 (at the beginning of this chapter), the `application.xml` file looks like Listing 1-10 when you deploy your application in exploded form. It will look different when you package your application into an EAR file.

Listing 1-10 The Sample application.xml Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN' 'http://java.sun.com/j2ee/dtds/
application_1_2.dtd'>

<application>
  <display-name></display-name>
  <module>
    <ejb>\ejb</ejb>
  </module>
  <module>
    <web>
      <web-uri>\web</web-uri>
      <context-root>banking</context-root>
    </web>
  </module>
```

```
</application>
```

Note that the value of `<context-root>` is becomes part of the URL you use to access the application, after the host and port names, but before the name of the accessed file or servlet:

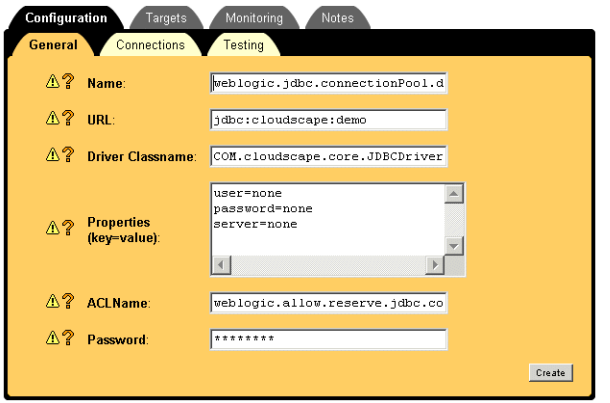
```
http://localhost:7001/banking/login.html
```

Step 6: Set Up a Connection Pool

If your J2EE application accesses a database, you need to set up a connection pool before you deploy the application. A connection pool is a named group of JDBC connections that your users will use to connect from your application to the database.

You can set up the connection pool graphically, through the Administration Console, which adds an entry for your application to the `config.xml` file in *`WL_HOME`*`\config\mydomain`.

Figure 1-6 Setting Up a Connection Pool in the Administration Console



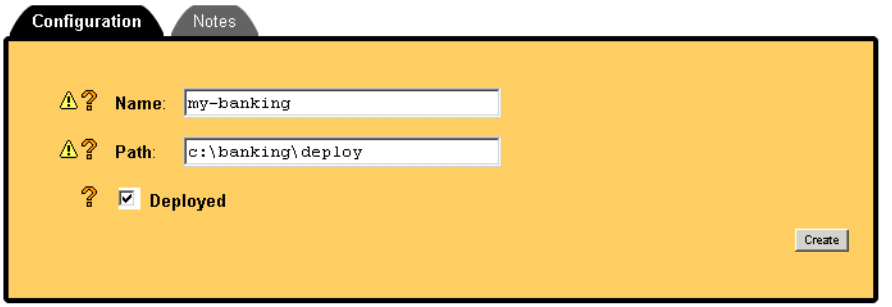
Hands On: Set Up a Connection Pool

Step 7: Deploy the Exploded Application

Once you set up a connection pool, while you are still using the Administration Console, you can use the console to deploy the application in exploded form. This tutorial describes how to deploy on a single server.

In this step, you leave your application in its development directory, because after deploying and testing it, you may decide to change source code and recompile. You are simply deploying by using the Administration Console once the server is running.

Figure 1-7 Deploying from the Administration Console

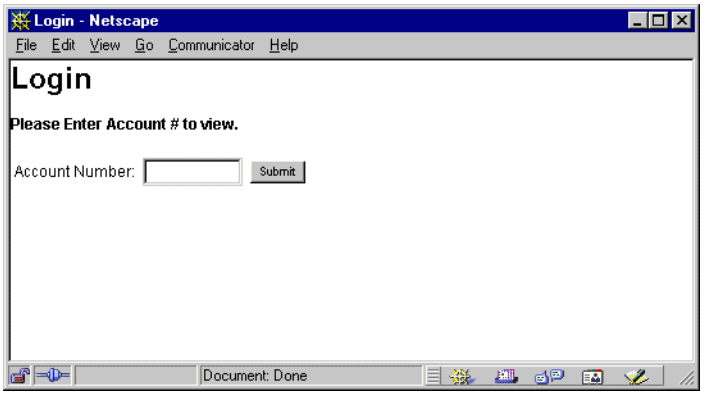


You have other ways of deploying as well:

- Automatic deployment, where you store an application in the `config\`*`somedomain`*`\applications` directory and it is automatically deployed when you start the server in development mode
- Editing **`config.xml`** in the domain in which you store your application
- Using the **`weblogic.deploy`** utility

Once the application is deployed successfully, you can run it. If it has a Web client, you would start it from a Web browser, using the value of `<context-root>` in `application.xml`. As an example, you would start the sample banking application by entering **http://localhost:7001/banking/login.html** in a Web browser.

Figure 1-8 Starting the the Banking Application



By the way, the valid account numbers you can enter are **1000** and **6000**.

Once the application is deployed, decide if and when to redeploy it. You may want to redeploy often if you are testing, or you may want to protect your application from being redeployed accidentally.

To control redeployment, create an empty text file named `REDEPLOY` in the application `META-INF` directory. Now you can control redeployment:

- If you modify the file (for example, by adding a space character) or using a `touch` command on it (on UNIX systems), you redeploy the application. This is useful during testing.
- If you change permissions on the `REDEPLOY` file to prevent it from being modified, you also prevent your application from being redeployed. Use this technique when you deploy an exploded application for production.

Hands On: Deploy the Exploded Application

A Hands-On: Deploying the Sample Banking Application

The following sections show you step by step how to deploy the sample banking application on WebLogic Server.

Prerequisites

To work through this tutorial, you need:

- WebLogic Server 6.1
  http://commerce.bea.com/downloads/weblogic_server.jsp

  This evaluation copy is WebLogic Server Premium Edition, with all features. You can also run this tutorial on WebLogic Server Advantage Edition, if you have that version installed.

  You cannot, however, use WebLogic Express for this tutorial.

- The sample banking application, `banking.zip`
  http://e-docs.bea.com/wls/docs61/samples.html

- The BEA XML Editor or XML editor:
  http://dev2dev.bea.com/resourcelibrary/utilitiestools/index.jsp

The instructions in this tutorial are specific to Microsoft Windows, but can be easily adapted to UNIX platforms.

Notes

- In the hands-on instructions, a pathname like this

  `yourWebLogic`\config\mydomain

  asks you to substitute the name of your WebLogic Server directory for *`yourWebLogic`*.

- When you see an instruction like this:

