



Developing Enterprise JavaBeans™



VERSION 9

Borland®
JBuilder®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997–2003 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JBE0090WW21002ejb 6E7R0503
0304050607-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter 1

Introduction

1-1

Documentation conventions	1-1
Developer support and resources	1-3
Contacting Borland Technical Support.	1-3
Online resources	1-4
World Wide Web	1-4
Borland newsgroups.	1-4
Usenet newsgroups	1-4
Reporting bugs	1-5

Chapter 2

An introduction to EJB development

2-1

Why we need Enterprise JavaBeans	2-1
Roles in the development of an EJB	
application	2-2
Application roles	2-2
Infrastructure roles	2-3
Deployment and operation roles	2-4
EJB architecture	2-4
The EJB server	2-5
The EJB container.	2-5
How an enterprise bean works	2-6
Types of enterprise beans	2-7
Session beans	2-7
Entity beans	2-7
Message-driven beans	2-7
Remote and local access.	2-8
Developing enterprise beans	2-8

Chapter 3

Creating 2.0 session beans and message-driven beans with the EJB Designer

3-1

Introducing EJB modules	3-2
Creating an EJB 2.0 module	3-2
Creating a module with the EJB Module wizard	3-2
Creating an EJB module from existing deployment descriptors	3-4
Displaying the EJB Designer	3-7
Quicken the display of your EJBs.	3-8
Preventing the saving of EJB Designer layout changes	3-9
How the EJB Designer names EJB files	3-9

Creating session beans	3-10
Viewing a bean's source code	3-11
Modifying the bean.	3-12
Editing bean attributes	3-12
Adding a new field	3-12
Deleting a field.	3-13
Adding a new method	3-13
Removing a method.	3-14
Working with ejbCreate() methods	3-14
Regenerating a bean's interfaces.	3-15
Setting packages for a enterprise beans	3-16
Importing beans.	3-16
Organizing beans with views	3-18
Finding beans	3-19
Arranging beans.	3-20
Creating message-driven beans	3-20
Removing beans.	3-21
Fixing errors in your bean	3-22
Viewing the deployment descriptors	3-23
Displaying the Deployment Descriptor editor.	3-23
Setting IDE options for the EJB Designer.	3-24
Taking the next step.	3-25
Chapter 4	
Creating 2.0 entity beans with the EJB Designer	4-1
Creating CMP 2.0 entity beans from an imported data source	4-2
Importing a data source	4-2
jndi-definitions.xml	4-3
Modifying the imported data source schema	4-4
Generating the entity bean classes and interfaces	4-6
Editing entity bean properties	4-7
Referencing another table.	4-8
WebLogic table mappings	4-11
Entity bean field and method inspectors	4-14
Creating entity bean relationships.	4-15
Using the relationship inspector to specify a relationship	4-16
Improving performance in large projects with many relationships.	4-20
Removing a relationship	4-20

Adding a finder method	4-20	Changing the build properties for an EJB module	8-4
Adding an ejbSelect method	4-21	Compiling	8-4
Adding a home business method.	4-22	The generated JAR file	8-5
Creating a schema from entity beans	4-23	Editing deployment descriptors.	8-6
Exporting a data source	4-23	Verifying descriptors	8-7
Creating entity beans with bean-managed persistence	4-23		
Chapter 5		Chapter 9	
Creating session facades for entity beans	5-1	Running and testing an enterprise bean	9-1
Using the DTO and Session Facade wizard . . .	5-2	Testing your bean	9-2
Examining the generated classes	5-7	Choosing the type of test client	9-3
Chapter 6		Working with test client applications	9-3
Creating EJB 1.x components with JBuilder	6-1	Creating a test client application.	9-3
EJB modules	6-1	Using the test client application	9-7
Creating an EJB 1.x module	6-2	Using your test client application to test your enterprise bean	9-9
Creating an EJB 1.x module with the EJB Module wizard.	6-2	Creating a Server runtime configuration	9-9
Creating an EJB module from existing deployment descriptors	6-3	Running your EJB test client application	9-11
Creating an enterprise bean.	6-5	Working with JUnit test cases	9-12
Creating a session bean	6-6	Creating a JUnit test case	9-12
Creating an entity bean	6-7	Creating a JUnit test case using the EJB Test Client wizard	9-13
Adding the business logic to your bean . .	6-8	Running your JUnit test case	9-15
Exposing business methods through the remote interface	6-11	Working with Cactus JUnit test cases	9-15
Generating the bean class from a remote interface	6-12	Creating a Cactus JUnit test case	9-15
Creating the home and remote interfaces for an existing bean	6-14	Configuring your project for testing an EJB with Cactus	9-17
Chapter 7		Creating a Cactus JUnit test case using the EJB Test Client wizard . . .	9-21
Creating EJB 1.x entity beans from an existing database table	7-1	Running your Cactus JUnit test case . .	9-23
Creating entity beans with the EJB 1.x Entity Bean Modeler	7-1		
Chapter 8		Chapter 10	
Compiling enterprise beans and creating a deployment module	8-1	Deploying enterprise beans	10-1
Compiling the bean	8-1	Creating a deployment descriptor file	10-2
Changing build properties for a deployment module	8-1	The role of the deployment descriptor	10-3
Changing the build properties for a bean .	8-4	The types of information in the deployment descriptor	10-3
		Structural information	10-4
		Application assembly information . .	10-5
		Security	10-5
		Application server-specific properties	10-6
		Creating an EAR file	10-6
		Deploying to an application server . . .	10-7
		Deploying one or more JAR files . .	10-8

Deploying to non-Borland servers	10-9
Setting deployment options with the Properties dialog box	10-9
Hot deploying to an application server	10-10
Chapter 11	
Using the Deployment Descriptor editor	
	11-1
Displaying the Deployment Descriptor editor	11-1
Viewing the deployment descriptor of an enterprise bean	11-2
Viewing an EJB module-level WebLogic 6.x, 7.x, or 8.x Properties page	11-3
Changing bean information	11-5
Enterprise bean information	11-5
General panel	11-5
Environment panel	11-8
EJB References panel	11-10
Resource References panel	11-12
Security Role References panel	11-13
Properties panel	11-13
Security Identity panel	11-15
EJB Local References panel	11-16
Resource Env Refs panel	11-18
WebLogic 6.x, 7.x, 8.x General panel	11-19
Server-specific Properties panel	11-20
WebLogic 6.x, 7.x, or 8.x Cache panel	11-21
Container transactions	11-22
Setting container transaction policies	11-22
WebLogic 6.x, 7.x, or 8.x Transaction Isolation panel	11-24
WebLogic 6.x, 7.x, or 8.x Idempotent Methods panel	11-25
Working with JDBC 1 data sources	11-26
Setting isolation levels	11-28
Setting data source properties	11-29
Adding security roles and method permissions	11-31
Creating a security role	11-31
Assigning method permissions	11-33
Adding container-managed persistence information for EJB 1.1 components	11-35
Finders panel	11-36
Specifying WebSphere 4.0 finders	11-38
Verifying descriptor information	11-39
Chapter 12	
Using the DataExpress for EJB components	
	12-1
The DataExpress EJB components	12-2
Components for the server	12-2
Components for the client	12-2
Creating the entity beans	12-3
Creating the server-side session bean	12-3
Adding provider and resolver components to the session bean	12-3
Writing the setSessionContext() method	12-4
Adding an EJB reference or EJB local reference to the deployment descriptor	12-5
Adding the providing and resolving methods	12-5
Calling the finder method	12-6
Building the client side	12-8
Handling relationships	12-9
The sample project	12-10
Chapter 13	
Developing session beans	
	13-1
Types of session beans	13-1
Stateful session beans	13-1
Stateless session bean	13-2
Writing the session bean class	13-2
Implementing the SessionBean interface	13-2
Writing the business methods	13-3
Adding one or more ejbCreate() methods	13-3
How JBuilder can help you create a session bean	13-4
The life of a session bean	13-6
Stateless beans	13-6
Stateful beans	13-7
The method-ready in transaction state	13-8
Chapter 14	
Developing entity beans	
	14-1
Persistence and entity beans	14-1
Bean-managed persistence	14-2
Container-managed persistence	14-2
Primary keys in entity beans	14-2
Writing the entity bean class	14-3
Implementing the EntityBean interface	14-4

Declaring and implementing the entity bean methods	14-5	Creating a home or local home interface for a session bean	16-2
Creating create methods	14-5	create() methods in session beans	16-3
Creating finder methods	14-7	Creating a remote home or local home interface for an entity bean	16-4
Writing the business methods	14-8	create() methods for entity beans	16-5
The life of an entity bean	14-8	Finder methods for entity beans	16-5
The nonexistent state	14-8	Creating the remote or local interface	16-7
The pooled state	14-9	The EJBObject and EJBLocal Object interfaces	16-8
The ready state	14-9		
Returning to the pooled state	14-9		
A bank entity bean example	14-10		
The entity bean home interface	14-10		
The entity bean remote interface	14-11		
An entity bean with container-managed persistence	14-12		
An entity bean with bean-managed persistence	14-14		
The primary key class	14-18		
The deployment descriptor	14-18		
Deployment descriptor for an entity bean with bean-managed persistence	14-20		
Deployment descriptor for an entity bean with container-managed persistence	14-20		
Chapter 15			
Developing message-driven beans 15-1			
How message-driven beans work	15-2		
The life of a management-driven bean instance	15-2		
Writing a message-driven bean class	15-3		
Implementing the MessageDrivenBean interface	15-3		
Implementing the MessageListener interface	15-4		
Writing the onMessage() method	15-4		
How JBuilder can help you create a message-driven bean	15-4		
Message-driven bean deployment descriptor attributes	15-6		
Using the SonicMQ Message Broker with message-driven beans	15-7		
Chapter 16			
Creating the home and remote/local interfaces 16-1			
Creating the home interface	16-1		
The EJBHome interface	16-2		
The LocalHome interface	16-2		
Creating a home or local home interface for a session bean	16-2		
create() methods in session beans	16-3		
Creating a remote home or local home interface for an entity bean	16-4		
create() methods for entity beans	16-5		
Finder methods for entity beans	16-5		
Creating the remote or local interface	16-7		
The EJBObject and EJBLocal Object interfaces	16-8		
Chapter 17			
Developing enterprise bean clients 17-1			
Locating the home interface	17-2		
Getting the remote/local interface	17-2		
Session beans	17-3		
Entity beans	17-3		
Finder methods and the primary key class	17-4		
Create and remove methods	17-4		
Calling methods	17-5		
Removing bean instances	17-6		
Referencing a bean with its handle	17-6		
Managing transactions	17-7		
Discovering bean information	17-8		
Creating a client with JBuilder	17-9		
Chapter 18			
Managing transactions 18-1			
Characteristics of transactions	18-1		
Transaction support in the container	18-2		
Enterprise beans and transactions	18-3		
Bean-managed versus container-managed transactions	18-3		
Local and global transactions	18-4		
Using the transaction API	18-4		
Handling transaction exceptions	18-6		
System-level exceptions	18-6		
Application-level exceptions	18-7		
Handling application exceptions	18-7		
Transaction rollback	18-8		
Options for continuing a transaction . .	18-8		
Chapter 19			
Creating JMS producers and consumers 19-1			
Using the JMS wizard	19-2		
Publish\subscribe message systems	19-3		
Point to point message systems	19-4		
Completing the code	19-5		

Chapter 20	
Exploring CORBA-based distributed applications	20-1
What is CORBA?	20-1
What is the VisiBroker ORB?	20-2
How JBuilder and the VisiBroker ORB work together	20-2
Setting up JBuilder for CORBA applications	20-4
Defining interfaces in Java	20-7
About the java2iop and java2idl compilers	20-8
Accessing the java2iop and java2idl compilers in JBuilder	20-9
RMI	20-10
Working with the java2iop compiler	20-10
Generating IIOP interfaces running java2iop	20-11
Mapping primitive data types to IDL	20-13
Mapping complex data types	20-13
Working with the java2idl compiler	20-15
Chapter 21	
Tutorial: Developing a session bean with the EJB Designer	21-1
Creating a new project	21-2
Specifying the target application server	21-2
Creating an EJB module	21-3
Building the bean	21-5
Setting bean properties	21-6
Adding fields to the Cart bean	21-6
Adding business methods to the Cart bean	21-9
Adding items to and removing items from the cart	21-10
Retrieving the items held by the bean and their cost	21-10
Adding a purchase() method	21-11
Working in the source code	21-11
Initializing the list of items	21-13
Adding the import statements	21-13
Implementing ejbCreate()	21-13
Implementing addItem() and removeItem()	21-14
Creating an Item class	21-15
Implementing the remaining methods	21-16
Working with the bean's deployment descriptors	21-17
Compiling your project	21-18
Running the Cart bean	21-18
Code for cart session bean	21-19
Chapter 22	
Tutorial: Creating a test client application	22-1
Opening the cart_session project	22-1
Using the EJB Test Client wizard	22-2
Examining the generated code	22-4
Adding your code to the test client	22-5
Creating a Cart bean instance	22-5
Adding items to and removing items from the cart	22-5
Completing the purchase	22-7
Removing the bean instance	22-7
Summarizing the items in the cart	22-9
Compiling the test client	22-10
Running the test client	22-10
Code for the test client application	22-12
Chapter 23	
Tutorial: Creating a CORBA application	23-1
Step 1: Setting up the project	23-2
Step 2: Defining the interfaces for the CORBA objects in IDL	23-3
Step 3: Generating client stubs and server servants	23-3
Generated files	23-4
Step 4: Implementing the client	23-5
Binding to the AccountManager object	23-6
Binding the wrapper class at runtime	23-7
Step 5: Implementing the server	23-9
What is the POA?	23-9
Step 6: Providing an implementation for the CORBA interface	23-10
Step 7: Compiling the application	23-11
Step 8: Running the Java application	23-11
Starting the VisiBroker ORB Smart Agent	23-11
Starting the server	23-12
Running the client	23-12
Deploying the application	23-13
Other sample applications	23-14
Index	I-1

Tutorials

Developing a session bean with the EJB Designer	21-1	Creating a test client application	22-1
		Creating a CORBA application	23-1

1

Introduction

Developing Enterprise JavaBeans explains how to create enterprise beans with JBuilder and use them in building distributed systems. JBuilder has a set of designers, wizards, and tools that greatly simplifies the creation, testing, and deploying of enterprise beans. You can create enterprise beans for deployment to Borland Enterprise Server 5.1.1 - 5.2.1, BEA WebLogic Servers 6.x, 7.x, and 8.x, IBM WebSphere Application Servers 4.0 and 5.0, the Sybase Enterprise Application Server 4.x, and the Sun-Netscape iPlanet Application Servers 6.x.

Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
Bold	Bold is used for java tools, bmj (Borland Make for Java), bjc (Borland Compiler for Java), and compiler options. For example: javac , bmj , -classpath .
<i>Italics</i>	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
<i>Keycaps</i>	This typeface indicates a key on your keyboard, such as “Press <i>Esc</i> to exit a menu.”

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
Monospaced type	Monospaced type represents the following: <ul style="list-style-type: none"> • text as it appears onscreen • anything you must type, such as “Type Hello World in the Title field of the Application wizard.” • file names • path names • directory and folder names • commands, such as <code>SET PATH</code> • Java code • Java data types, such as <code>boolean</code>, <code>int</code>, and <code>long</code>. • Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events • argument names • field names • Java keywords, such as <code>void</code> and <code>static</code>
[]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	Angle brackets are used to indicate variables in directory paths, command options, and code samples. For example, <code><filename></code> may be used to indicate where you need to supply a file name (including file extension), and <code><username></code> typically indicates that you must provide your user name. When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (< >). For example, you would replace <code><filename></code> with the name of a file, such as <code>employee.jds</code> , and omit the angle brackets. Note: Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <code></code> and <code><ejb-jar></code> . The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.
<i>Italics, serif</i>	This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, <code><url="jdbc:borland:jbuilder\\samples\\guestbook.jds"></code>
...	In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

Table 1.2 Platform conventions

Item	Meaning
Paths	Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).
Home directory	The location of the standard home directory varies by platform and is indicated with a variable, <home>. <ul style="list-style-type: none"> • For UNIX and Linux, the home directory can vary. For example, it could be /user/<username> or /home/<username> • For Windows NT, the home directory is C:\Winnt\Profiles\<username> • For Windows 2000 and XP, the home directory is C:\Documents and Settings\<username>
Screen shots	Screen shots reflect the Metal Look & Feel on various platforms.

Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

Contacting Borland Technical Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support on installation of the Borland product to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at <http://www.borland.com/devsupport/>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

Online resources

You can get information from any of these online sources:

World Wide Web	http://www.borland.com/ http://www.borland.com/techpubs/jbuilder/
Electronic newsletters	To subscribe to electronic newsletters, use the online form at: http://www.borland.com/products/newsletters/index.html

World Wide Web

Check www.borland.com/jbuilder regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- <http://www.borland.com/jbuilder/> (updated software and other files)
- <http://www.borland.com/techpubs/jbuilder/> (updated documentation and other files)
- <http://community.borland.com/> (contains our web-based news magazine for developers)

Borland newsgroups

When you register JBuilder you can participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at <http://www.borland.com/newsgroups/>.

Usenet newsgroups

The following Usenet groups are devoted to Java and related programming issues:

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases

- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

Note These newsgroups are maintained by users and are not official Borland sites.

Reporting bugs

If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

- Support Programs page at <http://www.borland.com/devsupport/namerica/>. Click the “Reporting Defects” link to bring up the Entry Form.
- Quality Central at <http://qc.borland.com>. Follow the instructions on the Quality Central page in the “Bugs Reports” section.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email jppubs@borland.com. This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

An introduction to EJB development

The “Enterprise JavaBeans (EJB) specification” formally defines a Java server-side component model and a programming interface for application servers. Developers build the components, called enterprise beans, to contain the business logic of the enterprise. Enterprise beans run on an EJB server that provides services such as transaction management and security to the beans. Developers don’t have to worry about programming these low-level and complex services, but can focus on encapsulating the business rules of an organization or system within the beans, knowing that the services are available to the beans when they are needed.

While the Enterprise JavaBeans specification is the ultimate authority on the EJB framework, it’s primarily useful to vendors such as Borland who build the EJB servers and containers the beans run in. This book will help you, the JBuilder developer, learn what you want to know about developing enterprise beans with JBuilder.

Why we need Enterprise JavaBeans

The client-server model of application development has enjoyed considerable popularity. The client application resides on a local machine and accesses the data in a data store such as a relational database management system. This model works well as long as the system has only a few users. As more and more users need access to the data, these applications don’t scale well to meet the demands. Because the client contains the logic, it must be installed on each machine. Management becomes increasingly difficult.

Gradually the benefits of dividing applications into more than the two tiers of the client-server model becomes apparent. In a multi-tier application, only the user interface stays on local machines while the logic of the application runs in the middle tier on a server. The final tier is still the stored data. When the logic of an application needs updating, changes are made to the software of the middle tier on the server, greatly simplifying the management of updates.

But creating reliable, secure, and easily managed distributed applications is notoriously difficult. For example, managing transactions over a distributed system is a major task. Fortunately, using components that follow the EJB specification to build distributed systems relieves much of the burden by:

- Dividing the development of a distributed system into specific tasks that are assigned to specialists.

For example, if the application is an accounting system, the enterprise bean developer would need to understand accounting. The system administrator must know about monitoring a deployed and running application. Each specialist assumes a particular role.

- Making EJB server and container services available to the enterprise bean and application developers.

The EJB server provider and EJB container provider (who are often the same vendor) handle many of the more difficult tasks so the developers don't have to. For example, the container an enterprise bean runs in can provide transaction and security services to the bean automatically.

- Making enterprise beans portable.

Once a bean is written, it can be deployed on any EJB server that adheres to the Enterprise JavaBeans standard. Each bean is likely to include vendor-specific elements, however.

Roles in the development of an EJB application

The work of developing a distributed EJB application is divided into six distinct roles. Each role is assumed by an expert in their domain. By dividing the work this way, the task of creating and managing a distributed system becomes much easier.

Application roles

Those who assume the application roles write the code for the enterprise beans and the applications that use them. Both roles require an

understanding of how the business runs, although at different levels. These are the two application roles:

- Bean provider

Bean providers (also called bean developers) create the enterprise beans and write the logic of the business methods within them. They also define the remote home or local home and remote or local interfaces for the beans and they create the beans' deployment descriptors. Bean providers don't necessarily need to know how their beans will be assembled and deployed.

- Application assembler

Application assemblers write the applications that use the enterprise beans. These applications usually include other components, such as GUI clients, applets, JavaServer Pages pages (JSP), and servlets. These components are assembled into a distributed application. Assemblers add assembly instructions to the bean deployment descriptors.

Although application assemblers must be familiar with the methods contained within the enterprise beans so they can call them, they don't need to know how those methods are implemented.

JBuilder users who are interested in Enterprise JavaBeans are usually bean providers and application assemblers. Therefore, this book is written primarily for them. JBuilder has wizards, designers, and other tools that simplify the development of enterprise beans and the applications that use them.

Infrastructure roles

Without a supporting infrastructure, the enterprise beans and the applications that use them cannot run. Although the two infrastructure roles are distinct, they are almost always assumed by the same vendor. Together they provide system-level services to the enterprise beans and provide an environment in which to run. These are the two infrastructure roles:

- EJB server provider

EJB server providers are specialists in distributed transaction management, distributed objects, and other low-level services. They provide an application framework in which to run EJB containers. EJB service providers must provide, at a minimum, a naming service and a transaction service to the beans.

- EJB container provider

EJB container providers provide the deployment tools required to deploy enterprise beans and the runtime support for the beans. A container provides management services to one or more beans. They

communicate for the beans with the EJB server to access the services the bean needs.

In almost all cases, the EJB server provider and the EJB container provider are the same vendor. The Borland Enterprise Server provides both the server and the container.

Deployment and operation roles

The final steps in the development of an EJB distributed application are to deploy the application and to monitor the enterprise computing and network infrastructure as it runs. These are the deployment and operation roles:

- Deployer

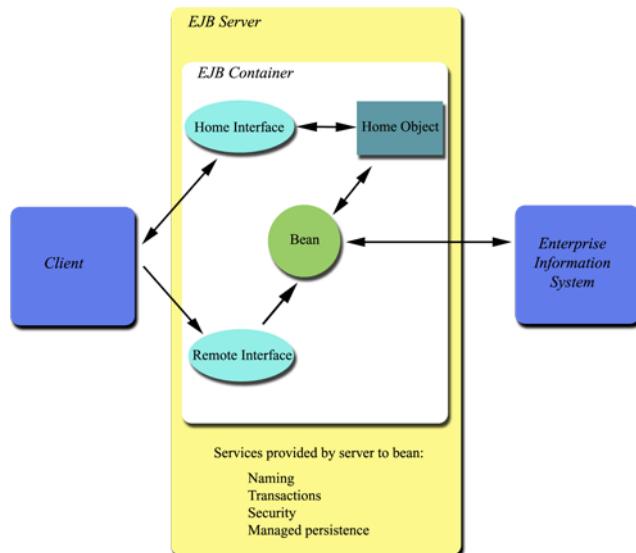
Deployers understand the operation environment for distributed applications. They adapt the EJB application to the target operation environment by modifying the properties of the enterprise beans using the tools provided by the container provider. For example, deployers set transaction and security policies by setting appropriate properties in the deployment descriptor. They also integrate the application with existing enterprise management software.

- System administrator

Once an application is deployed, the system administrator monitors it as it runs, and takes appropriate actions if the application behaves abnormally. System administrators are responsible for configuring and administrating the enterprise's computing and networking infrastructure that includes the EJB server and EJB container.

EJB architecture

Multi-tier distributed applications often consist of a client that runs on a local machine, a middle-tier that runs on a server that contains the business logic, and a backend-tier consisting of an enterprise information system (EIS). An EIS can be a relational database system, an ERP system, a legacy application, or any data store that holds the data that needs to be accessed. This figure shows a typical EJB multi-tier distributed system with three tiers: the client; the server, the container, and the beans deployed on them; and the enterprise information system.

Figure 2.1 EJB architecture diagram

Because our interest is how to develop enterprise beans, our focus is the middle tier.

The EJB server

The EJB server provides system services to enterprise beans and manages the containers in which the beans run. It must make available a JNDI-accessible naming service and a transaction service. Frequently an EJB server provides additional features that distinguish it from its competitors. The Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1 is an example of an EJB server.

The EJB container

A container is a runtime system for one or more enterprise beans. It provides the communication between the beans and the EJB server. It provides transaction, security, and network distribution management. A container is both code and a tool that generates code specific for a particular enterprise bean. A container also provides tools for the deployment of an enterprise bean, and a means for the container to monitor and manage the application.

The EJB server and EJB container together provide the environment for the bean to run in. The container provides management services to one or more beans. The server provides services to the bean, but the container

interacts on behalf of the beans to obtain those services. Almost always the EJB server and the EJB container are made by the same vendor and are simply two parts of an application server, such as Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1.

