Inside WebObjects

Enterprise JavaBeans

November 2002

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Listings, and Tables

About This Document

Enterprise JavaBeans (EJB) is a specification that provides an infrastructure through which solution providers can develop components that you can purchase and use in your WebObjects applications with minimal effort. In addition, the components can be configured to work with a variety of databases (as long as the database supports JDBC). The key ingredient in these components is enterprise beans. Enterprise beans are business objects that contain logic used to perform specific tasks. They are similar to enterprise objects in WebObjects, but can be used in application servers by multiple vendors.

Enterprise JavaBeans is part of Sun's Java 2 Platform, Enterprise Edition (J2EE) strategy. J2EE provides an abstraction from the implementation details of databases, directory services, communication protocols, and so on. EJB aims at providing you an abstraction layer between your application's business logic and the implementation-specific details of the data entities it uses. An enterprise-bean developer doesn't have to worry about which database is used when the bean is deployed, freeing her to concentrate on the business problem. WebObjects implements version 1.1 of the EJB specification.

Enterprise JavaBeans support in WebObjects lets you integrate third-party, enterprise-bean–based solutions in your WebObjects applications. This means you can purchase components that solve a particular problem, so that you can focus on issues specific to your business. In addition, you can develop your own enterprise beans using WebObjects tools. You must keep in mind, however, that Enterprise Object technology does not complement, nor can be efficiently combined with Enterprise JavaBeans. When you write enterprise beans, you use a persistence-management system that is completely separate from Enterprise Objects. You should not have enterprise beans that use the same database tables that enterprise-object classes are mapped to.

About This Document

You should read this document if you want to learn how to incorporate an EJB-based solution in a WebObjects application or you want to develop your own enterprise beans using WebObjects tools. However, it is not the purpose of this document to teach you EJB development. If you want to develop enterprise beans, you must already have a sound knowledge of the technology.

The document includes the following chapters:

- Chapter 2, "Introduction to Enterprise JavaBeans" (page 17), provides an overview of Enterprise JavaBeans technology and how it's implemented in WebObjects.
- Chapter 3, "Developing Session Beans" (page 21), walks you through the development of a simple session bean and its use in a client application.
- Chapter 5, "Developing Bean Frameworks" (page 71), lists the steps you take to create and maintain bean frameworks.
- Chapter 6, "Configuring Applications" (page 79), explains how to configure the transaction manager, the persistence manager, and the EJB container in your client applications.
- Chapter 7, "Configuration Reference" (page 101), provides explanations of the XML elements used in the configuration files of client applications.
- "Document Revision History" (page 143), lists changes made from previous editions of the document.

To get the most out of this document you should be an experienced WebObjects application developer. In particular, you need to know how to create applications using Project Builder and be familiar with the layout of a Project Builder project. To make use of enterprise beans in an application, you are required to edit configuration files written in XML; therefore, you should be familiar with XML's rules and syntax.

To streamline your learning experience, you can use take advantage of the companion resources that are included with this document in the `databases`, `models`, and `projects` directories in `/Developer/Documentation/WebObjects/Enterprise_JavaBeans` or in the TAR file that you can download from http://developer.apple.com/techpubs/webobjects/webobjects.html.

If you need to learn the basics about developing WebObjects applications, you can find pertinent documents and resources in http://developer.apple.com/webobjects.

If you need to learn about EJB development, these books provide you introductory information as well as development guidelines:

- *Enterprise JavaBeans* (O'Reilly)
- *Professional EJB* (Wrox Press)
- *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform* (Addison-Wesley)

WebObjects uses open-source implementations of the EJB container, the object request broker, the transaction manager, and the persistence manager. For details about those implementations in WebObjects, consult the following documents:

- *OpenEJB User Guide* provides details about the EJB container included with WebObjects. However, most of the information from it needed to develop or deploy enterprise beans is present in this document. You can find the document in `/System/Library/Frameworks/JavaOpenEJB.framework/Resources/English.lproj/Documentation/OpenEJB_User_Guide.pdf`. You can find more information at http://OpenEJB.sourceforge.net.
- *OpenORB Programmers Guide* and *RMI over IIOP for OpenORB* deal with the Object Request Broker (ORB) implementation used in WebObjects. They are located in `/System/Library/Frameworks/JavaOpenORB.framework/Resources/English.lproj/Documentation`. For more information, visit http://OpenORB.sourceforge.net.
- API documentation on the Tyrex transaction manager is located in `/System/Library/Frameworks/JavaOpenTM.framework/Resources/English.lproj/Documentation`. For more information on Tyrex, visit http://Tyrex.sourceforge.net.

CHAPTER 1

About This Document

CHAPTER 2

Introduction to Enterprise JavaBeans

WebObjects provides all the tools you need to develop and deploy enterprise applications. However, WebObjects is not the only technology available. Other companies provide tools that accomplish the same task, albeit using different methods and requiring specialized deployment environments. Therefore, it's difficult for a WebObjects application to talk to an application developed and deployed under a different environment. J2EE and EJB bridge the schism between environments from different vendors.

J2EE standardizes the way Web applications communicate with the resources they need to operate. Akin to JDBC, the goal of J2EE is to provide an infrastructure that applications from different developers can utilize to get their work done.

Enterprise JavaBeans

Enterprise JavaBeans is an important part of J2EE. It provides an environment in which components from several manufacturers can be assembled into a working application. The application assembler, with deep knowledge of the requirements of the business, can choose the component that best matches the task at hand. For instance, she could use transaction-processing beans from one company; customer, order, and product beans from another company; and shipping beans from a third company. She would then end up with an application capable of accepting orders, charging the customer, and process shipments without having to write code.

Enterprise beans are specialized components that can encapsulate session information, workflow, and persistent data. A bean client is an application or an enterprise bean that makes use of another bean. An enterprise bean has three parts:

- **The home interface** is used by the client to create and discard beans.
- **The remote interface** is used by the client to execute the bean's business methods.
- **The implementation or bean class** is where the bean's business and callback methods are implemented. The client never invokes these methods directly; they are invoked by the bean container.

The container is a conceptual entity that mediates between enterprise-bean instances and client applications. Clients never access bean instances directly. Instead, they interact with proxies provided by the container This allows the bean container to perform its duties in the most efficient way. The client doesn't have to know how the container implements its functions; all it needs to know is how to talk to the container.

In addition, beans have a deployment descriptor. This is an XML file that gives the container information about each bean and data-source connection details, among many other items.

There are two major types of enterprise beans: session beans and entity beans.

- **Session beans** come in two flavors: stateful and stateless. Stateful session beans maintain state between method invokations; stateless session beans do not.

  Stateless session beans are useful for grouping related methods in one place. Stateful session beans can be used to encapsulate workflow. In most cases it's more efficient for client applications to use session beans (stateless or stateful) to accomplish their tasks than to use entity beans directly because network traffic is reduced.
- **Entity beans** are similar to enterprise objects (the objects that represent an instance of a data entity in Enterprise Objects). They encapsulate access to data entities.

An enterprise-bean developer can focus on the high-level business logic needed to implement the services that a bean provides instead of on low-level system or data-store calls (those functions can be left to the container).

One of the most important functions of the container is transaction management and access control. WebObjects includes the OpenEJB open-source container system. OpenEJB consists of four main components:

- **EJB container:** The EJB container implements the lifecycle of enterprise beans and the server contracts in the EJB specification.

- **Object Request Broker (ORB):** The OpenORB object request broker implements RMI-over-IIOP, naming service, and CORBA ORB.
- **Transaction manager:** The Tyrex transaction manager implements a transaction manager compliant with the Java Transaction API (JTA) and Object Transaction Service (OTS) specifications.
- **Persistence manager:** The Castor JDO persistence manager implements bean persistence for entity beans. It's used in the implementation of CMP (container-managed persistence) beans.

Enterprise JavaBeans in WebObjects

You can use WebObjects development tools to develop enterprise beans from scratch or to integrate third-party EJB-based solutions in a WebObjects application. Bean development in WebObjects is divided in two phases: development and deployment.

You develop enterprise beans by writing the `.java` and deployment descriptor files. Project Builder provides you with templates for all these files. You can also obtain the source code or JAR files for enterprise beans from a third party. As an alternative, you can develop data models from which entity-bean source files can be generated. For more information, see "Developing an Entity Bean From a Data Model" (page 45).

You deploy one or more beans by generating a bean framework, which contains the beans' JAR and deployment descriptor files, and placing it somewhere in a development computer's file system; for example, in `/Library/Frameworks`. After you deploy a bean framework, it's available to be integrated in client applications for their use.

Client applications can be developed in two ways: using an internal bean container, or using an external container:

- **Internal container:** This approach is the most scalable because each application instance has its own container and naming-service object.

If the user load of your site becomes too large for one instance to handle, all you have to do is add more instances of it. Each container answers only to one application, so there is no application-to-container bottleneck.

- **External container:** This approach is beneficial if you already have a robust bean container, running on a fast computer, that you want to leverage. In this case, no configuration files should be present in the bean-client application project. For more on the configuration files, see "Configuring Applications" (page 79).

Developing Session Beans

Before developing WebObjects applications that use enterprise beans, you have to create enterprise-bean frameworks. These frameworks contain the JAR files (which include the deployment descriptor files) needed to deploy enterprise beans.

You can develop a bean framework by writing the beans yourself or by using third-party beans (either from Java source and deployment descriptor files or JAR files). Project Builder helps you develop beans by providing you with bean templates that get you started.

In most cases, you save both time and money when you purchase enterprise beans from EJB vendors instead of developing your own. This is because you obtain a solution that has been tested by the solution vendor and other developers. Also remember that you cannot take advantage of Enterprise Object technology in your enterprise beans; for example, you may have to implement primary-key classes, finder methods, primary-key–value generation, and so on in your entity beans. In addition, you have to choose between implementing a bean as an entity bean or a session bean. It's a bean provider's job to design an effective and efficient bean solution for you. You can then compare similar solutions from various vendors and purchase the one that most closely addresses your situation.

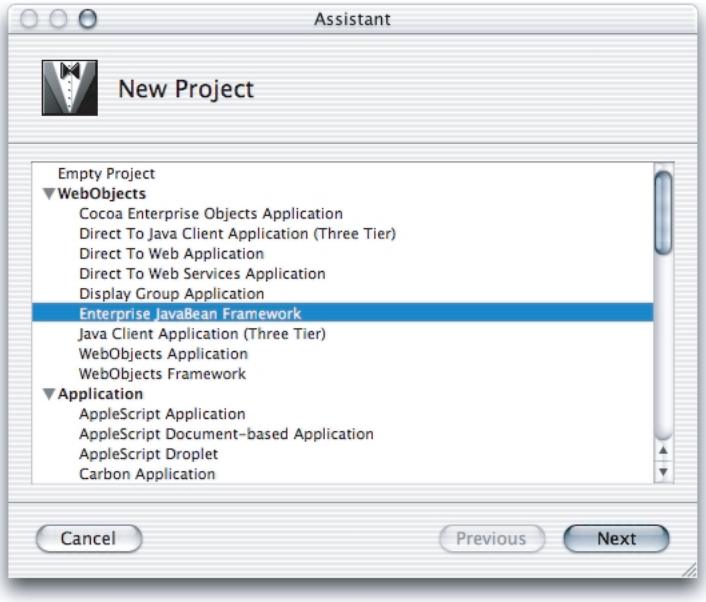
The following sections show how to develop a stateless session bean for use in a WebObjects application both on Mac OS X and Windows.

Developing a Session Bean in Mac OS X

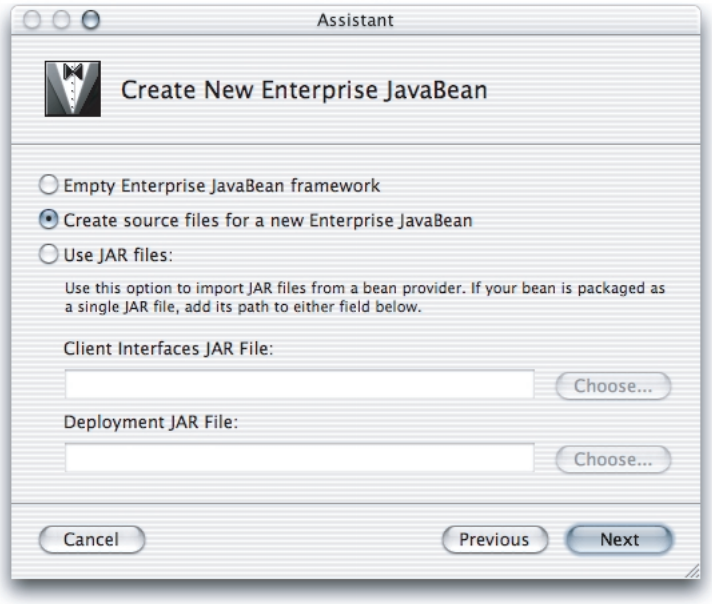
This section shows you how you develop a stateless session bean for use in a WebObjects application in Mac OS X.

Creating the Bean Framework

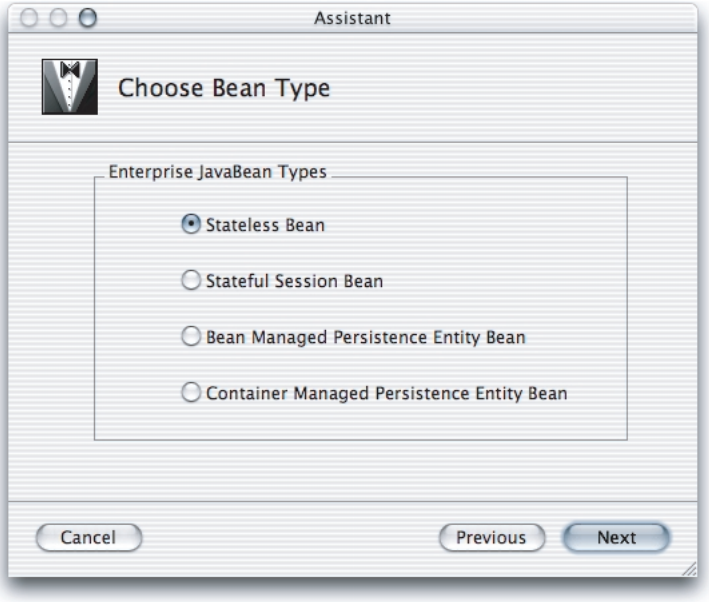
1. Launch Project Builder.
2. Choose File > New Project.
3. In the New Project pane of the Project Builder Assistant, select Enterprise JavaBean Framework from the list of project types, and click Next.



4. Name the project `Hello`.
5. In the Create New Enterprise Java Bean pane of the Assistant, select "Create source files for a new Enterprise Java Bean."



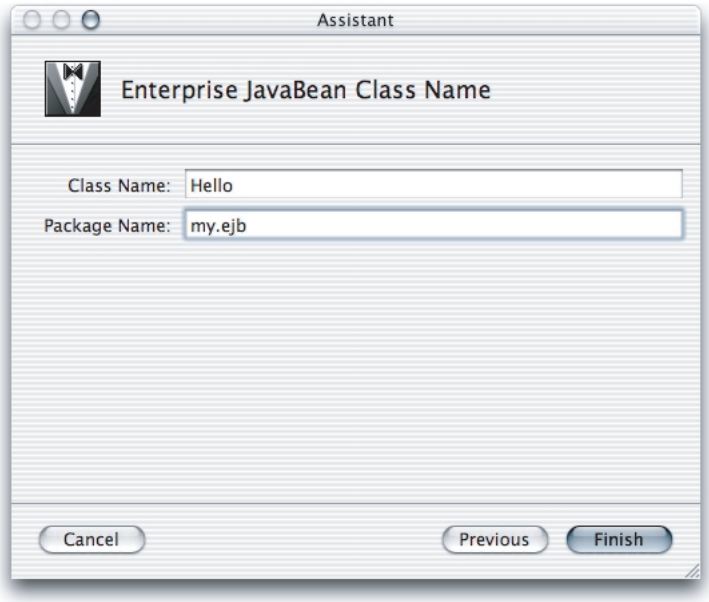
6. In the Choose Bean Type pane, make sure Stateless Bean is selected under Enterprise JavaBean Types.



7. In the Enterprise JavaBean Class Name pane:

    a. Enter `Hello` in the Class Name text field.

    b. Enter `my.ejb` in the Package Name text field.



Analyzing the Hello Bean's Files

The Hello project has templates for the home and remote interfaces, as well as for the implementation class of the Hello enterprise bean in the Classes group of the Groups & Files list. In addition, the Resources group contains the bean's deployment descriptor file.

Listing 3-1 shows the template for the home interface of the Hello enterprise bean (`HelloHome.java`):

Listing 3-1 `HelloHome.java` file

```
package my.ejb;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface HelloHome extends EJBHome {
```

```
    /** Creation methods **/

    /* Stateful session beans may have multiple create methods taking
    * different parameters. They must all be reflected in identically
    * named methods in the home interface without the 'ejb' prefix
    * and initial cap.
    *
    * Stateless session bean create methods never have parameters.
    */

    public Hello create() throws RemoteException, CreateException;
}
```

Listing 3-2 shows the template for the bean's remote interface (`Hello.java`):

Listing 3-2 `Hello.java` file

```
package my.ejb;
import javax.ejb.*;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface Hello extends EJBObject {

    //
    // Business Logic Interfaces
    //

    // Example:
    // public String hello() throws java.rmi.RemoteException;

}
```

Listing 3-3 shows the template for the bean's implementation class (`HelloBean.java`):

Listing 3-3 HelloBean.java file

```
package my.ejb;
import javax.ejb.*;

public class HelloBean implements SessionBean {

    //
    // Creation methods
    //

    public HelloBean() {
    }

    public void ejbCreate() throws CreateException {
        /* Stateless session bean create methods never have parameters */
    }

    //
    // SessionBean interface implementation
    //

    private SessionContext _ctx;

    public void setSessionContext(SessionContext ctx) {
        this._ctx = ctx;
    }

    public void ejbPassivate() {
        /* does not apply to stateless session beans */
    }

    public void ejbActivate() {
        /* does not apply to stateless session beans */
    }

    public void ejbRemove() {
    }

    //
    // Business Logic Implementations
```

```
    //

    // Example:
    // public String hello() { return "hello"; }
}
```

Listing 3-4 shows the bean's deployment descriptor file(`ejb-jar.xml`):

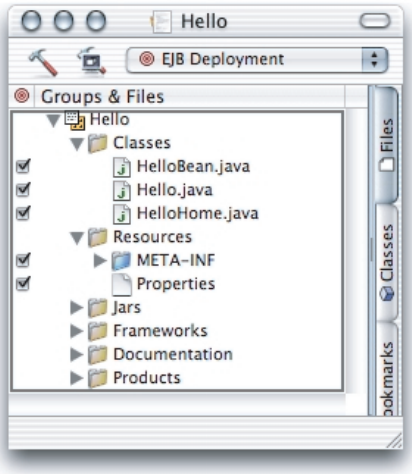
Listing 3-4 `ejb-jar.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
  <description>deployment descriptor for Hello</description>
  <display-name>Hello</display-name>
  <enterprise-beans>
        <session>
            <description>deployment descriptor for HelloBean</description>
            <display-name>HelloBean</display-name>
            <ejb-name>HelloBean</ejb-name>
            <home>my.ejb.HelloHome</home>
            <remote>my.ejb.Hello</remote>
            <ejb-class>my.ejb.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

In addition to providing you with most of the code needed to deploy a bean, Project Builder also partitions the source code appropriately between two targets: EJB Deployment and EJB Client Interfaces.