  ```
  cd %WL_HOME%\config\mydomain
  ```

  it means enter the command using an environment variable: `%WL_HOME%` on Windows, or `$WL_HOME` on Unix.

- When you see *`bankingHome`*, it means the root location of the banking application.

Set Up the Application Directory

1. Download `banking.zip` from the tutorials website and unzip it to your `c:` drive.

   You should see a directory structure like this:

   ```
   banking\
     dev\
       ejb\
       web\
     deploy\
         ejb\
           META-INF\
         web\
           WEB-INF\
             classes\
             lib\
   ```

   If you are working on Windows, the `META-INF` directory might look like it has a mixed case name, like `Meta-inf`.

2. Open a command window.

3. Set up your shell environment to run the build script:

   ```
   cd yourWeblogicDirectory\config\mydomain
   setEnv
   ```

4. Add the location of the compiled EJB classes to your class path:

   ```
   set CLASSPATH=bankingHome\deploy\ejb;%CLASSPATH%
   ```

5. Run the build script to compile the servlet and enterprise bean classes to the correct locations:

   ```
   cd bankingHome\dev
   ```

   ```
   build
   ```

6. Copy the JSP and HTML files and images to the `deploy` directory:

   ```
   cd bankingHome\dev\web
   copy images bankingHome\deploy\web
   copy html bankingHome\deploy\web
   ```

   Note that you are moving the JSP and HTML files to `deploy\web`, not `deploy\web\html`.

Generate the Descriptors

1. Move to the `deploy` directory:

   `cd` *`bankingHome`*`\deploy`

2. Generate the web application descriptors:

   `java weblogic.ant.taskdefs.war.DDInit web`

   This step generates the `web.xml` and `weblogic.xml` descriptors and stores them in `web\WEB-INF`.

3. Generate the EJB descriptors:

   `java weblogic.ant.taskdefs.ejb.DDInit ejb`

   This step generates `ejb-jar.xml`, `weblogic-ejb-jar.xml`, and `weblogic-cmp-rdbms-jar.xml` and stores them in `ejb\META-INF`.

Edit web.xml

1. With the XML editor of your choice, open *`bankingHome`*`\deploy\web\WEB-INF\web.xml`.

2. Delete all <servlet> and <servlet-mapping> entries referring to:

   `AccountDetail`

   `error`

   They are unnecessary.

3. Map BankAppServlet in <servlet-mapping> as follows:
   `<url-pattern>/banking</url-pattern>`

4. Save the file.

Edit weblogic.xml

1. Open the generated `weblogic.xml` in your XML editor.
2. Replace the existing `DOCTYPE` statement with this one, all on one line:

   ```
   <!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
   Application 6.1//EN"
   "http://www.bea.com/servers/wls610/dtd/weblogic-web-jar.dtd">
   ```

3. Save the file.

Edit weblogic-ejb-jar.xml

1. Open the generated *`bankingHome`*`\deploy\ejb\META-INF\weblogic-ejb-jar.xml` in your XML editor.
2. Change the value of `<jndi-name>` to:

   ```
   <jndi-name>containerManaged.AccountHome</jndi-name>
   ```

3. Save the file.

Edit weblogic-cmp-rdbms-jar.xml

1. Open *`bankingHome`*`\deploy\ejb\META-INF\weblogic-cmp-rdbms-jar.xml` in your XML editor.
2. Search for `findByPrimaryKey`. Delete `<finder>findByPrimaryKey</finder>`.
3. Change the value of `<pool-name>` to:

```
<pool-name>demoPool</pool-name>
```

4. Change the value of `<table-name>` to:

   ```
   <table-name>ejbAccounts</table-name>
   ```

5. Change the following `<dbms-column>` values:

   a. from `<dbms-column>accountIdColumn</dbms-column>`

      to `<dbms-column>id</dbms-column>`

   b. from `<dbms-column>balanceColumn</dbms-column>`

      to `<dbms-column>bal</dbms-column>`

   c. from `<dbms-column>accountTypeColumn</dbms-column>`

      to `<dbms-column>type</dbms-column>`

6. Search for `findAccount` and change the value of `<finder-query>` to:

   ```
   <![CDATA[(= balance $0)]]>
   ```

7. Search for `findBigAccounts` and change the value of `<finder-query>` to:

   ```
   <![CDATA[(> balance $0)]]>
   ```

8. Search for `findNullAccounts` and change the value of `<finder-query>` to:

   ```
   <![CDATA[(isNull accountType)]]>
   ```

9. Save the file.

Edit application.xml

1. Copy the `application.xml` file included in the zip file to the right location for deployment:

   ```
   copy bankingHome\dev\application.xml
   bankingHome\deploy\META-INF
   ```

2. Open `application.xml` in your XML editor.

3. Edit the values of `<ejb>` and `<web-uri>` to use the names of your EJB and web module directories.

If you are using the sample directory structure, you do not need to change `application.xml`. The elements would look like this:

```
<module>
  <ejb>\ejb</ejb>
</module>
<module>
  <web>
    <web-uri>\web</web-uri>
    <context-root>banking</context-root>
  </web>
```

4. Save the file.

Insert Cloudscape Data

1. Set your environment:

   `setEnv.cmd`

2. Add `%WL_HOME%\samples\eval\cloudscape\lib\cloudscape.jar` to CLASSPATH.

3. Execute `banking.ddl`:

```
java utils.Schema jdbc:cloudscape:%WL_HOME%\samples\eval\cloudscape\data\demo
COM.cloudscape.core.JDBCDriver -verbose bankingHome\dev\ejb\banking.ddl
```

Server Startup

1. Before starting the server, add the following.

   a. To `startWebLogic.cmd`:

      ```
      -Dcloudscape.system.home=%WL_HOME%/samples/eval/cloudscape/d
      ata
      ```

   b. To the startup CLASSPATH in the `startWebLogic.cmd`:

```
%WL_HOME%\samples\eval\cloudscape\lib\cloudscape.jar;
bankingHome\deploy\ejb
```

2. Start WebLogic Server:

```
cd %WL_HOME%\config\mydomain
startWebLogic
```

Set Up a Connection Pool

1. Start the Administration Console by opening a Web browser and browsing to

   `http://localhost:7001/console`

   Or, if you specified a different host name or listen port when you installed the server, use those.

2. In the left pane, click **JDBC --> Connection Pools**.
3. In the right pane, click **Configure a New JDBC Connection Pool**.
4. Enter these values:
5. **Name**`demoPool`
   URL`jdbc:cloudscape:demo`
   Driver Classname`COM.cloudscape.core.JDBCDriver`
6. Click **Create**.
7. Click the Target tab.
   a. In the right pane, click the Targets tab.
   b. Select myserver from the Available column.
   c. Click the right arrow, then click Apply.

Configure RMI Logger Startup Class

1. Go to Startup & Shutdown under Deployments.
2. Click Configure a new Startup Class.
3. Enter the following:

```
Name: rmilogger
```

```
ClassName: examples.tutorials.migration.banking.RMILoggerImpl
```

4. Click Create
5. Target the Startup Class:
   a. Click the Target tab.
   b. In the right pane, click the Targets tab.
   c. Select myserver from the Available column.
   d. Click the right arrow, then click Apply.

Deploy the Exploded Application

1. In the Administration Console, click **Deployments > Applications** in the left pane.
2. In the right pane, click **Configure a New Application**.
3. For **Name**, enter `banking-exploded`.
4. For **Path**, enter *`bankingHome`*`\deploy` .
5. Make sure the **Deployed** box is checked.
6. Click **Create**.
7. In the left pane, expand `banking-exploded`. You should see the `ejb` and `web` modules listed beneath it.

8. Target the server for the `web` module:
   - In the left pane, click `\web`.
   - In the right pane, click the **Targets** tab.
   - Select **myserver** from the **Available** column.
   - Click the right arrow, then **Apply**.
9. Target the server for the `ejb` module:
   - In the left pane, click `\ejb`.
   - In the right pane, click the **Targets** tab.
   - Select **myserver** from the **Available** column.
   - Click the right arrow, then **Apply**.
10. To start the banking application, open a Web browser and go to this URL:

    ```
    http://localhost:7001/banking/login.html
    ```