Although it is a vital part of the Enterprise JavaBeans architecture, enterprise bean developers and application assemblers don't have to think about the container. It remains a behind-the-scenes player in an EJB distributed system. Therefore, this book goes no further explaining what a container is and how it works. For more information about containers, refer to the "Enterprise JavaBeans Specification" itself at <http://java.sun.com/products/ejb/docs.html>. For specific information about the Borland EJB container, see the Borland Enterprise Server's documentation.

How an enterprise bean works

The bean developer must create these interfaces and classes:

- The remote home and/or local home interface for the bean

The home interface defines the methods a client uses to create, locate, and destroy instances of an enterprise bean.

- The remote and/or local interface for the bean

The remote or local interface defines the business methods implemented in the bean. A client accesses these methods through the remote interface.

- The enterprise bean class

The enterprise bean class implements the business logic for the bean. The client accesses these methods through the bean's remote interface.

Once the bean is deployed in the EJB container, the client calls the `create()` method defined in the home interface to instantiate the bean. The home interface isn't implemented in the bean itself, but by the container. Other methods declared in the home interface permit the client to locate an instance of a bean and to remove a bean instance when it is no longer needed. EJB 2.0 beans also allow the home interface to have business methods called `ejbHome` methods.

When the enterprise bean is instantiated, the client can call the business methods within the bean. The client never calls a method in the bean instance directly, however. The methods available to the client are defined in the remote or local interface of the bean, and the remote or local interface is implemented by the container. When the client calls a method, the container receives the request and delegates it to the bean instance.

Types of enterprise beans

An enterprise bean can be a session bean, an entity bean, or a message-driven bean.

Session beans

Session beans can be either stateful or stateless. Stateless beans don't maintain state for a particular client. Because they don't maintain conversational state, stateless beans can be used to support multiple clients.

A stateful session bean executes on behalf of a single client. In a sense, the session bean represents the client in the EJB server. Stateful session beans can maintain the client's state, which means they can retain information for the client. The classic example where a session bean might be used is a shopping cart for an individual shopping at an online store on the web. As the shopper selects items to put in the "cart," the session bean retains a list of the selected items.

Session beans can be short-lived. Usually when the client ends the session, the bean is removed by the client.

Entity beans

An entity bean provides an object view of data in a database. Usually the bean represents a row in a set of relational database tables. An entity bean usually serves more than one client.

Unlike session beans, entity beans are considered to be long-lived. They maintain a persistent state, living as long as the data remains in the database, rather than as long as a particular client needs it.

The container can manage the bean's persistence, or the bean can manage it itself. If the persistence is bean-managed, the bean developer must write code that includes calls to the database.

Message-driven beans

The EJB 2.0 specification introduced message-driven beans. They behave as a Java Message Service (JMS) listener, processing asynchronous messages. The EJB container manages the bean's entire environment.

Message-driven beans are similar to stateless session beans because they maintain no conversational state. Unlike session and entity beans, clients don't access them through interfaces. A message-driven bean has no interfaces, just a bean class. A single message-driven bean can process

messages from more than one client. A message-driven bean is essentially a block of application code that executes when a message arrives at a particular JMS destination.

Remote and local access

An EJB 2.0 component can be accessed remotely or locally. Clients that access a remote bean use the bean's remote and remote home interfaces. A remote home is often referred to as the home interface. A client with remote access to a bean can run on a different machine and use a different Java Virtual Machine (JVM) than the bean itself. In method calls to a remote bean, parameters are passed by value, which helps maintain loose coupling between the client and the bean.

A client with local access to a bean must run in the same JVM as the bean it accesses. A local client won't be an external client application, but rather another enterprise bean or web component. In method calls to a local bean, parameters are passed by reference, resulting in a tighter coupling between the calling bean or web component and the called bean.

Like the remote interface, the local interface provides access to the bean's business methods, while its local home interface provides access to the methods that control the life cycle of the bean as well as its finder methods. Often entity beans that have a container-managed relationship with other entity beans have local access to them.

Because beans with local interfaces must run in the same JVM, there is no need for remote calls. Therefore, the overhead of serializing and transporting objects is reduced. Usually this means greater performance.

Developing enterprise beans

The next few chapters explain how to use the JBuilder wizards, designers, and tools that make it easier and quicker to create your enterprise beans. It assumes that you understand what enterprise beans are, how they work, and what they require.

If your EJB knowledge is limited or you want more information about EJB development before you begin using JBuilder's EJB wizards and tools, start reading [Chapter 13, "Developing session beans"](#) and the chapters that follow it before beginning this chapter.

Developing Enterprise JavaBeans with JBuilder has several steps:

- 1 "Configuring the target server settings" in *Developing J2EE applications*"
- 2 "Introducing EJB modules" on page 3-2
- 3 Chapter 3, "Creating 2.0 session beans and message-driven beans with the EJB Designer" and
Chapter 4, "Creating 2.0 entity beans with the EJB Designer"
or
[Chapter 6, "Creating EJB 1.x components with JBuilder"](#) and
[Chapter 7, "Creating EJB 1.x entity beans from an existing database table"](#)
- 4 Chapter 8, "Compiling enterprise beans and creating a deployment module"
- 5 Chapter 11, "Using the Deployment Descriptor editor"
- 6 Chapter 9, "Running and testing an enterprise bean"
- 7 "Deploying to an application server" on page 10-7

Creating 2.0 session beans and message-driven beans with the EJB Designer

The JBuilder WebLogic Edition provides support for WebLogic Servers only

This chapter explains how to use JBuilder to create session beans and message-driven beans that are compliant with Sun Microsystems' Enterprise JavaBeans™ 2.0 specification. Even if you want to create EJB 2.0 entity beans and you are new to JBuilder and its EJB Designer, you should start with this chapter as it covers the basics of working with the EJB Designer. Then you can go to [Chapter 4, "Creating 2.0 entity beans with the EJB Designer"](#) to learn about the special features for creating entity beans.

For information about how to create components that are compliant with the Enterprise JavaBeans™ 1.1 specification, see [Chapter 6, "Creating EJB 1.x components with JBuilder."](#)

To help you create your EJB 2.0-compliant beans, the EJB Designer provides a rapid application development (RAD) environment for EJB 2.0 development. The EJB Designer is a true Two-Way Tool™ that allows you to design your enterprise bean visually as JBuilder generates the code from your design. You can make changes to your design either through the EJB Designer, or by editing the generated source code directly. Your source code and your design remain synchronized. As you work with the EJB Designer, your deployment descriptors are being created for you, preparing your bean for deployment to your target application server.

Introducing EJB modules

Each enterprise bean you create must belong to a JBuilder EJB module. An EJB module is a logical grouping of one or more beans that will be deployed in a single JAR file. An EJB module contains the information that is used to produce the deployment descriptor(s) for that JAR file. You can edit the content of an EJB module using the Deployment Descriptor editor.

Once you have an EJB module and have edited it to your liking with the Deployment Descriptor editor, you can Make or Build an EJB module to produce the JAR. JBuilder uses the deployment descriptor to help identify the class files to be packaged.

An EJB module can be one of two formats: XML or binary. Because an EJB module in XML format is essentially a text file, it's easier to work with if you are using a version control system. An EJB module in binary format is essentially the deployment descriptors in a .zip archive.

You can have more than one EJB module in a project. All the EJB modules in a single project use the same project classpath and JDK, and they are configured for the same target application server.

If you haven't done so already, follow the instructions in the chapter, "Configuring the target server settings" in *Developing J2EE Applications*.

Creating an EJB 2.0 module

You have multiple ways to create an EJB module:

- Use the EJB Module wizard to create an EJB module when you haven't created your enterprise beans yet.
- Create an EJB module from existing deployment descriptors.

Creating a module with the EJB Module wizard

If you haven't created your enterprise beans yet, begin by creating an EJB module with the EJB Module wizard.

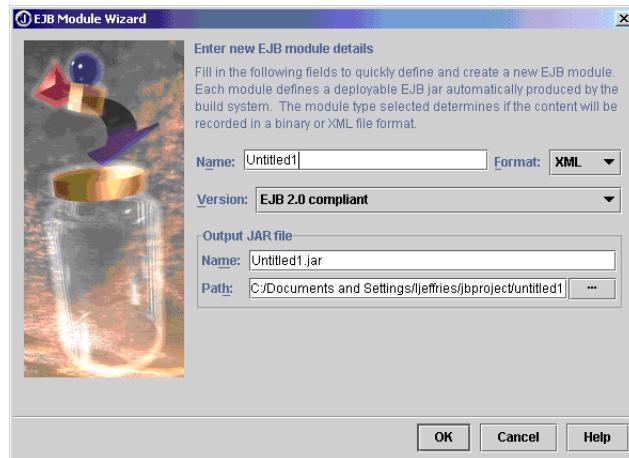
To create an EJB module,

- 1 Choose File | New and click the Enterprise tab.

Note

If the EJB wizards on the Enterprise page are disabled, you don't have the Enterprise version of JBuilder installed or you haven't selected a configured application server for your project. Also, check to ensure that you have selected an application server, rather than a web server. See "Configuring the target server settings" in *Developing J2EE Applications* for information about configuring and selecting an application server.

- 2** Double-click the EJB Module icon and the wizard appears:



If you don't have an open project before you begin the EJB Module wizard, the Project wizard appears first. After you create a new project, the EJB Module wizard then appears.

- 3** Specify the name of the EJB module.

- 4** Specify the format of the new module.

Your choices are binary, which internally stores the deployment descriptors in .zip format and was used before JBuilder 5, or XML, which stores the deployment descriptors in XML format. XML format allows users to merge changes if they are checking them into a source control system. Using the XML format is recommended.

- 5** Specify the version as EJB 2.0 compliant using the Version drop-down list.

The EJB 2.0 compliant option is disabled if your target application server does not support EJB 2.0

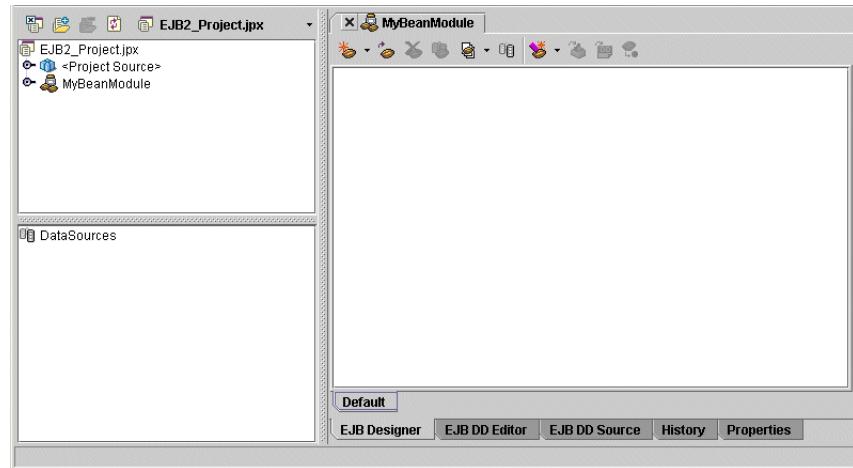
- 6** Specify the name of the JAR file your enterprise bean(s) will be in.

JBuilder entered a default name that is the same as the name of your EJB module. You can simply accept that name or specify another. JBuilder also entered a path based on your project path. You can change it to your liking or accept the default path.

- 7** Click OK to create the EJB module.

The EJB Designer appears. For example, if you are starting a project named `EJB2_Project` and you specified `MyBeanModule` as the name of the EJB

module, the EJB Designer, the project pane, and the structure pane would look like this:



Creating an EJB module from existing deployment descriptors

If you already have existing EJB 2.0 deployment descriptors for enterprise beans you created previously, you can create a new EJB module that contains the descriptors. You have two options:

- Use the EJB Module From Descriptors wizard, which creates a new EJB module that contains the deployment descriptors.
- Use the Project For Existing Code wizard, which creates a new JBuilder project containing one or more new EJB modules that contain the deployment descriptors. The wizard recognizes only Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1 or WebLogic 6.x or later vendor-specific deployment descriptors.

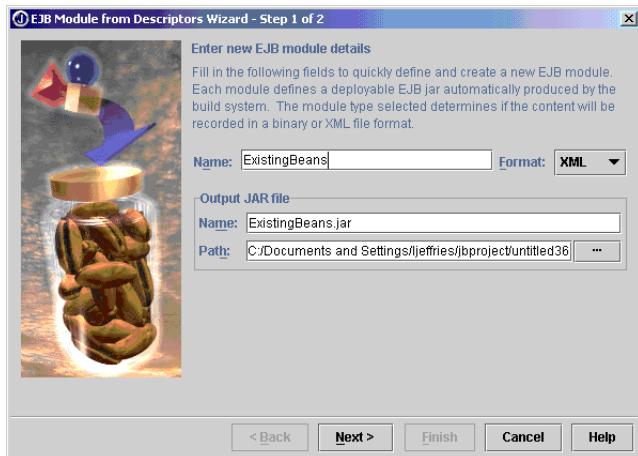
Both the EJB Module From Descriptors wizard and the Project For Existing Code wizard both merge the vendor-specific information into a JBuilder common format, which is then used to generate the deployment descriptors for the application server you have selected for your project as needed.

Using the EJB Module From Descriptors wizard

The EJB Module From Descriptors wizard imports the `ejb-jar.xml` deployment descriptor and BEA WebLogic and Borland Enterprise Server proprietary deployment descriptors only into a new EJB module.

To use the EJB Module From Descriptors wizard,

- 1** Choose File | New and click the Enterprise tab.
- 2** Double-click the EJB Module From Descriptors wizard icon and the wizard appears:



If you don't have any open project before you begin the EJB Module From Descriptors wizard, the Project wizard appears first. After you create a new project, the EJB Module From Descriptors wizard then appears.

- 3** Specify the name of your new EJB module.

- 4** Specify the format of the new module.

Your choices are binary, which internally stores the deployment descriptors in .zip format and was used before JBuilder 5, or XML, which stores the deployment descriptors in XML format. XML format allows users to merge changes if they are checking them into a source control system. Using the XML format is recommended.

- 5** Specify the name and path of the JAR file your enterprise bean will be in.

JBuilder entered a default name that is the same as the name of your EJB module. You can simply accept that name or specify another.

- 6** Click Next and specify the directory that contains the existing deployment descriptor(s) you want to make up the module. (Frequently this is in the META-INF directory of a JAR.) When you do,

the wizard lists the deployment descriptors in the specified directory in the Identified EJB Descriptors field. It also lists the root source directory for these descriptors in the Root Source Directory field.



- 7 If you want to add the root source directories to your project, check the Add Root Source Directories To Project check box.
- 8 Click Finish to create the EJB module that contains the deployment descriptors for the existing bean(s).

Using the Project For Existing Code wizard

The Project For Existing Code wizard creates a new JBuilder project from an existing body of work. That body of work can include EJB 1.1 or 2.0 vendor-specific deployment descriptors if they were developed using the Borland Enterprise Server 5.1.1 - 5.2.1 or the WebLogic Server 6.x or later.

To use the Project For Existing Code wizard to create a new JBuilder project and a new EJB module for each directory containing EJB deployment descriptors the wizard scans, follow these steps:

- 1 Choose File | New to display the object gallery.
- 2 Click the Project tab.
- 3 Double-click the Project From Existing Code icon to start the Project From Existing Code wizard.

- 4 Use the ... button to specify the root directory where you want the scan to begin.

For example, suppose you have a `clients` directory that contains two subdirectories named `personal_info` and `accounts`, each containing EJBs with deployment descriptors. You want to create both a `personal_info` and an `accounts` EJB module. If you specify `clients` as the Directory, the wizard starts its scan with the `clients` directory, then scans the `personal_info` and `accounts` directories. Each generated EJB module is given a default name generated from the subdirectory name where the descriptors were found. So the new project the wizard creates will contain two EJB modules, `personal_info` and `accounts`.

For complete information about using remaining pages of the Project From Existing Code wizard, click the wizard's Help button or see "Creating a project from existing files" in the "Creating and managing projects" chapter in *Building Applications with JBuilder*.

Displaying the EJB Designer

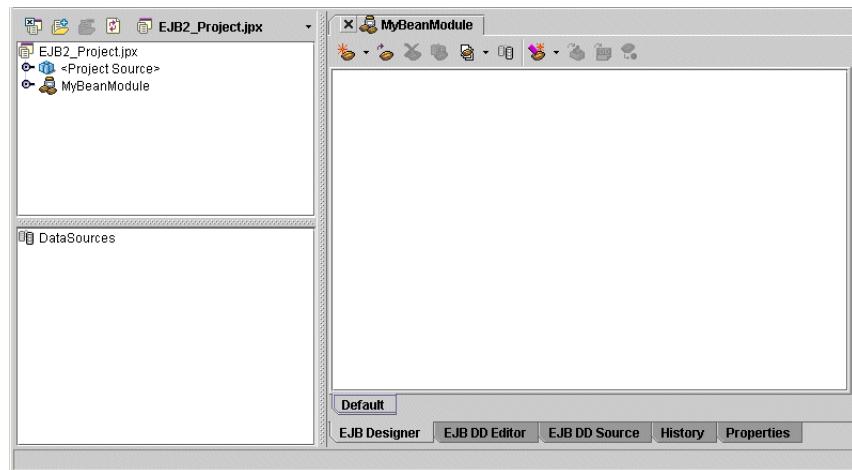
You use the EJB Designer to develop EJB 2.0 enterprise beans. There are two ways to display the EJB Designer to begin creating an enterprise bean:

- Use the EJB Module wizard to create a new module. If you choose EJB 2.0-compliant as the Version of EJB module you are creating, the wizard displays the new module it creates in the project pane and displays the EJB Designer, ready for you to begin a new enterprise bean. See "[Creating a module with the EJB Module wizard](#)" on page 3-2 for more information.
- Use the EJB 2.0 Designer wizard. These are the steps you follow:
 - a** Choose File | New, click the Enterprise tab, and double-click the EJB 2.0 Designer.
 - b** Select the EJB module you want the bean you create to become part of from the list of Available EJB Modules.

Only the available EJB 2.0 modules appear in the list. If you don't have a module yet, you can choose the New button, which starts the EJB Module wizard. When the wizard completes, the EJB 2.0 Designer wizard continues.

c Click OK.

The EJB Designer appears. For example, if you are starting a project named `EJB2_Project` and you specified `MyBeanModule` as the name of the EJB module, the EJB Designer, the project pane, and the structure pane would look like this:



Any time another view, such as the Source view or a Deployment Descriptor panel, is present, you can switch to the EJB Designer view by using one of these two methods:

- Double-click the EJB module's node in the project pane and click the EJB Designer tab in content pane.
- Click the EJB module's tab at the top of the content pane. This option is available only if the module is open in the current project.

Quickening the display of your EJBs

If you have many EJBs in the EJB Designer, you might want to collapse the bean representations so that you can see only the names, making it easier to locate the bean you want and quickening the display of the beans. Choose Tools | IDE Options and check the Only Expand Selected EJBs option. If you do this, the EJB Designer will display the beans with their names only. To see the fields, methods, etc. of the bean, you then click it to select it in the EJB Designer. To see the all of the beans in their entirety again, uncheck the Only Expand Selected EJBs option. When you do, the Lazily Load EJBs option is enabled.

If you do select the Only Expand Selected EJBs option, you can then choose to have the EJB Designer load your beans into memory only as they are selected and expanded. Check the Lazily Load EJBs option, which can speed up the EJB Designer's initial display.

Preventing the saving of EJB Designer layout changes

If, as you work with the EJB Designer, you find you are inadvertently moving beans around when you don't intend to, you can prevent JBuilder from saving the layout changes you make in the EJB Designer. Choose Tools | IDE Options, click the EJB Designer tab, and uncheck the Dirty Module On Layout Changes option, which is on by default. When this option is on, all changes to the layout of the beans in the EJB Designer are automatically saved. When this option is off, layout changes are stored only if you make changes in your code.

Caution If you uncheck the Dirty Module On Layout Changes option, none of your layout changes are saved.

How the EJB Designer names EJB files

The names of the files JBuilder generates when you use the EJB Designer to create enterprise beans varies depending on the type of enterprise bean you are creating. In JBuilder, session beans are remote by default and entity beans are local by default. Therefore, the EJB Designer generates a remote home interface for a session bean and a local home interface for an entity bean. In each case the name of the home interface, whether remote for a session bean or local for an entity bean, will be <Bean Name>Home. Likewise, the interface that declares the business methods is the remote interface for a session bean and the local interface for an entity bean. For example, if you decide to generate both the local and remote interfaces for a *session bean* named Component, these are the files the EJB Designer generates for you:

- ComponentHome - the remote home interface
- ComponentBean - the bean class
- Component - the remote interface
- ComponentLocalHome - the local home interface
- ComponentLocal - the local interface

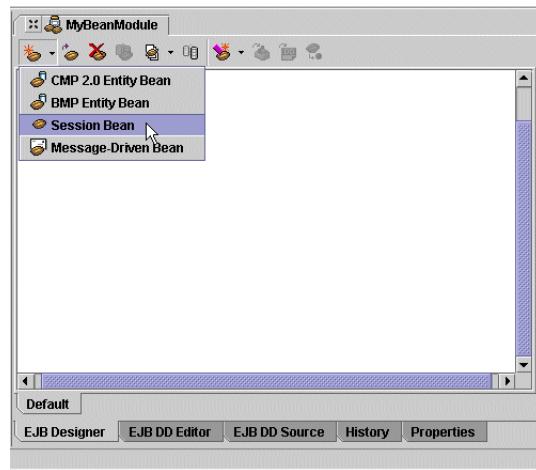
If you do the same thing for an *entity bean* with the same name of Component, these are the files the EJB Designer generates for you:

- ComponentHome - the local home interface
- ComponentBean - the bean class
- Component - the local interface
- ComponentRemoteHome - the remote home interface
- ComponentRemote - the remote interface

Creating session beans

To create a session bean,

- 1 Right-click the EJB Designer pane and choose Create EJB | Session Bean or click the Create EJB icon on the toolbar and choose Session Bean:

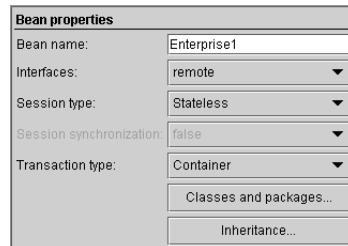


A representation of the session bean appears in the EJB Designer named `Enterprise<n>` (where `<n>` stands for a number) and three files appear in the project pane.

These are the generated files that appear in the project pane:

- `Enterprise<n>` - The remote interface of the session bean.
- `Enterprise<n>Bean` - The bean class of the session bean.
- `Enterprise<n>Home` - The home interface of the session bean.

- 2 Click the representation of the bean in the EJB Designer on the top row, the name of the bean. An inspector appears:



- 3 Within the inspector, change the name of the bean to a name of your choosing.

The files in the project pane are renamed to reflect the name change.

- 4 Use the Interfaces drop-down list to select from Remote, Local, or Remote/Local.

Selecting Remote means that the EJB Designer generates the bean class with a remote home and remote interface. This is the default value.

Selecting Local means that the EJB Designer generates a bean class with a local home and a local interface. Because the EJB Designer generates the Remote interfaces by default for session beans, selecting Local changes the name and content of the bean's files in the project pane to reflect a bean that can be accessed locally only. The local home interface name will have `LocalHome` appended to it, and local interface name will have `Local` appended to it.

If you select the Remote/Local option, all the remote and local files will appear in the project pane giving you a total of five bean files.

- 5 Specify the type of session bean in the Session Type field: Stateless or Stateful. For more information about session bean types, see ["Types of session beans" on page 13-1](#).
 - 6 Specify the Transaction Type: Container (for container-managed) or Bean (for bean-managed). For more information about container-managed and bean-managed transactions, see ["Bean-managed versus container-managed transactions" on page 18-3](#).
 - 7 If you want the bean to implement the `SessionSynchronization` interface, set the Session Synchronization attribute to true. This option is available for stateful session beans only. For more information about the `SessionSynchronization` interface, see ["The SessionSynchronization interface" on page 13-8](#).
 - 8 Click the Classes And Packages button to make any changes to the package name, bean and bean class names, and the interface names. Click OK.
- Changes you make appear in the project pane.
- 9 Click the Inheritance button if you want to change the parent of the bean class to some other than `java.lang.Object`.

Viewing a bean's source code

At any time as you work within the EJB Designer, you can double-click a generated file (`.java`) in the project pane to see its source code. Or you can right-click the bean representation in the EJB Designer and choose View Bean Source in the context menu or simply click the View Bean Source icon on the toolbar; JBuilder displays the source code for the bean class. If you right-click on a specific field or method in the bean representation and choose View Bean Source, your cursor will be located where the field or method is defined in the bean class source code when the editor appears. To return to the EJB Designer, double-click the EJB module node in the project pane.