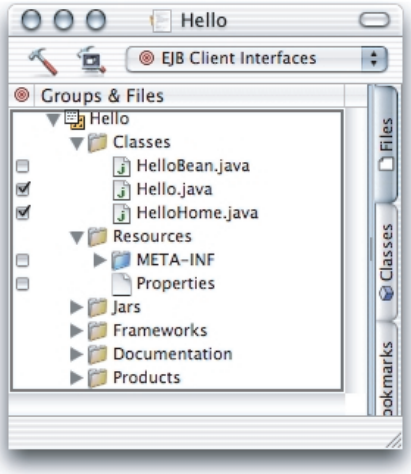
Figure 3-1 shows how all the bean's source files are assigned to the EJB Deployment target.

Figure 3-1 Hello project—the EJB Deployment target's members



When you view the EJB Client Interfaces target, however, you see that the implementation class and the deployment descriptor files are not assigned to it, as shown in Figure 3-2.

Figure 3-2 Hello project—the EJB



Adding Business Logic to the Bean

Now you're ready to add the business logic required for the Hello bean to provide a message to its clients.

Edit `Hello.java` by adding the following method declaration:

```
public String message() throws RemoteException;
```

Edit `HelloBean.java` by adding the implementation of the `message` method, which is listed below.

```
public String message() {
    return "Hello, World.";
}
```

Building the Bean Framework

To build the Hello bean framework, all you have to do is click Build in the toolbar or choose Build > Build. (Make sure that the Hello target is selected in the target pop-up menu before you build.)

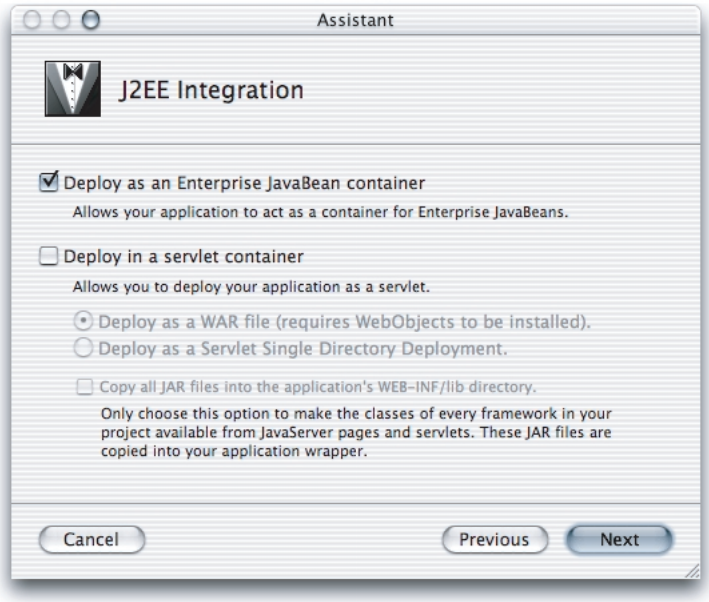
After the framework is built, you can find it in the project's `build` directory:

```
Hello/
    build/
        Hello.framework
```

Creating the Client Application

Now that the Hello bean framework is built, you're ready to use it in an application. In this case, the client application is an Web application that invokes the bean's `message` method, and displays its return value in a WOString element.

1. Create a WebObjects application project.
2. Name the project `Hello_Client`.
3. In the Enable J2EE Integration pane of the Project Builder Assistant, select "Deploy as an Enterprise JavaBean container."



When you deploy the client application as an EJB container, each application instance has its own EJB container. For more information on internal and external containers, see "Enterprise JavaBeans in WebObjects" (page 19).

4. This example doesn't require the use of any data-source adaptors, so make sure no adaptors are selected in the Choose EOAdaptors pane.

   You need to select a data-source adaptor only when you plan to use enterprise objects in your application. Entity beans that use bean-managed persistence (BMP) are responsible for interfacing with the necessary data stores. For entity beans that use container-managed persistence (CMP), the bean container has this responsibility. The Hello_Client application does not use enterprise objects.

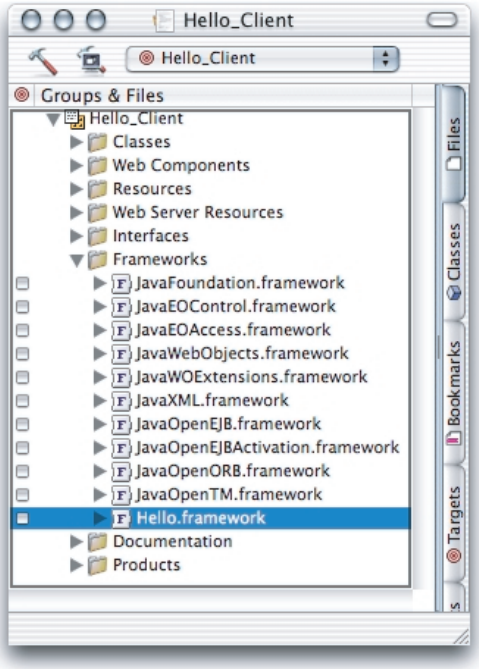
5. Add the Hello framework to the project.

   a. In the Choose Frameworks pane of the Assistant, click Add.

   b. Select `Hello.framework` in the build folder of the Hello project folder, and click Choose.

Figure 3-3 highlights the `Hello.framework` in the Hello_Client project.

Figure 3-3 Hello_Client project—`Hello.framework` in Frameworks group



Adding Business Logic to the Client Application

You have generated an application that, when run, instantiates its own EJB container. This container behaves like a standard EJB container. To access bean instances, you use standard EJB methods.

Modify `Session.java`

Now, edit `Session.java` so that each new session creates a Hello bean proxy that your components can access.

First, add these `import` statements:

```
import my.ejb.Hello;
import my.ejb.HelloHome;
import java.rmi.RemoteException;
```

```
import java.util.Properties;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
```

Now, add two instance variables: one to hold a Hello bean instance and another to hold a Hello home-interface object.

```
// Holds a Hello bean instance.
protected Hello hello;

// Holds the Hello bean's home interface.
private HelloHome _helloHome = null;
```

Modify the `Session` constructor so that it looks like this

```
public Session() {
    super();

    // Instantiate a Hello bean object.
    try {
        hello = helloHome().create();
    }
    catch (RemoteException re) {
        re.printStackTrace();
    }
    catch (CreateException ce) {
        ce.printStackTrace();
    }
}
```

Finally, add the following method:

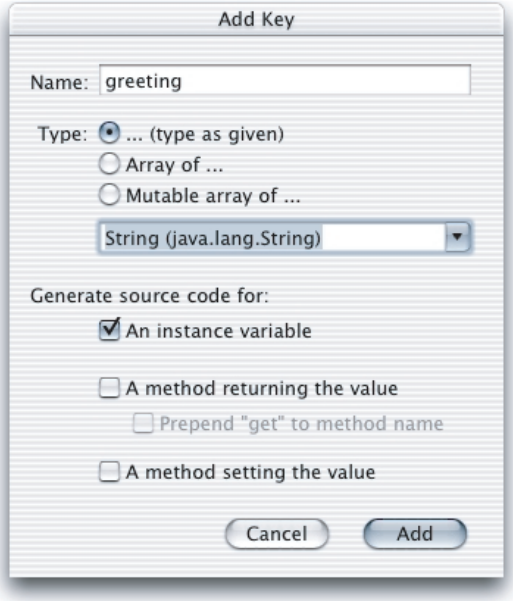
```
/**
 * Obtains Hello bean's home interface.
 * @return Hello bean's home interface.
 */
public HelloHome helloHome() {
    if (_helloHome == null) {
        try {
```

```
            Context jndiContext = new InitialContext();
            _helloHome =
(HelloHome)PortableRemoteObject.narrow(jndiContext.lookup("HelloBean"),
HelloHome.class);
        }
        catch (NamingException ne) {
            ne.printStackTrace();
        }
    }
    return _helloHome;
}
```

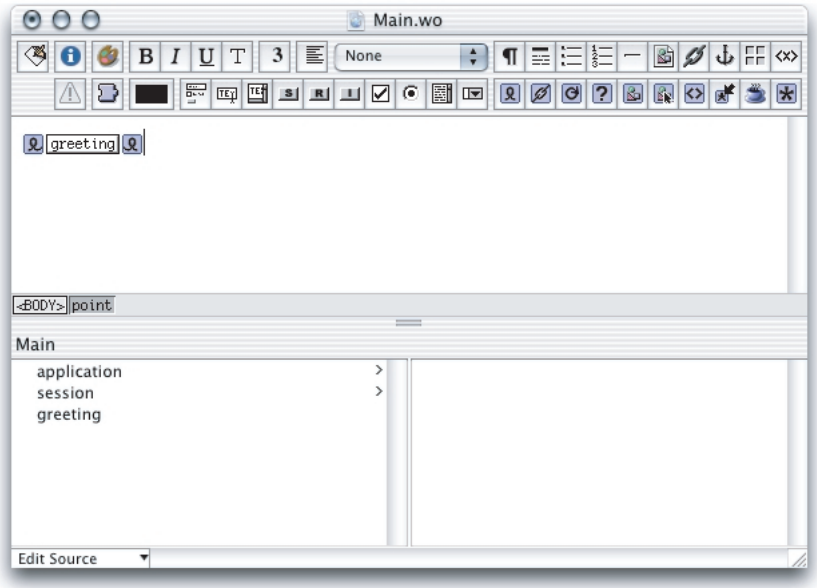
Modify `Main.wo`

Open `Main.wo` in WebObjects Builder by double-clicking `Main.wo`, which is located under the Main subgroup of the Web Components group in the Groups & Files list.

Add a String key called `greeting` to `Main.wo` through the Edit Source pop-up menu.



Add a WOString element to the component, and bind it to the `greeting` key.



Modify `Main.java`

When a Main page is about to be displayed, the Main object needs to invoke the `message` method of its Hello bean proxy to obtain the bean's greeting, and store the value returned in its `greeting` instance variable. When the WOString element is rendered on the page, its `value` binding provides the text to be displayed; in this case, the value comes from `greeting` in the Main object.

Add the following `import` statements to `Main.java`:

```
import my.ejb.Hello;
import java.rmi.RemoteException;
```

Edit the `Main` constructor so that it looks like this

```
public Main(WOContext context) {
    super(context);

    Session session = (Session)session();

    try {
```

```
            greeting = session.hello.message();
        }
        catch (RemoteException re) {
            re.printStackTrace();
        }
    }
```

Configuring the Container

This simple session bean project doesn't make use of bean persistence. Therefore, it requires no container configuration. The text you need to delete from the `TransactionManagerConfiguration.xml` file starts at the line numbered 1 and ends at the line numbered 2 in (everything between the `<resources>` and `</resources>` tags and the tags themselves). See "Transaction Manager Configuration" (page 85) for more information.

Listing 3-5 `TransactionManagerConfiguration.xml` file of the Hello_Client project with container-configuration information

```
<domain>
  <name>default</name>
   <resources>                                                                    //1
     <dataSource>
        <name>DefaultDatabase</name>
        <class>tyrex.resource.jdbc.xa.EnabledDataSource</class>
        <!-- Path to the database-driver JAR File if not in the extensions directory-->
        <jar>file:/FAKEPATHNAME</jar>
        <config>
            <driverName>jdbc:oracle:thin:@HOSTNAME:PORTNAME:DATABASENAME</driverName>
            <driverClassName>oracle.jdbc.OracleDriver</driverClassName>
            <user>ejb</user>
            <password>ejb</password>

            <!-- Transaction timeout in seconds. -->
            <transactionTimeout>60</transactionTimeout>
            <!-- Specifies the JDBC transaction isolation attribute. -->
            <isolationLevel>Serializable</isolationLevel>
        </config>
```

```
        <limits>
            <maximum>100</maximum>
            <minimum>10</minimum>
            <initial>10</initial>
            <maxRetain>300</maxRetain>
            <timeout>50</timeout>
        </limits>
    </dataSource>
  </resources>                                                    //2
</domain>
```

After removing the irrelevant information, the `TransactionManagerConfiguration.xml` file should look like this:

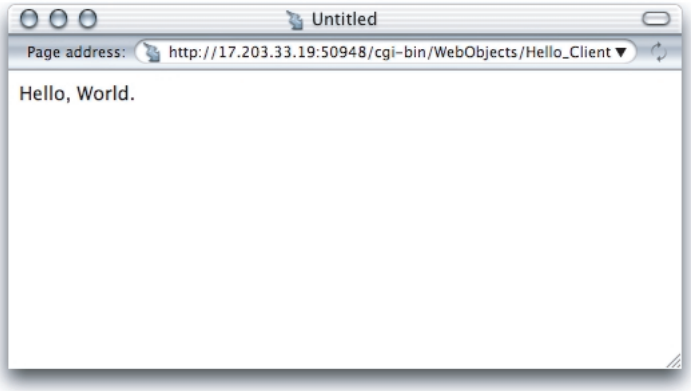
Listing 3-6 `TransactionManagerConfiguration.xml` file of the Hello_Client project without container-configuration information

```
<domain>
  <name>default</name>
</domain>
```

Running the Hello_Client Application

After you build and run the application, you should see a Web page similar to the one in Figure 3-4 in your Web browser.

Figure 3-4 Output of the Hello_Client application



Developing a Session Bean in Windows

This section shows how to develop a stateless session bean for use in a WebObjects application in Windows.

Creating the Bean Framework

1. Launch Project Builder.
2. Choose Project > New.
3. Choose Java WebObjects EJB Framework from the Project Type pop-up menu in the New Project dialog, and click Browse.
4. Select a path for your project, name it `Hello`, and click Save.
5. In the Specify Enterprise JavaBeans pane of the EJB Framework Wizard, select "Create source files for a new Enterprise Java Bean" and click Next.
6. Make sure that Stateless Session Bean is selected in the Chose Enterprise JavaBeans Type pane of the wizard and click Next.

7. In the Create New Enterprise JavaBeans class pane:
    a. Enter `Hello` in the Class Name text field.
    b. Enter `my.ejb` in the Package Name text field.
    c. Click Finish.

Adding Business Logic to the Bean

Now you're ready to add the business logic required for the Hello bean to provide a message to its clients.

Edit `Hello.java` by adding the following code (the file is located in the Classes bucket):

```
public String message() throws RemoteException;
```

Edit `HelloBean.java` by adding the implementation of the `message` method, which is listed below (the file is located in the Classes bucket of the EJBServer subproject).

```
public String message() {
    return "Hello, World.";
}
```

Building the Framework

To build the Hello framework, click the Build button or choose Tools > Project Build > Build.

After the framework is built, you'll find it in the project's directory:

```
Hello/
    Hello.framework
```

Creating the Client Application Project

Now that the Hello framework is built, you're ready to use it in an application. In this case, the client application is a Web application that invokes the bean's `message` method, and displays its return value in a WOString element.

1. Create a Java WebObjects Application project and name it Hello_Client.
2. Choose None in the "Choose type of assistance in your Java project" pane of the WebObjects Application Wizard.
3. Choose "Deploy as an EJB Container" in the Enable J2EE Integration pane.
4. In the Choose EOAdaptors pane, click Select None, and then click Finish.

Adding the Hello Bean Framework to the Hello_Client Project

You need to add the Hello bean framework to the Hello_Client project in order to use the services provided the Hello enterprise bean—mainly providing a greeting. To accomplish that, follow these steps:

1. Select the Frameworks bucket and choose Project > Add Files.
2. Navigate to the Hello project directory, select `Hello.framework`, and click Open.
3. Click Add in the search order dialog.

Creating the Container Configuration Files

To create the configuration files that the client application needs to interact with its environment, you need to run an application named OpenEJBTool, whose launch script is located in `/Apple/Library/WebObjects/JavaApplications/OpenEIBTool.woa`.

Using the Bourne shell, execute the following commands:

```
cd /Apple/Library/WebObjects/JavaApplications/OpenEJBTool.woa

./OpenEJBTool.cmd -o c:/<Hello_Client_path>
                  c:/<Hello_path>Hello.framework
```

When the tool is finished, you need to add the configuration files it generated (`OpenEJBConfiguration.xml` and `TransactionManagerConfiguration.xml`) to the Resources bucket of the Hello_Client project.

Note: You have to run OpenEJBTool manually every time you add bean frameworks to your project or when the deployment descriptor file in any of the bean frameworks your project uses changes.

Adding Business Logic to the Client Application

You have generated a WebObjects application that, when run, instantiates its own EJB container. This container behaves like a standard EJB container. To access bean instances, you use standard EJB methods.

Modify `Session.java`

Here you edit `Session.java` so that each new session creates a Hello proxy and provides access to it to components.