Modifying the bean

So far, JBuilder has generated just skeleton classes for the bean you are developing. Continue working within the EJB Designer to make the bean what you want. You can add new fields and methods and modify their attributes. At any time you can switch directly to the source code for the bean and add new code and modify existing code. All your changes will be represented in the EJB Designer when you switch back to it.

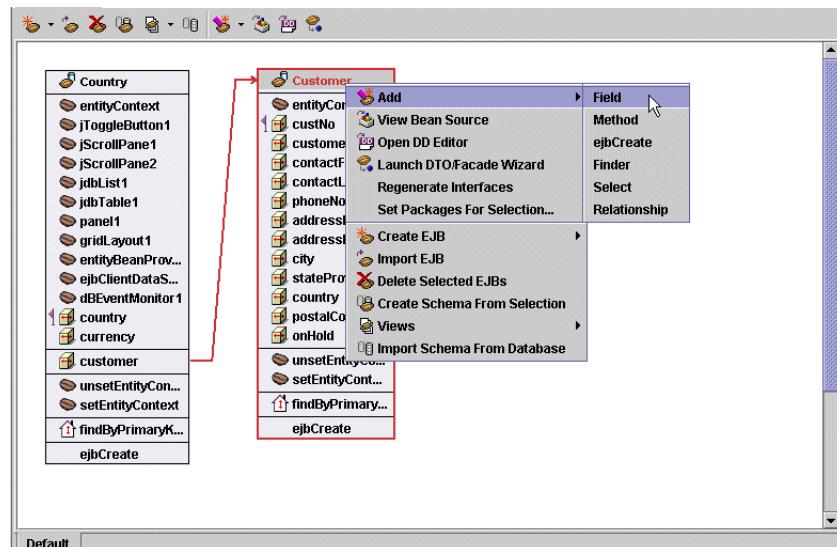
Editing bean attributes

Edit bean attributes by clicking the element of the bean you want to modify. When you do, an inspector appears that you can use to make your changes.

Adding a new field

To add a new field to a bean,

- 1 Right-click the bean representation in the EJB Designer and choose Add | Field or click the Add icon on the toolbar and choose Field:



A new field titled `untitledField1` appears in the bean. The field's inspector appears:

Field name:	<code>untitledField1</code>
Type:	<code>java.lang.String</code>
Getters:	<code>none</code>
Setters:	<code>none</code>

- 2** Use the inspector to modify the attributes of `untitledField<n>`, including the name of the field:

- Give the field a meaningful name using the Field Name field.
- Specify the Java type of the field you are declaring in the Type box: for example, `java.lang.String`. You can use the ... button to browse to a Java object.
- Specify where the getter and setter access methods are declared in the drop-down lists: in the local interface, in the remote interface, in both interfaces, or in no interface.

To complete your field, double-click the bean class in the project pane to display the source code. Find where the field's getter and setter methods are defined and add any additional logic you might want.

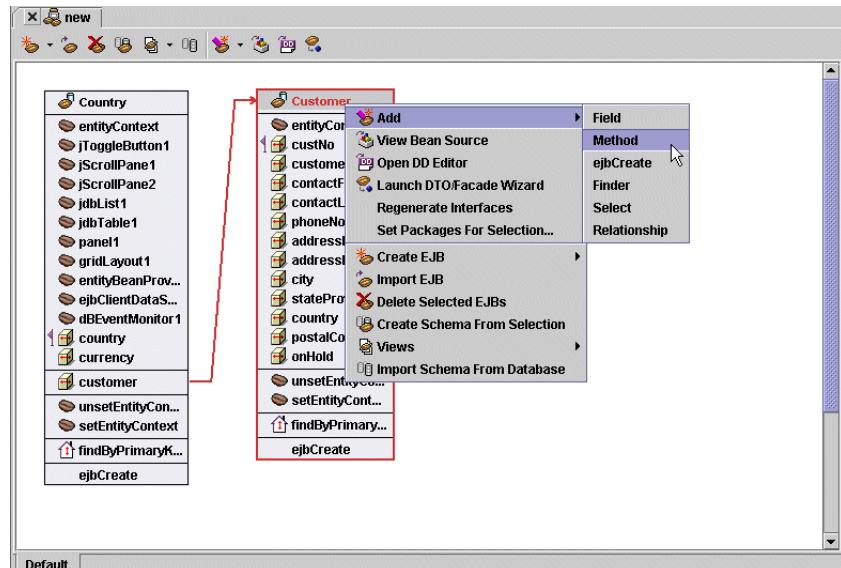
Deleting a field

To remove a field from a bean, right-click the field in the bean representation and choose Delete Field.

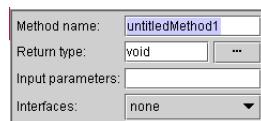
Adding a new method

To add a new method to a bean,

- 1** Right-click the bean representation in the EJB Designer and choose Add | Method or click the Add icon on the toolbar and choose Method:



A new method titled `untitledMethod<n>` appears in the bean. The method's inspector appears:



- 2 Use the inspector to modify the attributes of `untitledMethod<n>`:
 - a Give the method a meaningful name in the Method Name field.
 - b Specify the return type for the method by typing in an appropriate Java return type or by clicking the ... button to browse to a Java object.
 - c Enter any arguments you want passed to the method in the Input Parameters field. You declare the arguments as you would in code; for example, `Integer age`. Separate multiple entries with commas.
 - d Use the Interfaces drop-down list to specify where you want the method declared: in the home interface (the remote home), the local home interface, the home and local home interfaces, the local interface, the remote interface, the remote and local interfaces, or in no interface at all.
- 3 To complete your method, double-click the bean class in the project pane to display its source code. Find the skeleton method you just added in the source and, in the method's body, write the logic of the method.

Removing a method

To remove a method from a bean, right-click the method in the bean representation and choose Delete Method.

Working with `ejbCreate()` methods

You can use the EJB Designer to specify parameters to pass to an `ejbCreate()` method and to generate multiple `ejbCreate()` methods for your stateful session beans and entity beans. (Stateless session beans can have one `ejbCreate()` method only.)

To specify parameters to pass to the default `ejbCreate()` method,

- 1 Click `ejbCreate` in the bean representation in the EJB Designer to display an inspector.
- 2 Specify the parameters you want passed to the method in the Input Parameters field. Enter the type of the parameter followed by the parameter name with commas between the parameters as if you were

defining the parameters in code. For example, you might use `String group_id, String name` to pass `group_id` and `name` parameters to `ejbCreate()`.

Note You won't be able to specify parameters for the default `ejbCreate()` method of a stateless session bean. Stateless session beans never need more than a single parameterless `ejbCreate()` method because they don't maintain state.

To add another `ejbCreate()` method to your stateful session or entity bean,

- 1 Right-click the bean representation in the EJB Designer.
- 2 Choose Add | ejbCreate to add a new `ejbCreate` method to the bean representation.
- 3 Accept the default name or specify a different one as the value of the Method Name in the new method's inspector.
- 4 Specify parameters you want passed to the method as the value of Input Parameters.

Now you must go into the source code of your bean class and add the logic you want to the `ejbCreate()` method(s).

Regenerating a bean's interfaces

When you work directly in the bean's source code, you might make changes to the bean class, but forget to make those changes in the home/local home and remote/local interfaces. You can choose to have the EJB Designer regenerate those interfaces for you based on the current state of your bean class.

To regenerate the interfaces of your bean,

- 1 Right-click the top of the bean representation in the EJB Designer.
- 2 Choose Regenerate Interfaces from the context menu that appears.

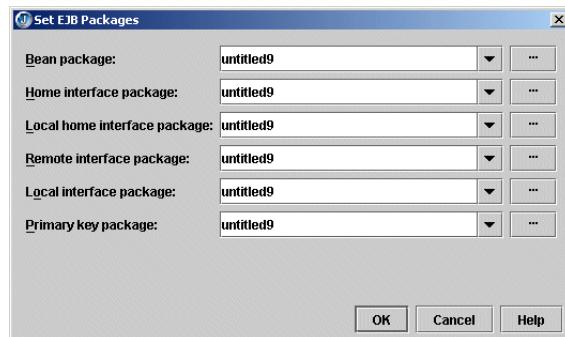
If you prefer to have the EJB Designer regenerate the bean's interfaces automatically each time you make changes to the source code of the bean class, follow these steps:

- 1 Choose Tools | IDE Options.
- 2 Click the EJB Designer tab.
- 3 Check the Always Regenerate Interfaces check box.
- 4 Click OK to close the dialog box.

Setting packages for a enterprise beans

You can quickly change packages for all the classes and interfaces of an enterprise bean all at once. This is especially useful if, for example, you want to move all the classes and interfaces of a bean to another package. Follow these steps:

- 1 Right-click the bean in the EJB Designer and choose Set Packages For Selection. This dialog box appears:



- 2 Use the dialog box to change the package names for the enterprise bean parts. If you want all the parts of the bean to be in the same package, simply specify the name of the package you want as the Bean Package Name and the value of all the remaining fields will change to the name of that package. If you want the parts of the bean to be in different packages, you can edit each field individually. If a package exists in your current project, you can select it as a field's value by using the drop-down arrow to display a list of existing packages. You can also use the ... button to display a Select A Package dialog box to select an existing package. If the package doesn't exist yet, type a new package name in a field, and when you choose OK, you will see the new package created in the project pane that contains the classes and/or interfaces you specified for that package.

- 3 Choose OK.

You can see the results of changing the package for the bean classes and interfaces in the project pane.

Importing beans

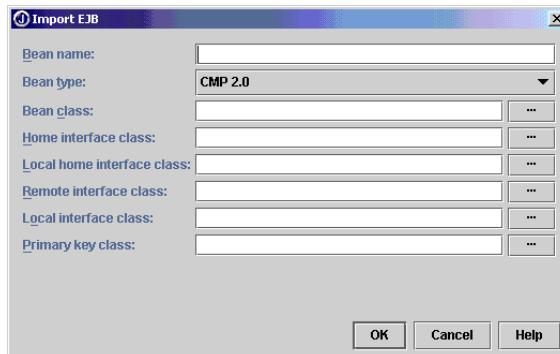
You can import an enterprise bean into an EJB module using the EJB Designer. You'll find this useful if you want to import a bean from one EJB module to another or if you've obtained a bean that has no accompanying

deployment descriptors. The source code of the bean you are importing must be in the project's source path.

To import an enterprise bean,

- 1 Open the EJB module you want to import the bean into in the EJB Designer.
- 2 Right-click the EJB Designer pane and choose Import EJB or click the Import EJB icon on the toolbar.

The Import EJB dialog box appears:



- 3 Type in the name you want for the bean in the Bean Name field.
- 4 Select the type of bean you are importing from the Bean Type drop-down list. Your choices are CMP 2.0, BMP, Session, or Message-Driven.
- 5 Specify the bean class name and location in the Bean Class field. You can use the ... button to locate the bean and the package information will be filled in automatically.
- 6 Use the interface class fields to specify the interfaces you want to import and their location. You can use the ... button to locate the interfaces and the package information is filled in for you automatically.
- 7 If you want to import a primary key class for an entity bean, specify the primary key class and its location in the Primary Key Class field.
- 8 Click OK.

The new bean is imported into the EJB module. Its bean representation appears in the EJB Designer along with its inspector, ready for you to edit bean properties.

Organizing beans with views

The EJB Designer allows you to group sets of enterprise beans on views, which help you organize and develop complex EJB projects. Each view acts much like a worksheet in a spreadsheet or a page in a multi-page dialog box. While your EJB module can contain many enterprise beans, only the ones on the current view are visible at any one time in the EJB Designer. Therefore, using views removes clutter on the screen and allows you to focus on the beans you are interested in at any one moment.

When you create a new EJB module, the default view appears in the EJB Designer. You can rename the default view or create a new view, if you choose.

To rename the default view,

- 1 Right-click the EJB Designer pane and choose Views | Rename View or click the View icon on the toolbar and choose Rename View.
- 2 Type in the new name you want to use for the view in the dialog box that appears and click OK or press *Enter*.

The new name for the view appears on the view tab at the bottom of the EJB Designer pane.

To create a new view,

- 1 Right-click the EJB Designer pane and choose Views | New View or click the Views icon on the toolbar and choose New View.
- 2 Type in the name you want to use for the new view in the dialog box that appears and click OK or press *Enter*.

A new tab appears with the name you specified near the bottom of the EJB Designer pane. To make the new view you added the current view, click its tab.

To move one or more EJBs to another view,

- 1 Select one or more EJBs on a view.
- 2 Right-click one of the selected EJBs.
- 3 Choose Views | Move Selection or click the Views icon on the toolbar and choose Move Selection.
- 4 Select the view you want to move the bean(s) to from the drop-down list.

To copy one or more EJBs to another view,

- 1** Select one or more EJBs on a view.
- 2** Right-click one of the selected EJBs.
- 3** Choose Views | Copy Selection or click the Views icon on the toolbar and choose Copy Selection.
- 4** Select the view you want to move the bean(s) to from the drop-down list.

To remove one or more EJBs from a view,

- 1** Select one or more EJBs on a view.
- 2** Right-click one of the selected EJBs.
- 3** Choose Views | Remove Selection.

When you use the Views | Remove Selection command, the EJB representations are removed from the current view; they aren't permanently removed from the EJB module or project. If the EJBs you selected to remove don't exist on any other view, they will reappear on the default view after you have removed them.

If you want to delete one or more EJBs and the corresponding source code altogether, see "["Removing beans" on page 3-21](#)".

To remove a view,

- 1** Make the view you want to remove the current view.
- 2** Right-click the EJB Designer pane.
- 3** Choose Views | Delete View or click the Views icon on the toolbar and choose Delete View.

The view disappears. If the deleted view had beans on it, the beans will reappear on the default view.

Finding beans

If you have several views in an EJB module, it's possible to lose track of the location of a bean. To quickly find a particular bean,

- 1** Choose Search | Find EJB or *Ctrl+F*.

A Find EJB dialog box appears. All the EJBs in the module are listed with the view they are in parentheses.

- 2 Select the EJB you are looking from the EJB Names list or type the EJB name in the EJB Name field at the top of the dialog box. You may enter wildcard characters (*) and (?) in the EJB name.
- 3 Click OK.

The view that contains the EJB you are searching for becomes the current view, the specific EJB appears, and its inspector opens.

Arranging beans

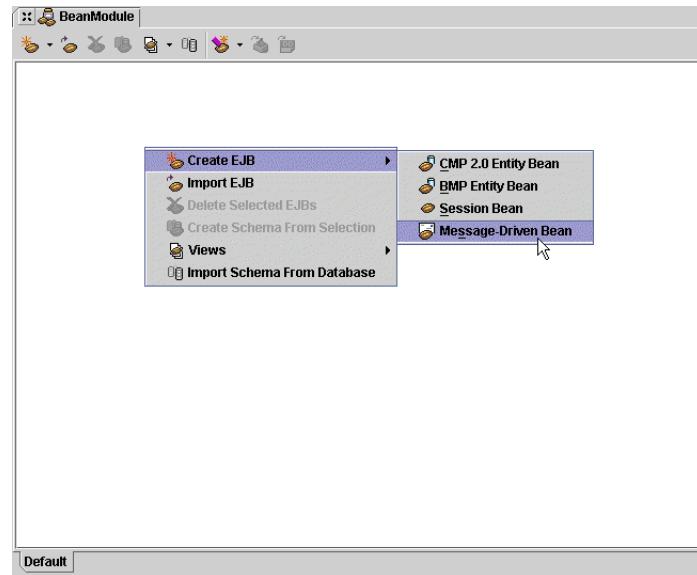
You can choose to have the EJB Designer rearrange your enterprise beans for you on a view:

- 1 Right-click the EJB Designer pane.
- 2 Choose Views | Arrange EJBs or click the Views icon on the toolbar and choose Arrange EJBs.

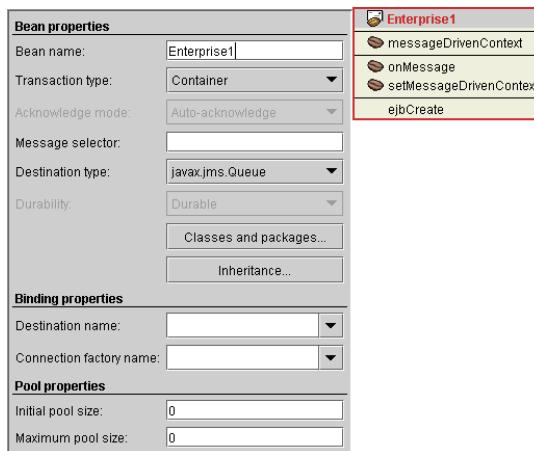
The EJB Designer attempts to rearrange the beans in a logical pattern.

Creating message-driven beans

To create a message-driven bean, begin by right-clicking the EJB Designer and choosing Create EJB | Message-Driven Bean or by clicking the Create EJB icon on the toolbar and choose Message-Driven Bean:



A representation of the message-driven bean appears in the EJB Designer along with its inspector:



Most of the properties displayed in the inspector are deployment descriptor attributes. The standard properties appear in the Bean Properties section of the inspector. The vendor-specific properties appear below the Bean Properties and will vary depending on your selected application server. For information about some of these properties, see [Chapter 15, "Developing message-driven beans."](#)

As with session beans, you add new fields and methods by right-clicking the bean representation and selecting a menu item from the menu that appears. Use the inspectors that appear when you click a field or method to modify field and method attributes. See ["Modifying the bean" on page 3-12](#) for complete details.

Note that you won't be able to specify interfaces when you add a new method because message-driven beans don't have interfaces.

To complete the bean, double-click the new message bean node in the project pane to switch to the source code for the bean class. Find the `onMessage()` method in the class and, in the method's body (double-clicking `onMessage()` in the structure pane is a quick way to move your cursor to it in the source code), add the logic that responds to an incoming message.

Removing beans

To remove one or more EJBs from the EJB Designer and to delete the corresponding source files in the project pane,

1 Select the bean representation in the EJB Designer:

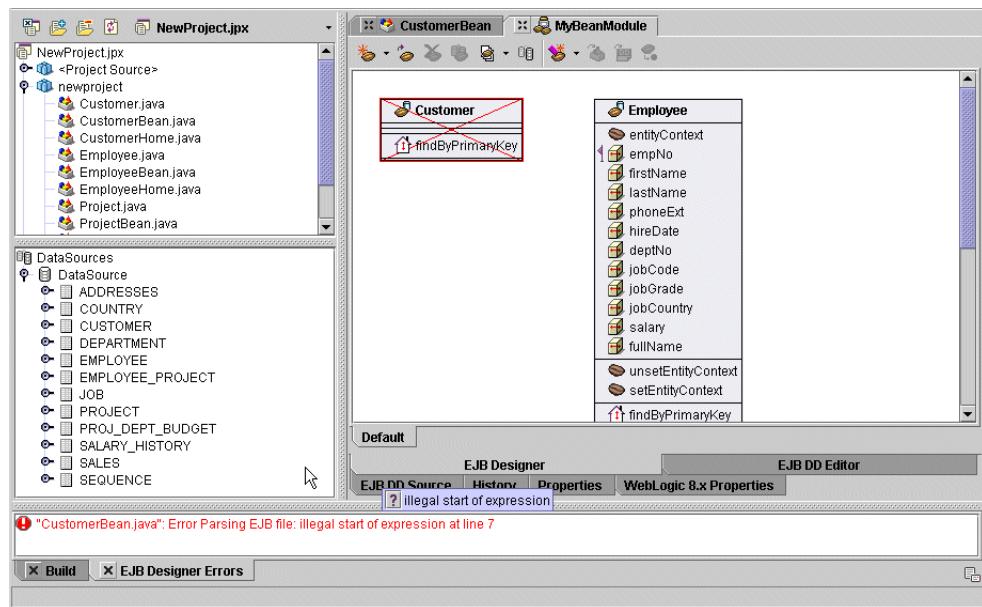
- To select a single bean, click it.
- To select multiple beans, *Ctrl-click* each bean you want to select or drag the mouse pointer around those you want to select.

- 2 Right-click one of the selected beans and choose Delete Selected EJBs, press the Delete key, or choose Delete Selected EJBs on the EJB Designer toolbar.

A dialog box appears explaining that the deletion will be permanent and asking you to confirm the deletion. By default, the Delete Source Files From Project check box is checked in the dialog box. If you just want to delete the bean deployment descriptors and leave the source files untouched, uncheck this option. You might find unchecking this option useful if you are sharing source code between modules or if you are simply removing beans for which the source code no longer exists.

Fixing errors in your bean

While you are working between the EJB Designer and your source code to create your bean, it's possible you might introduce a syntax error in your code. If you do, when you display the EJB Designer, an error message appears in the message pane and the bean representation of that bean appears crossed out.

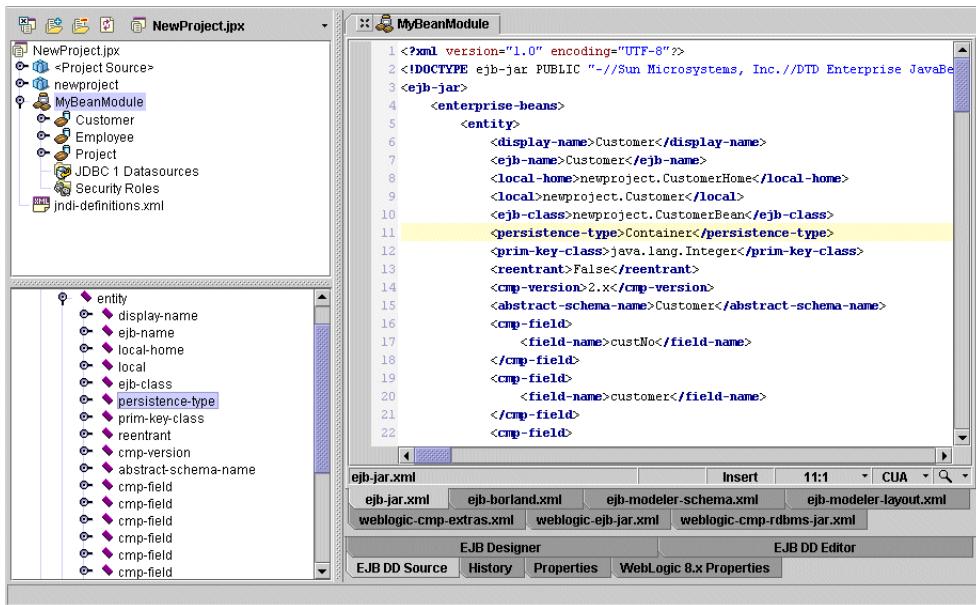


To rapidly go to the offending line in your source code, double-click the error message. Fix the error in your code, then return to the EJB Designer. Your bean representation should now appear normally.

To remove the EJB Designer Errors tab in the message pane, right-click the tab and choose Remove EJB Designer Errors Tab.

Viewing the deployment descriptors

As you work with the EJB Designer to build enterprise beans and modify their attributes, the deployment descriptors for your beans are being created for you. To see the XML source code of the deployment descriptors, click the EJB DD Source tab at the bottom of the content pane. You'll see another series of tabs at the bottom of the content pane. Which tabs are present depend on which application server you are targeting:



Displaying the Deployment Descriptor editor

JBuilder provides a Deployment Descriptor editor that allows you to view the current settings and edit them. A bean's deployment descriptor information is spread among several panels. There are multiple ways to display the Deployment Descriptor editor:

- Double-click the EJB module in the project pane and click the EJB DD Editor tab at the bottom of the content pane. The panel that appears lets you set a few JAR file attributes.
- To view and edit the deployment descriptor information for a single bean, expand the EJB module node in the project pane and double-click the bean in the EJB module's list of beans. The General panel of the Deployment Descriptor editor appears. You can click any of the other tabs at the bottom of the Deployment Descriptor editor to view other panels.

- To view and edit the deployment descriptor for a single bean while you are working on it in the EJB Designer, right-click the top of the bean representation in the EJB Designer and choose Open DD Editor in the context menu that appears, or simply click the Open DD Editor icon on the toolbar.

To return to the EJB Designer, double-click the EJB module node in the project pane or click the EJB module's tab at the top of the content pane.

Tip Pressing *F12* switches between the EJB Designer and the Deployment Descriptor editor.