First, add these `import` statements:

```
import my.ejb.Hello;
import my.ejb.HelloHome;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
```

Now, add two instance variables: one to hold a Hello bean object and another to hold a Hello home-interface object.

```
// Holds a Hello bean instance.
protected Hello hello;

// Holds a Hello bean home-interface object.
private HelloHome _helloHome;
```

Modify the `Session` constructor so that it looks like this:

```
public Session() {
    super();

    // Instantiate a HelloBean object.
    try {
        hello = helloHome().create();

    } catch (RemoteException re) {
```

```
        re.printStackTrace();
    } catch (CreateException ce) {
        ce.printStackTrace();
    }
}
```

Finally, add the following method:

```
/**
 * Obtains HelloBean's home interface.
 * @return HelloBean's home interface.
 */
public HelloHome helloHome() {
    if (_helloHome == null) {
        try {
            Context jndiContext = new InitialContext();
            _helloHome =
(HelloHome)PortableRemoteObject.narrow(jndiContext.lookup("HelloBean"),
HelloHome.class);

        } catch (NamingException ne) {
            ne.printStackTrace();
        }
    }

    return _helloHome;
}
```

Modify `Main.wo`

Open `Main.wo` in WebObjects Builder by double-clicking `Main.wo`, which is located under the Web Components bucket.

Add a String key called `greeting` to `Main.wo` through the Edit Source pop-up menu.

Add a WOString element to the component, and bind it to the `greeting` key.

Modify Main.java

When a Main page is about to be displayed, the Main object needs to invoke the message method of its Hello bean proxy to obtain the bean's greeting, and store the value returned in its greeting instance variable. When the WOString element is rendered on the page, its value binding provides the text to be displayed; in this case, the value comes from greeting in the Main object.

Add the following import statements to Main.java:

```
import com.my.ejb.Hello;
import java.rmi.RemoteException;
```

Edit the Main constructor so that it looks like this:

```
public Main(WOContext context) {
    super(context);

    Session session = (Session)session();

    try {
        greeting = session.hello.message();

    } catch (RemoteException re) {
        re.printStackTrace();
    }
}
```

Configuring the Container

This simple session bean project doesn't make use of bean persistence. Therefore, it requires no database configuration. You need to edit the TransactionManagerConfiguration.xml file of the project to remove extraneous data-source configuration information, which is everything between the <resources> and </resources> tags, and the tags themselves.

After removing the irrelevant information, the TransactionManagerConfiguration.xml file should look like this

```
<domain>
  <name>default</name>
</domain>
```

Running the Hello_Client Application

After you build and run the application, you should see a Web browser window with the message “Hello, World.”

CHAPTER 4

Developing Entity Beans

This chapter guides through the development an entity-bean framework based on a data model and its use in an application. It also shows you the entity-bean source files that the Project Builder Assistant can generate for you, which include source files for CMP (container-managed persistence) entity beans, BMP (bean-managed persistence) entity beans, and DAO (Data Access Object) source files used to customize data-store access.

The chapter contains the following sections:

Developing an Entity Bean From a Data Model

This section shows how to use EOBeanAssistant to create an enterprise-bean framework based on entities defined in a model file created using EOModeler.

As part of this document's companion files is a simple model file named `Person.eomodeld`, located in the `models` directory.

Creating an Empty Bean Framework

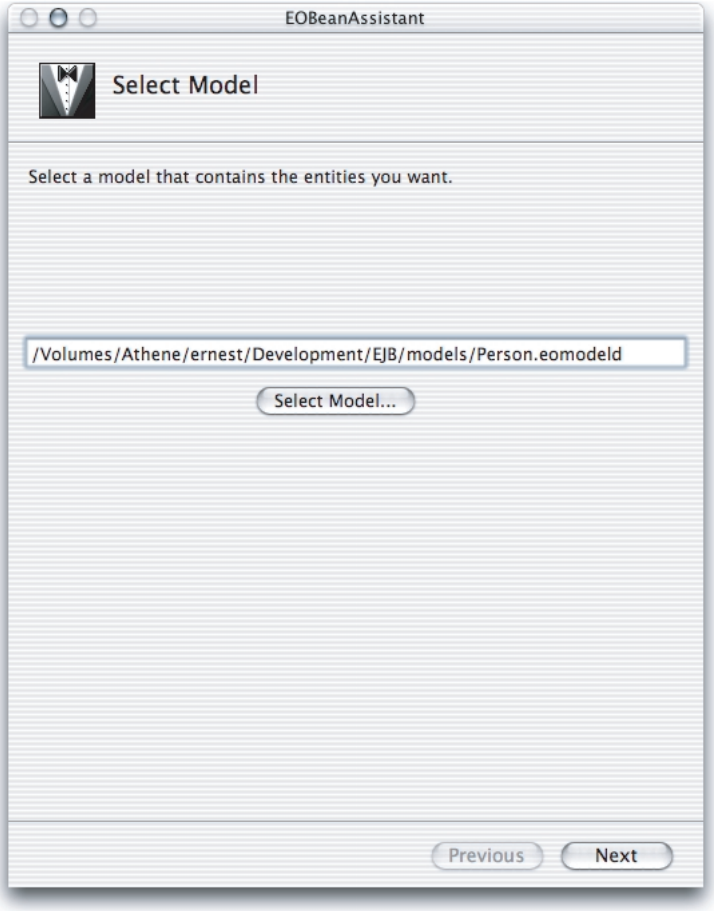
Before generating the source files for an entity bean, there needs to be a project to which you add the files to. You can create an empty bean-framework project or you may add the generated files to an existing bean-framework project. This section walks you through the creating an empty bean framework.

1. In Project Builder, choose File > New Project.
2. Choose Enterprise JavaBean Framework as the project type.
3. Name the project `Person`.
4. Make sure "Empty Enterprise JavaBean framework" is selected in the Create New Enterprise JavaBean pane of the Project Builder Assistant.

Generating Enterprise-Bean Source Files From a Model File Using EOBeanAssistant

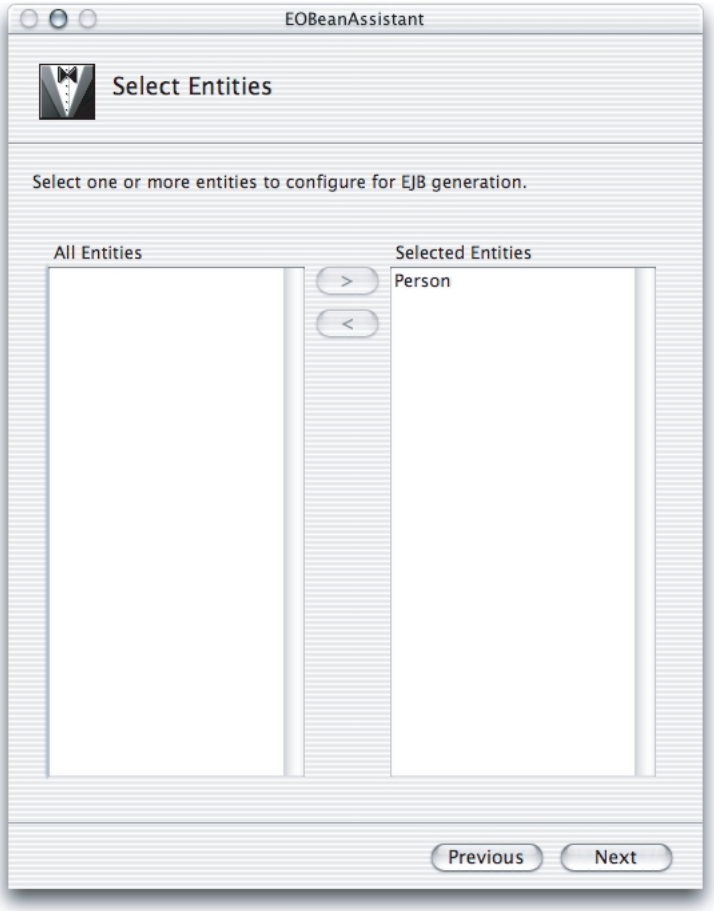
EOBeanAssistant is an application that creates all the Java source files and the `ejb-jar.xml` file you need to implement an entity bean. It's located in `/Developer/Applications`. Figure 4-1 shows the Select Model pane of EOBeanAssistant. Enter the path to your model in the text field or click Select Model and navigate to it.

Figure 4-1 The Select Mode pane of EOBeanAssistant



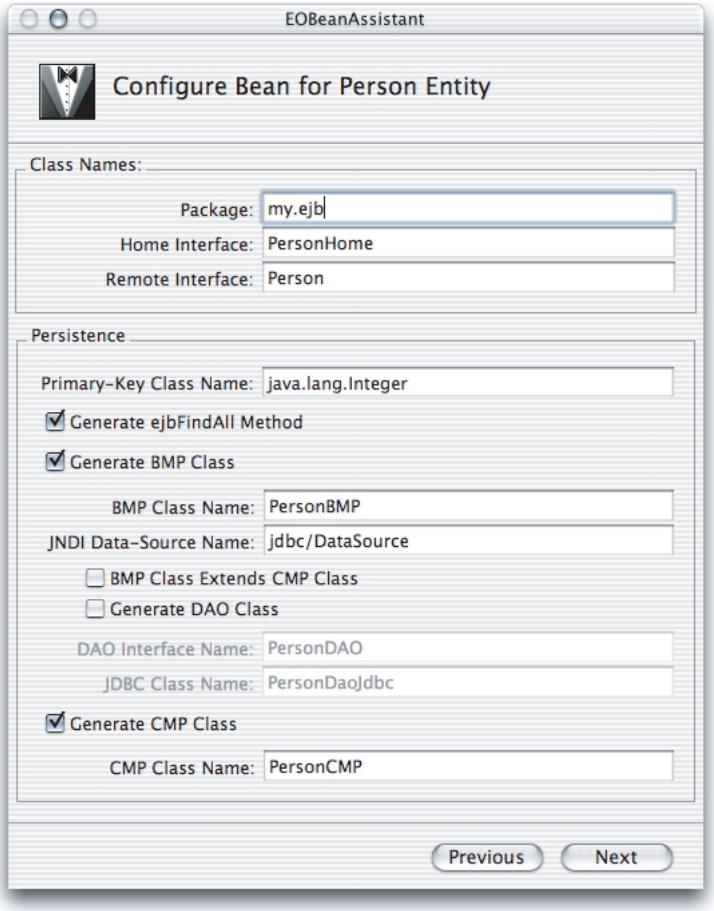
In the Select Entities pane (Figure 4-2), select the entity or entities you want to generate enterprise-bean source files for and click the right-pointing arrow so that they move from the box on the left to the one on the right.

Figure 4-2 The Select Entities pane of EOBeanAssistant



The Configure Bean pane (Figure 4-3) provides several options. Make sure Generate `ejbFindAll` Method, Generate BMP Class, and Generate CMP Class are selected.

Figure 4-3 The Configure Bean pane of EOBeanAssistant



Package
: The package name to use in the Java source files and the deployment descriptor.

Home Interface
: The name of the home-interface class of the entity bean.

Remote Interface
: The name of the remote-interface class of the entity bean.

Primary Key Class Name
: The data type you want to use for the primary key of the entity bean.

Generate `ejbFindAll` Method
: When selected, EOBeanAssistant generates `findAll` and `ejbFindAll` methods.

Generate BMP Class
: When selected, EOBeanAssistant generates a BMP-source file for the entity bean.

BMP Class Name
: The name of the BMP-source file.

JNDI Data-Source Name
: The name of the JNDI (Java Naming and Directory Service) data source that identifies the data store.

BMP Class Extends CMP Class
: When selected, the BMP-class file generated by EOBeanAssistant extends the CMP class.

Generate DAO Class
: When selected, EOBeanAssistant generates a DAO class files that implement database-specific logic.

DAO Interface Name
: The name of the DAO interface.

JDBC Class Name
: The name of the JDBC class used to communicate with the data store.

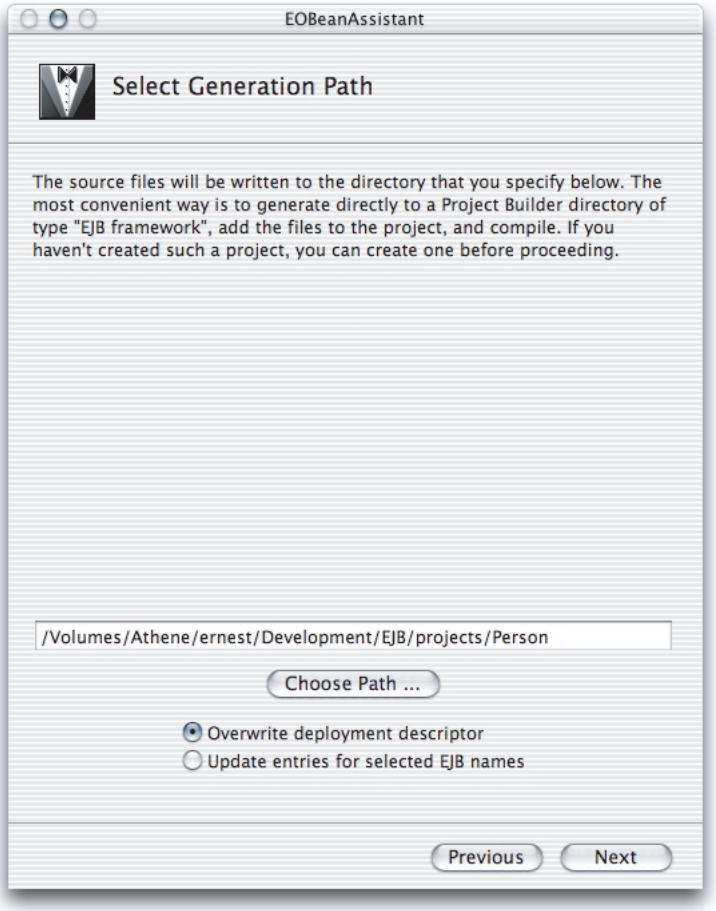
Generate CMP Class
: When selected, EOBeanAssistant generates a CMP-class file for the entity bean.

CMP Class Name
: The name of the CMP-class file.

The Common Bean Configuration pane allows you to enter header information that you want all the source files generated to contain.

The Select Generation Path pane (Figure 4-4) allows you to enter or choose the path of the bean-framework project folder you want to add the generated source files to. You can choose to overwrite the existing `ejb-jar.xml` file or to add or replace only the entries corresponding to the added entity beans.

Figure 4-4 The Select Generation Path pane of EOBeanAssistant



Listing 4-1 shows the files that EOBeanAssistant adds to the Person project directory.

Listing 4-1 Enterprise bean files added by EOBeanAssistant to the Person project directory

```
Person/
```

```
EOBeanBuilder.xml
Person.java
PersonBMP.java
PersonCMP.java
PersonHome.java
META-INF/
    ejb-jar.xml
```

Using an Entity-Bean Framework

In this section you create an application that uses the entity-bean framework developed in "Developing an Entity Bean From a Data Model" (page 45).

Before you can successfully run the application, you must create the Person database. Use OpenBaseManager to import the database in `/Developer/Documentation/WebObjects/Enterprise_JavaBeans/databases`. Alternatively, you can do the following:

1. Create a database named `Person` using OpenBaseManager.
    a. Launch OpenBaseManager, which is located in `/Applications/OpenBase/OpenBaseManager`.
    b. Choose Database > New.
    c. In the Configure Database dialog, enter `Person` in the Database Name text input field, select Start Database at Boot, choose ASCII from the Internal Encoding pop-up menu, and click Set.
    d. In the OpenBaseManager main window, select the Person database under localhost and click Start Database.
2. Add the PERSON table to the Person database.
    a. Open `/Developer/Documentation/WebObjects/Enterprise_JavaBeans/models/Person.eomodeld` in EOModeler.
    b. Choose Property > Generate SQL.
    c. Make sure only the Create Tables option is selected in the SQL Generation dialog and click Execute SQL.

Developing the Application Project

Follow these steps to develop the client-application project:

1. Create a project named `Person_Client`.
2. In the J2EE Integration pane of the Project Builder Assistant, select "Deploy as Enterprise JavaBean container."
3. In the Chose EOAdaptors pane of the Assistant, deselect the JDBC adaptor.
4. In the Choose Frameworks pane, add `Person.framework`, which is located in the build directory of the Person project directory.

Defining Data Sources

In Project Builder, select `GlobalTransactionConfiguration.xml` under the Resources group. You should see the file shown in Listing 4-2.

Listing 4-2 Person_Client project—`GlobalTransactionConfiguration.xml` file

```
<!DOCTYPE databases PUBLIC "-//EXOLAB/Castor JDO Configuration DTD Version
1.0//EN"
                          "http://www.apple.com/webobjects/5.2/DTDs/jdo-
conf.dtd">
<database name="Global_TX_Database" engine="oracle">                    //1
    <jndi name="java:comp/env/jdbc/DefaultCMPDatasource" />
    <mapping href="Contents/Resources/CMPConfiguration.xml" />
</database>
```

In the line numbered 1, change the value of the `engine` attribute to `"generic"`.

For more information on data-source definition, see "Defining Data Sources" (page 92).

Mapping Enterprise Beans to Data-Store Tables

Select `CMPConfiguration.xml` under the Resources group of the Groups & Files list. You should see the file shown in Listing 4-3.