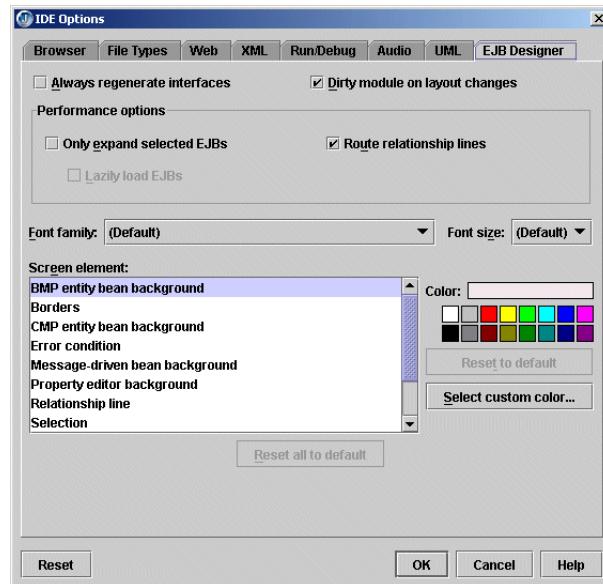
For more information about using the Deployment Descriptor editor, see [Chapter 11, "Using the Deployment Descriptor editor."](#)

Setting IDE options for the EJB Designer

You can modify the font and change the color of elements of the EJB Designer as well as select how EJBs are displayed and loaded in the EJB Designer:

- 1 Choose Tools | IDE Options.
- 2 Click the EJB Designer tab.

This page appears in the IDE Options dialog box:



- 3 Make your selections to change the screen elements as you wish and click OK when you are done. Click the dialog's Help button if you need assistance using the dialog box or and for more information about the options.

Taking the next step

Now that you've designed your session or message-driven beans, you're ready to compile them. See [Chapter 8, "Compiling enterprise beans and creating a deployment module."](#)

If you are seeking information about creating entity beans in the EJB Designer, see [Chapter 4, "Creating 2.0 entity beans with the EJB Designer."](#)

Creating 2.0 entity beans with the EJB Designer

The JBuilder WebLogic Edition provides support for WebLogic Servers only

This chapter explains how to use JBuilder to create entity beans that are compliant with Sun Microsystems' Enterprise JavaBeans™ 2.0 specification. For information about how to create entity beans that are compliant with the Enterprise JavaBeans™ 1.1 specification, see [Chapter 6, "Creating EJB 1.x components with JBuilder"](#) and [Chapter 7, "Creating EJB 1.x entity beans from an existing database table."](#)

You should be familiar with the material in [Chapter 3, "Creating 2.0 session beans and message-driven beans with the EJB Designer"](#) before you read this chapter. That chapter explains how to use the features of the EJB Designer. You will use those same features to build entity beans. The EJB Designer has additional features that help you create entity beans, however. This chapter describes using those features.

You can begin creating your entity beans in one of two ways:

- 1 Import an existing data source into the EJB Designer and use it to create CMP 2.0 entity beans.
- 2 Right-click the EJB Designer and choose CMP 2.0 Entity Bean or BMP Entity Bean. Or you can click the Create EJB icon on the EJB Designer toolbar and choose either CMP 2.0 Entity Bean or BMP Entity bean. Add and modify fields and methods as you would with session and message-driven beans.

Creating CMP 2.0 entity beans from an imported data source

Within the EJB Designer, you can import an existing JDBC data source and then use it to create your entity beans.

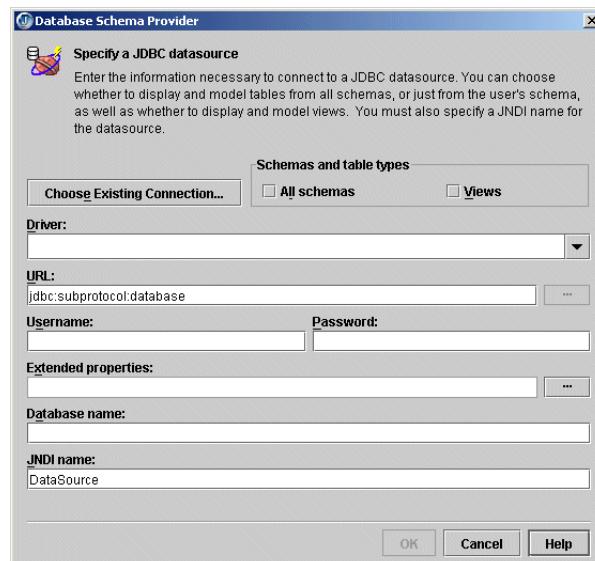
Importing a data source

When you import a data source, you are actually importing the structure of the data source, its schema. You can then modify that schema to meet your needs.

To import a data source,

- 1 Right-click the DataSources node in the structure pane and choose Import Schema From Database, or right-click the EJB Designer pane and choose Import Schema From Database (or simply click the Import Schema From Database icon on the EJB Designer toolbar).

The Database Schema Provider dialog box appears:



- 2 Specify a JDBC data source.

Enter the information that's needed to connect to a JDBC data source.

To use an existing connection, click the Choose Existing Connection button and select a connection. Other required information for this page is then filled in automatically except the password, which you must enter yourself if your connection requires one.

If you don't have an existing connection or want to create another, select a driver from the Driver drop-down list and specify an URL. The

drivers that appear are those you set up using Tools | Enterprise Setup on the Database Drivers page. See “Setting up JDBC drivers” in the “Configuring the target application server settings” chapter of *Developing J2EE Applications* for more information.

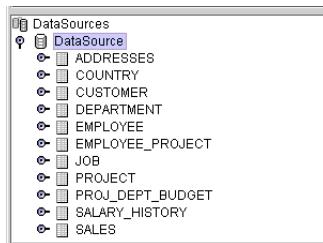
Specify the Username required to access the data source, and if a password is required, type in the password. Select any extended properties you need. Finally, specify the name of the database and specify a JNDI name for the data source or accept JBuilder’s default name.

3 Specify which Schemas And Table Types options you want.

If you check the All Schemas option, all schemas the user has rights to for the connection will be used. If you leave All Schemas unchecked, just the schemas with the same name as the username are used, potentially reducing the time required to make the connection and load the data.

Check the Views option if you want to have views loaded into the EJB Designer. If you don’t want to load views, leave the Views option unchecked.

The EJB Designer attempts to connect to the specified data source. If the attempt to connect succeeds, the specified data source appears in the structure pane:



If the EJB Designer detects another data source in the project with the same URL, it will ask if you want to use the same name.

jndi-definitions.xml

The JBuilder WebLogic Edition provides support for WebLogic Servers only

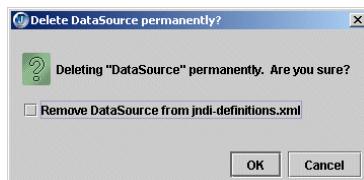
All data source definitions for a project are stored in `jndi-definitions.xml`.

To remove a data source from `jndi-definitions.xml`,

- 1 Right-click a `DataSource` node in the structure pane.

2 Choose Delete.

The Delete DataSource Permanently? dialog box appears:



3 Check the Remove DataSource From jndi-definitions.xml check box and choose OK.

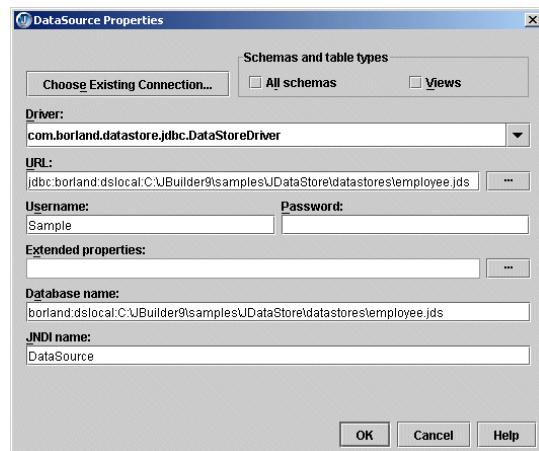
Once a data source is removed from the file, it won't be deployed if your target application server is Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1.

Modifying the imported data source schema

Once you have imported a data source, you can make changes to it before you use it to create your entity beans. Right-click elements in the structure pane to see the options available to make changes. These are the options for the various elements:

- For the top data source node, use the right-click context menu to add a table, edit data source properties, or to rename or delete the imported data source.

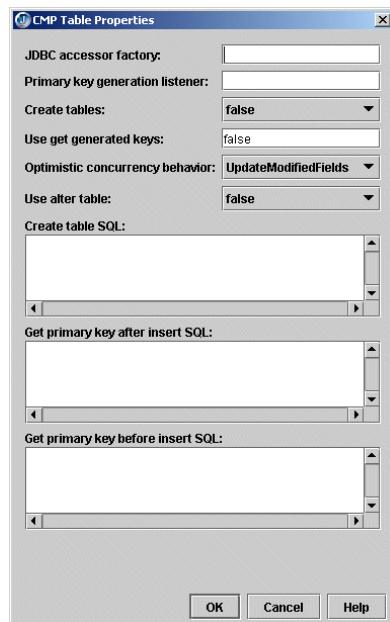
Choosing Edit DataSource Properties displays the DataSource Properties dialog box that you can use to edit the properties that give you access to the data source:



- For table nodes, use the context menu to add a column to the table, edit table properties, or to rename or delete the table node from the data source.

Choosing Add Column adds a column to the table. Right-click the new column node added and choose Rename to give the new column a meaningful name.

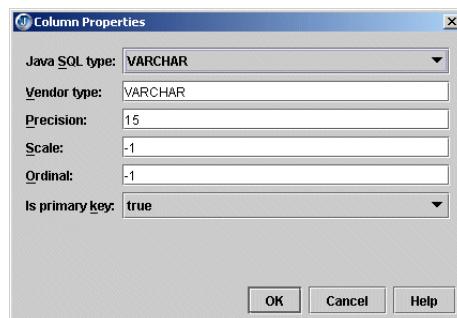
Choosing Edit Table CMP Properties displays a CMP Table Properties dialog box you can use to specify the container-managed persistence property settings you want your entity bean to have. These properties are specific to the Borland Enterprise Server only:



For help understanding the meaning of these properties, click the Help button in the dialog box.

- For column nodes, use the context menu to edit column properties, or to rename or delete the column node from the table.

Choose Edit Column Properties to change the data type of the column and specify if the column is a primary key.



Choosing Edit Column CMP Properties displays a Column CMP Properties dialog box you can use to specify the container-managed persistence property settings you want the column to have. There properties are specific to the Borland servers only:



For help understanding the meaning of these properties, click the Help button in the dialog box.

To delete a database schema, right-click the data source in the structure pane and choose Delete. A dialog box appears asking you to confirm the deletion. It also gives you the opportunity to specify whether you want the data source removed from the `jndi-definitions.xml` file.

Generating the entity bean classes and interfaces

Once your data source is as you want it, you're ready to generate the entity bean classes and interfaces. In the structure pane, right-click a table you want to use to create an entity bean and choose Create CMP 2.0 Entity Bean to create a 2.0 container-managed entity bean, or choose Create BMP Entity Bean to create a bean-managed entity bean. A bean representation appears in the EJB Designer and the entity bean classes and interfaces appear in the project pane. These are the files that are generated:

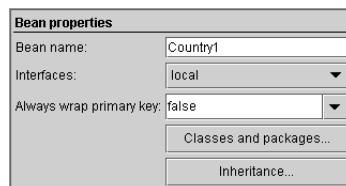
- `<Entity>` - The local interface of the entity bean.
- `<Entity>Bean` - The bean class of the entity bean.
- `<Entity>Home` - The local home interface of the entity bean.

Note that the EJB Designer's file-naming convention for an entity bean differs from that of a session bean. While a client application can access remote session beans, usually entity beans should be accessed through session beans only. The entity bean the session bean accesses is therefore likely to be local to it. So the interface that declares the business methods is usually the local interface for an entity bean and the remote interface for a session bean. The interface of an entity bean that is named with just the

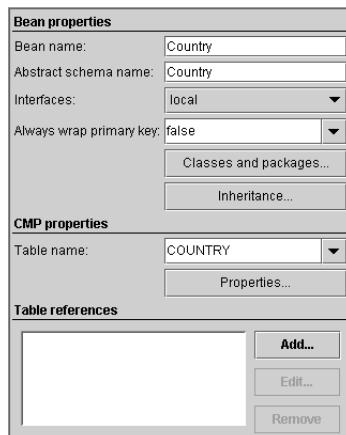
name of the entity is the local interface. For a session bean, it is the remote interface. Likewise, the interface that ends with the Home suffix for an entity bean is the local home interface, while for a session bean it is the remote home interface. For more information, see “[How the EJB Designer names EJB files](#)” on page 3-9.

Editing entity bean properties

As with session beans and message-driven beans, you can use the EJB Designer’s inspectors to edit the properties of your entity bean. Click the top row of the bean representation in the EJB Designer to make this inspector appear if it isn’t already visible. Here is an inspector for a BMP entity bean:



Here is an inspector for a CMP 2.0 entity bean:



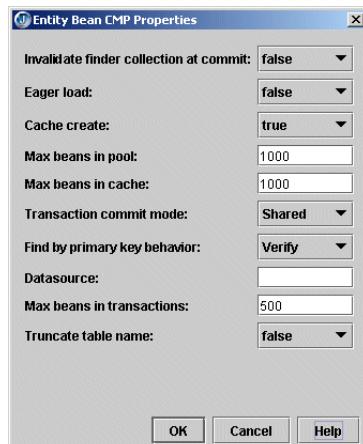
The inspectors for both BMP and CMP 2.0 beans allow you to change the name of the bean, and to specify whether the bean has remote interfaces (home and remote), local interfaces (local home and local), or both remote and local interfaces (home, remote, local home, and local). The inspector for CMP 2.0 beans also allow you to change the name of your abstract schema name, which is used in EJB-QL for finder and select methods.

By default, JBuilder creates a wrapped primary key class only when the primary key consists of more than one primary key field or if the single field is a Java primitive. If you want JBuilder to always create a wrapped primary key class in all cases, set the Always Wrap Primary Key option to true. For example, if you have an entity bean with a single primary key field of type `java.lang.Integer`, you can have this wrapped into a custom primary key class that contains a reference to this field.

Use the Classes and Packages button to make any desired changes to the names of the classes and packages, and use the Inheritance button if you want to specify the parent of the bean as other than `java.lang.Object`.

All the fields below the CMP Properties line in the can vary depending on your selected target server.

Click the Properties button to view the CMP Properties of a CMP 2.0 entity bean:



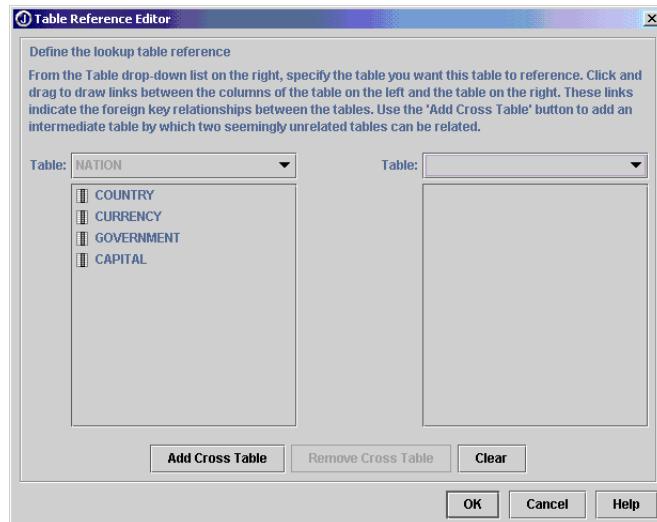
The CMP Properties in the inspector set the container-managed persistence properties of the bean. These properties are specific to Borland application servers. They determine how the container manages the bean's persistence. Please see your Borland Enterprise Server documentation for information on these properties.

Referencing another table

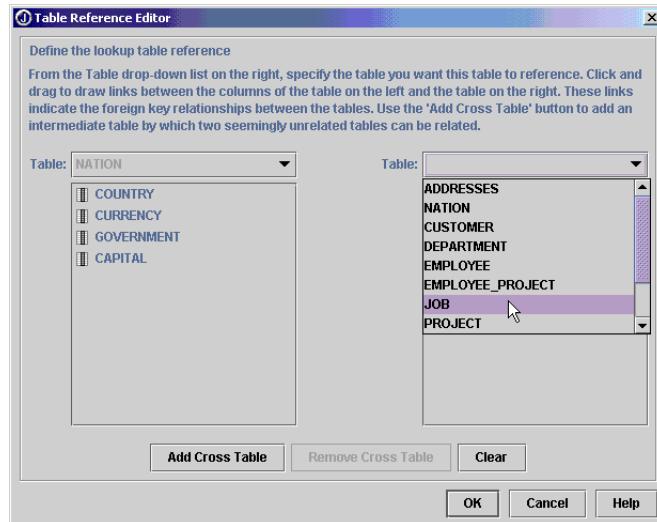
Table References let you reference another table and map columns in that table to fields in this entity bean. Table references are available for entity beans that target the Borland Enterprise Server only.

To create a table reference,

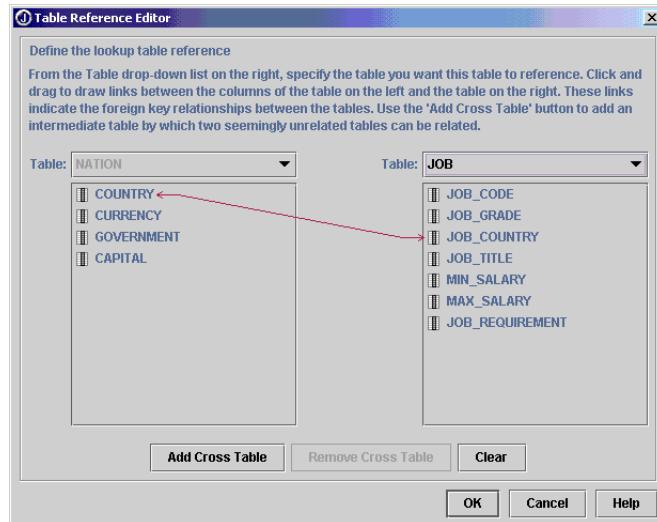
- 1 Click the Add button next to the Table Reference box in the entity bean inspector. The Table Reference editor appears:



- 2 Use the right drop-down list to specify the table you want to reference:



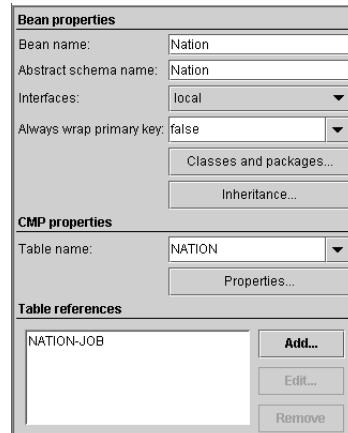
- 3** Click and hold down the mouse button on a column in the left table you want to link to a column in the right table. Drag a line to the column in the right table you want to link to:



If you need a third table to make the connection between two tables (as you might in a many to many relationship), click the Add Cross Table button and select the table that has fields that can be used to link to both tables. Then click and drag between the columns of the three tables to complete the table reference.

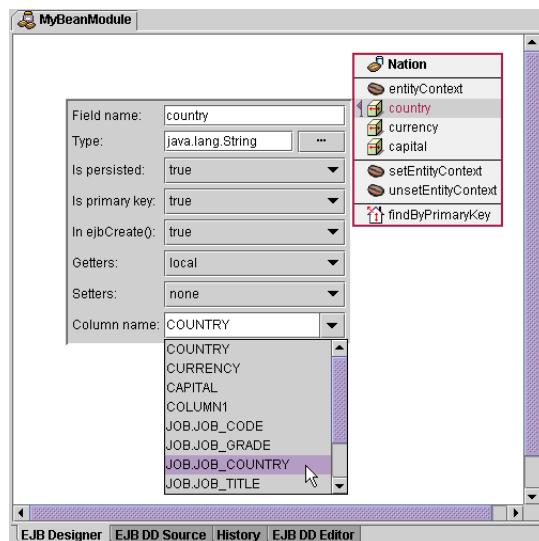
You can link multiple columns if it makes sense to do so. Choose OK when you are done.

The table reference you created appears in the Table References box:



- 4** In the entity bean representation, click one of the fields you want to map to a column in the referenced table to display its field inspector.

- 5 In the Column Name drop-down list, select the column you want to map the field to:



WebLogic table mappings

If your target application server is WebLogic Server 7.x or 8.1, the entity bean inspector looks like this:

Bean properties	
Bean name:	Employee
Abstract schema name:	Employee
Interfaces:	local
Always wrap primary key:	false
Classes and packages...	
Inheritance...	

CMP properties							
<input checked="" type="radio"/> Single table mapping:	EMPLOYEE						
Verify columns:	▼						
Optimistic column:	▼						
<input type="radio"/> Multiple table mapping:							
<table border="1"> <tr> <td>EMPLOYEE</td> <td>Add...</td> </tr> <tr> <td></td> <td>Edit...</td> </tr> <tr> <td></td> <td>Remove</td> </tr> </table>		EMPLOYEE	Add...		Edit...		Remove
EMPLOYEE	Add...						
	Edit...						
	Remove						

Field groups							
<table border="1"> <tr> <td></td> <td>Add...</td> </tr> <tr> <td></td> <td>Edit...</td> </tr> <tr> <td></td> <td>Remove</td> </tr> </table>			Add...		Edit...		Remove
	Add...						
	Edit...						
	Remove						

Note that the table mapping options are available only for WebLogic Server 7.x and 8.1. If you are using a WebLogic Server 6.x version, the table mapping options won't appear in the bean's inspector.

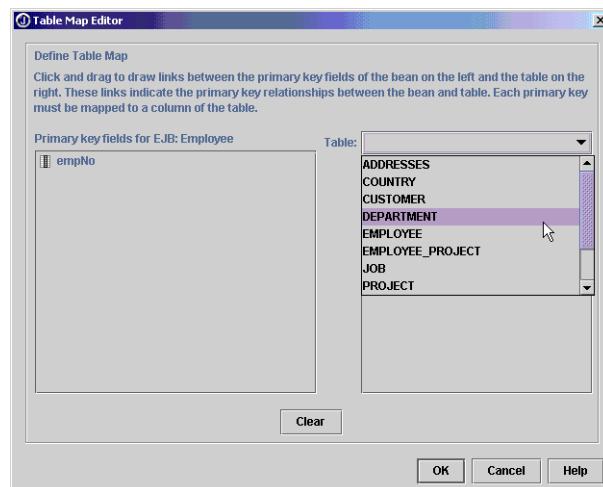
The Verify Columns field is used to specify that the columns should be checked for validity just before a transaction is committed to make sure no other transaction has changed the column's data. Select Read if you want all columns that have been read during the transaction to be checked. Select Modified if you want only the columns that have been updated during the transaction to be checked. Select either Version or Timestamp to specify that a column containing version or timestamp data exists in the table and that this column is to be used to implement optimistic concurrency.

The Optimistic Column field is enabled if you selected Version or Timestamp as a Verify Columns value. Select the column from the drop-down list that contains the version or timestamp value. For more complete information about optimistic concurrency, see your WebLogic documentation.

You can use the inspector to reference another table and map columns in that table to fields in this entity bean using the Multiple Table Mapping option. The Single Table Mapping option is the default.

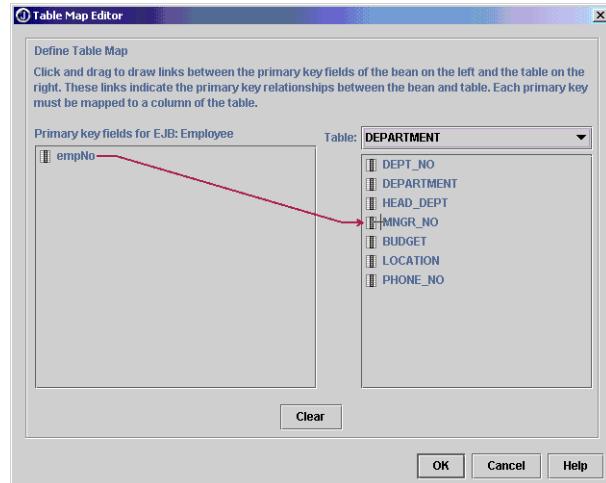
To reference another table and map columns to fields in your bean,

- 1 Select the Multiple Table Mapping option.
- 2 Click the Add button to display a Table Map Editor.
- 3 Select the table you want to map columns to from the Table drop-down list:



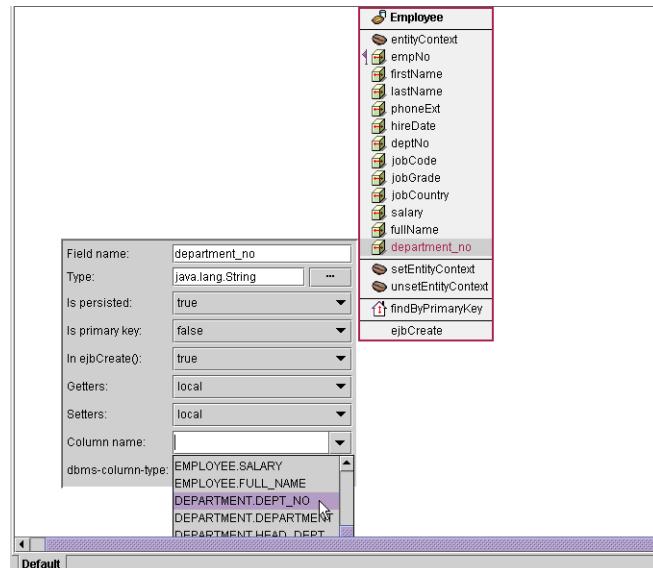
- 4 Click and drag between the primary keys in the table on the left to the column you want to map the column to in the table on the right.