Listing 4-3 Person_Client project—`CMPConfiguration.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                    "http://www.apple.com/webobjects/5.2/DTDs/mapping.dtd">
<mapping>
    <class identity="person_ID" name="my.ejb.PersonCMP">
        <map-to table="*** MARKER table name ***"/>                    //1
        <field direct="true" name="personName" type="java.lang.String">
            <sql name="personName" type="varchar"/>                    //2
        </field>
        <field direct="true" name="person_ID" type="java.lang.Integer">
            <sql name="person_ID" type="integer"/>                     //3
        </field>
    </class>
</mapping>
```

Change the numbered lines according to these instructions:

1. Set the `table` attribute of the `map-to` element to `"PERSON"`.
2. Set the `name` attribute of the `sql` element to `"PERSON_NAME"`.
3. Set the `name` attribute of the `sql` element to `"PERSON_ID"`.

For more information on mapping enterprise beans to tables, see "Mapping Enterprise Beans to Data-Store Tables" (page 87).

Configuring the Transaction Manager

Select `TransactionManagerConfiguration.xml` and edit it so that it looks like Listing 4-4.

Listing 4-4 Person_Client project—`TransactionManagerConfiguration.xml` file

```
<domain>
  <name>default</name>
   <resources>
     <dataSource>
        <name>DefaultDatabase</name>
```

```
        <class>tyrex.resource.jdbc.xa.EnabledDataSource</class>
        <jar>/Library/Java/Extensions/OpenBaseJDBC.jar</jar>
        <config>
            <driverName>jdbc:openbase://localhost/Person</driverName>
            <driverClassName>com.openbase.jdbc.ObDriver</driverClassName>
            <transactionTimeout>60</transactionTimeout>
            <isolationLevel>Serializable</isolationLevel>
        </config>
        <limits>
            <maximum>100</maximum>
            <minimum>10</minimum>
            <initial>10</initial>
            <maxRetain>300</maxRetain>
            <timeout>50</timeout>
        </limits>
    </dataSource>
  </resources>
</domain>
```

For more information on transaction manager configuration, see "Transaction Manager Configuration" (page 85).

Creating, Retrieving, and Removing Person Beans

Now, add the application's business logic. Select `Application.java` under Classes in the Groups & Files list and modify it to match Listing 4-5.

Listing 4-5 Person_Client project—`Application.java` file

```
import my.ejb.Person;
import my.ejb.PersonHome;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```
import javax.rmi.PortableRemoteObject;

import com.webobjects.foundation.*;
import com.webobjects.appserver.*;

public class Application extends WOApplication {

    private PersonHome _personHome = null;

    public static void main(String argv[]) {
        WOApplication.main(argv, Application.class);
    }

    public Application() {
        super();
        System.out.println("Welcome to " + this.name() + "!");

        // Create Person records.
        createPeople();

        // Show Person records.
        showPeople();

        // Remove Person records.
        removePeople();
    }

    /**
     * Creates Person records.
     */
    public void createPeople() {
        System.out.println();
        System.out.println("Creating Person records.");

        String[] names = {"Susana", "Charles", "Maria", "August"};
        NSArray personNames = new NSArray(names);
        for (int id = 1; id <= personNames.count(); id++) {
            addPerson(id, (String)personNames.objectAtIndex(id - 1));
        }
        System.out.println();
    }
```

```
/**
 * Displays Person records.
 */
public void showPeople() {
    System.out.println("Showing Person records:");
    for (int id = 1; ; id++) {
        try {
            Integer personID = new Integer(id);
            Person person = personHome().findByPrimaryKey(personID);
            System.out.println("Name: " + person.getPersonName());
        }
        catch (FinderException fe) {
            break;
        }
        catch (RemoteException re) {
            re.printStackTrace();
        }
    }
    System.out.println();
}

/**
 * Removes Person records.
 */
public void removePeople() {
    int id = 1;
    System.out.println("Removing Person records:");
    while (removePerson(id++));
    System.out.println();
}

/**
 * Adds a Person record.
 */
public void addPerson(int id, String name) {
    try {
        Integer personID = new Integer(id);
        if (personIDIsAvailable(personID)) {
            Person person = personHome().create(personID, name);
            System.out.println("Added " + name + ".");
```

```
            }
            else {
                System.out.println(name + " not added because ID " + personID + " is in
use.");
            }
        }
        catch (RemoteException re) {
            re.printStackTrace();
        }
        catch (CreateException ce) {
            // Unable to create record: Do nothing.
        }
    }

    /**
     * Removes a Person record.
     * @return <code>true</code> when successful, <code>false</code> otherwise.
     */
    public boolean removePerson(int id) {
        boolean removed = false;
        try {
            Integer personID = new Integer(id);

            // FinderException is thrown when the record doesn't exist.
            Person person = personHome().findByPrimaryKey(personID);

            System.out.println("Deleting " + person.getPersonName() + ".");
            personHome().remove(personID);
            removed = true;
        }
        catch (FinderException fe) {
            // Record not found: Do nothing.
        }
        catch (RemoteException re) {
            re.printStackTrace();
        }
        catch (RemoveException re) {
            re.printStackTrace();
        }
       return removed;
    }
```

```
    /**
     * Determines whether a personID has been used.
     * @param personID the value to check;
     * @return <code>true</code> when personID is available,
     * <code>false</code> otherwise.
     */
    public boolean personIDIsAvailable(Integer personID) {
        boolean personID_available = true;
        try {
            personHome().findByPrimaryKey(personID);
            personID_available = false;
        }
        catch (FinderException fe) {
            // personID is available: Do nothing.
        }
        catch (RemoteException re) {
            re.printStackTrace();
            personID_available = false;
        }
        return personID_available;
    }

    /**
     * Obtains Person's home interface.
     * @return Person's home interface.
     */
    public PersonHome personHome() {
        if (_personHome == null) {
            try {
                Context jndiContext = new InitialContext();
                _personHome =
(PersonHome)PortableRemoteObject.narrow(jndiContext.lookup("PersonCMP"),
PersonHome.class);
            }
            catch (NamingException ne) {
                ne.printStackTrace();
            }
        }
        return _personHome;
    }
}
```

Build and run the application. You should see the following output in the console:

```
Welcome to Person_Client!

Creating Person records.
Added Susana.
Added Charles.
Added Maria.
Added August.

Showing Person records:
Name: Susana
Name: Charles
Name: Maria
Name: August

Removing Person records:
Deleting Susana.
Deleting Charles.
Deleting Maria.
Deleting August.
```

Advanced Entity-Bean Development

This section addresses support for advanced entity-bean development, mainly bean-managed persistence (BMP) and Data Access Objects (DAO).

Bean-Managed Persistence

CMP beans are easy to use because the bean container performs the data-store operations; you don't need to worry about executing SQL statements. However, you can use BMP beans when you need to customize the way your beans store and retrieve data from a data store.

If you select Generate BMP Class in the Configure Bean pane of EOBeanAssistant when you develop an entity bean from a data model, you get a file similar to the one listed in Listing 4-6.

Listing 4-6 PersonBMP.java file

```
package my.ejb;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import javax.ejb.*;
import java.sql.*;

public class PersonBMP implements EntityBean {
    protected DataSource _datasource;
    protected EntityContext _entityContext;
    protected java.lang.String personName;
    protected java.lang.Integer person_ID;

    /**
     * Empty constructor as required by the EJB spcification.
     */
    public PersonBMP() {
    }

    /**
     * This method creates a new entity from all required (e.g. non-NULL)
     * attributes.
     * @returns the primary key for this bean instance.
     * @throws javax.ejb.CreateException
     */
    public java.lang.Integer ejbCreate(java.lang.Integer person_ID) throws
CreateException{
        this.person_ID = person_ID;
        Connection connection = null;
        PreparedStatement statement = null;
        try {
            String sqlString = "INSERT INTO PERSON (PERSON_ID) VALUES (?)";
            connection = _datasource.getConnection();
            statement = connection.prepareStatement(sqlString);
```

```
            statement.setInt(1, person_ID.intValue());
            int ret=statement.executeUpdate();
            if ( ret != 1 ) {
                throw new CreateException();
            }
        }
        catch (SQLException e) {
            throw new EJBException(e);
        }
        finally {
            cleanup(connection, statement);
        }
        return person_ID;
    }

    /**
     */
    public void ejbPostCreate(java.lang.Integer person_ID) {
    }

    /**
     * This method creates a new entity from all attributes.
     * @returns the primary key for this bean instance.
     * @throws javax.ejb.CreateException
     */
    public java.lang.Integer ejbCreate(java.lang.Integer person_ID, java.lang.String
personName) throws CreateException {
        this.personName = personName;
        this.person_ID = person_ID;
        Connection connection = null;
        PreparedStatement statement = null;
        try {
           String sqlString = "INSERT INTO PERSON (PERSON_NAME, PERSON_ID) VALUES (?,?)";
            connection = _datasource.getConnection();
            statement = connection.prepareStatement(sqlString);

            statement.setString(1, personName);
            statement.setInt(2, person_ID.intValue());
            int ret=statement.executeUpdate();
            if (ret != 1 ) {
                throw new CreateException();
```

```
        }
    }
    catch (SQLException e) {
        throw new EJBException(e);
    }
    finally {
        cleanup(connection, statement);
    }
    return person_ID;
}

/**
 */
public void ejbPostCreate(java.lang.Integer person_ID, java.lang.String personName) {
 }

/**
 * This method returns all entities in this table.
 * @returns all entities in the table in a java.util.Collection.
 * @throws javax.ejb.FinderException if there are problems connecting
 *              to the database or executing the query.
 */
public java.util.Collection ejbFindAll() throws FinderException {
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        String sqlString = "SELECT PERSON_ID FROM PERSON";
        connection = _datasource.getConnection();
        statement = connection.prepareStatement(sqlString);
        ResultSet resultSet = statement.executeQuery();
        java.util.Collection primaryKeys = new java.util.ArrayList();
        while(resultSet.next()) {
          java.lang.Integer primaryKey = new java.lang.Integer(resultSet.getInt(1));
            primaryKeys.add(primaryKey);
        }
        resultSet.close();
        return primaryKeys;
    }
    catch (SQLException e) {
        throw new EJBException(e);
    }
```

```
        finally {
            cleanup(connection, statement);
        }
    }

    /**
     * This method returns the bean associated with the primaryKey argument,
     * or throws a javax.ejb.ObjectNotFoundException.
     * @returns the bean associated with the primaryKey argument.
     * @throws javax.ejb.ObjectNotFoundException if an entity with the given
     *                   primary key doesn't exist.
     */
    public java.lang.Integer ejbFindByPrimaryKey(java.lang.Integer primaryKey) throws
FinderException {
        Connection connection=null;
        PreparedStatement statement=null;
        try {
            String sqlString = "SELECT PERSON_ID FROM PERSON WHERE PERSON_ID = ? ";
            connection = _datasource.getConnection();
            statement = connection.prepareStatement(sqlString);
            statement.setInt(1, primaryKey.intValue());
            ResultSet resultSet = statement.executeQuery();
            boolean found = resultSet.next();
            resultSet.close();
            if (found) {
                return primaryKey;
            }
            else {
                throw new ObjectNotFoundException("Could not find: " + primaryKey);
            }
        }
        catch (SQLException e) {
            throw new EJBException(e);
        }
        finally {
            cleanup(connection, statement);
        }
    }

    /**
     */
```

```
public java.lang.String getPersonName(){
   return personName;
}

/**
 */
public void setPersonName(java.lang.String personName){
   this.personName = personName;
}

/**
 */
public java.lang.Integer getPerson_ID() {
    return person_ID;
}

public void ejbRemove() throws RemoveException {
    Connection        connection = null;
    PreparedStatement statement  = null;
    try {
        String sqlString = "DELETE FROM PERSON WHERE PERSON_ID = ? ";
        connection = _datasource.getConnection();
        statement  = connection.prepareStatement(sqlString);

        statement.setInt(1, person_ID.intValue());
        if (statement.executeUpdate() != 1) {
            throw new RemoveException();
        }
    }
    catch (SQLException e) {
        throw new EJBException(e);
    }
    finally {
        cleanup(connection, statement);
    }
 }

public void ejbActivate() {
   java.lang.Integer person_ID = (java.lang.Integer) _entityContext.getPrimaryKey();
    this.person_ID = person_ID;
}
```

```
    public void ejbPassivate() {
    }

    public void ejbLoad() {
        Connection connection = null;
        PreparedStatement statement = null;
        try {
            String sqlString = "SELECT PERSON_NAME FROM PERSON WHERE PERSON_ID = ? ";
            connection = _datasource.getConnection();
            statement = connection.prepareStatement(sqlString);

            statement.setInt(1, person_ID.intValue());
            ResultSet resultSet = statement.executeQuery();
            if(!resultSet.next()) {
                resultSet.close();
               throw new NoSuchEntityException("ejbLoad failed. Primary key not found:
"+_entityContext.getPrimaryKey());
            }
            personName = resultSet.getString(1);
            resultSet.close();
        }
        catch(SQLException e) {
            throw new EJBException(e);
        }
        finally {
            cleanup(connection, statement);
        }
    }

    public void ejbStore() {
        Connection connection=null;
        PreparedStatement statement=null;
        try {
           String sqlString = "UPDATE PERSON SET PERSON_NAME = ?  WHERE PERSON_ID = ? ";
            connection = _datasource.getConnection();
            statement = connection.prepareStatement(sqlString);

            statement.setString(1, personName);
            statement.setInt(2, person_ID.intValue());
            if(statement.executeUpdate() != 1) {
```

```
                throw new NoSuchEntityException("ejbStore failed. Primary key not found:
"+_entityContext.getPrimaryKey());
            }
        }
        catch (SQLException e) {
            throw new EJBException(e);
        }
        finally {
            cleanup(connection, statement);
        }
    }

    public void setEntityContext(EntityContext entityContext) {
        _entityContext = entityContext;
        try {
            InitialContext context = new InitialContext();
            _datasource = (DataSource) context.lookup("java:comp/env/jdbc/DataSource");
        }
        catch(Exception ne) {
            throw new EJBException(ne);
        }
    }

    public void unsetEntityContext() {
        _entityContext = null;
    }

    private void cleanup(Connection connection, PreparedStatement statement) {
        if(statement != null) {
            try{
                statement.close();
            }
            catch(SQLException e) {
                // Do nothing.
            }
        }
        if(connection != null) {
            try {
                connection.close();
            }
            catch(SQLException e) {
```

```
                // Do nothing.
            }
        }
    }
}
```

Data Access Objects

When you select Generate DAO Class in the Configure Bean pane of EOBeanAssistant, you get a file similar to the one listed in Listing 4-7.

Listing 4-7 PersonDAO.java file

```
package my.ejb;

public interface PersonDAO {

    /**
     * This method creates a new entity from its required attributes.
     * @throws javax.ejb.CreateException;
     * @returns the primary key for this bean instance.
     */
    public java.lang.Integer ejbCreate(java.lang.Integer person_ID) throws
javax.ejb.CreateException;

    /**
     * This method creates a new entity from all attributes.
     * @throws javax.ejb.CreateException;
     * @returns the primary key for this bean instance.
     */
    public java.lang.Integer ejbCreate(java.lang.Integer person_ID, java.lang.String
personName) throws javax.ejb.CreateException;



    /**
     * This method returns the bean associated with the primaryKey argument,
     * or throws a javax.ejb.ObjectNotFoundException.
     * @throws javax.ejb.ObjectNotFoundException if an entity with the given
```

```
     *               primary key doesn't exist;
     * @returns the bean associated with the primaryKey argument.
     */
    public java.lang.Integer ejbFindByPrimaryKey(java.lang.Integer primaryKey) throws
javax.ejb.FinderException;

    public void ejbRemove() throws javax.ejb.RemoveException;

    public void ejbLoad();

    public void ejbStore();

    public void setBeanInstance(PersonBMP bean, javax.sql.DataSource datasource);

}
```

CHAPTER 4

Developing Entity Beans

CHAPTER 5

Developing Bean Frameworks

This chapter tells you how to create enterprise-bean frameworks to be used by client applications. It contains the following sections:

Adding Source Files to a Bean- Framework Project

To add new enterprise-bean source files to an existing enterprise-bean framework project, follow these steps:

1. Choose File > New File.
2. Choose Enterprise Java Bean from the New File pane of the Project Builder Assistant.
3. In the New Enterprise Java Bean pane of the Assistant:
   a. Enter the name of the bean in the File Name text field.
   b. Enter the location where you want to place the bean's source files in the Location text field.

c. Choose the project you want to add the bean to from the Add to Project pop-up menu.

4. Select "Create source files for a new Enterprise Java Bean" in the Create New Enterprise Java Bean pane.
5. Select the type of bean you want to create in the Choose Bean Type pane of the Assistant.
6. In the Enterprise Java Bean Class Name pane:

   a. Enter the class name of the bean in the Class Name text field.

   b. Enter the package name in the Package Name text field.

Adding JAR Files to a Bean Framework Project

To add an enterprise-bean JAR file to an existing enterprise-bean framework project follow these steps:

1. Choose File > New File.
2. Choose Enterprise JavaBean form the New File pane of the Project Builder Assistant.
3. In the New Enterprise JavaBean pane of the Assistant:

   a. Enter the name of the bean in the File Name text field

   b. Choose the project you want to add the bean to from the Add to Project pop-up menu

4. In the Create New Enterprise JavaBean pane:

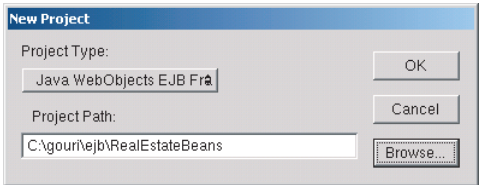
   a. Select Use JAR Files

   b. Enter the location of the JAR files you want to add in the Client Interfaces JAR and Deployment JAR text fields (client-interface JAR files contain helper classes that facilitate communication between clients and bean containers).

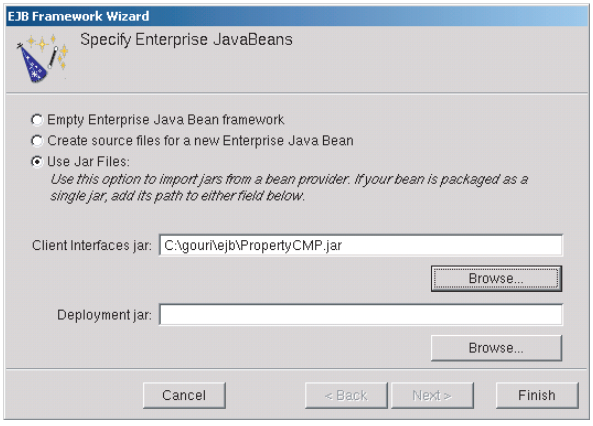
Creating Frameworks From Bean JAR Files in Windows

In Windows, you have to create one enterprise-bean framework per JAR file. Follow these steps to create an enterprise-bean framework:

1. Launch Project Builder.
2. Choose Project > New.
3. Choose Java WebObjects EJB Framework from the Project Type pop-up menu in the New Project dialog and enter a location for your project.

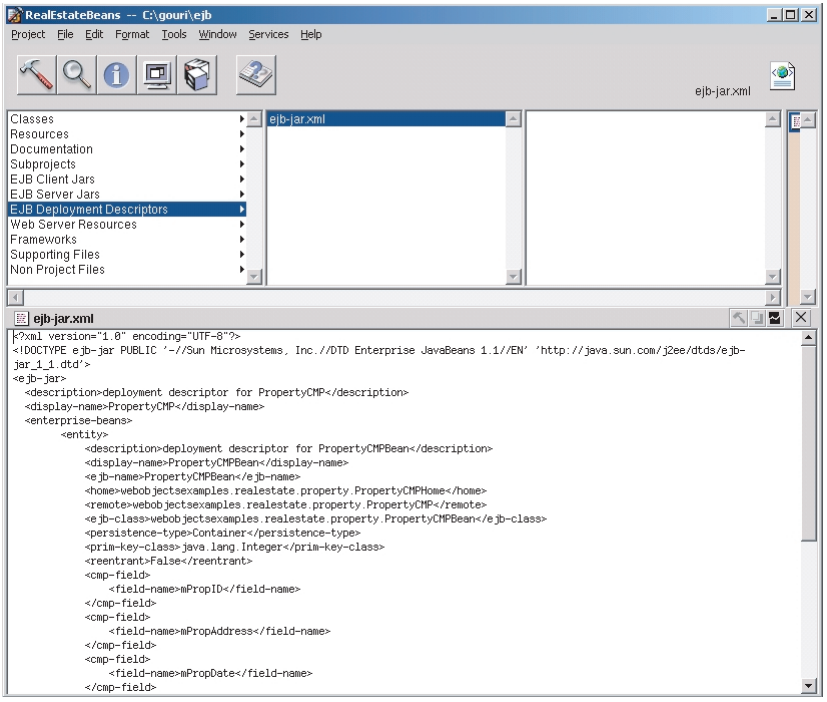


4. In the Specify Enterprise JavaBeans pane of the EJB Framework Wizard:
    a. Select "Use JAR Files."
    b. Enter the location of the JAR file you want to use in the "Client Interfaces jar" text field.



After WebObjects finishes generating the project, you should see a window like the one in Figure 5-1.

Figure 5-1 Bean framework project in Windows



Adding CMP Fields to an EJB Deployment Descriptor

After creating a bean framework using Project Builder, you have to add to the deployment descriptor the fields whose persistence is to be managed by the EJB container. To accomplish this, you add `cmp-field` elements to the `ejb-jar.xml` file in the META-INF directory of your project.

Listing 5-1 lists the deployment descriptor of a simple entity bean with container-managed persistence. The numbered lines show the `cmp-field` elements.

Listing 5-1 Deployment descriptor for a CMP entity bean

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
  <description>deployment descriptor for PersonBean</description>
  <display-name>PersonBean</display-name>
  <enterprise-beans>
        <entity>
            <description>deployment descriptor for PersonBean</description>
            <display-name>PersonBean</display-name>
            <ejb-name>PersonBean</ejb-name>
            <home>com.my.ejb.PersonHome</home>
            <remote>com.my.ejb.Person</remote>
            <ejb-class>com.my.ejb.PersonBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>com.my.ejb.PersonPK</prim-key-class>
            <reentrant>False</reentrant>
            <resource-ref>
                <description>the default data source for a CMP bean.</description>
                <res-ref-name>jdbc/DefaultCMPDatasource</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
            <cmp-field>                                                    //1
              <field-name>PersonID</field-name>                            //2
            </cmp-field>                                                   //3
            <cmp-field>                                                    //4
              <field-name>firstName</field-name>                           //5
            </cmp-field>                                                   //6
            <cmp-field>                                                    //7
              <field-name>lastName</field-name>                            //8
            </cmp-field>                                                   //9
```

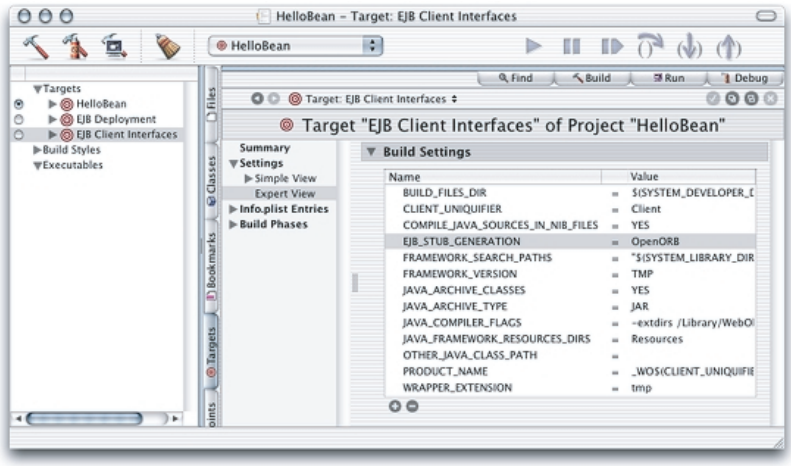
```
            <cmp-field>                                                    //10
              <field-name>middleInitial</field-name>                       //11
            </cmp-field>                                                   //12
            <cmp-field>                                                    //13
              <field-name>dateOfBirth</field-name>                         //14
            </cmp-field>                                                   //15
        </entity>
    </enterprise-beans>
</ejb-jar>
```

Generating EJB Stubs

When you build a bean-framework project you have the option of generating the EJB stubs or not generating them. The `EJB_STUB_GENERATION` build setting of the EJB Client Interfaces target is how you tell Project Builder whether to create the stubs. The build setting can have two values: `OpenORB` and `None`. By default, the build setting is set to `OpenORB`. This means that stubs are generated. By setting the build setting to `None`, you tell project builder not to generate the stubs. This makes building bean-frameworks faster. However, this setting requires that the `WOEJBTransport` property be set to `IntraVM`. For more on the `WOEJBTransport` property, see "Communication Transport Between Bean Clients and Containers" (page 98).

You access the `EJB_STUB_GENERATION` build setting through the Expert View of the EJB Client Inte.rfaces target, as shown in Figure 5-2.

Figure 5-2 Viewing the value of the `EJB_STUB_GENERATION` build setting



In Windows, you find the `EJB_STUB_GENERATION` build setting in the `Makefile.preamble` file of the EJBServer subproject.

In bean-framework projects created with a version of WebObjects earlier than 5.2, you may have to add the build setting yourself. Just click “Add new build setting” in the lower-left corner of the Build Settings pane.

CHAPTER 6

Configuring Applications

This chapter shows you how to configure a bean-client application project so that it can use resources like databases and JavaMail connections, explains how to ensure that each enterprise bean is bound to the appropriate resources, and helps you improve the performance of your applications through the use of the `WOEJBTransport` property.

You need to configure three major items before deploying a bean-client application:

- **Transaction manager.** This is where you define the data sources that your enterprise beans use to store their data and the JavaMail connections they use for messaging.
- **Persistence manager.** Here you map the fields of your CMP (container-managed persistence) beans to columns in tables of your data stores so that the container can perform database transactions for the beans.
- **EJB container.** This is where you set bean-deployment properties such as method transactions and permissions.

You perform this configuration by editing the files in Table 6-1.

Table 6-1 The configuration files of a bean-client application

Filename	Purpose
`TransactionManagerConfiguration.xml`	Defines data sources and JavaMail connections.
`GlobalTransactionConfiguration.xml`	Defines the JNDI name of a remote data store.
`CMPConfiguration.xml`	Defines bean-to-table and field-to-column mapping.
`OpenEJBConfiguration.xml`	Defines enterprise-bean deployment behavior for the container.

The chapter is divided in the following sections:

- "Configuration Overview" (page 81) provides you with a checklist of items you need to review in the configuration files that WebObjects creates by default.
- "Transaction Manager Configuration" (page 85) explains how to configure the transaction manager.
- "Persistence Manager Configuration" (page 86) tells you how to map enterprise-bean fields to table columns.
- "Container Configuration" (page 93) explains how you configure the EJB container.
- "Using External Containers" (page 96) shows you how to configure your client application to use an external EJB container.
- "Communication Transport Between Bean Clients and Containers" (page 98) tells you how communication between client applications and bean containers can be streamlined.

Configuration Overview

This section gives you a quick look at the configuration process for bean-client applications. It lists the major points you need to look at before deploying your application. It's divided in the following sections:

- "Configuring the Transaction Manager" (page 81).
- "Configuring the EJB Container" (page 82).
- "Configuring the Persistence Manager" (page 84).

Configuring the Transaction Manager

The `TransactionManagerConfiguration.xml` file is where you enter connection information, such as user name and password, for the data stores that your application's CMP beans use. You also configure JavaMail. The `OpenEJBConfiguration.xml` file determines what you need to configure in this file: the data stores, or JavaMail.

If your enterprise beans use container-managed persistence (the `OpenEJBConfiguration.xml` file contains the string "`<res-type>javax.sql.DataSource</res-type>`"), you need to configure at least one data source.

Listing 6-1 Example `dataSource` element of the `TransactionManagerConfiguration.xml` file

```
<dataSource>
    <name>DefaultDatabase</name>
    <class>tyrex.resource.jdbc.xa.EnabledDataSource</class>
    <jar>file:/Users/Shared/JDBCDrivers/oracle/oracle8.1.7.1.zip</jar>
    <config>
        <driverName>jdbc:oracle:thin:@xsrv3.apple.com:1521:sqa</driverName>
        <driverClassName>oracle.jdbc.OracleDriver</driverClassName>
        <user>ejb</user>
```

```
        <password>ejb</password>
    </config>
    <limits>
        <maximum>100</maximum>
        <minimum>10</minimum>
        <initial>10</initial>
        <maxRetain>300</maxRetain>
        <timeout>50</timeout>
    </limits>
</dataSource>
```

If your enterprise beans do not use container-managed persistence, you need to delete the `resources` element from the `TransactionManagerConfiguration.xml` file, which includes everything between the `<resources>` and `</resources>` tags as well as the tags themselves.

If you need to define more than one data store, you can add `dataSource` elements for each additional data store. See "Transaction Manager Configuration" (page 85) for more information.

If your enterprise beans make use of JavaMail (the `OpenEJBConfiguration.xml` file contains the string "`<resource-type>javax.mail.Session</resource-type>`"), you need to configure JavaMail.

To configure JavaMail in the `TransactionManagerConfiguration.xml` file, add this to the data-source section and customize as necessary:

```
<javamail>
    <name>DefaultSMTPServer</name>
    <property>
        <key>mail.smtp.host</key>
        <value>post.office.com</value>
    </property>
</javamail>
```

Configuring the EJB Container

Once you have defined the resources that your enterprise beans utilize, you have to review the container environment that WebObjects defined for you in the `OpenEJBConfiguration.xml` file:

- Data sources and JavaMail-connection factories.

If your enterprise beans use more than one data source or rely on JavaMail (as defined in the `TransactionManagerConfiguration.xml` file, you have to make sure that each bean is linked to the appropriate data source or JavaMail connection factory (through the `res-id` element inside `resource-ref`) in the `OpenEJBConfiguration.xml` file. See "resource-ref element" (page 133).

- Environment entries.

  Scan the file for `env-entry` elements and make sure that they contain the appropriate values for your situation. See "env-entry element" (page 125).

- Method-transaction settings.

  Make sure that the `trans-attribute` element of `method-transaction` elements is set to the appropriate transaction type. If the enterprise bean does not define the transaction type for a method, WebObjects sets it to `Required`. See "Containers Section" (page 93), and "method-transaction element" (page 130) for details.

- One `entity-container` element per database.

  When your CMP beans use more than one database, you need to

  - group the CMP beans that use the same data store under the same `entity-container` element
  - create a global transaction manager configuration file by duplicating the `GlobalTransactionConfiguration.xml` file and changing "`Global_TX_Database`" so that it names the additional data source (for example, "`Global_TX_Personnel`")

  In general, you should use the following grouping:

  - one `entity-container` element per distinct database that encloses its corresponding CMP beans (for more information, see "entity-container element" (page 124))
  - one `stateless-session-container` element that encloses all stateless beans
  - one `stateful-session-container` element that encloses all stateful beans

Configuring the Persistence Manager

This section explains how to configure the container for CMP beans. If your application doesn't use CMP beans, you don't need to configure the files mentioned here. In fact, the files are only present in your project when at least one of your enterprise beans uses container-managed persistence.

GlobalTransactionConfiguration.xml

This is where you define the JNDI name of a remote data store. It must be identical to the name used in the `resource-ref` element of a bean in the `CMPConfiguration.xml` file. You must have one global-transaction configuration file per data store.

CMPConfiguration.xml

This is where you map enterprise beans to tables and their fields (or instance variables) to columns in those tables. You also define a bean's identity or primary key and configure key-value generators. This is an example of a `CMPConfiguration.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                         "http://castor.exolab.org/mapping.dtd">
<mapping>
    <class key-generator="MAX" identity="mPropID"
name="webobjectsexamples.realestate.property.PropertyCMPBean">
        <map-to table="EJB_PROPERTY"/>
        <field direct="true" name="mPropID" type="java.lang.Integer">
            <sql name="PROP_ID" type="integer"/>
        </field>
        <field direct="true" name="mPropAddress" type="java.lang.String">
            <sql name="PROP_ADDR" type="varchar"/>
        </field>
        <field direct="true" name="mPropDate" type="java.util.Date">
            <sql name="PROP_LIST_DATE" type="date"/>
        </field>
        <field direct="true" name="mPropPrice" type="float">
            <sql name="PROP_ASK_PRICE" type="real"/>
        </field>
    </class>
</mapping>
```

For more information, see "Persistence Manager Configuration" (page 86).

Transaction Manager Configuration

WebObjects includes the Tyrex transaction manager. You configure it through the `TransactionManagerConfiguration.xml` file.

The only item you need to configure for the transaction manager is the domain. A transaction domain provides centralized management of transactions. It defines the policy for all transactions created from that domain, such as default timeout, maximum number of open transactions, support, and journaling. In addition, the domain maintains resource managers such as JDBC data sources and JCA (J2EE Connector Architecture) connectors.

Note: If your application uses only session beans and does not need to access a data store, you must remove the `resources` element from the `TransactionManagerConfiguration.xml` file.

This is an example of a `TransactionManagerConfiguration.xml` file:

```
<domain>
  <name>default</name>
   <resources>
     <dataSource>
        <name>DefaultDatabase</name>
        <class>tyrex.resource.jdbc.xa.EnabledDataSource</class>
        <jar>/Library/Java/Extensions/OpenBaseJDBC.jar</jar>
        <config>
            <driverName>jdbc:openbase://localhost/Person</driverName>
            <driverClassName>com.openbase.jdbc.ObDriver</driverClassName>
            <transactionTimeout>60</transactionTimeout>
            <isolationLevel>Serializable</isolationLevel>
        </config>
        <limits>
            <maximum>100</maximum>
            <minimum>10</minimum>
            <initial>10</initial>
```

```
            <maxRetain>300</maxRetain>
            <timeout>50</timeout>
        </limits>
    </dataSource>
  </resources>
</domain>
```

For details on how to write the transaction manager configuration file, see "Elements of the Transaction Manager Configuration File" (page 114).

Persistence Manager Configuration

Container-managed persistence is handled by the Castor JDO component. It generates SQL statements that the container uses to update your database. All you have to do is map your data store's table columns to entity beans' fields.

The persistence manager configuration files specify how the persistence manager obtains a connection to a data source, the mapping between Java classes and tables in the data store, and the service provider to use to talk to the data store.

These are supported database servers:

- Generic JDBC engine
- Oracle 7 and Oracle 8
- Sybase 11 and SQL Anywhere
- Microsoft SQL Server
- DB/2
- PostgreSQL 6.5 and 7
- Hypersonic SQL
- InstantDB
- Interbase
- MySQL
- SAP DB

You configure the persistence manager by editing three files:

- `CMPConfiguration.xml`

  This file defines the correspondence between table columns and the fields of your enterprise beans. It also defines how CMP beans are made persistent. This mapping is used in global transaction configuration files.

- `GlobalTransactionConfiguration.xml`

  This file defines the configuration that the persistence manager uses when a client uses an enterprise bean with a transaction context. This configuration requires that the data source be specified in the JNDI registry. The persistence manager creates the data source connection, which can be used in bean-managed as well as container-managed persistence beans.

Mapping Enterprise Beans to Data-Store Tables

One of the tasks you need to perform to accomplish bean persistence is to map the enterprise bean fields to be persisted to table columns or other types of permanent storage. You accomplish this by editing the `CMPConfiguration.xml` file.

The Mapping File

The mapping information you enter in the `CMPConfiguration.xml` file is written from the point of view of the enterprise bean and describes how the contents of the bean's fields are translated to and from permanent storage.

This is an example of the contents of the `CMPConfiguration.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                         "http://castor.exolab.org/mapping.dtd">
<mapping>
    <class key-generator="MAX" identity="mPropID"
name="webobjectsexamples.realestate.property.PropertyCMPBean">
        <map-to table="EJB_PROPERTY"/>
        <field direct="true" name="mPropID" type="java.lang.Integer">
            <sql name="PROP_ID" type="integer"/>
        </field>
        <field direct="true" name="mPropAddress" type="java.lang.String">
            <sql name="PROP_ADDR" type="varchar"/>
```

```
        </field>
        <field direct="true" name="mPropDate" type="java.util.Date">
            <sql name="PROP_LIST_DATE" type="date"/>
        </field>
        <field direct="true" name="mPropPrice" type="float">
            <sql name="PROP_ASK_PRICE" type="real"/>
        </field>
    </class>
</mapping>
```

For details in how to write the `CMPConfiguration.xml` file, see "Elements of the Component-Managed Persistence Configuration File" (page 102).

Primary Keys

The persistence manager can generate the values of identity properties automatically with the key generator. When the enterprise bean's `create` method is invoked, the persistence manager sets the value of the identity property to the value obtained from the key generator. The key generator can use one of several algorithms available to generate the value. You can use generic algorithms or algorithms specific to your data source. For details on setting the algorithm to use for an enterprise bean's identity property, see "class element" (page 105) and "key-generator element" (page 110).

You can use the key generator only under the following conditions:

- The primary-key value is not determined from the arguments to the bean's `ejbCreate` method.
- The bean's identity can be determined through a single field of numeric (`byte` through `long`) or String type.