Here you see the empNo field of the Employee bean being linked to the MNGR_NO column of the DEPARTMENT table.



- 5 Now you can return to your entity bean, add new fields, and map those fields to the columns in the table you established a reference to.

For example, here a new field named department_no has been added to the Employee bean. The inspector shows the new field being mapped to the DEPT_NO column in the DEPARTMENT table.

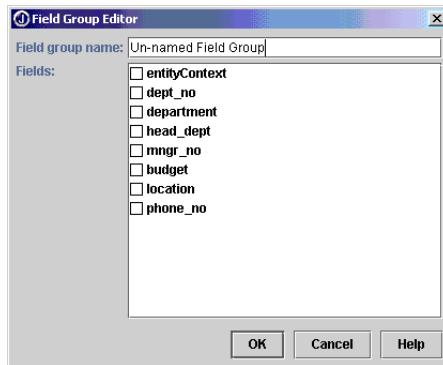


Note that a Field Groups box appears in a WebLogic entity bean inspector. A field-group element represents a subset of the container-managed persistence fields and the container-managed relationship fields of an entity bean. For more information about field groups and how they are used, consult your WebLogic documentation.

To add a field group,

- 1 Click the Add button.

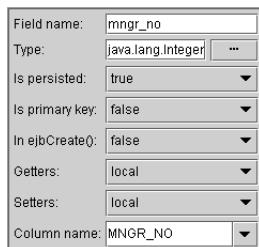
The Field Group editor appears:



- 2 Give the new group a name in the Field Group Name field.
- 3 Check the fields you want to be included in the group.
- 4 Click OK.

Entity bean field and method inspectors

The inspector for an entity bean field looks like this:



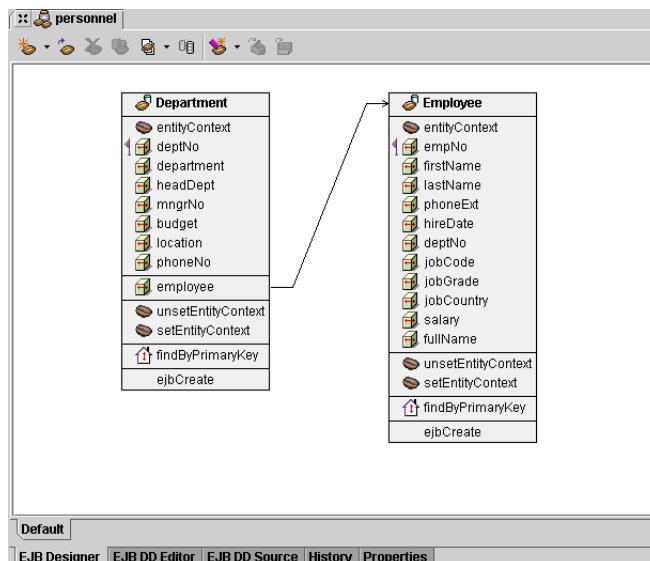
Use the inspector to change the field name, if you wish. You can specify the field's type, specify whether its value will be persisted, specify whether the field is a primary key for the table, specify if you want the field's value set in the `ejbCreate()` method in the bean class, specify where the field's access methods are defined, and map the field to the appropriate column in the table. If WebLogic Server 6.x, 7.x, or 8.x is your

target, you can also specify the database column type (dbms-column-type); your choices are OracleClob, OracleBlob, LongString, and SybaseBinary, or the default value of none.

The method inspector for entity beans is the same one used for session and message-driven beans. See “[Adding a new method](#)” on page 3-13 if you want more detail.

Creating entity bean relationships

You can create relationships between EJB 2.0 entity beans. For example, if you have a `Department` table and an `Employee` table, you might want to create a relationship that reflects which employees are in which departments. In this example, you would right-click the `Department` bean representation in the EJB Designer pane and choose Add | Relationship or click the Add icon on the toolbar and choose Relationship. When you do, a new field appears in the `Department` bean. Then click the `Employee` bean and a line connects the two beans. The new field added in the `Department` bean is now named `employee` to reflect the new relationship.



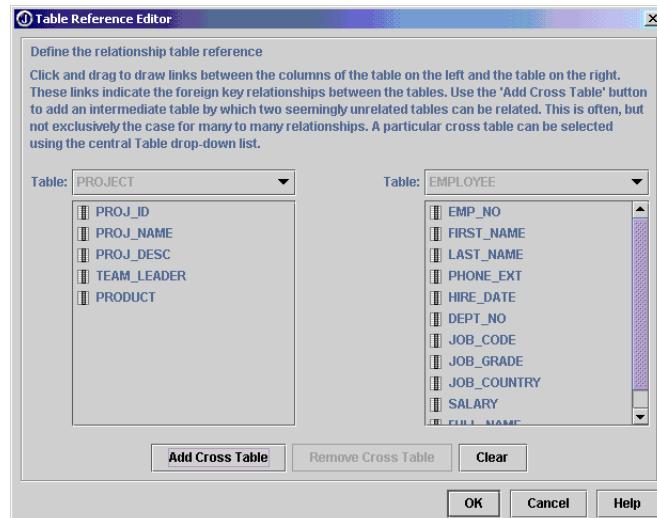
By default, the EJB Designer creates relationships by looking for columns (or fields) with the same name in the two tables you are connecting. Often the two tables won't have a common column name. You, therefore, must specify how the relationship is to be created.

Using the relationship inspector to specify a relationship

To specify how the relationship is to be created, click the new field that has been added to the bean you began specifying the relationship from. A relationship inspector appears:

Relationship properties	
Relationship name: nation-job	
Multiplicity:	one to many
Navigability:	unidirectional
Cascade delete:	false
CMR field properties	
Field name:	job
Return type:	java.util.Collection
Getters:	local
Setters:	local
Edit Table Reference...	

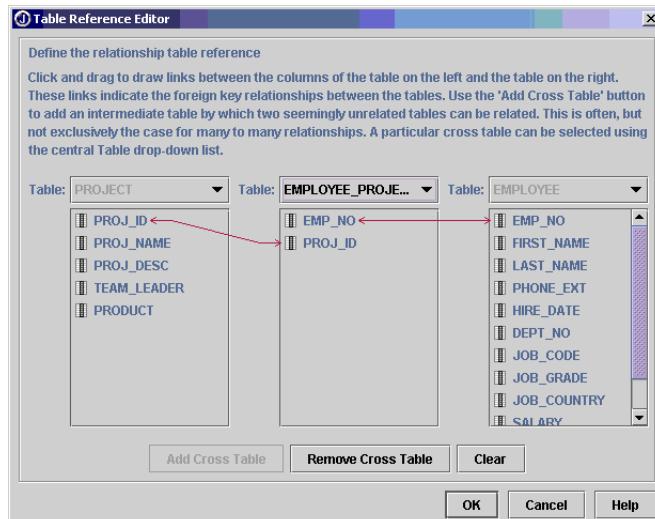
Click the Edit Table Reference button to display a table reference editor:



If the two tables have one or more columns with the same name, the table reference editor will show lines drawn between the common columns of the two tables. If no common column names exists, you must draw a line between the columns you want to use to create the relationship. For example, if the Department table had a column called DEPT_NO, and the Employee table had a column called DEPT, you would click and drag from the DEPT_NO column of the Department table to the DEPT column of the Employee table. When you release the mouse button, a line appears between the two columns, indicating the relationship.

For many-to-many relationships you need a third table to make the connection between two tables. For example, you might have an Employee table that includes a column named EMP_NO. You might also have a Project

table that includes a column named `PROJ_ID`. By themselves, the two tables have no columns you can use to create a relationship directly. If you had a Employee-Project table that consisted of two columns, `EMP_NO` and `PROJ_ID`, you could create a relationship between the Employee and Project table. In this case, in the table reference editor, you would click the Add Cross Table button. From the drop-down list that appears between the two tables, you would select the table that contains the columns to make the connection between the two tables. In this case, you would select the Employee-Project table. Then you would click and drag between the `EMP_NO` columns of the Employee table and the Employee-Project table, and do the same between the `PROJ_ID` columns of the Project table:



Click OK when you are done with the Edit Table Reference editor to return to the new field's inspector. You can use this inspector to specify if the relationship is one to one, one to many, or many to many.

You can also specify whether the relationship is unidirectional or bidirectional. If you select bidirectional, a new field also appears in the second bean. For example, if you have a Employee entity bean and a Project entity bean, you might want to create a relationship that allows an employee to have multiple projects as well as keeping track of all the employees on a single project. In this case, the direction of the relationship is bidirectional. Therefore, the Employee bean will have a new field named `project`, and the Project bean will have a new field named `employee`.

If you want the related rows in the other table in a relationship to be deleted when the first row of a table is deleted, set the Cascade Delete field to true. For example, a customer may shop in an online store and create an order, so there may be a Customer table and an Order table. If the Customer table has a relationship with the Order table through its `order` field and the Cascade Delete field is true, when the customer is deleted, all

orders made by that customer are deleted also. The Cascade Delete field can be set for one to one or one to many relationships only.

Use the CMR Field Properties in the inspector to specify in which interface(s) the getter and setter access methods are declared. The Return Type is the return type of the getter method and the type of the parameter passed to the field's setter method.

Specifying a WebLogic relationship

To specify how the relationship is to be created, click the new field that has been added to the bean you began specifying the relationship from. A relationship inspector appears:

Relationship properties	
Relationship name:	project-employee
Multiplicity:	one to many
Navigability:	unidirectional
Cascade delete:	false
db-cascade-delete	false
CMR field properties	
Field name:	employee
Return type:	java.util.Collection
Getters:	local
Setters:	local
Edit RDBMS Relation...	

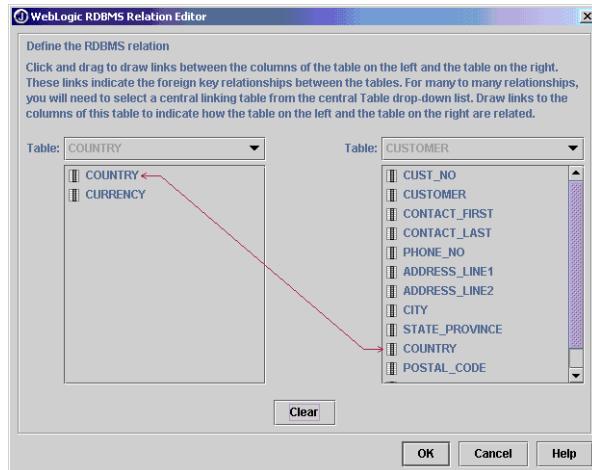
Specify whether the relationship will be one to one, one to many, or many to many using the Multiplicity field.

You can also specify whether the relationship is unidirectional or bidirectional. If you select bidirectional, a new field also appears in the second bean. For example, if you have a `Employee` entity bean and a `Project` entity bean, you might want to create a relationship that allows an employee to have multiple projects as well as keeping track of all the employees on a single project. In this case, the direction of the relationship is bidirectional. Therefore, the `Employee` bean will have a new field named `project`, and the `Project` bean will have a new field named `employee`.

If you want the related rows in the other table in a relationship to be deleted when the first table is deleted, set the Cascade Delete field to true. For example, a project has employees. If the `Project` table has a relationship with the `Employee` table through its `employee` field and the Cascade Delete field is true, when the project is deleted, all employees on that project are deleted also. The Cascade Delete field can be set for one to one or one to many relationships only.

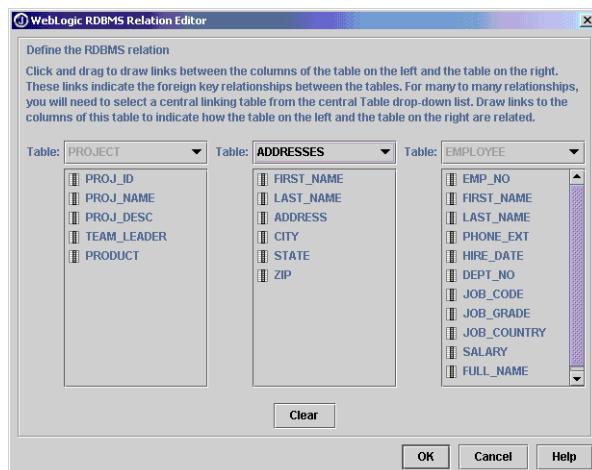
Use the CMR Field Properties in the inspector to specify in which interface(s) the getter and setter access methods are declared. The Return Type is the return type of the getter method and the type of the parameter passed to the field's setter method.

Click the Edit RDBMS Relation button to display the WebLogic RDBMS Relation Editor. If, in the relationship inspector you specified the Multiplicity as One to One or One to Many, the editor looks like this:

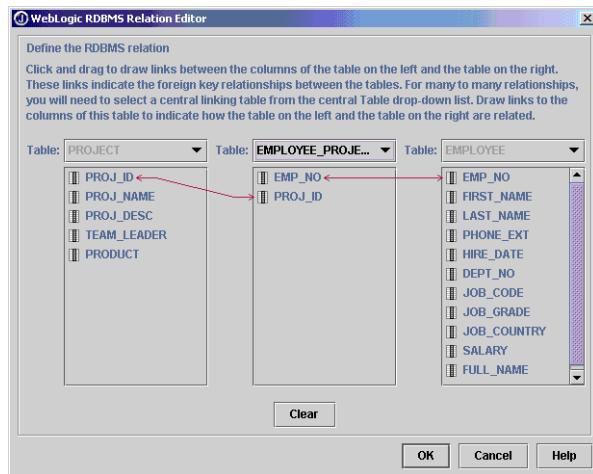


From the first drop-down Table field, select one of the tables you want to use to establish a relationship with another table. From the second drop-down Table field, select the other table in the relationship. When a table is selected in either of the Table fields, the tables columns are listed in the list box below the Table field. In the list of the columns in the first table, click the column you are using to create the relationship and drag to the column that contains the same data type in the second table that you are using to complete the relationship.

If you specified Many to Many as the value of the Multiplicity field, the editor looks like this:



In this case, you select the table in the middle that contains the columns that join the other two tables. Then click and drag between the columns that establish the relationship:



Improving performance in large projects with many relationships

If your project is quite large with many defined relationships, you might want to stop JBuilder from attempting to find the best route when it draws relationship lines as you work with the EJB Designer. Doing so can improve performance. Choose Tools | IDE Options, click the EJB Designer tab, and uncheck the Route Relationship Lines option, which is on by default.

Removing a relationship

To remove a relationship between two entity beans, right-click the new field that was added to the entity bean when the relationship was created and choose Delete Relationship on the context menu.

Adding a finder method

Finder methods allow the client to access the results of an EJB QL query. EJB QL is the query language for container-managed persistence query methods defined in the EJB 2.0 specification. An EJB QL query is a string that must contain a SELECT clause and a FROM clause, and may include a WHERE clause.

To add a finder method to an entity bean,

- 1 Right-click the name of the bean in the bean representation in the EJB Designer and choose Add | Finder on the context menu that appears or click the Add icon on the toolbar and choose Finder.
- A new finder method appears at the bottom of the bean along with the finder's inspector.
- 2 Use the inspector to give the finder a new name.
- 3 Specify whether the finder should return an instance of the bean, or `java.util.Collection`.
- 4 Specify the parameters passed to the finder, giving both the data types and the parameter names as the value of Input Parameters. For example, `java.lang.String lastName`.
- 5 Specify whether the finder should be declared in the home or local home interface or both in the Home Interfaces field.
- 6 If you choose to enter a query, enter an EJB QL statement. For example,

```
SELECT DISTINCT OBJECT(o)
FROM Order AS o, IN(o.lineItems) AS l
WHERE l.shipped = FALSE
```

Some application servers also permit you to use proprietary extensions. Check your application server documentation for more information about specifying queries.

The finder's inspector may have additional fields, depending on your target application server. If you need help filling in these additional fields, consult your application server documentation.

For more information about writing queries with the EJB 2.0 query language, see “Enterprise JavaBeans Query Language” in the J2EE tutorial on the Sun web site at http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html.

Adding an ejbSelect method

EJB QL is also used to create abstract `ejbSelect()` methods. Such methods allow you to use EJB QL to find objects or values related to the state of an entity bean without exposing the results to the client.

To add an `ejbSelect()` method,

- 1 Right-click the name of the bean in the bean representation in the EJB Designer and choose Add | Select on the context menu that appears.
A new select method appears at the bottom of the bean along with the method's inspector.
- 2 Use the inspector to give the method a new name. The EJB Designer will add a prefix of `ejbSelect` to the method name you specify. For example, if you specify the name `AllLineItems` in the inspector, the EJB Designer places an abstract `ejbSelectAllLineItems()` method in the bean class.
- 3 Specify whether the method should return an instance of the bean, a `java.util.Collection`, or `java.util.Set`.
- 4 Specify the parameters passed to the `ejbSelect()` method, giving both the data type and the parameter name as the value of Input Parameters. For example, `java.lang.String lastName`.
- 5 Select a Result Type Mapping from the drop-down list.
- 6 Enter an EJB QL statement. An EJB QL query is a string that must contain a SELECT clause and a FROM clause, and may include a WHERE clause.

Some select method's inspector may have additional fields, depending on your target application server. If you need help filling in these additional fields, consult your application server's documentation.

For more information about writing queries with the EJB 2.0 query language, see "Enterprise JavaBeans Query Language" in the J2EE tutorial on the Sun web site at http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html. Also consult your application server documentation.

Adding a home business method

EJB 2.0 components (both CMP and BMP) can have business methods, also called `ejbHome` methods, declared in the home interface. To create a home business method,

- 1 Right-click the entity bean representation in the EJB Designer and choose Add | Method or click the Add icon on the toolbar and choose Method on the menu that appears.
- 2 Specify the name of the method, its return type, and its parameters as you would when creating any other method using the method's inspector.

- 3 Specify the Interface value as Home, Local Home, or Home/Local Home.

When you specify a home interface, the EJB Designer declares the method in the home interface(s) you specified with the method name you specified. It also adds the method to the bean class with a prefix of ejbHome.

Like other business methods, you must then add the logic required to implement the ejbHome method in the bean class.

Creating a schema from entity beans

You can use an entity bean in the EJB Designer to create a schema to add to a DataSource in the structure pane.

To create a schema from entity beans,

- 1 Select the entity bean(s) you want to use to create a schema.
- 2 Right-click the EJB Designer pane and choose Create Schema From Selection or click the Create Schema From Selection icon on the toolbar.
- 3 Select the DataSource you want the schema added to and choose OK.

Exporting a data source

If you've modified a data source, you might want to use it to create database tables that reflect the changes you've made. Right-click the data source's node in the structure pane and choose Export Schema To SQL DDL (DDL stands for Data Definition Language.) In the dialog box that appears, specify a name for the file. You can then use that file to create a database that uses your modified data source.

Creating entity beans with bean-managed persistence

The EJB Designer can also help you create bean-managed persistence (BMP) entity beans:

- 1 Right-click the EJB Designer pane.
- 2 Choose Create EJB | BMP Entity Bean or click the Create EJB icon on the EJB Designer toolbar and choose BMP Entity Bean on the menu that appears.

Design your BMP bean as you would other beans by adding methods and fields and adding necessary logic. Unlike CMP 2.0 entity beans, you must then add code to manage the bean's persistence.

Creating session facades for entity beans

When you are designing your EJB applications, it's a good idea to avoid having an EJB client directly access your entity beans. There are at least two very good reasons for this:

- Each time the client requests data from an entity bean, an RMI-IIOP network call is made. If, for example, the client requests the customer's ID number and the customer's name, two network calls are required. With multiple entity beans and many data requests, your runtime performance can slow markedly.
- A client that accesses an entity bean directly must know the entity bean's implementation intimately. The client and the entity beans it calls are tightly coupled, which means changes you make in the entity bean will affect the client, too.

A popular solution is to use the Session Facade design pattern in which session facades intercede between the client and the entity beans. The client sends requests to the session facade, which forwards the request to the appropriate entity bean. A session facade doesn't do work itself, but delegates the work to other objects. A session facade provides a simple interface. It is the client interface to the underlying system, encapsulating specific knowledge and not exposing it unnecessarily. You can read about the Session Facade design pattern on the Web. Here are some links to get you started:

- Core J2EE Pattern Session Facade at <http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html>
- EJB Best Practices: Entity Bean Projection at <http://www-106.ibm.com/developerworks/java/library/j-ejb1008.html>

- Best Practices to Improve Performance Using Patterns in J2EE at <http://www.precisejava.com/javaperf/j2ee/Patterns.htm>
- Rules and Patterns for Session Facades at http://www7b.software.ibm.com/wsdd/library/techarticles/0106_brown/sessionfacades.html

JBuilder can help you create stateless session facades and data transfer objects (DTOs) for your entity beans using the DTO and Session Facade wizard.

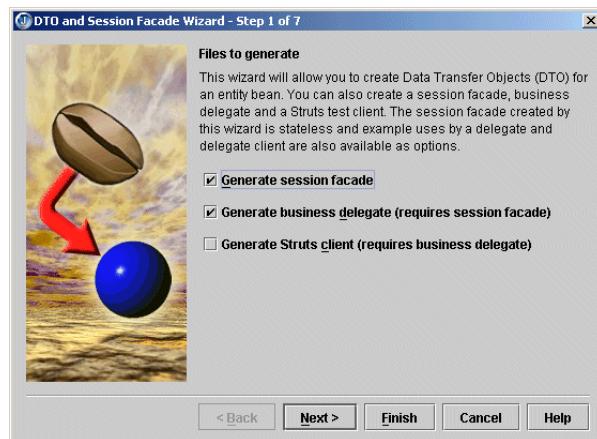
Using the DTO and Session Facade wizard

The wizard can generate these types of classes:

- DTOs - Data transfer objects. Their purpose is to pass information into and out of the EJB layer. One DTO is created for the bean the facade is created for, and additional DTOs are created for all beans with relationships to the initial bean. DTOs are sometimes referred to as Transfer Objects and other times as Value Objects.
- DtoAssemblers - Factory classes that assemble DTOs from beans. They require a bean and use its data to populate a DTO.
- ServiceLocator - A single class used to make looking up EJB home interfaces easier. You might need to edit the `getInitialContext()` method or make a subclass if you need to make a custom initial context for a particular application server.
- SessionFacade - A stateless session bean used to interact with an entity bean. It contains methods for creating, retrieving, updating, and deleting instances of the entity bean.
- Business Delegate - A delegate file created so you can call the methods of the session facade without having to know about EJBs or application servers. This is the primary class that clients will use to access and interact with entity bean data. When you write a test client, you can just use the business delegate to interact with the EJBs, which should make using EJBs easier.
- A Struts web client, which demonstrates the simple usage of the Business Delegate and its use in Struts. For information about Struts, see “Using the Struts framework in JBuilder” in the *Web Application Developer’s Guide*.