The following sections describe the key-generator algorithms you can use.

MAX

This generic algorithm fetches the maximum value of the primary key (`MAX`) and locks the record found until the end of the transaction. When the transaction ends, the value generated is `(MAX + 1)`. Because of the lock, concurrent transactions that use the same algorithm wait until the end of the original transaction to obtain a new primary-key value. Note that it is still possible to perform multiple inserts during the same transaction.

With this algorithm, duplicate-key exceptions are almost completely avoided. The only case in which they might occur is when inserting a row into an empty table because there are no rows to lock. In this case, the value generated is `1`.

This is an example of a definition of a key generator using the MAX algorithm:

```
<key-generator name="MAX">
    <param name="table" value="PERSON"/>
    <param name="key-column" value="PERSON_ID"/>
</key-generator>
```

HIGH/LOW

This generic algorithm needs an auxiliary table or sequence table containing a unique column (the key column) that stores table names and, a numeric (`integer`, `bigint`, or `numeric`) column used to reserve primary-key values.

The following table describes the parameters used by the HIGH/LOW key generator.

Table 6-2 HIGH/LOW key generator parameters

Parameter	Description	Use
`table`	Sequence-table name.	Mandatory
`key-column`	Name of the column containing table names.	Mandatory
`value-column`	Name of the column used to reserve primary-key values.	Mandatory
`grab-size`	Number of primary-key values the key generator reserves at a time.	Optional; default="`10`"
`same-connection`	Indicates whether the key generator must use the same connection when accessing the sequence table. Values: (`true` or `false`). Must be set to `true` when working in an EJB environment.	Optional; default="`false`"
`global`	Indicates whether the key generator produces globally unique keys. Values: (`true` or `false`).	Optional; default="`false`"

The first time the key generator is called, it finds the row for the target table in the sequence table, locks it, reads the last reserved primary-key value, increases it by the grab size (the number of primary-key values to reserve at a time), and unlocks the row. In subsequent requests for primary-key values for the same target table, the key generator provides primary-key values from the reserved values until it runs out. When it has no more primary-key values, it accesses the sequence table to obtain a new group of primary-key values.

Note: The sequence table must be in the same database as the table for which primary-key values are to be generated. When working with multiple databases, you must have one sequence table in each database that contains a table for which the key generator is to provide primary-key values.

If *grab-size* is set to `1`, the sequence tables contain the true maximum primary-key value at all times. In this case, the HIGH/LOW key generator is essentially equivalent to the MAX key generator.

If the *global* is set to `true`, the sequence table contains only one row instead of one row per table. The key generator uses this row for all tables.

UUID

This algorithm generates global unique primary-key values. The value generated is a combination of the host's IP address, the current time in milliseconds since 1970, and a static counter. The complete key consists of a 30-character, fixed-length string. This algorithm has no parameters. The primary-key column must be of type `char`, `varchar`, or `longvarchar`.

IDENTITY

The IDENTITY key generator can be used only with auto-increment primary-key columns (identities) in Sybase ASE/ASA, MS SQL Server, MySQL, and Hypersonic SQL.

After an insert, except when using MySQL or Hypersonic SQL, the key generator obtains the primary-key value from the `@@identity` system variable, which contains the last identity value for the current database connection. When using MySQL, the system function `LAST_INSERT_ID()` is used. For Hypersonic SQL, `IDENTITY()` is used.

SEQUENCE

This algorithm can be used with only Oracle, Oracle8i, PostgreSQL, Interbase, and SAP DB. It generates keys using sequences.

The following table describes the parameters for the SEQUENCE key generator.

Table 6-3

Parameter	Description	Use
`sequence`	Sequence name.	Optional; default="`{0}_seq`"
`returning`	RETURNING mode for Oracle8i. Values: (`true` or `false`)	Optional; default="`false`"
`increment`	Increment for Interbase.	Optional; default="`1`"
`trigger`	Indicates whether there is a trigger that generates primary-key values.	Optional; default="`false`"

Usually a sequence is used for only one table. Therefore, in general, you have to define one key generator per table. However, if you adhere to a naming convention for sequences, you can use one key generator for multiple tables.

For example, if you always obtain sequence names by adding `_seq` to the name of the corresponding table, you can set *sequence* to "`{0}_seq`" (the default).

The way this key generator performs its function depends on the data-source server being used.

With PostgreSQL, this key generator performs `SELECT nextval(sequence_name)` before the insert and produces the identity value that is then used when it performs `INSERT`.

With Interbase, the key generator performs `SELECT gen_id(sequence_name, increment) from rdb$database` before the insert.

With Oracle, by default (`returning="false"`) and with SAP DB, the key generator transforms the insert statement generated by the persistence manager to the form `INSERT INTO table_name (pk_name, ...) VALUES (sequence_name.nextval, ...)`, executes it, and then it performs `SELECT sequene_name.currval FROM table_name` to obtain the identity value.

With Oracle8i, when you set `returning` to "`true`", `RETURNING primary_key_name INTO ?` is appended to the insert statement shown above, which is a more efficient procedure to generate primary-key values. Therefore, the persistence manager fetches the identity value when it executes the insert statement (both the insertion and the procurement of the identity value occur in one statement).

If your table has an `on_Insert` trigger, like the one listed below, that already generates values for the table's primary key, you can set *trigger* to "`true`".

```
create or replace trigger "trigger_name"
before insert on "table_name" for each row
begin
    select "sequence_name".nextval into :new."pk_name" from dual;
end;
```

This prevents `"sequence_name".nextval` from being pulled twice: first during the insert and then in the trigger. It's also useful in combination with `returning="true"` for Oracle, in which case you may not specify the sequence name.

Defining Data Sources

Global data-source configuration files tell the transaction manager how to locate a data store using JNDI. They contain the mapping between enterprise beans and tables in a data store. The transaction manager then uses the information in `TransactionManagerConfiguration.xml` to create database connections.

The persistence manager can obtain a connection to a data store in one of three ways:

- using a JDBC 2.0 driver and URL
- using a JDBC 2.0 data source
- using a JNDI data source

If you are deploying the application inside a J2EE environment, you should use the JNDI method because it allows the application server to manage connection pooling and distributed transactions.

To allow for concurrent transactions and to ensure data integrity, two data-store definitions should never use overlapping mappings.

The following is the JNDI configuration of a global data store:

```
<database name="ebiz" engine="oracle">
    <jndi name="java:comp/env/jdbc/mydb"/>
    <mapping href="Contents/Resources/CMPConfiguration.xml"/>
</database>
```

Container Configuration

The `OpenEJBConfiguration.xml` file contains deployment information, as well as transaction and security details. Its contents are divided in two sections: containers, and facilities. WebObjects writes this file for you. However, you need to make additions, especially regarding the transaction type for methods and mapping physical roles to logical roles.

Containers Section

This section of the EJB configuration file holds four types of elements: `containers`, `security-role`, `method-permission`, and `method-transaction`.

The `containers` element can contain three types of elements: `stateless-session-container`, `stateful-session-container`, and `entity-container`. Each of these elements holds definitions for the corresponding types of enterprise beans: stateless session bean, stateful session bean, and entity bean (CMP and BMP).

One or more logical security roles are defined using `security-role` elements. Physical security roles are mapped to logical security roles in the facilities section of the file. You have to define the logical security roles that you want to use in your application. Then you assign those roles to the methods of the enterprise beans—using `method-permission` elements—as you see fit.

Note: WebObjects generates a default role and assigns it to all method permissions. You have to add the roles adequate for your situation and assign them to each of the methods of your enterprise beans through `method-permission` elements.

The `method-transaction` element tells the container how to manage transactions for each method invocation. You must determine what kind of transaction attribute each enterprise bean's methods should have, and modify the contents of the `method-transaction` element as appropriate. Table 6-4 provides a brief explanation of transaction attributes.

Table 6-4 Transaction attributes

Transaction attribute	Meaning
`NotSupported`	The current transaction is suspended until the method ends.
`Supports`	If in a transaction, the method is included in it.
`Required`	The method must be invoked within a transaction. Otherwise, a new transaction is created for the method.
`RequiresNew`	A new transaction is always created for the method.
`Mandatory`	The method must be invoked within a transaction. Otherwise, a `javax.transaction.TransactionRequiredException` is thrown.
`Never`	The method must never be invoked within a transaction. Otherwise, a `java.rmi.RemoteException` is thrown.

Listing 6-2 shows an example of the containers section of the EJB configuration file:

Listing 6-2 Example containers section of the `OpenEJBConfiguration.xml` file

```
<container-system>
    <containers>
        <stateless-session-container>
            <container-name>Basic Stateless Container</container-name>
            <stateless-bean>
                <description>deployment descriptor for HelloBean</description>
                <display-name>HelloBean</display-name>
                <ejb-deployment-id>HelloBean</ejb-deployment-id>
                <home>com.my.ejb.HelloHome</home>
                <remote>com.my.ejb.Hello</remote>
                <ejb-class>com.my.ejb.HelloBean</ejb-class>
                <transaction-type>Container</transaction-type>
            </stateless-bean>
        </stateless-session-container>
    </containers>
    <security-role>
        <role-name>everyone</role-name>
    </security-role>
    <method-permission>
        <role-name>everyone</role-name>
        <method>
            <ejb-deployment-id>HelloBean</ejb-deployment-id>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <method-transaction>
        <method>
            <ejb-deployment-id>HelloBean</ejb-deployment-id>
            <method-intf>Remote</method-intf>
            <method-name>message</method-name>
            <method-params/>
        </method>
        <trans-attribute>NotSupported</trans-attribute>
    </method-transaction>
</container-system>
```

Facilities Section

This section of the EJB configuration file specifies the runtime environment: proxy-generation attributes, remote JNDI contexts, database connections, and J2EE services. The elements used are `intra-vm-server`, `remote-jndi-contexts`, `connectors`, and `services`, respectively. You should not edit this part of the `OpenEJBConfiguration.xml` file.

Using External Containers

You may want to use an external EJB container instead of an internal one in your bean-client applications when you already have a powerful, reliable container. In this case, you need to remove all the configuration files listed at the beginning of this chapter from your project.

To configure your application to use a single, external EJB container, you need to set system properties when you launch your application. You can set them through the command line. The following list details the properties you need to set for various EJB containers:

- OpenEJB

```
-Djava.naming.factory.initial=org.openorb.rmi.jndi.CtxFactory
-Djava.naming.provider.url=
corbaloc::1.2@<HOST>:<NAMESERVICE_PORT>/NameService"
-Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB
-Dorg.omg.CORBA.ORBSingletonClass=org.openorb.CORBA.ORBSingleton
-Djavax.rmi.CORBA.StubClass=org.openorb.rmi.system.StubDelegateImpl
-Djavax.rmi.CORBA.UtilClass=org.openorb.rmi.system.UtilDelegateImpl
-Djavax.rmi.CORBA.PortableRemoteObjectClass=
org.openorb.rmi.system.PortableRemoteObjectDelegateImpl
```

- iPlanet

```
-Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
-Djava.naming.provider.url=iiop://$<HOST>:$<NAMESERVICE_PORT>
```

- Web Logic

```
-Djava.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
-Djava.naming.provider.url=t3://<HOST>:<NAMESERVICE_PORT>
```

- WebSphere

```
-Djava.naming.factory.initial=
com.ibm.websphere.naming.WsnInitialContextFactory
-Djava.naming.provider.url=iiop://<HOST>:<NAMESERVICE_PORT>"
```

If you want to use more than one EJB container in your application, you have to set these properties through application code. For example, to set the JNDI context for the Web Logic EJB container, you would add the following method listed in Listing 6-3.

Listing 6-3 The `initialContext` method setting external-container properties

```
/**
 * Obtains the JNDI context.
 * @return the JNDI context.
 */
public static Context initialContext() throws NamingException {
    Properties properties = new Properties();

    properties.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");

    properties.put(Context.PROVIDER_URL, "t3://<HOST>:<NAMESERVICE_PORT>");

    return new InitialContext(properties);
}
```

Communication Transport Between Bean Clients and Containers

Bean-client applications can communicate with bean containers using one of two transports: Common Object Request Broker Architecture (CORBA) or intra virtual machine. You determine how clients communicate with bean containers through the `WOEJBTransport` property. It can have one of two values: `OpenORB` for the CORBA transport or `IntraVM` for the intra-VM transport.

Using CORBA requires that bean frameworks be built with EJB stubs. See "Generating EJB Stubs" (page 76) for information on generating EJB stubs when building bean frameworks. You must use CORBA when the client application and the bean container do not run on the same virtual machine.

When you know that the bean client and the container run on the same virtual machine, you should set the `WOEJBTransport` property to `IntraVM`. This streamlines communication between bean client and container as well as facilitate debugging your enterprise beans in Project Builder.

Generating the EJB Configuration Files

After you add bean frameworks to an existing bean-client project, you need to regenerate the EJB configuration files. Also, WebObjects 5.2, bean-client projects do not require the `LocalTransactionConfiguration.xml` file. Therefore, you need delete the file from the projects and re-generate the remaining configuration files.

To re-generate the configuration files of a bean-client project, you run OpenEJBTool with the bean-client project path and the path of each of the bean frameworks it uses, as shown below.

```
% cd /System/Library/WebObjects/JavaApplications/OpenEJBTool.woa
% ./OpenEJBTool -o <bean-client_project_path> <bean-framework1_path> ...
<bean-frameworkN_path>
```

EJB Container Operation Logging

EJB-container operations are logged using Log4J, which is an open-source package that allows you to turn on logging for an application without changing its source code. Logging is configured through the `logging.conf` file, which is placed in the Resources group of a project. Listing 6-4 shows the logging.conf file. You modify this file to change the debugging level for the container. For information and documentation on Log4J, see http://jakarta.apache.org/log4jl.

Listing 6-4 `Logging.conf` file

```
# This file sets up log4j logging for the EJB container
#
# The default setup will log error messages to stdout

log4j.rootCategory=warn, R                                              //1

# Fileappender
log4j.appender.F=org.apache.log4j.FileAppender                          //2

# Edit this line to suit you application name
log4j.appender.F.file=/tmp/application.log                              //3
log4j.appender.F.layout=org.apache.log4j.PatternLayout                  //4
log4j.appender.F.layout.ConversionPattern=%5p [%t] (%C:%L) - %m%n       //5


# Console Appender
log4j.appender.R=org.apache.log4j.ConsoleAppender                       //6
log4j.appender.R.layout=org.apache.log4j.PatternLayout                  //7
log4j.appender.R.layout.ConversionPattern=%5p [%t] (%C:%L) - %m%n       //8
log4j.appender.R.Target=System.err                                      //9

# General logging for the EJB container
#log4j.category.OpenEJB=debug                                          //10
```

```
### logging for Container-Managed Persistence
#log4j.category.CastorCMP=debug                                          //11

### CORBA layer logging
#log4j.category.CORBA-Adapter=warn                                       //12

### logging of transaction handling
#log4j.category.Transactions=info                                        //13

### Transaction Manager and Connection Pool logging
#log4j.category.tyrex.default=debug                                      //14
#log4j.category.tyrex.ots=debug                                         //15
#log4j.category.tyrex.security=debug                                    //16
#log4j.category.tyrex.resource=debug                                    //17
#log4j.category.tyrex.resource.castor=debug                             //18
#log4j.category.tyrex.resource.DefaultDatabase=debug                    //19
```

The line numbered 1 configures the logging level of the root category and the output channel. In this case, `warn` tells Log4J that it should log warnings only. This setting applies to all the subcategories of `rootCategory` that do not override it. The second argument indicates which appender to use: `R` for the console output and `F` for file output.

The lines numbered 2 through 5 configure file logging. You must only change lines 3 through 5, however.

The lines numbered 6 through 9 configure console logging.

The lines numbered 10 through 19 configure the logging level of several components.

Configuration Reference

The elements (defined by tags) of an XML file can include attributes, other elements, or both. The sections below include tables that describe those elements. In the Members column, element names are between < and > characters. Table 7-1 describes the meaning of the symbols in the Use column in the tables that describe an element's members.

Table 7-1 Element usage symbols

Symbol in Use column	Meaning
Nothing	The tag or attribute is required by the parent tag.
?	The element or attribute can be omitted.
*	The element can be present zero or more times within the parent element.
+	The element must be present at least once within the parent element.

Elements of the Component-Managed Persistence Configuration File

The DTD for the `CMPConfiguration.xml` file is located at http://castor.exolab.org/mapping.dtd, and is shown in Listing 7-1.