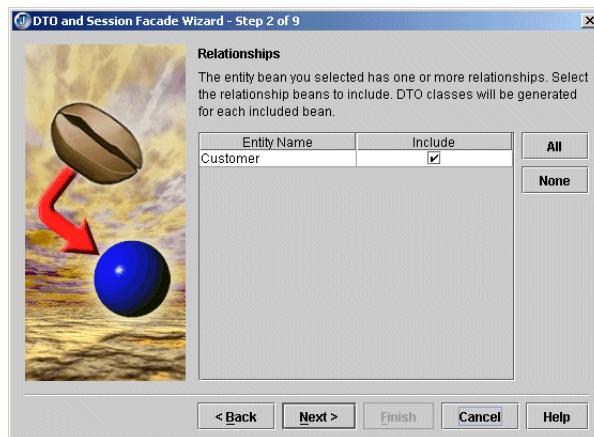
To create data transfer objects and a session facade for an entity bean,

- 1 Right-click an 2.0 CMP or BMP entity bean in the EJB Designer.
- 2 Choose DTO/Facade wizard on the context menu to display the wizard:



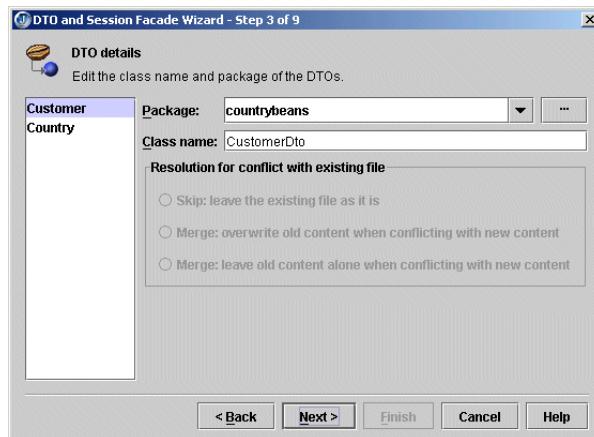
- 3 Work through the steps of the wizard:

- a The first page of the wizard allows you to specify whether you just want to create a stateless session facade for the entity bean, to generate a business delegate, and to also generate a Struts web client that uses the generated business delegate. Which pages appear in the wizard depend on which files you select to generate on this page. Click Next to go to the next step.
- b If the entity bean for which you are created a session facade has relationships defined to other entity beans, this page appears in the wizard:

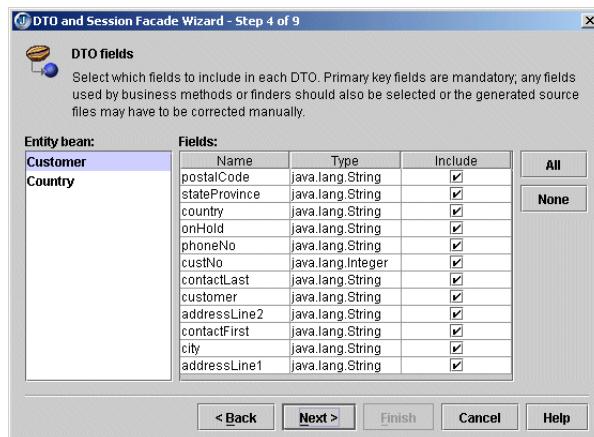


Using the DTO and Session Facade wizard

In the list of relationships on this page, check the Include check box to specify which relationships you want the session facade to know about. Click Next to go to the next step:

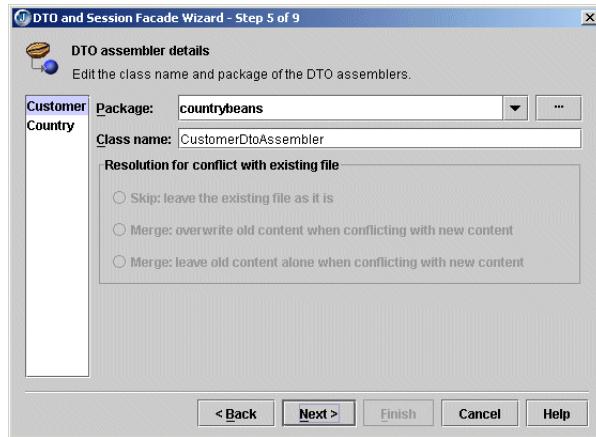


- c JBuilder suggests names for the data transfer objects (DTOs) and the package where they are stored. Edit these details if the default names aren't satisfactory. If your entity bean has relationships to other beans that you specified in the previous step, you'll see them listed in the box on the left. Select each bean and fill in the information for all beans. If a file with the name you specify for the data transfer object already exists, the page allows you to specify how you want the file conflict resolved by enabling the conflict resolution options. If necessary, select the resolution option you prefer. You'll see these options on other pages through the wizard. Whenever they are enabled, you must select how you want the conflict resolved. Click Next to go to the next step:

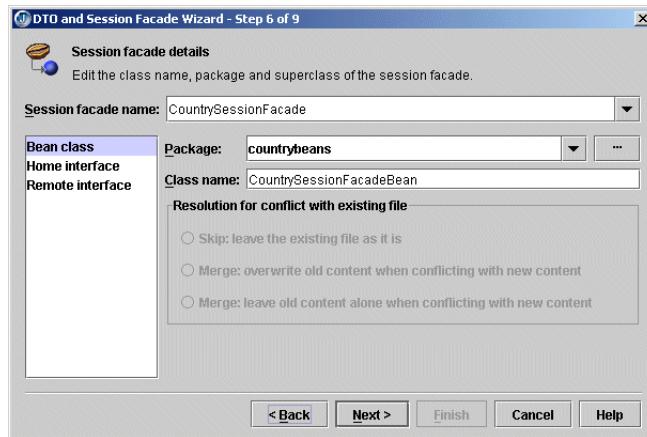


- d Use this page to specify which fields you want included in the generated data transfer objects. Check the Include check box next to

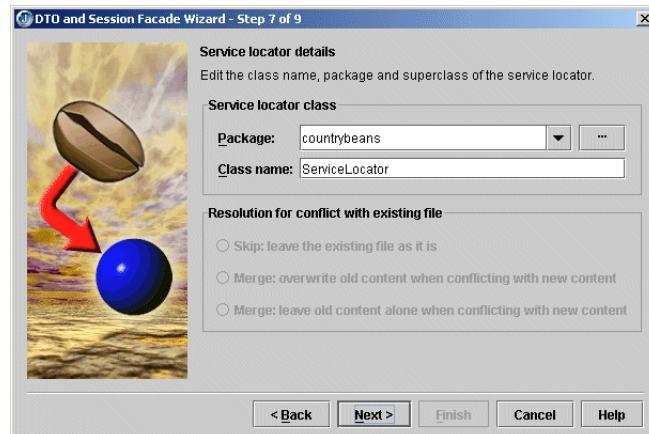
each field you want to include and uncheck the check box for those you want to omit. By default all the primary key fields are checked and you won't be able to uncheck them. Include any fields used by the business methods of your entity bean and its finders. The list box on the left will list multiple entity beans if the bean has relationships with others. Make sure you have selected each bean in the list so that you have the opportunity to include or omit each field. Click Next to go on to the next step:



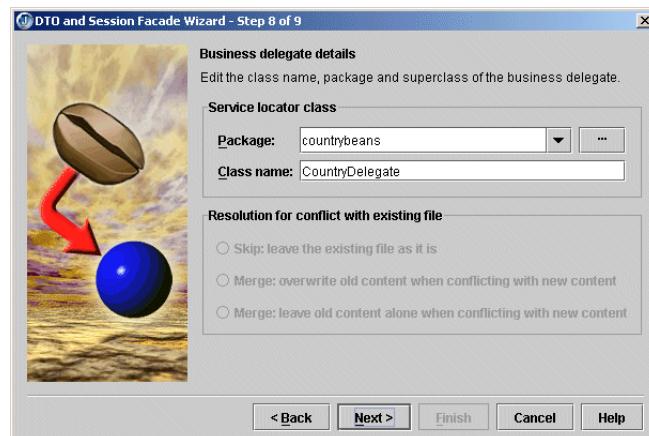
- e JBuilder suggests names for the data transfer object assemblers and the package where they are stored. Edit these details if the default names aren't satisfactory. If your entity bean has relationships to other beans that you specified in the previous step, you'll see them listed in the box on the left. Select each bean and fill in the information for all beans. If a file with the name you specify for the data transfer object already exists, the page allows you to specify how you want the file conflict resolved. If necessary, select the resolution option you prefer. Click Next to go to the next step:



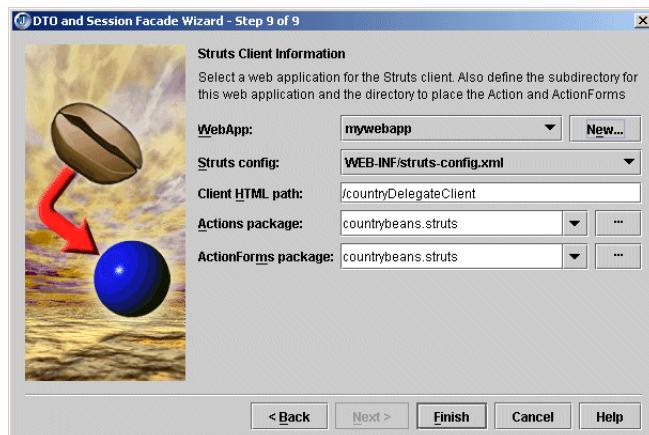
- f JBuilder suggests names for the session facade and the package where they will be stored. Edit these if you want to make changes. You can use the drop-down combo box for the Session Facade Name to select an existing session bean. This allows you to add the create, update, and delete methods of multiple entity beans to a single session bean. The box on the left lists the bean class, the remote interface and the home interface. Select each element in turn and specify the information for each. Make sure you have looked at each page to give you the chance to see the default name and to decide if you need to change it for each part of the session facade. Click Next to go to the next step:



- g JBuilder suggests a name for the service locator class. Accept the default name and package or edit them to meet your needs. Click Next to go to the next step. This page appears if you chose to generate a business delegate:



- h** JBuilder suggests a name for the business delegate. Accept the default name and package or edit them to meet your needs. Click Next to go to the next step. This page appears if you chose to generate a Struts client:



- i** Specify the web application you want to use using the WebApp drop-down list or click New to start the Web Application wizard to create one. If you are creating a new web application with the Web Application wizard, make sure that you check the Struts option in the list of frameworks. When the Web Application wizard completes, the DTO and Session Facade wizard returns. Specify the client HTML path and the Actions and ActionForms packages for your Struts client.

- j** Click Finish.

Examining the generated classes

The wizard generates the classes you specified. To see them, open the package node that contains the source code in the project pane. You'll find a data transfer object for each entity bean and for each bean with which that entity bean has a relationship. You'll also see an assembler class for each DTO, which creates the DTO. There will be a service locator class and a service locator exception class. Examine a session facade bean class and you'll see that it contains methods to create, remove, update, and retrieve instances of the entity bean. If you elected to create a business delegate, you'll find that file too, which contains the methods to create a session facade instance and calls the creating, removing, updating, and retrieving methods of the session facade. If you elected to create a Struts client, the project pane will have a web application node that contains the client files.

Creating EJB 1.x components with JBuilder

The JBuilder WebLogic Edition provides support for WebLogic Servers only

This chapter explains how to use JBuilder to create components that are compliant with the Sun Microsystems' Enterprise JavaBeans 1.1 specification. For information about how to create components that are compliant with the Enterprise JavaBeans 2.0 specification, see [Chapter 3, "Creating 2.0 session beans and message-driven beans with the EJB Designer"](#) and [Chapter 4, "Creating 2.0 entity beans with the EJB Designer."](#)

EJB modules

Each enterprise bean you create must belong to a JBuilder EJB module. An EJB module is a logical grouping of one or more beans that will be deployed in a single JAR file. In previous versions of JBuilder, an EJB module was known as an EJB group. They are the same thing. An EJB module contains the information that is used to produce the deployment descriptor(s) for that JAR file. You can edit the content of an EJB module using the Deployment Descriptor editor.

Once you have an EJB module and have edited it to your liking with the Deployment Descriptor editor, you can Make or Build an EJB module to produce the JAR. JBuilder uses the deployment descriptor to help identify the class files to be packaged.

An EJB module can be one of two formats: XML or binary. Because an EJB module in XML format is essentially a text file, it's easier to work with if you are using a version control system. An module in binary format is essentially the deployment descriptors in a .zip archive.

You can have more than one EJB module in a project. All the EJB modules in a single project use the same project classpath and JDK, and they are configured for the same target application server.

If you haven't done so already, follow the instructions in the chapter, "Configuring the target application server" in *Developing J2EE Applications*.

Creating an EJB 1.x module

There are two ways to create an EJB module:

- Use the EJB Module wizard to create an EJB module when you haven't created your enterprise beans yet.
- Use the EJB Module From Descriptors wizard to create an EJB from the deployment descriptors of existing enterprise beans you have.

If you don't have an open project before you begin an EJB module wizard, JBuilder displays the Project wizard first. After you create a new project, the EJB wizard you selected then appears.

Creating an EJB 1.x module with the EJB Module wizard

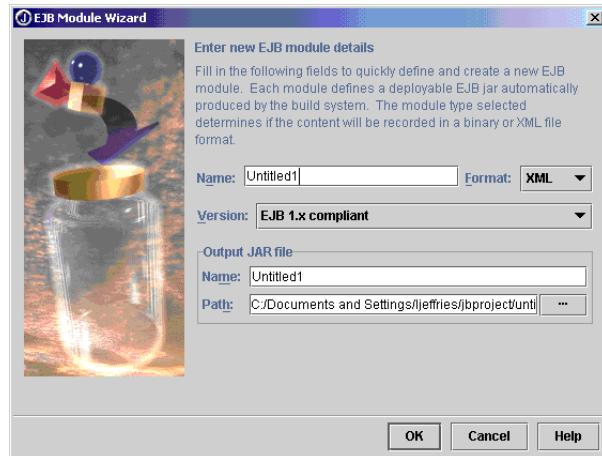
If you haven't created your enterprise beans yet, begin by creating an EJB module. To create an EJB module,

- 1 Choose File | New and click the Enterprise tab.

Note

If the EJB wizards on the Enterprise page are disabled, you don't have the Enterprise version of JBuilder installed or you haven't selected a configured application server for your project or the selected application server is not configured. See "Configuring the target application server" in *Developing J2EE Applications* for information about configuring and selecting an application server for your project.

- 2 Double-click the EJB Module wizard icon and the wizard appears:



- 3 Specify the name of the EJB module.

4 Specify the format of the new module.

Your choices are binary, which internally stores the deployment descriptors in .zip format and was used before JBuilder 5, or XML, which stores the deployment descriptors in XML format. XML format allows users to merge changes if they are checking them into a source control system. Using XML is recommended unless you are sharing the file with an older version of JBuilder.

5 Specify the version as EJB 1.x compliant. The options available to you depend on which application server is your target.

6 Specify the name of the JAR file your enterprise bean(s) will be in.

JBuilder entered a default name that is the same as the name of your EJB module. You can simply accept that name or specify another. JBuilder also entered a path based on your project path. You can change it to your liking or accept the default path.

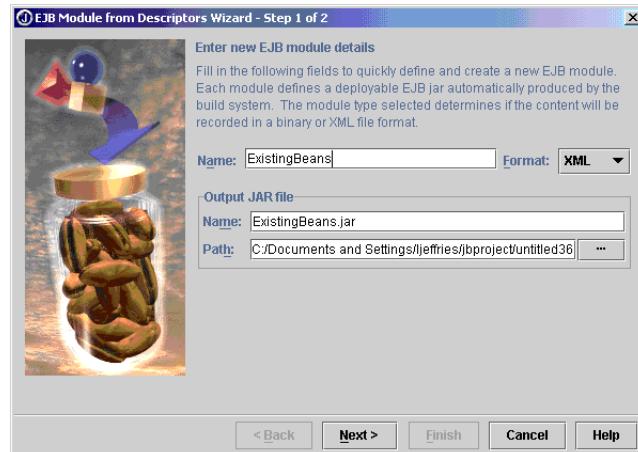
7 Click OK to create the EJB module.

Creating an EJB module from existing deployment descriptors

The EJB Module From Descriptors wizard imports the ejb-jar.xml deployment descriptor and BEA WebLogic and Borland Enterprise Server proprietary deployment descriptors only into a new EJB module.

To create an EJB module from existing descriptors,

- 1** Choose File | New and click the Enterprise tab.
- 2** Double-click the EJB Module From Descriptors wizard icon and the wizard appears:



If you don't have any open project before you begin the EJB Module From Descriptors wizard, the Project wizard appears first. After you create a new project, the EJB Module From Descriptors wizard then appears.

3 Specify the name of your new EJB module.

4 Specify the format of the new module.

Your choices are binary, which internally stores the deployment descriptors in .zip format and was used before JBuilder 5, or XML, which stores the deployment descriptors in XML format. XML format allows users to merge changes if they are checking them into a source control system. Using the XML format is recommended.

5 Specify the name and path of the JAR file your enterprise bean will be in.

JBuilder entered a default name that is the same as the name of your EJB module. You can simply accept that name or specify another.

6 Click Next and specify the directory that contains the existing deployment descriptor(s) you want to make up the module.

(Frequently this is in the META-INF directory of a JAR.) When you do, the wizard lists the deployment descriptors in the specified directory in the Identified EJB Descriptors field. It also lists the root source directory for these descriptors in the Root Source Directory field.



7 If you want to add the root source directories to your project, check the Add Root Source Directories To Project check box.

- 8 Click Finish to create the EJB module that contains the deployment descriptors for the existing bean(s).

Creating an enterprise bean

The JBuilder object gallery contains two wizards you can use to create 1.x enterprise beans: the Enterprise JavaBean 1.x wizard and the EJB 1.x Entity Bean Modeler. The Wizards menu contains another: the EJB 1.x Bean Generator. This section discusses creating an enterprise bean with the Enterprise JavaBean 1.x wizard. To read about using the EJB 1.x Entity Bean Modeler to create entity beans, see [Chapter 7, “Creating EJB 1.x entity beans from an existing database table.”](#)

The Enterprise JavaBean 1.x wizard and the EJB 1.x Entity Bean Modeler create the home and remote interfaces at the same time the bean class is created. If you prefer to begin your enterprise bean development by creating your remote interface first, see [“Generating the bean class from a remote interface” on page 6-12](#) for information about using the EJB 1.x Bean Generator to generate your bean class from a remote interface you have created.

To begin creating an enterprise bean with the Enterprise JavaBean 1.x wizard,

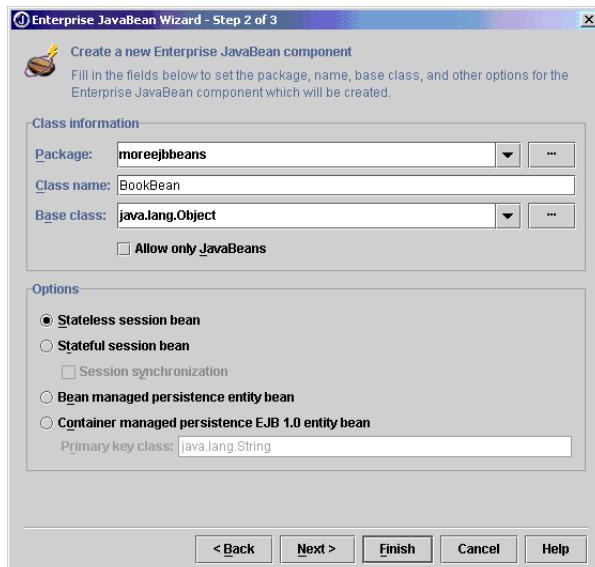
- 1 Choose File | New and click the Enterprise tab.
- 2 Double-click the Enterprise JavaBean 1.x wizard icon.

The wizard appears.



- 3** In the drop-down list, select the EJB module you want your enterprise bean to belong to. Choose Next to display page 2 of the wizard.

If you don't have an EJB module defined before you start the Enterprise JavaBeans wizard or you want to create another, click the New button to start the EJB Module wizard. You must have at least one EJB module defined in your project before you can create an enterprise bean. Once you've created an EJB module with the EJB Module wizard, select the new module and choose Next to continue with the Enterprise JavaBean 1.x wizard.



- 4** Specify the class name of your bean class, the package it will be in, and the bean's base class.

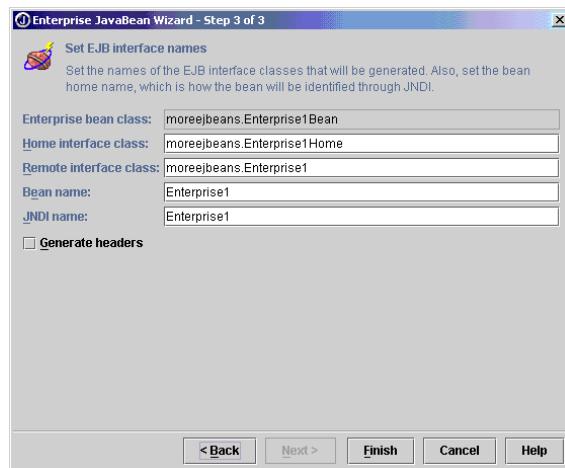
Next you must decide whether you are creating a session bean or an entity bean.

Creating a session bean

If you are creating a session bean,

- 1** Click either the Stateless Session Bean or Stateful Session Bean.
- 2** If you select a Stateful Session Bean, you can also choose to implement the `SessionSynchronization` interface by checking the Session Synchronization check box.

- 3** Click Next to go Step 3.



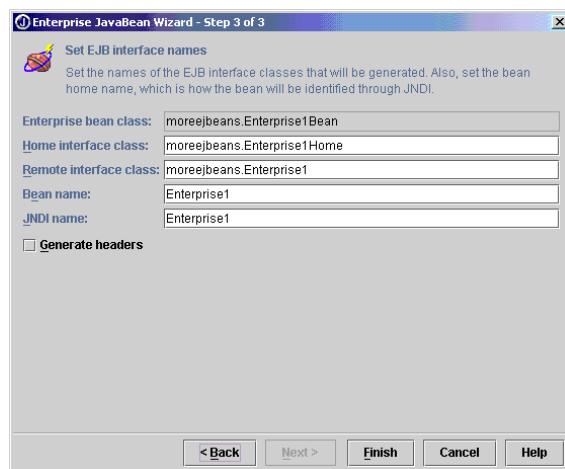
- 4** Specify names for the Home Interface Class, the Remote Interface Class, and the Bean Home Name; JBuilder suggests default names based on the name of your bean class.

- 5** Click Finish.

Creating an entity bean

If you are creating an entity bean,

- 1** Select either the Bean Managed Persistence Entity Bean option or the Container Managed Persistence 1.x Entity Bean option.
- 2** Specify a Primary Key Class.
- 3** Click Next to go Step 3.



- 4 Specify names for the Home Interface Class, the Remote Interface Class, and the Bean Home Name; JBuilder suggests default names based on the name of your bean class.
- 5 Click Finish.

After you click the Finish button, JBuilder creates the bean class and its home and remote interfaces. You'll see them appear in the project pane. Examine the source code of the bean class and you'll see that the class implements the `SessionBean` interface if it's a session bean, and it implements the `EntityBean` interface if it's an entity bean. JBuilder has added methods with empty bodies for the methods all enterprise beans must implement. You can add code to these method bodies to supply the logic your bean requires when these methods are called.

The home interface extends the `EJBHome` interface and contains a `create()` method needed to create the bean. The remote interface extends `EJBObject` but is empty otherwise because you have yet to declare any business logic methods for your bean.

Although you can begin your entity beans using the Enterprise JavaBeans wizard, the preferred way to create entity beans is to use the EJB 1.x Entity Bean Modeler. Entity beans you create with the Enterprise JavaBean 1.x wizard aren't likely to pass verification with the Deployment Descriptor editor until you complete the bean more fully.

Adding the business logic to your bean

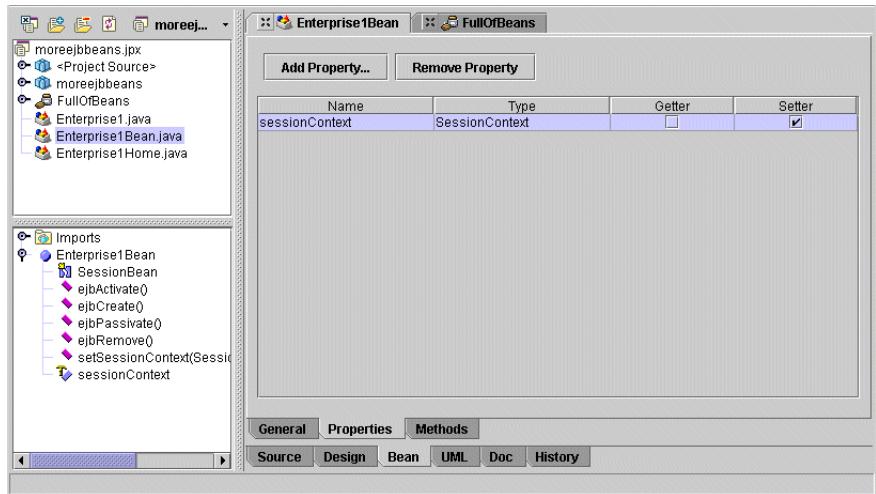
In the source code of your bean class, define and write the methods that implement the logic your enterprise bean needs.

If you need to add properties to the bean, you can either add them directly in the source code, or you can use the Properties page of the Bean designer.

To use the Bean designer to work with properties,

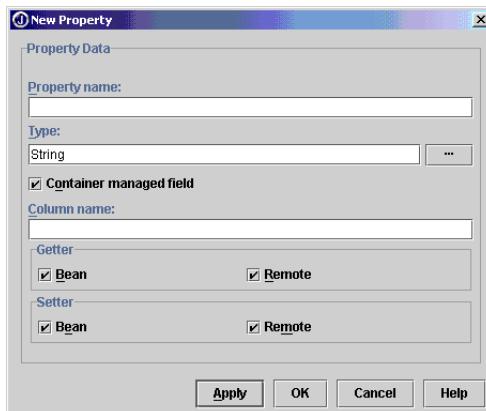
- 1 Double-click the bean class in the project pane.
- 2 Click the Bean tab to display the Bean designer.

- 3** Click the Properties tab to display the Properties page.



To add a new property,

- 1** Click the Add Property button to display the New Property dialog box.



- 2** Specify the Property Name and its Type.

- 3** If your bean is an entity bean with container-managed persistence, the Container Managed Field and the Column Name options are available in the New Property dialog. If the property you are creating is a column in a database table, check the Container Managed Field check box and specify the column name in the table as the value of the Column Name field.