Listing 7-1 DTD for CMPConfiguration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT mapping ( description?, include*, class*, key-generator* )>

<!ELEMENT include EMPTY>
<!ATTLIST include
    href  CDATA  #REQUIRED>

<!ELEMENT class ( description?, cache-type?, map-to?, field+ )>
<!ATTLIST class
    name            ID       #REQUIRED
    extends         IDREF    #IMPLIED
    depends         IDREF    #IMPLIED
    identity        CDATA    #IMPLIED
    access          ( read-only | shared | exclusive | db-locked )  "shared"
    key-generator   IDREF    #IMPLIED >

<!ELEMENT cache-type EMPTY>
<!ATTLIST cache-type
   type          ( none | count-limited | time-limited | unlimited ) "count-
limited"
    capacity       NMTOKEN  #IMPLIED>

<!ELEMENT map-to EMPTY>
<!ATTLIST map-to
    table      NMTOKEN  #IMPLIED
    xml        NMTOKEN  #IMPLIED
```

```
    ns-uri     NMTOKEN  #IMPLIED
    ns-prefix  NMTOKEN  #IMPLIED
    ldap-dn    NMTOKEN  #IMPLIED
    ldap-oc    NMTOKEN  #IMPLIED>

<!ELEMENT field ( description?, sql?, bind-xml?, ldap? )>
<!ATTLIST field
    name           NMTOKEN  #REQUIRED
    type           NMTOKEN  #IMPLIED
    required       ( true | false )  "false"
    direct         ( true | false )  "false"
    lazy           ( true | false )  "false"
    get-method     NMTOKEN  #IMPLIED
    set-method     NMTOKEN  #IMPLIED
    create-method  NMTOKEN  #IMPLIED
    collection     ( array | vector | hashtable | collection | set | map )
#IMPLIED>

<!ELEMENT sql EMPTY>
<!ATTLIST sql
    name        NMTOKEN  #IMPLIED
    type        CDATA    #IMPLIED
    many-key    NMTOKEN  #IMPLIED
    many-table  NMTOKEN  #IMPLIED
    dirty  ( check | ignore )  "check">

<!ELEMENT bind-xml EMPTY>
<!ATTLIST bind-xml
    name  NMTOKEN  #IMPLIED
    type  NMTOKEN  #IMPLIED
    matches  NMTOKEN  #IMPLIED
    node  ( attribute | element | text )  #IMPLIED>

<!ELEMENT ldap EMPTY>
<!ATTLIST ldap
    name  NMTOKEN  #IMPLIED>

<!ELEMENT key-generator  ( param* )>
<!ATTLIST key-generator
          name   CDATA  #REQUIRED
          alias  CDATA  #IMPLIED>
```

```
<!ELEMENT param EMPTY>
<!ATTLIST param
          name   CDATA  #REQUIRED
          value  CDATA  #REQUIRED>

<!ELEMENT description ( #PCDATA )>
```

The following sections describe the elements of the `CMPConfiguration.xml` file.

`bind-xml` element

The attribute or element name and XML schema must be specified for all XML-dependent fields. The `node` attribute indicates whether the field maps to an attribute, another tag, or the textual content of this tag. Only simple types (primitives, date, string, and so on) can be used for attribute values. Only one field can be specified as the content model in a given object. Table 7-2 describes the `bind-xml` element's members.

Table 7-2 Members of the `bind-xml` element

Member	Use	Description
`name`	?	Table-column name.
`type`	?	
`matches`	?	
`node`	?	Value: `attribute`, `element`, or `text`.

cache-type element

This element tells the container how to cache instances of this enterprise bean. Table 7-3 describes the members of this element.

Table 7-3 Members of the cache-type element

Member	Use	Description
type		Value: none, count-limited, time-limited, or unlimited. Default = "count-limited".
capacity	?	The maximum number of instances of this bean the container is to create.

class element

This element describes the mapping between a Java class (enterprise bean implementation) and an SQL table, an XML element, an LDAP entry, or any other engine. To map a class into LDAP, an identity field must be specified.

A class is specified by its Java class name, including the package name; for example, com.my.ejb.Person. If a class extends another class for which a mapping file exists, you should use the extends attribute to include the class being extended. Do not use the extends attribute to describe class inheritance that is not reflected in any mapping.

The class mapping specifies each field in the class that is mapped to a table column. Fields that are not mapped are not stored, read, or otherwise processed. Table 7-4 describes the `class` element's members.

Table 7-4 Members of the `class` element

Member	Use	Description
`name`		Class name.
`extends`	?	Implied by the persistence manager. It's the name of the class this class extends. Used only if this class extends another class for which mapping information is provided.
`depends`	?	Implied by the persistence manager.
`identity`	?	Implied by the persistence manager.
`access`		Value: `read-only`, `shared`, `exclusive` or `db-locked`. Default = `"shared"`
`key-generator`	?	Name or alias of the key generator to use. Use only for classes with single-property, numeric ID fields. If your class uses a compound primary key or the primary key contains strings, you must use a custom key generator; that is, the bean itself must create the primary-key values. See "key-generator element" (page 110).
`<description>`	?	Optional class description.
`<cache-type>`	?	See "cache-type element" (page 105).
`<map-to>`	?	Used if the name of the element this class maps to is not the same as the name of the class. By default, the persistence manager infers the name of the element from the name of the class: a class named `SocialEvent` is mapped to an element called `social-event`. See "map-to element" (page 112).
`<field>`	+	Describes the properties of an enterprise bean. See "field element" (page 107).

field element

This element specifies the mapping between an enterprise bean's field and an SQL table column, an XML element or attribute, an LDAP attribute, and so on. Table 7-5describes its members.

Table 7-5 Members of the field element

Item	Use	Description
name		Name of the enterprise bean's field being mapped.
type	?	Java type of the field. For example, java.lang.Integer.
required	?	Value: true or false. Default = "false". Indicates whether the field is optional or required.
direct	?	Value: true or false. Default = "false".
lazy	?	Value: true or false. Default = "false".
get-method	?	Implied by the persistence manager.
set-method	?	Implied by the persistence manager.
create-method	?	Implied by the persistence manager.
collection	?	Value: array, vector, hashtable, collection, set, or map . Implied by the persistence manager.
<description>	?	Optional field description.
<sql>	?	See "sql element" (page 113).
<bind-xml>	?	See "bind-xml element" (page 104).
<ldap>	?	See "ldap element" (page 111).

The mapping is specified from the perspective of the bean's implementation class. The field name is required even if no such field exists in the class in order to support field references. A field is an abstraction of an enterprise bean's property: It can refer to a property directly (by mapping to a public instance variable, not static nor transient) or indirectly by using accessor methods.

Unless specified otherwise, the persistence manager accesses the field through *get* and *set* methods, whose names are derived from the field name. For example, for a field called `lastName`, the accessors `String getName()` and `void setName(String)` are used. Collection fields require only a *get* method, except an array requires both a *get* and a *set* method. If the accessors are specified through the `get-method` and `set-method` attributes, the persistence manager accesses the field only through those methods. The methods must be `public` and not `static`.

If the `direct` attribute is `true`, the field is accessed directly. The field must be `public`, not `static` nor `transient`.

The `type` attribute indicates the type of the instance variable being mapped or the type of each a collection's elements. You can use fully qualified class names or a short name, as Table 7-6 illustrates.

Table 7-6 Values for the `type` attribute of the `field` element for CMP beans

Short name	Fully qualified name
`other`	`java.lang.Object`
`string`	`java.lang.String`
`integer`	`integer`
`long`	`long`
`boolean`	`boolean`
`double`	`double`
`float`	`float`
`big-decimal`	`java.math.BigDecimal`
`byte`	`byte`
`date`	`java.util.Date`
`short`	`short`
`char`	`char`
`bytes`	`byte[]`

Table 7-6 Values for the `type` attribute of the `field` element for CMP beans

Short name	Fully qualified name
`chars`	`char[]`
`strings`	`string[]`
`locale`	`java.lang.Locale`

If the field is a collection, you specify the collection type through the `collection` attribute and the type of each element of the collection through the `type` attribute. Use the following table to determine the appropriate value for the `collection` attribute.

Table 7-7 Values for the `collection` attribute of the `field` element CMP beans

Collection attribute value	Type of collection	Default implementation
`array`	`<type>[]`	`<type>[]`
`vector`	`java.util.Vector`	`java.util.Vector`
`hashtable`	`java.util.Hashtable`	`java.util.Hashtable`
`collection`	`java.util.Collection`	`java.util.Arraylist`
`set`	`java.util.Set`	`java.util.Hashset`
`map`	`java.util.Map`	`java.util.Hashmap`

The "Default implementation" column indicates the type used if the object holding the collection is `null` and needs to be instantiated. For `hashtable` and `map` collections, the persistence manager adds an object with the `put(Object, Object)` method: The object added is both the key and the value.

Table 7-5 (page 107) describes the members of the field element.

key-generator element

This element specifies parameters for the key generator (if needed). For example, to obtain sequential values from the table SEQTAB, use

```
<key-generator name="SEQUENCE">
    <param name="table" value="SEQTAB">
    <param name="global" value="0">
</key-generator>
<class key-generator="SEQUENCE">
    ...
</class>
```

If you have to use several key generators of the same type for the same data store, use aliases:

```
<key-generator name="SEQUENCE" alias="seq1">
    <param name="table" value="SEQTAB">
    <param name="global" value="0">
</key-generator>
<key-generator name="SEQUENCE" alias="seq2">
    <param name="table" value="SEQGLOBAL">
    <param name="global" value="1">
</key-generator>
<class key-generator="seq2">
    ...
</class>
```

Table 7-8 describes the members of the `key-generator` element.

Table 7-8 Members of the `key-generator` element

Member	Use	Description
`name`		Sequence-table name.
`alias`	?	Additional identifier for the key generator.
`<param>`	*	See "param element" (page 113).

Table 7-9 lists the key-generator names supported in the persistence manager.

Table 7-9 Key-generator names supported in the persistence manager

Name	Description
`MAX`	*MAX(pk) + 1* generic algorithm.
`HIGH/LOW`	HIGH/LOW generic algorithm.
`UUID`	UUID generic algorithm.
`IDENTITY`	Supports auto-increase identity fields in Sybase ASE/ASA, MS SQL Server, MySQL, and Hypersonic SQL.
`SEQUENCE`	Supports the SEQUENCE algorithm in Oracle, PostgreSQL, Interbase, and SAP DB.

`ldap` element

This element contains field mapping information for fields mapped to LDAP resources. Table 7-10 describes its members.

Table 7-10 Members of the `ldap` element

Member	Use	Description
`name`	?	LDAP-resource name.

map-to element

This element specifies the mapping between an enterprise bean and an SQL table. Table 7-11 describes the element's members.

Table 7-11 Members of the map-to element

Members	Use	Description
table	?	SQL table name.
xml	?	
ns-uri	?	
ns-prefix	?	
ldap-dn	?	
ldap-oc	?	

mapping element

This element is the root element of the entire file. It defines a collection of class mappings. Its members are described in Table 7-12.

Table 7-12 Members of the mapping element

Member	Use	Description
<description>	?	Optional description of the mapping.
<include>	*	Used to include other mappings in this mapping. The tag's sole member is the href attribute, set to the URL that indicates the location of the mapping file.
<class>	*	See "class element" (page 105).
<key-generator>	*	See "key-generator element" (page 110).

param element

This element is used to provide named parameters to the containing element. Table 7-13 describes the members of this element.

Table 7-13 Members of the param element

Member	Use	Description
name		Parameter name.
value		Parameter value.

sql element

This element provides field mapping information that is relevant only for fields mapped to SQL tables. The type can be the proper Java-class type returned by the JDBC driver or the SQL type without precision, for example, "java.math.BigDecimal" or "numeric". However, the type could contain the parameter for the SQL-to-Java type convertors in square brackets, for example, "char[01]" for false=0, true=1 conversion from the boolean Java type to the char SQL type.

Table 7-14 Members of the sql element

Member	Use	Description
name	?	Table-column name.
type	?	SQL type of the column.
many-key	?	
many-table	?	
dirty	?	Value: check or ignore. Default = "check".

Elements of the Transaction Manager Configuration File

The following sections describe the elements of the `TransactionManagerConfiguration.xml` file.

`config` element

The `config` element provides the configuration for a JDBC data source. Table 7-15 describes its members.

Table 7-15 Data members of the `config` element

Member	Use	Description
`<serverName>`		Server name.
`<portNumber>`		Port number.
`<databaseName>`	?	Database name.
`<driverType>`	?	Driver type. Value: `thin`.
`<user>`		User ID.
`<password>`		Password.

connector element

The connector element specifies a database-connection factory. Table 7-16 describes its members.

Table 7-16 Members of the connector element

Member	Use	Description
<name>		Connector name.
<jar>		Connector JAR filename.
<paths>	?	Paths to additional JAR and dependent files.
<config>	?	See "config element" (page 114).
<limits>	?	See "limits element" (page 116).

dataSource element

The dataSource element contains a specification for a JDBC data source. Table 7-17 describes the members of this element.

Table 7-17 Members of the dataSource element

Member	Use	Description
<name>		Data-source name.
<jar>		Data-source JAR filename.
<paths>	?	Paths to additional JAR and dependent files.
<class>		Class name of the data-source implementation.
<config>	?	See "config element" (page 114).
<limits>	?	See "limits element" (page 116).

domain element

The domain element is the root tag of the entire file. Table 7-18 describes the members of this element.

Table 7-18 Members of the domain element

Member	Use	Description
<name>		Domain name.
<maximum>	?	Maximum number of open transactions allowed.
<timeout>	?	Default timeout (in seconds) for transactions.
<resources>	?	See "resources element" (page 117).

limits element

The limits element provides resource limits for a data source or a connector. Table 7-19 describes its members.

Table 7-19 Data members of the limits element

Member	Use	Description
<maximum>	?	Maximum number of connections allowed.
<minimum>	?	Minimum number of connections allowed.
<initial>	?	Initial pool size.
<maxRetain>	?	Maximum period (in seconds) to retain open connections.
<timeout>	?	Maximum timeout (in seconds) to wait for a new connection.
<trace>	?	Turns tracing on ("true") or off ("false").

resources element

The resources element is the top-level tag of a list of JDBC data sources and JCA connectors. Table 7-20 describes the members of this element.

Table 7-20 Members of the resources element

Member	Use	Description
<dataSource>	*	See "dataSource element" (page 115).
<connector>	*	See "connector element" (page 115).

Elements of the Container Configuration File

The DTD for the deployment configuration file, OpenEJBConfiguration.xml, is stored in /System/Library/WebObjects/JavaApplications/OpenEJBTool.woa/Contents/Resources. The DTD is also added to the Resources group of an enterprise-bean–client application project. The file, called openejb_config.dtd, is shown in Listing 7-2 . (You must never edit this file.)

Listing 7-2 DTD for OpenEJBConfiguration.xml

```
<?xml encoding="US-ASCII"?>
<!ELEMENT entity-bean (description?, display-name?, small-icon?,large-icon?, ejb-
deployment-id, home, remote, ejb-class, persistence-type, prim-key-class, reentrant, cmp-
field-name*, primkey-field?, jndi-enc?, security-role-ref*, query*)>
<!ELEMENT query (description?, method, query-statement)>
<!ELEMENT query-statement (#PCDATA)>
<!ELEMENT entity-container (class-name?, codebase?, description?, display-name?,
container-name, properties?, entity-bean+)>
<!ELEMENT codebase (#PCDATA)>
<!ELEMENT class-name (#PCDATA)>
```

```
<!ELEMENT cmp-field-name (#PCDATA)>
<!ELEMENT connection-manager (connection-manager-id, class-name,properties?)>
<!ELEMENT connectors (connector*, connection-manager+)>
<!ELEMENT connector (connector-id, connection-manager-id, managed-connection-factory)>
<!ELEMENT connector-id (#PCDATA)>
<!ELEMENT connection-manager-id (#PCDATA)>
<!ELEMENT containers      (stateful-session-container|stateless-session-
container|entity-container)+>
<!ELEMENT container-name (#PCDATA)>
<!ELEMENT container-system (containers, security-role*, method-permission*, method-
transaction*)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT display-name (#PCDATA)>
<!ELEMENT ejb-class (#PCDATA)>
<!ELEMENT ejb-deployment-id (#PCDATA)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT home (#PCDATA)>
<!ELEMENT env-entry (env-entry-name, env-entry-type, env-entry-value)>
<!ELEMENT env-entry-name (#PCDATA)>
<!ELEMENT env-entry-type (#PCDATA)>
<!ELEMENT env-entry-value (#PCDATA)>
<!ELEMENT facilities (intra-vm-server, remote-jndi-contexts?, connectors?, services)>
<!ELEMENT remote-jndi-contexts (jndi-context+)>
<!ELEMENT jndi-context (jndi-context-id, properties)>
<!ELEMENT jndi-context-id (#PCDATA)>
<!ELEMENT ejb-ref (ejb-ref-name, home, ejb-ref-location)>
<!ELEMENT ejb-ref-location (ejb-deployment-id | (remote-ref-name, jndi-context-id))>
<!ELEMENT remote-ref-name (#PCDATA)>
<!ELEMENT factory-class (#PCDATA)>
<!ELEMENT intra-vm-server (proxy-factory, codebase?, properties?)>
<!ELEMENT jndi-enc (env-entry*, ejb-ref*, resource-ref*)>
<!ELEMENT large-icon (#PCDATA)>
<!ELEMENT logical-role-name (#PCDATA)>
<!ELEMENT managed-connection-factory (class-name, properties?)>
<!ELEMENT method (description?, ejb-deployment-id?, method-intf?, method-name, method-
params?)>
<!ELEMENT method-intf (#PCDATA)>
<!ELEMENT method-name (#PCDATA)>
<!ELEMENT method-param (#PCDATA)>
<!ELEMENT method-params (method-param*)>
<!ELEMENT method-permission (description?, role-name+, method+)>
```

```
<!ELEMENT method-transaction (description?, method+, trans-attribute)>
<!ELEMENT openejb (container-system, facilities)>
<!ELEMENT persistence-type (#PCDATA) >
<!ELEMENT physical-role-name (#PCDATA)>
<!ELEMENT prim-key-class (#PCDATA)>
<!ELEMENT primkey-field (#PCDATA)>
<!ELEMENT properties (property+)>
<!ELEMENT property (property-name, property-value)>
<!ELEMENT property-name (#PCDATA)>
<!ELEMENT property-value (#PCDATA)>
<!ELEMENT proxy-factory (#PCDATA)>
<!ELEMENT reentrant (#PCDATA)>
<!ELEMENT role-mapping (logical-role-name+, physical-role-name+)>
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT role-link (#PCDATA)>
<!ELEMENT remote (#PCDATA)>
<!ELEMENT res-auth (#PCDATA)>
<!ELEMENT res-id (#PCDATA)>
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT res-type (#PCDATA)>
<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth, (res-id |
properties | connector-id))>
<!ELEMENT resource (description?, res-id, properties)>
<!ELEMENT security-role (description?, role-name)>
<!ELEMENT security-role-ref (description?, role-name, role-link)>
<!ELEMENT security-service (description?, display-name?, service-name, factory-class,
codebase?,properties?, role-mapping*)>
<!ELEMENT security-service-name (#PCDATA)>
<!ELEMENT services (security-service, transaction-service)>
<!ELEMENT service-name (#PCDATA)>
<!ELEMENT small-icon (#PCDATA)>
<!ELEMENT stateful-bean (description?, display-name?, small-icon?,large-icon?, ejb-
deployment-id, home, remote, ejb-class, transaction-type, jndi-enc?, security-role-
ref*)>
<!ELEMENT stateless-bean (description?, display-name?, small-icon?,large-icon?, ejb-
deployment-id, home, remote, ejb-class, transaction-type, jndi-enc?, security-role-
ref*)>
<!ELEMENT stateful-session-container (codebase?, description?, display-name?, container-
name, properties?, stateful-bean+)>
<!ELEMENT stateless-session-container (codebase?, description?, display-name?,
container-name, properties?, stateless-bean+)>
```

```
<!ELEMENT transaction-service (description?, display-name?, service-name, factory-class,
codebase?, properties?) >
<!ELEMENT transaction-service-name (#PCDATA)>
<!ELEMENT transaction-type (#PCDATA)>
<!ELEMENT trans-attribute (#PCDATA)>
```

The following sections describe the elements of the `OpenEJBConfiguration.xml` file.

`connection-manager` element

This element specifies a connection manager. Table 7-21 describes its members.

Table 7-21 Members of the `connection-manager` element

Member	Use	Description
`<connection-manager-id>`		Name of the connection manager.
`<class-name>`		Class name of the data source.
`<properties>`		Properties required by the data source.

`connector` element

This element defines a connector. Table 7-22 describes its members.

Table 7-22 Members of the `connector` element

Member	Use	Description
`<connector-id>`		Name of the connector.
`<connection-manager-id>`		Specifies a connection manager. See "connection-manager element" (page 120).
`<managed-connection-factory>`	*	See "managed-connection-factory element" (page 127).

connectors element

This element encloses connectors or connection managers. Table 7-23 describes its members.

Table 7-23 Members of the connectors element

Member	Use	Description
<connector>	*	See "connector element" (page 120).
<connection-manager>	+	See "connection-manager element" (page 120).

container-system element

This element delimits the container configuration section of the deployment configuration file. Table 7-24 describes its members.

Table 7-24 Members of the container-system element

Member	Use	Description
<containers>		See "dataSource element" (page 115).
<security-role>	+	See "connector element" (page 115).
<method-permission>	+	Assigns a logical role to methods of the enterprise beans defined in the containers element.
<method-transaction>	+	Specifies a method's transaction attribute.

containers element

This element encloses containers for the three types of enterprise beans: stateless session beans, stateful session beans, and entity beans. Table 7-25 describes its members.

Table 7-25 Members of the containers element

Member	Use	Description
<stateful-session-container> <stateless-session-container> <entity-container>	+	At least one of these items must be present.

ejb-ref element

This element defines a reference to a bean so that the bean can be accessed using JNDI calls. Table 7-26 describes its members.

Table 7-26 Members of the ejb-ref element

Member	Use	Description
<ejb-ref-name>		JNDI name for the bean. For example, ejb/agent/Agent.
<home>		Home interface of the bean. For example, webobjectsexamples.realestate.agent.AgentHome.
<ejb-ref-location>		See "ejb-ref-location element" (page 123).

ejb-ref-location element

This element identifies a bean through its name (using its <ejb-deployment-id> member) or through its remote interface and JNDI context ID. Table 7-27 describes its members.

Table 7-27 Members of the ejb-ref-location element

Member	Use	Description
<ejb-deployment-id> or (<remote-ref-name>, <jndi-context-id>)		Either <ejb-deployment-id> or <remote-ref-name> and <jndi-context-id> must be specified.

entity-bean element

This element defines an entity session bean. Table 7-28 describes its members.

Table 7-28 Members of the entity-bean element

Member	Use	Description
<description>	?	Description for the bean.
<display-name>	?	
<small-icon>	?	
<large-icon>	?	
<ejb-deployment-id>		Name of the bean.
<home>		Home interface (for example, com.my.ejb.PersonHome).
<remote>		Remote interface (for example, com.my.ejb.Person).
<ejb-class>		Implementation class (for example, com.my.ejb.PersonBean).

Table 7-28 Members of the `entity-bean` element

Member	Use	Description
`<persistence-type>`		Value: `Container` or `Bean`.
`<prim-key-class>`		Fully qualified class name of the primary key.
`<reentrant>`		Value: `true` or `false`. Should be `false`.
`<cmp-field-name>`	*	Container-managed–persistence field name.
`<primkey-field>`	?	Primary-key field name.
`<jndi-enc>`	?	See "jndi-enc element" (page 126).
`<security-role-ref>`	*	See "security-role-ref element" (page 136).
`<query>`	*	Specifies a query for a finder method.

`entity-container` element

This element defines an entity-bean container and encloses the definition of entity beans. Table 7-29 describes its members.

Table 7-29 Members of the `entity-container` element

Member	Use	Description
`<codebase>`	?	
`<description>`	?	Description of the container.
`<display-name>`	?	
`<container-name>`		Name for the container.
`<properties>`	?	Used to tell the container how to handle instances of entity beans. See "properties element" (page 131).
`<entity-bean>`	+	Entity bean definitions. See "entity-bean element" (page 123).

env-entry element

This element defines an environment variable and its value (which can be accessed by other beans through JNDI). Table 7-30 describes its members.

Table 7-30 Members of the env-entry element

Member	Use	Description
<env-entry-name>		Name of the variable.
<env-entry-type>		Java type of the variable.
<env-entry-value>		Value for the variable.

facilities element

This element specifies the runtime environment: proxy-generation attributes, remote JNDI contexts, data-source connections, and J2EE services. You should not change the information within <facilities> and </facilities> tags. Table 7-31 describes its members.

Table 7-31 Members of the facilities element

Member	Use	Description
<intra-vm-server>		
<remote-jndi-contexts>	?	
<connectors>	?	
<services>		

jndi-context element

This element defines one external JNDI context to be used by the application. Table 7-32 describes its members.

Table 7-32 Members of the jndi-context element

Member	Use	Description
<jndi-context-id>		Name of the JNDI context.
<properties>		Required properties.

jndi-enc element

This element encloses naming information so that this bean can be located through JNDI. Table 7-33 describes the members of the jndi-enc element.

Table 7-33 Members of the jndi-enc element

Member	Use	Description
<env-entry>	*	
<ejb-ref>	*	Defines a reference to this bean. See "ejb-ref element" (page 122).
<resource-ref>	*	Defines the beans data source. See "resource-ref element" (page 133).

intra-vm-server element

This element specifies the dynamic factory proxy to use to create client proxies of the real EJB objects. Table 7-34 describes its member.

Table 7-34 Member of the intra-vm-server element

Member	Use	Description
<proxy-factory>		Dynamic proxy factory. Values: org.openejb.util.proxy.jdk13.Jdk13ProxyFactory or org.openejb.util.proxy.DynamicProxy.

managed-connection-factory element

This element defines a managed-connection factory. Table 7-35 describes its members.

Table 7-35 Members of the managed-connection-factory element

Member	Use	Description
<class-name>		Class name of the data source.
<properties>	?	Properties required by the data source.

method element

This element specifies a home or remote interface method of an enterprise bean. Table 7-36 describes its members.

Table 7-36 Members of the method element

Member	Use	Description
<description>	?	Description for the method.
<ejb-deployment-id>	?	Must specify the ID (name) of one of the enterprise beans declared in the <container-system> tag. If this element isn't specified, this method declaration applies to all matching bean methods (home and remote interfaces) of all the enterprise beans defined in the <container-system> tag.
<method-intf>	?	Value: Home or Remote. Distinguishes between a method with the same signature that is defined in both the home and remote interface.
<method-name>		Specifies the method name.
<method-params>	?	Identifies a single method among multiple methods with an overloaded method name. If the method takes no input arguments, this element can be empty or omitted.

These examples of the three possible styles of the method element's syntax:

- Referring to all the methods (home and remote interfaces) defined within the <container-system> tag.

```
<method>
    <method-name>*</method-name>
</method>
```

- Referring to a specific method defined within the <container-system> tag.

```
<method>
    <method-name>METHOD</method-name>
</method>
```

- Referring to a single method within a set of methods (home and remote interfaces) with an overloaded name.

```
<method>
    <method-name>METHOD</method-name>
    <method-params>
        <method-param>PARAM-1</method-param>
        <method-param>PARAM-2</method-param>
                ...
        <method-param>PARAM-n</method-param>
    </method-params>
</method>
```

`method-params` element

This element is used when further identification of a method is needed due to method-name overloading. Table 7-37 describes its members.

Table 7-37 Members of the `method-params` element

Member	Use	Description
`<method-param>`	*	Fully qualified Java type. Specify arrays by following the array element's type with one or more pairs of square brackets (for example, `int[]`).

method-permission element

This element maps security roles to methods. Table 7-38 describes its members.

Table 7-38 Members of the method-permission element

Member	Use	Description
<description>	?	Description for the method permission.
<role-name>	+	Logical role name corresponding to a <security-role> tag.
<method>	+	See "method element" (page 128).

method-transaction element

This element specifies how the container manages transaction scopes when delegating a method invocation to an enterprise bean's implementation class. Table 7-39 describes its members.

Table 7-39 Members of the method-transaction element

Member	Use	Description
<description>	?	Description for the method and the transaction.
<method>	+	Methods to apply the transaction type to.
<trans-attribute>		Value: NotSupported, Supports, Required, RequiresNew, Mandatory, Never, or Bean.

openejb element

This is the root tag of the deployment configuration file. Table 7-40 describes its members.

Table 7-40 Members of the openejb element

Member	Use	Description
<container-system>		See "container-system element" (page 121).
<facilities>		See "facilities element" (page 125).

properties element

This element encloses a set of property-value definitions. Table 7-41 describes its member.

Table 7-41 Member of the properties element

Member	Use	Description
<property>		See "property element" (page 131).

property element

This element encloses a property-value definition. Table 7-42 describes its members.

Table 7-42 Members of the property element

Member	Use	Description
<property-name>		The name of the property.
<property-value>		The value for the property.

query element

This element can be used to declare a query statement and bind it to a specific finder method. The value can be retrieved using the org.openejb.core.DeploymentInfo.getQuery method. Table 7-43 describes the members of the query element.

Table 7-43 Members of the query element

Member	Use	Description
<description>	?	Description for the query.
<method>		The <ejb-deployment-id> tag of <method> is ignored (should not be used). See "method element" (page 128).
<query-statement>	*	SQL statement.

remote-jndi-contexts element

This element groups external JNDI contexts. Table 7-44 describes its members.

Table 7-44 Member of the remote-jndi-contexts element

Member	Use	Description
<jndi-context>	+	See "jndi-context element" (page 126).

resource element

This element defines a resource. Table 7-45 describes its members.

Table 7-45 Member of the resource element

Member	Use	Description
<description>	?	Description for the resource.
<res-id>		Maps this resource to a <connector-id> element in the corresponding <connectors> section.
<properties>		See "properties element" (page 131).

resource-ref element

This element specifies a reference to an external resource. Table 7-46 describes its members.

Table 7-46 Members of the resource-ref element

Member	Use	Description
<description>	?	Description for the resource.
<res-ref-name>		Specifies the name of a resource manager connection-factory reference (for example, comp/env/jdbc/Employee).

Table 7-46 Members of the `resource-ref` element

Member	Use	Description
`<res-type>`		Specifies the type of the data source, that is, the Java class or interface expected to be implemented by the data source (for example, `javax.sql.DataSource`).
`<res-auth>`		Value: `Application` or `Container`. Specifies who signs on to the resource manager: the enterprise bean or the container.
`<res-id>` or `<connector-id>` or `<properties>`		You can map this resource reference to a resource, a connector, or to a set of properties.

This is an example of a `<resource-ref>` definition using properties:

```
<resource-ref>
    <res-ref-name>comp/env/jdbc/Employee</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <properties>
        <property>
            <property-name>url</property-name>
            <property-value>jdbc:odbc:orders</property-value>
        </property>
        <property>
            <property-name>username</property-name>
            <property-value>Admin</property-value>
        </property>
        <property>
            <property-name>password</property-name>
            <property-value></property-value>
        </property>
    </properties>
</resource-ref>
```

role-mapping element

This element maps a logical security role to a physical security role. Table 7-47 describes its members.

Table 7-47 Members of the role-mapping element

Member	Use	Description
<logical-role-name>	+	Logical security-role name.
<physical-role-name>	+	Physical security-role name.

security-role element

This element defines a logical role name. Table 7-48 describes its members.

Table 7-48 Members of the security-role element

Member	Use	Description
<description>	?	Description of the logical role.
<role-name>		Logical role name (for example, everyone or admin).

security-role-ref element

This element specifies a security-role reference. Table 7-49 describes its members

Table 7-49 Members of the security-role-ref element

Member	Use	Description
<description>	?	Description of the security role.
<role-name>		Security-role name used in code. It must be the String used as the argument in the invocation of the isCallerInRole(String) method of EJBContext.
<role-link>		Name of a security role (<security-role> tag). Links this security-role reference to a defined security role. See "security-role element" (page 135).

security-service element

This element defines a security service. Table 7-50 describes its members.

Table 7-50 Members of the security-service element

Member	Use	Description
<description>	?	Description of the service.
<display-name>	?	
<service-name>	?	Name of the service.
<factory-class>		Name of the factory class for the service.
<codebase>	?	
<properties>	?	Properties needed by the service.
<role-mapping>	+	See "role-mapping element" (page 135).

services element

This element encloses services used by the container. Table 7-51 describes its members.

Table 7-51 Members of the services element

Member	Use	Description
<security-service>		Description of the services.
<transaction-service>	*	See "transaction-service element" (page 140).

stateful-bean element

This element defines a stateful session bean. Table 7-52 describes its members.

Table 7-52 Members of the stateful-bean element

Member	Use	Description
<description>	?	Description for the bean.
<display-name>	?	
<small-icon>	?	
<large-icon>	?	
<ejb-deployment-id>		Name of the bean (for example, HelloBean).
<home>		Home interface (for example, com.my.ejb.HelloHome).
<remote>		Remote interface (for example, com.my.ejb.Hello).
<ejb-class>		Implementation class (for example, com.my.ejb.HelloBean).

Table 7-52 Members of the `stateful-bean` element

Member	Use	Description
`<transaction-type>`		Value: `Container` or `Bean`.
`<jndi-enc>`	?	See "jndi-enc element" (page 126).
`<security-role-ref>`	*	See "security-role-ref element" (page 136).

`stateful-session-container` element

This element defines a stateful session bean container and encloses the definitions of stateful session beans. Table 7-53 describes its members.

Table 7-53 Members of the `stateful-session-container` element

Member	Use	Description
`<codebase>`	?	
`<description>`	?	Description of the container.
`<display-name>`	?	
`<container-name>`		Name for the container.
`<properties>`	?	Used to tell the container how to handle instances of stateful session beans. See "properties element" (page 131).
`<stateful-bean>`	+	Stateful bean definitions. See "stateful-bean element" (page 137).

stateless-bean element

This element defines a stateless session bean. Table 7-54 describes its members.

Table 7-54 Members of the stateless-bean element

Member	Use	Description
<description>	?	
<display-name>	?	
<small-icon>	?	
<large-icon>	?	
<ejb-deployment-id>		Name of the bean.
<home>		Home interface (for example, com.my.ejb.HelloHome).
<remote>		Remote interface (for example, com.my.ejb.Hello).
<ejb-class>		Implementation class (for example, com.my.ejb.HelloBean).
<transaction-type>		Value: Container or Bean.
<jndi-enc>	?	See "jndi-enc element" (page 126).
<security-role-ref>	*	See "security-role-ref element" (page 136).

stateless-session-container element

This element defines a stateless session bean container and encloses the definitions of stateless session beans. Table 7-55 describes its members.

Table 7-55 Members of the stateless-session-container element

Member	Use	Description
<codebase>	?	
<description>	?	Description of the container.
<display-name>	?	
<container-name>		
<properties>	?	Used to tell the container how to handle instances of stateless session beans. See "properties element" (page 131).
<stateless-bean>	+	Stateless bean definitions. See "stateless-bean element" (page 139).

transaction-service element

This element defines a transaction service. Table 7-56 describes its members.

Table 7-56 Members of the transaction-service element

Member	Use	Description
<description>	?	Description of the transaction service.
<display-name>	?	
<service-name>		Name of the transaction service.

Table 7-56 Members of the `transaction-service` element

Member	Use	Description
`<factory-class>`		Name of the factory class for the service.
`<codebase>`	?	
`<properties>`	?	Properties needed by the service.

APPENDIX A

Document Revision History

Table A-1 describes the revisions to *Inside WebObjects: Enterprise JavaBeans*.

Table A-1

Date	Notes
October 2002	Revised for WebObjects 5.2.
	Document name changed to *Inside WebObjects: Enterprise JavaBeans*.
	Chapter 5, "Developing Bean Frameworks" (page 71), added information on EOBeanBuilder usage.
	Removed references to `LocalTransactionConfiguration.xml` file, as it not used in WebObjects 5.2. See "Generating the EJB Configuration Files" (page 98) for more information.
	Added information on EJB-stub generation ("Generating EJB Stubs" (page 76)).
	Added information on EJB transport ("Communication Transport Between Bean Clients and Containers" (page 98)).
	Added information on EJB-container logging ("EJB Container Operation Logging" (page 99)).
January 2002	Reorganized Chapter 7, "Configuration Reference" (page 101), in alphabetical order.
	Added index and glossary.
December 2001	Document published as *Inside WebObjects: Developing EJB Applications*.

APPENDIX A

Document Revision History

Glossary

bean class The bean class implements the methods defined in an enterprise bean's business methods, which are defined in the remote interface.

bean client An application or enterprise bean that makes use of an enterprise bean.

deployment descriptor XML file that describes the configuration of a Web application. It's located in the WEB-INF directory of the application's WAR file and named web.xml. See also WAR.

EJB (Enterprise JavaBeans) Specification that provides an infrastructure through which data-based components can be developed and deployed in a variety of platforms.

J2EE (Java 2 Platform, Enterprise Edition) Specification that define a platform for the development and deployment of Web applications. It defines an environment under which enterprise beans, servlets, and JSP pages can share resources and work together.

home interface The home interface defines an enterprise bean's life-cycle methods, used to create, remove, and find beans.

ORB (Object Request Broker) Facility through which a client application can locate and use distributed objects.

remote interface The remote interface defines an enterprise bean's business methods, which are used by its clients to interact with the bean.

Web application, Web app File structure that contains servlets, JSP pages, HTML documents and other resources. This structure can be deployed on any servlet-enabled HTTP server. See also servlet container.

Index

A, B

C

D

E, F

G

H

I

J, K, L,

M, N

O

P, Q

R

S

T, U, V

W. X, Y, Z