4 Specify your access methods by setting the Getter and Setter options.

If you decide your property needs a getter access method, you can also decide if it appears in the bean class and/or in the remote interface. If you decide your property needs a setter access method, you can also decide if it appears in the bean class and/or in the remote interface.

5 Choose Apply to immediately add the new property definition to the source code of your bean. The access methods you specified are added to bean class and/or the remote interface, depending on the options you selected.

6 You can continue adding new properties in the dialog box. When you are finished, choose OK.

If you use the Enterprise JavaBean 1.x wizard to begin an entity bean with container-managed persistence, you will be adding properties to your bean. Keep in mind that at least one property must be the primary key and that you must specify which field or fields makes up the primary key on the General panel of the Deployment Descriptor editor. If you fail to do so, the Deployment Descriptor editor won't be able to verify the deployment descriptor as valid.

You can also use the Properties page to modify a property. For example, if you didn't specify a setter for your property when you were declaring it and you decide your bean needs one, you can simply check the Setter box for that property on the Properties page and JBuilder adds the setter method to your source code. Or you can remove a getter or setter by unchecking the appropriate check box.

To remove a property from your bean using the Properties page,

1 Select the property listed in the table of properties.

2 Click the Remove button.

JBuilder asks if you want to remove the property and its associated code.

3 Choose Yes.

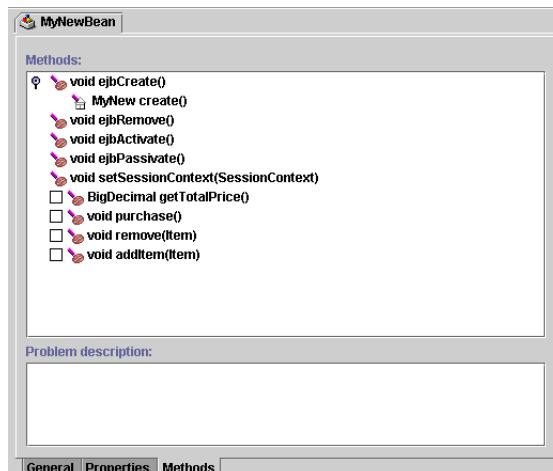
You can also use the Properties page to change the name of the property and its type. The Bean designer is a Two-Way Tool™, so changes you make on the Properties page are reflected in your code and changes you make in your code are reflected on the Properties page.

Exposing business methods through the remote interface

Once you've declared your business logic methods in the source code of your bean, you must specify which methods you want to add to the remote interface. The client can call only those methods exposed through the remote interface of the bean.

To add methods to the remote interface,

- 1** Double-click the enterprise bean in the project pane.
- 2** Click the Bean tab to display the Bean designer.
- 3** Click the Methods tab.
- 4** In the Methods box, check the check box next to the methods you want to expose in the remote interface.



As you check methods in the Methods box, the methods are added to the remote interface.

To remove a method from the remote interface, uncheck the check box next to the method in the Methods box.

To edit one of the methods, right-click it to display a context menu and choose Edit Selected. The file opens in the code editor and your cursor is positioned on that method, ready for you to edit it.

The context menu has other commands you'll find useful. You can choose Remove Selected to remove a method from the bean class. Choosing Check All checks all the methods so that they are all added to the remote

interface; choosing Uncheck All unchecks all the methods so that no methods are added to the remote interface.

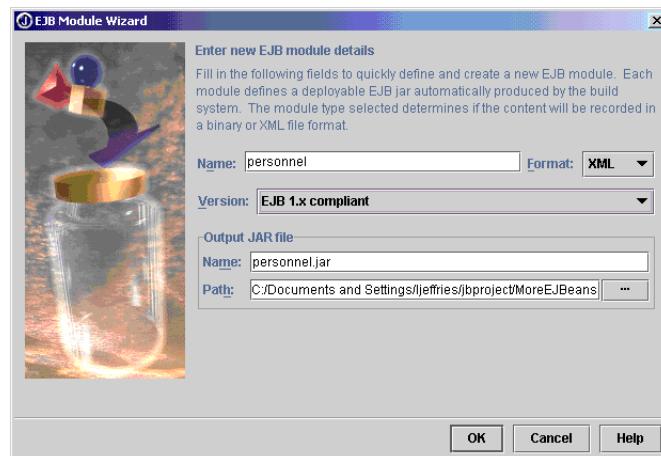
You can use the Methods page to verify that the methods declared in your bean class have the same method signature as they do in the home and remote interface. For example, suppose you add a parameter to the `ejbCreate()` method in your bean class, but neglect to add it to the `create()` method in the home interface. The Methods box will show both the `ejbCreate()` method and `create()` method in red text. If you then click a method displayed in red text, the Problem Description box explains what the problem is. You could then add the additional parameter to the `create()` method to make the method signatures match and fix the problem. Or, if you remove methods from your bean class but forget to do so in the remote interface, the Methods box will display those methods in red text to remind you to remove them from the remote interface.

Generating the bean class from a remote interface

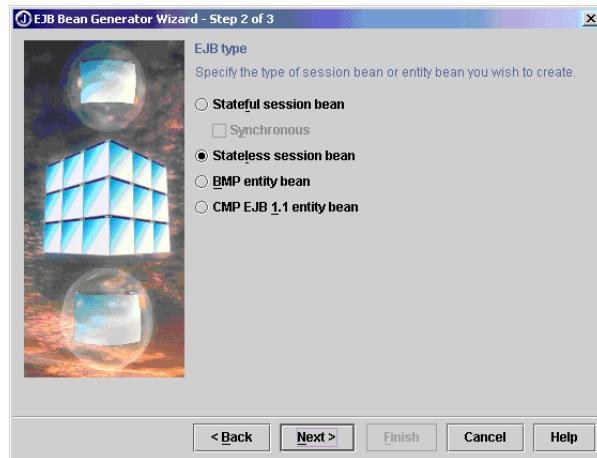
Some developers prefer to start their development of an enterprise bean by designing the remote interface first. If you favor this approach, you can use the EJB 1.x Bean Generator to generate a skeleton bean class from your existing remote interface.

To generate a bean class from a remote interface,

- 1 Display the remote interface in the editor.
- 2 Choose Wizards | EJB | EJB 1.x Bean Generator to display the EJB 1.x Bean Generator wizard:

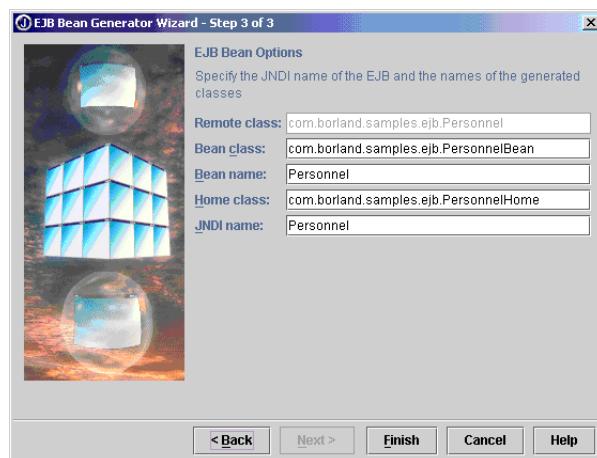


- 3 Select the EJB module the bean belongs to and click Next.



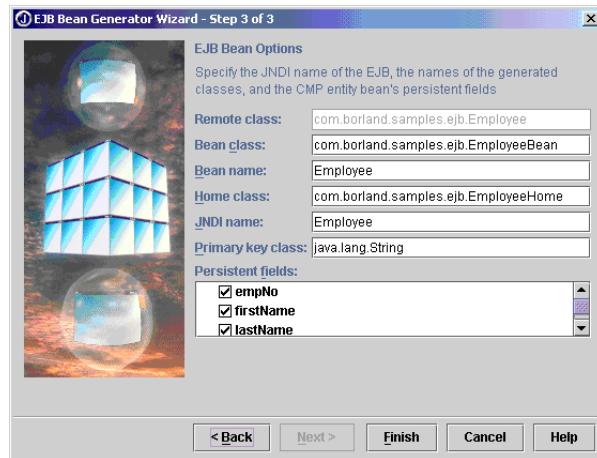
- 4 Select the type of EJB you want generated and click Next.

If you selected one of the session bean options, this page appears:



Specify the EJB Bean Options: the Bean Class, the Bean Name, the Home Interface, and the JNDI Name.

If you selected the CMP entity bean option, this screen appears:



5 Specify the EJB Bean options: the Bean Class, the Bean Name, the Home Interface, the JNDI Name, the Primary Key Class, and which fields you want to be persistent.

6 Choose Finish.

The EJB 1.x Bean Generator creates the skeleton bean class you specified that includes the methods found in the remote interface. In the generated bean class, these methods include a comment reminding you to fill in their implementations. You must add your code to the methods to implement them as you wish.

The EJB 1.x Bean Generator also creates a home interface if one did not previously exist. If a home interface did exist, the EJB 1.x Bean Generator asks you if you want to overwrite the home interface and responds according to your answer.

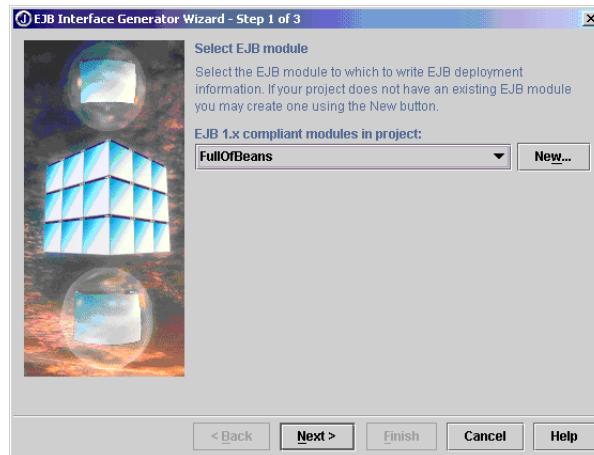
Creating the home and remote interfaces for an existing bean

If you already have a bean class, but don't have the required home and remote interfaces, you can use the EJB 1.x Interface Generator wizard to create them. You can also use the wizard if you've made significant changes to the source code of your bean and you want the changes

reflected in the interfaces. By using the EJB 1.x Interface Generator, you regenerate new interfaces based on the revised bean class source code.

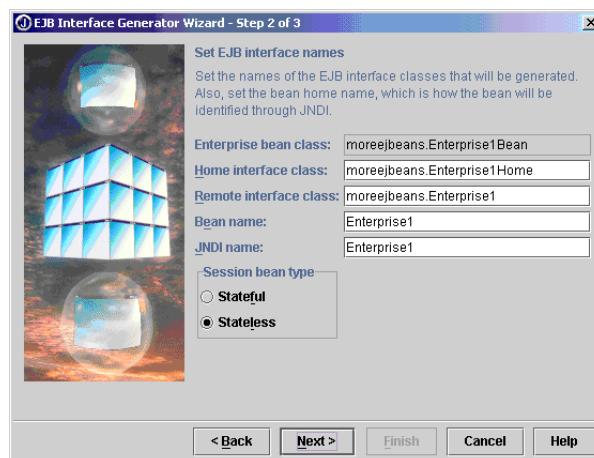
To use the EJB 1.x Interface Generator wizard,

- 1 Open the source code of your bean class in the code editor.
- 2 Choose Wizards | EJB | EJB 1.x Interface Generator.

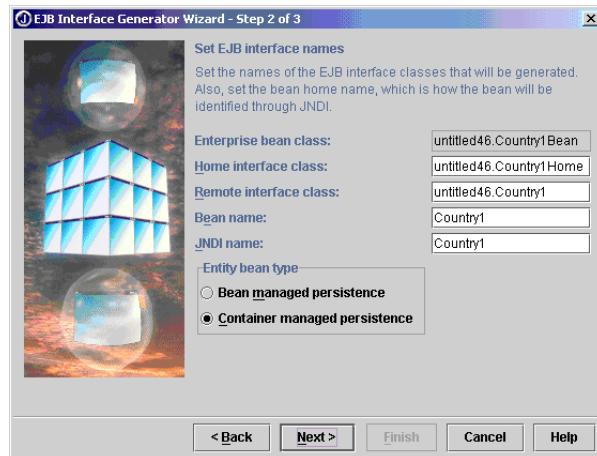


- 3 Select the EJB module the bean belongs to and click Next.

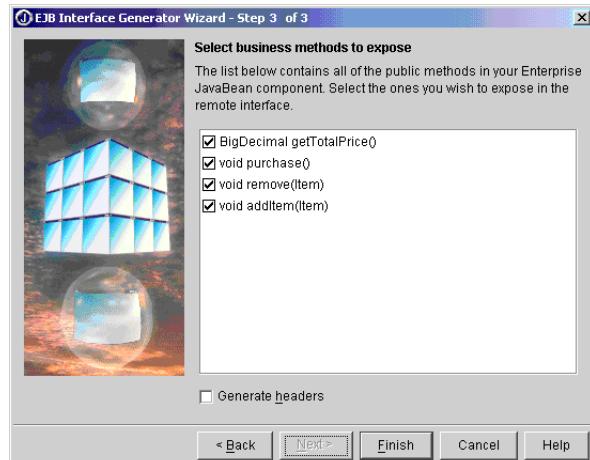
This page appears if the bean is a session bean:



If the bean is an entity bean, this page appears:



- 4 Accept the default names or enter new ones.
- 5 If the enterprise bean is a session bean, select either the Stateless or Stateful option. If the enterprise bean is an entity bean, select either Bean Managed Persistence or Container Managed Persistence.
- 6 Click Next to display Step 3, which displays the bean methods:



- 7 Leave those methods that you want exposed in the remote interface checked and uncheck those you don't want to appear in the remote interface.
- 8 Choose Finish.

Your next step is to compile your beans, debug them, and create a JAR file. See [Chapter 8, “Compiling enterprise beans and creating a deployment module.”](#)

Creating EJB 1.x entity beans from an existing database table

The JBuilder WebLogic Edition provides support for WebLogic Servers only

Often the data you want to model with an entity bean already exists in a database. You can use JBuilder's Entity Modeler to create EJB 1.x entity beans. To create EJB 2.0 entity beans that model data in a database, see "[Creating CMP 2.0 entity beans from an imported data source](#)" on page 4-2.

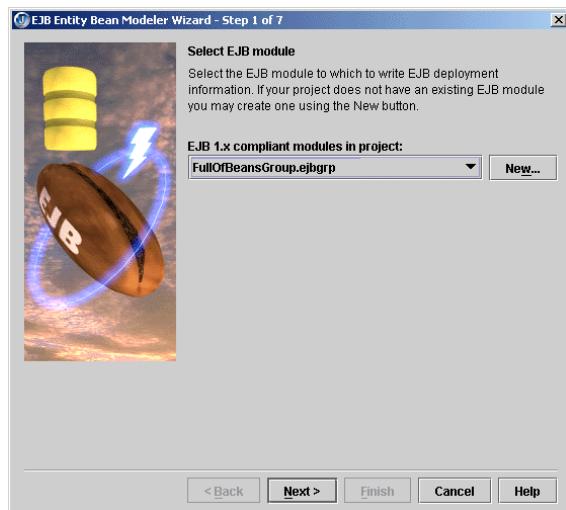
Creating entity beans with the EJB 1.x Entity Bean Modeler

The EJB 1.x Entity Modeler wizard creates entity beans based on existing tables in any database accessible through JDBC. You can use the wizard to create several entity beans at once and you can specify any relationships between those beans.

Once you've used the EJB 1.x Entity Bean Modeler to generate the code that makes up the entity beans, their primary keys, their home and remote interfaces, and the appropriate entries in the deployment descriptor, you can then modify the results using other JBuilder tools, such as the Bean designer, the Deployment Descriptor editor, and the JBuilder code editor.

To display the EJB 1.x Entity Modeler, choose File | New, click the Enterprise tab, and choose EJB 1.x Entity Bean Modeler. If you have at

least one EJB module defined in your project, the EJB 1.x Entity Bean Modeler appears.

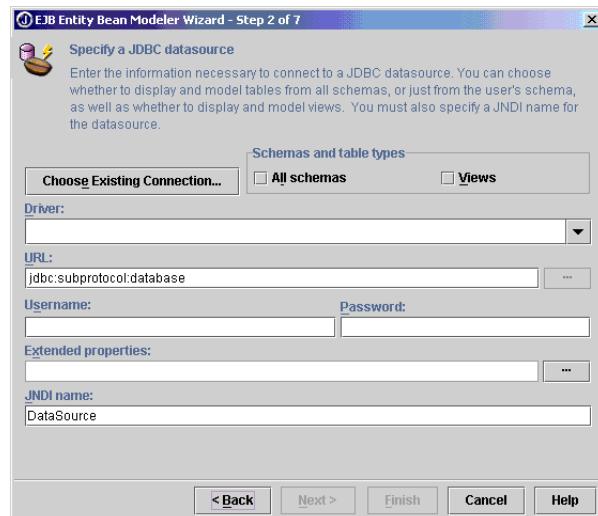


All enterprise beans developed with JBuilder must belong to an EJB module. If you don't have at least one EJB module in your current project, click the New button to start the EJB Module wizard. Once you've created an EJB module with the EJB Module wizard, you can continue with the Entity Bean Modeler.

To create one or more beans from existing database tables, follow these steps:

- 1 Select an EJB module to put your bean in and choose Next to go to Step 2.

The EJB module you select is used to determine where the deployment information is written.



2 Specify a JDBC data source.

Enter the information that's needed to connect to a JDBC data source.

To use an existing connection, click the Choose Existing Connection button and select a connection. Other required information for this page is then filled in automatically except the password, which you must enter yourself if your connection requires one.

If you don't have an existing connection or want to create another, select a driver from the Driver drop-down list and specify an URL. The drivers that appear are those you set up using Tools | Enterprise Setup on the Database Drivers page. See "Setting up JDBC drivers" in the "Setting up the target application server" chapter of *Developing J2EE Applications* for more information.

Specify the Username for the data source, and if a password is required, type in the password. Select any extended properties you need. Finally, specify a JNDI name for the data source.

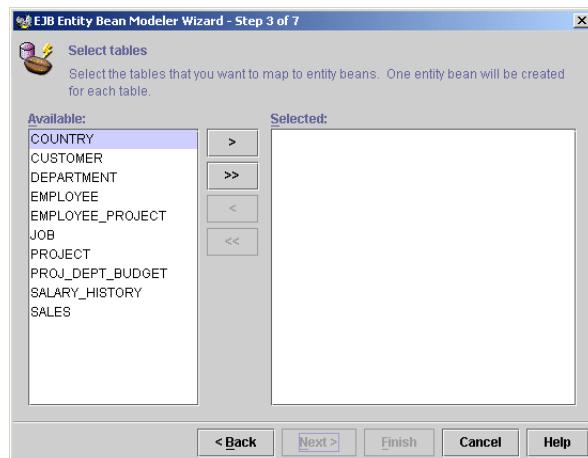
3 Specify which Schemas And Table Types options you want.

If you check the All Schemas option, the EJB 1.x Entity Bean Modeler will load all schemas the user has rights to for the connection. If you leave All Schemas unchecked, just the schemas with the same name as the username are loaded, potentially reducing the time required to make the connection and load the data.

Check the Views option if you want to have views loaded into the EJB 1.x Entity Bean Modeler. If you don't want to load views, leave the Views option unchecked.

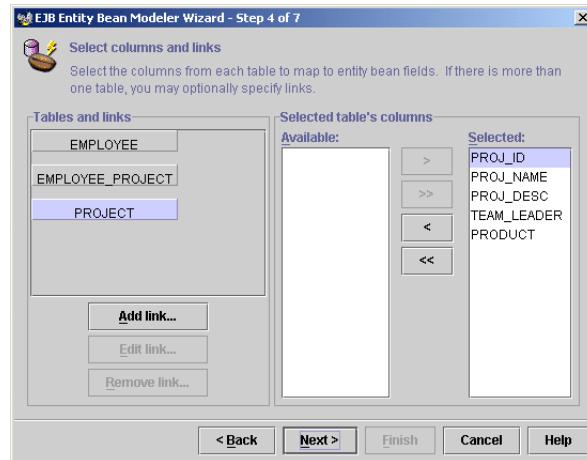
4 Click Next.

The EJB 1.x Entity Bean Modeler attempts to connect to the specified data source. Only if the connection is successful does the next page appear.



5 Select the tables you want to map to entity beans.

For each table you select, one entity bean will be created. From the Available list select the tables you want and move them to the Select list by using the > and >> buttons. When you've selected all your tables, choose Next.



6 Select the columns from each table to map to entity bean fields and specify any relationships you want to establish between the tables .

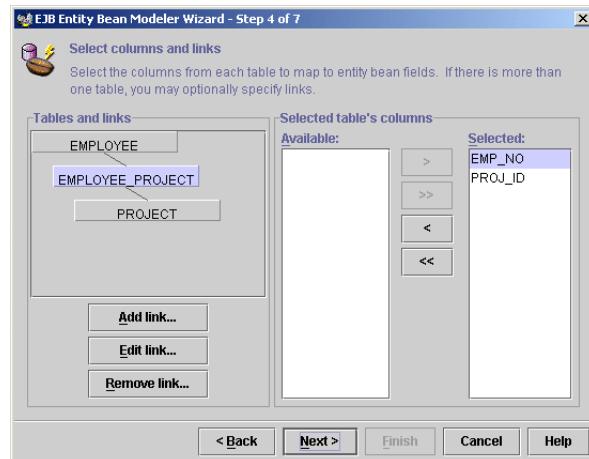
In the Tables and Links section, you'll see all the tables you selected in the previous step. Select each table in turn by clicking on it and then use the Selected Table's Columns section to move any columns of the table between the Available and Selected lists. By default, all columns in every table are selected.

You can also specify relationships between the tables by dragging the mouse pointer between the tables in the Tables and Links box on the left. Or you can use the Add Link button to do the same thing. When you use either method, a dialog box appears that proposes a relationship based on foreign keys, primary keys, unique indexes, and field names and types in the two tables. You can accept the suggested relationship or modify it to create the relationship you want. To remove a link between tables, choose Remove Link.

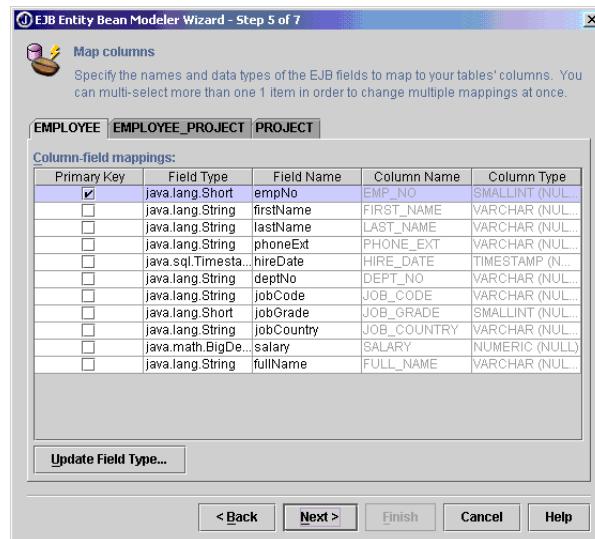
Note

The EJB 1.x Entity Bean Modeler supports CMP relationships only for the Borland servers only.

Here's an example of three tables linked together:



When you've selected all columns in each table that you want mapped to fields in entity beans you're creating, choose Next.



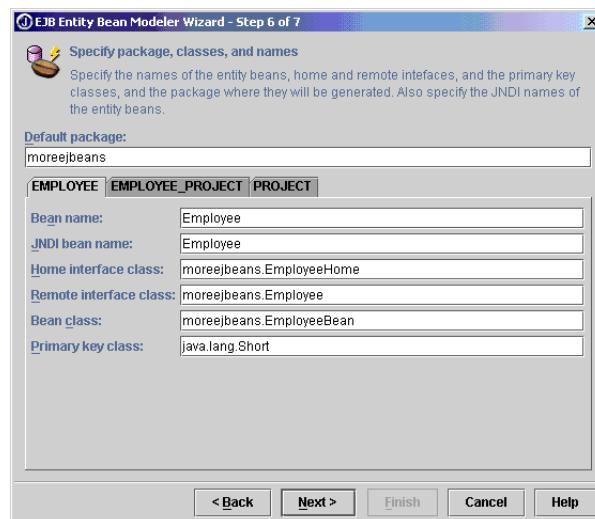
- 7 Specify the names and data types for the entity bean fields to map to your table's columns.

Click the appropriate tab to select the table you want to begin the mapping process on. For each column in the table a suggested Field Name and Field Type appears. You can simply accept the suggested

name or edit the suggested names and types as you want them to be in your bean.

To change the data type of multiple fields at one time, select the fields you want to change and choose Update Field Type. A dialog box appears in which you can type the new field type. When you choose Apply or OK, the field type for each selected field changes.

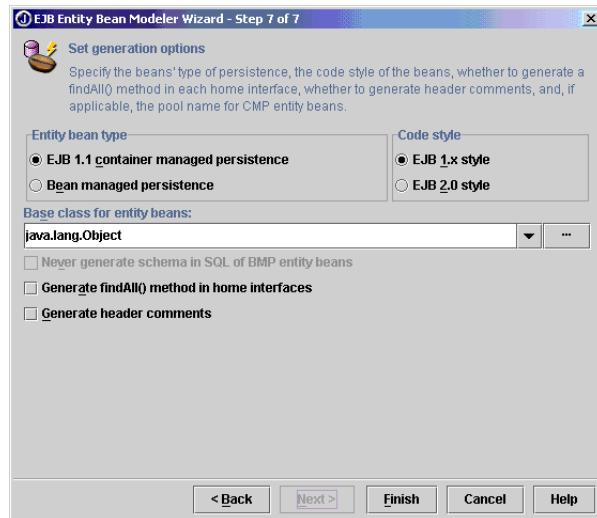
If the table already has a primary key, that field or set of fields is selected when the Map Columns page first appears. If no primary key exists, you must select one or more fields to make up primary key by checking the check box for those fields in the Primary Key column. When you finish mapping all the selected columns to the field names and types you want in your entity bean for each table, choose Next.



- 8 Specify the package, the classes and interfaces, and the JNDI name for each bean you are creating.

For each table, JBuilder suggests a name for the entity bean, the name used by JNDI, the name of the home and remote interfaces, the name of the bean class, and the type of the primary key class. You can specify a different package for each of these; by default, the project package is suggested. You can accept these values as they are, or you can modify

them as you wish. When you have finished specifying the information for each table, choose Next.



- 9** Select whether you want the entity beans to have container-managed or bean-managed persistence.

If you want to prepare for EJB 2.0 and want the code generated to follow the EJB 2.0 style, select the EJB 2.0 code style option. For more information about these options, choose the Help button in the EJB 1.x Entity Bean Modeler.

By default, the base class for your bean is `java.lang.Object`. If you want to use another class as the foundation of your entity beans, use the Base Class For Entity Bean field to specify another class.

If you want your entity beans capable of returning all rows in a data set, check the FindAll() Method In Home Interface option. The EJB 1.x Entity Bean Modeler places a `findAll()` method in the home interfaces of your beans. You can also choose whether you want header comments to appear in the resulting files.

The options available on this screen depend on your target application server.

- 10** Choose Finish.

JBuilder creates an entity bean for each table and all the supporting interfaces. You can now add the business logic you want to the beans, define the methods you want the client to be able to call in the remote interface, compile the beans, and edit the deployment descriptors for the beans.

Compiling enterprise beans and creating a deployment module

The JBuilder WebLogic Edition provides support for WebLogic Servers only

After you've used JBuilder to create either EJB 1.x or EJB 2.0 enterprise beans, you must compile your beans and prepare them for deployment.

Compiling the bean

When you've written and saved your enterprise bean, its interfaces, and any supporting classes, you're almost ready to compile. This chapter explains how to compile your bean classes and create deployment modules such as JARs or EARs.

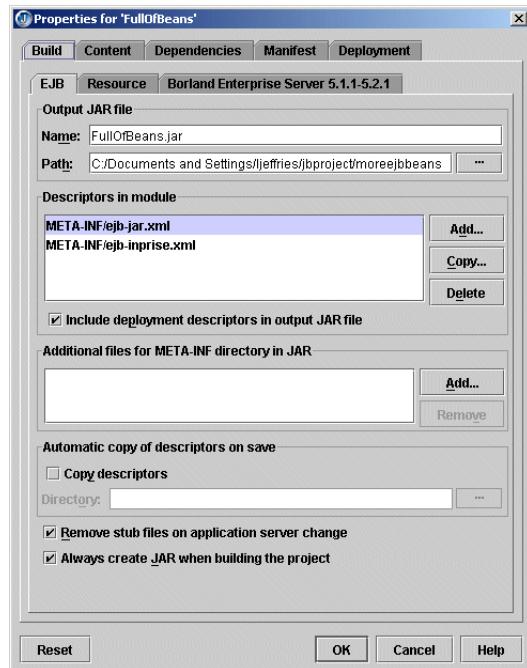
Changing build properties for a deployment module

Before you begin compiling, you might want to change build properties that determine how the JAR file is generated, although it's not required.

To change the build properties for an EJB module or EAR group,

- 1 Right-click the EJB module or EAR group in the project pane and choose Properties.
- 2 Select the Build tab.

3 Select the EJB or EAR tab.



4 Edit the build properties as you wish.

You can change the name of the output JAR or EAR file and where it is generated.

You can also insert deployment descriptors into an EJB module or EAR group. Clicking the Add button displays the Insert Deployment Descriptor dialog box; use it to navigate to and select the file you want to add to the EJB module. Clicking the OK button displays a message box that suggests you add the file to the `META-INF` directory. You can accept that suggestion or choose an existing directory that is a subdirectory of `META-INF`.

You can copy deployment descriptors, including the current descriptors' directory structure, to a new location. You can also delete deployment descriptors from an EJB module or EAR group.

If you want to specify that additional files should be added to the JAR file, click the Add Button and specify the location of the files. You'll need to do this if you've added a new class to your project, for example, and you want it to become part of the JAR file. Or if you have deployment descriptors you have edited outside of JBuilder, you can add them here and uncheck the Include Deployment Descriptors In Output JAR File. The deployment descriptors shown in the Deployment Descriptors In Module list won't be added to the JAR, but

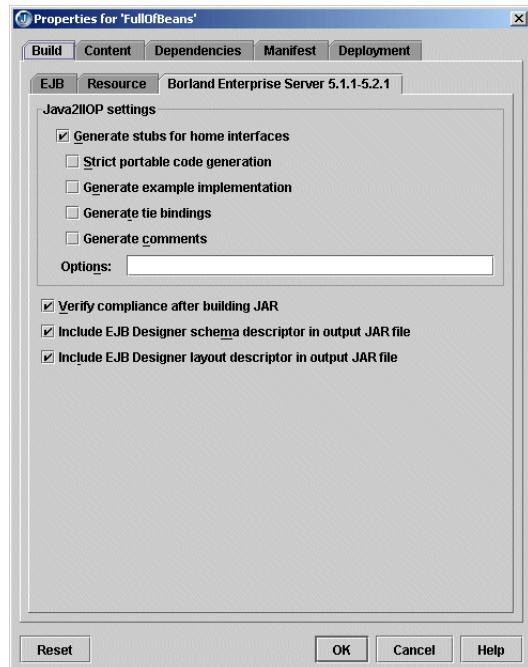
those you specified in the Additional Files For META-INF Directory In JAR list will be.

The Automatic Copy Of Descriptors On Save options allow you to save a copy of the deployment descriptors in their current state each time you save the EJB module, EAR group, or your project. To enable this feature, check the Copy Descriptors check box and specify the directory where they should be saved using the Directory field. Developers who write their own build tasks might find this capability desirable.

If you might target different application servers, you can use the Remove Stubs Files On Application Server Change option to remove client stubs used by the old application server when you select a new application server. This prevents the stubs meant for another application server from being copied into the newly generated JAR file.

The Always Create JAR When Building The Project option is on by default. By unchecking this option, you can prevent building the JAR file every time you choose to make or rebuild the project. When this option is not checked, you can still create the JAR by right-clicking the EJB module and choosing Make or Rebuild.

- 5 Click the tab of the application server you are targeting. For example, this image shows the Borland Enterprise Server 5.1.1 - 5.2.1 tab selected:



- 6 Specify the build options you want. If you need more information about the available options, click the Help button.
- 7 Click OK when you are done.

Changing the build properties for a bean

If you're targeting the Borland Enterprise Server 5.1.1 - 5.2.1 and you're going to test your bean locally, generate and add the client stubs to your classpath. You do this by changing the build properties of the home interface before compiling:

- 1 Right-click the home interface of the bean and choose Properties.
- 2 Click the Build tab.
- 3 Click the VisiBroker tab.
- 4 Check the Generate IIOP check box and select any other Java2IIOP options you want.
- 5 Click OK.

Changing the build properties for an EJB module

Most application servers have a server-specific Build Properties page you can use to set build options for an EJB module.

To see the Build Properties page for your selected target application server,

- 1 Right-click the EJB module in the project pane.
- 2 Choose Properties.
- 3 Click the Build tab, then on the Build page, click the tab with the name of your application server on it.

The options on the server-specific build properties page differs with each server. The page for WebLogic servers, for example, include EJBC or APPC compiler settings, depending on which version of WebLogic you are using.

Compiling

To compile all the classes in the project, right-click the project file (<project>.jpx) and choose Make, or simply choose Project | Make Project.

During the compiling process, JBuilder might detect that a problem exists in a deployment descriptor that makes it invalid. If this happens, you'll see a message appear in the message pane that tells you to verify the bean

in the Deployment Descriptor editor. For more information about verifying a deployment descriptor, see “[Verifying descriptor information](#)” on page 11-39.

If you’ve chosen to generate the client stubs, you’ll see that the home interface node in the project pane now has several files listed below it if you click its icon to expand it. These generated files are the required client stubs and helper classes that make enterprise beans work.

The build process is customized for the target application server and runs the tools specific to that application server in addition to compiling the .java files.

**Note for WebSphere
4.0 users**

Two deployment descriptors (`Map.mapxmi` and `Schema.dbxmi`) are generated for entity beans with container-managed persistence for the WebSphere 4.0 Advanced Edition only. If you are using one edition of WebSphere 4.0 and change your target server to the other version, you must recompile your project to ensure JBuilder generates the correct deployment descriptors for you.

The generated JAR file

Each enterprise bean that adheres to the EJB 2.0 specification requires a deployment descriptor entry in XML format. This format wasn’t mandatory in the EJB 1.1 specification, but nearly all application server vendors adopted this format for their 1.1 bean deployment descriptors. As you used the JBuilder wizards to create one or more enterprise beans, you also created one or more deployment descriptors that are in XML format.

When you compile your project, JBuilder creates a JAR file based on the configured name and displays it as a node under the module in the project pane.

You can also create the JAR file without compiling your entire project. Right-click the EJB module node in the project pane and choose Make to compile the EJB module node. If you want to modify the build properties before choosing Make, select the Properties menu item on the same popup menu and make any modifications you want in the Build Properties dialog box before choosing Make to generate the JAR file.

The JAR file contains all the deployment descriptors for the targeted application server. Each deployment descriptor is an XML file. Each JAR file can contain one or more deployment descriptors.

JBuilder will target one of multiple application servers. The application server you are targeting determines the number of deployment descriptors that are in the generated JAR file. Every JAR file will have an

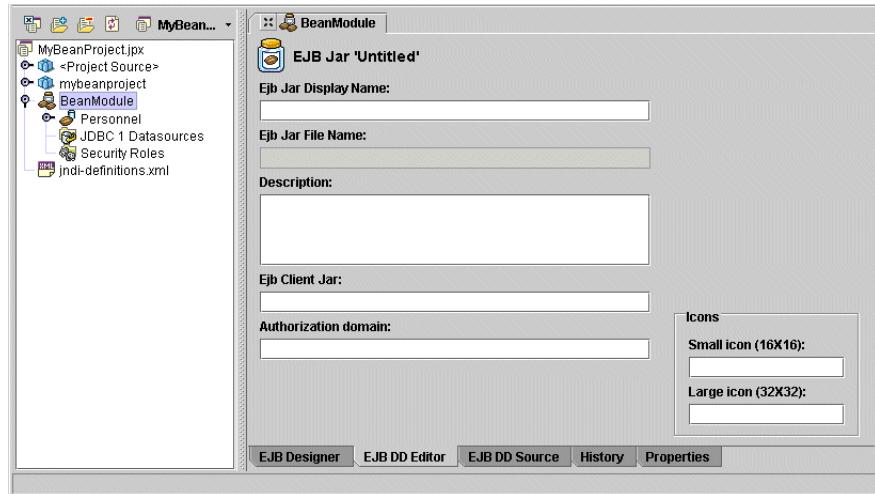
`ejb-jar.xml`, which describes the deployment attributes for the beans in the module that are common among all application servers. `ejb-jar.xml` is the EJB 1.1- or EJB 2.0-compliant deployment descriptor.

Vendor-specific information for an EJB 2.0- module is kept in the `ejb-borland.xml` file, even when the application server is some other than a Borland server. For EJB 1.1 modules, the file name is `ejb-inprise.xml`. When you compile, additional vendor-specific XML files are generated from this information. They are also generated when you click the Deployment Descriptor editor Source tab.

Editing deployment descriptors

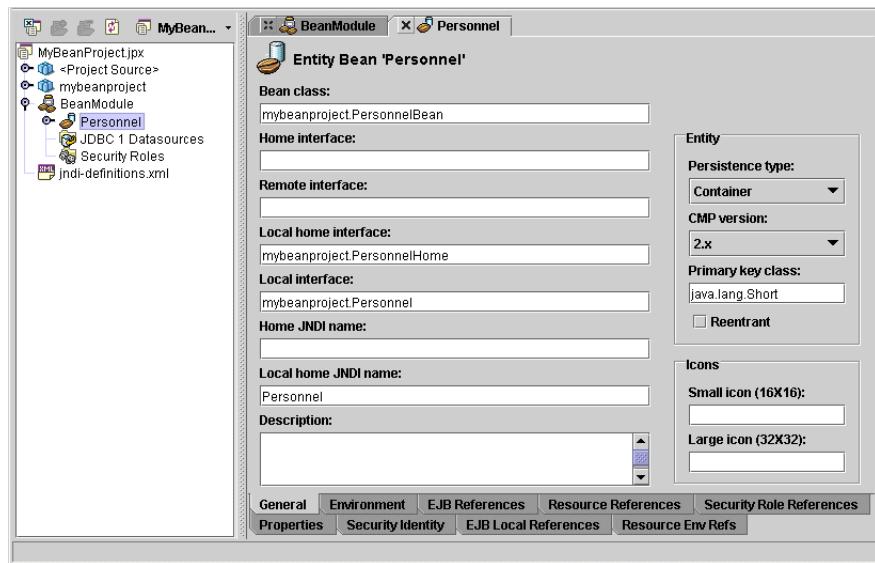
JBuilder's Deployment Descriptor editor provides a way to modify the existing deployment descriptors. You can, however, choose to use any other deployment descriptor editing tool you want.

To display the Deployment Descriptor editor, double-click the EJB module in the project pane and click the EJB DD Editor tab at the bottom of the content page. The Deployment Descriptor editor appears:



To view information about an enterprise bean in the Deployment Descriptor editor, open the EJB module in the project pane by clicking the far left icon next to the module name. You'll see the beans contained in the module listed. Double-click the name of the enterprise bean you want to edit. When a bean is selected in the editor, several tabs appear in the

Deployment Descriptor editor. You use these tabs to go to panels where you edit deployment descriptor information.



For detailed information about using the Deployment Descriptor editor, see [Chapter 11, “Using the Deployment Descriptor editor.”](#)

Verifying descriptors

After you've finished editing the descriptor, you can verify the file to make sure the descriptor information is correct, the required bean class files are present, and so on. Verification occurs on the `ejb-jar.xml` and `ejb-borland.xml` deployment descriptors. Use it when your target application server is a Borland server only.

To verify descriptor information, right-click the module in the project pane and choose Verify.

Verify does the following:

- Ensures that the descriptor conforms to the EJB 1.1 or 2.0 specification, depending on the type of the EJB module.
- Ensures that the classes referenced by the deployment descriptors conform to the EJB 1.1 or 2.0 specification, depending on the type of the EJB module.

If the verification fails, one or more messages appear in a Log panel describing the failures.

Running and testing an enterprise bean

The JBuilder WebLogic Edition provides support for WebLogic Servers only

Once you've finished creating an enterprise bean, you're ready to run it. The quickest way to do that is to right-click its EJB module or the JAR file the module contains and select Run Using Defaults or Debug Using Defaults from the context menu. Or you can create a runtime configuration for your bean and select Run Using <runtime configuration> or Debug Using <runtime configuration> from the context menu. This starts the container for the currently selected application server using the JAR for this EJB module. Be patient as the start-up process takes a while.

You can view the progress of the start-up process in the message window. Any errors that occur will also appear there. If you want to run multiple JARs on the current application server, select multiple EJB modules.

Note for WebLogic 7.x and 8.1 users

If WebLogic 7.x or 8.1 is your target application server, context menu items for running or debugging don't appear when you right-click the JAR file or EJB module in the project pane. Deployment is not available at startup time for WebLogic 7.x and 8.1. Deploy your deployable modules after the server starts up.

Note for WebSphere 4.0 Advanced Edition users

You must create an EAR group that contains your beans before deploying. For more information about EAR groups, see ["Creating an EAR file" on page 10-6](#).

Note for WebSphere users

After choosing Run or Debug from the context menu, you must then take the extra step of choosing a Deploy menu command to deploy your beans. See ["Hot deploying to an application server" on page 10-10](#) for more information.

Notes for iPlanet users

Before you begin running or debugging an application, start the iPlanet server outside of JBuilder.

When you choose to run or debug an enterprise bean when iPlanet is the target application server, the iPlanet startup process is more complicated. You will see additional message boxes appear. JBuilder must stop the iPlanet KJS processes running outside of JBuilder, deploy the selected archive, and start a single KJS process within JBuilder. At each step, you'll see a message box. As with other servers, you can view the progress of the start-up process in the message window. You will not be able to interact with JBuilder until you see the ENGINE-ready message. Once you do see the ENGINE-ready message, you should probably scroll back in the message pane to verify that no errors have occurred.

To stop the iPlanet server, click the red stop button at the bottom right of the message pane. The normal KJS processes are then restarted outside of JBuilder.

If an iPlanet error occurs, you must use iPlanet's administration tool to shutdown and restart the server. You can access it from JBuilder's Tools menu if you have checked the Add iPlanet Tools To The Tools Menu option when you set up iPlanet for JBuilder using Tools | Configure Servers.

To launch Web View with iPlanet, stop the web server outside of JBuilder, launch the server within JBuilder, and restart the web server after you see the message ENGINE IS READY message in JBuilder.

When you stop the iPlanet server, the running JAR is undeployed. To deploy the JAR again, restart the server and deploy the JAR.

Testing your bean

To test your bean, you can use the EJB Test Client wizard to generate test clients that make calls to your bean. The EJB Test Client wizard can generate three different types of test clients:

- An EJB test client application calls the methods of your bean.
- A JUnit test case makes use of the JUnit testing framework. It runs as a client in a separate VM from your EJB, passing parameters by value.
- A Cactus JUnit test case runs on the server, so it can access server-side resources and make local calls to your EJB from a web application.

Choosing the type of test client

With three different types of test clients to choose from, it may be difficult to decide which type is right for you. The following discussion of the merits of each type of test client may help you decide.

- The Application type of EJB test client is the easiest to set up. It works like an application and connects to the server. Validating the test results must be done manually, however, because it does not make use of the JUnit testing framework.
- Use of the JUnit type of test client requires knowledge of the JUnit framework. This type of test client is highly automated and can be run as part of a test suite. Use of the JUnit framework allows automatic checking of results and good reporting capabilities. It is not as powerful as the Cactus type, because it runs the test as a client.
- The Cactus type of test client is the most comprehensive among the three different types. It requires Cactus and the JUnit framework, which makes setup more complex than for the other types. Cactus test cases run on the server, so they can access server-side resources. This is the only type of test client that can test an EJB 2.0 entity bean with a local interface.

Working with test client applications

This section explains how to use the EJB Test Client wizard to create a simple test client application and how to use the resulting application to test your enterprise bean.

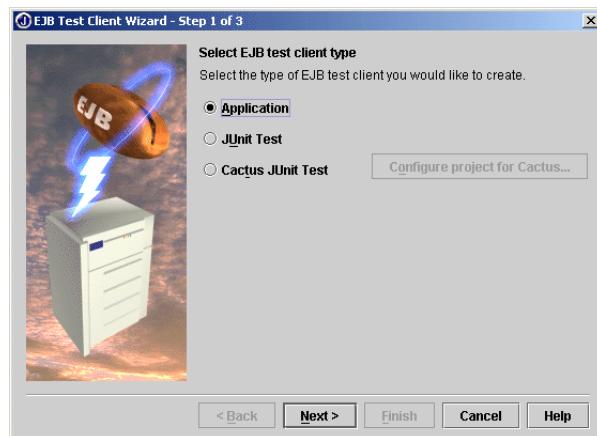
Creating a test client application

JBuilder can help you create a test client application that makes calls to your new bean.

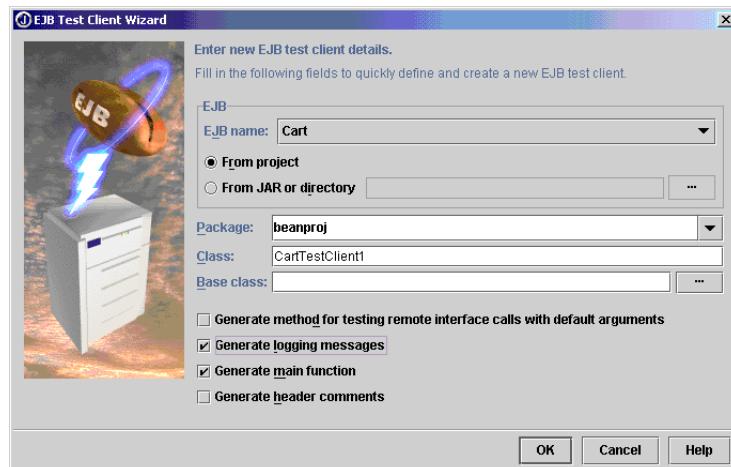
To create a test client application,

- 1 Open the project that contains the EJB module for your enterprise bean.

- 2 Choose File | New, click the Enterprise tab, and double-click the EJB Test Client icon to display the EJB Test Client wizard:



- 3 Select the Application test client type and choose Next.



- 4 Select the bean you want to create a client for using one of the Select EJB options and specifying the bean:
- Select From Project if your bean is in the current project and specify which bean by selecting it from the drop-down list.

- Select From JAR Or Directory if your bean is not in the current project, but exists elsewhere in a JAR file or a directory. Use the ... button to navigate to where the JAR is located and select the JAR, then use the drop-down list to select the bean you want.

Note

You will see only EJBs with remote interfaces in these lists because enterprise beans with local interfaces cannot be accessed by a client application, only by another bean or a web component.

- 5 Select the package name from the list of packages. The current package is the default value.
- 6 Enter a name for the test client class or accept the default name.
- 7 Select the options you want:

- Generate Method For Testing Remote Interface Calls With Default Arguments

Adds a `testRemoteCallsWithDefaultArguments()` method that tests the remote interface calls with default argument values. For example, the default argument for a String is "", the default argument for an int is 0, and so on.

- Generate Logging Messages

Adds code that displays messages reporting on the bean's status as the client runs. For example, a message is displayed when bean initialization is begun and another when it completes. This option also generates wrappers for all the methods declared in the home and remote interfaces and initialization functions. Finally, the messages report how long each method call takes to complete.

- Generate Main Function

Adds the main function to the client.

- Generate Header Comments

Adds JavaDoc header comments to the client you can use to fill in information such as title, author, and so on.

8 Choose Next.



- 9** Check the Create a Runtime Configuration check box and supply the information needed to create a runtime configuration. JBuilder suggest a default name you can change to any name of your choosing.

10 Choose Finish.

The EJB Test Client wizard generates a test client that creates a reference to the enterprise bean.

If the Generate Logging Messages option is selected, for each method declared in the bean's remote interface, the wizard also declares and implements a method that calls the remote method. Each of these methods reports its success in invoking the remote method and how long the remote method took to execute.

There are multiple ways to use the generated test client application. If you added a `main()` function to the test client application, you can write the code that invokes the calls to the enterprise bean's methods in the `main()` function. You do this by first calling either a `create` or `find` method, and, if a remote reference is returned, by using that remote reference to call the bean's business methods. Or, because the wizard has declared a `client` object in the `main()` function, you can use that `client` object to simply call the methods declared in the test client application that call the bean's remote methods.

If you selected the Generate Method For Testing Remote Interface Calls With Default Arguments option, your client class now contains a `testRemoteCallsWithDefaultArguments()` method. If you selected the logging option, this method calls the remote method wrappers that were

generated from the logging option. To test each remote method, you can then simply call `testRemoteCallsWithDefaultArguments()` after you create a remote interface reference in either the client class `create()` method or in one of its `findByXXX()` methods.

If you did not select the logging option, the `testRemoteCallsWithDefaultArguments()` method requires a remote interface passed as a parameter. You must then create a remote interface reference in either the home reference's `create()` method or in one of its `findByXXX()` methods. Then add the code to the client class to call the `testRemoteCallsWithDefaultArguments()` method, passing it the remote reference as a argument.

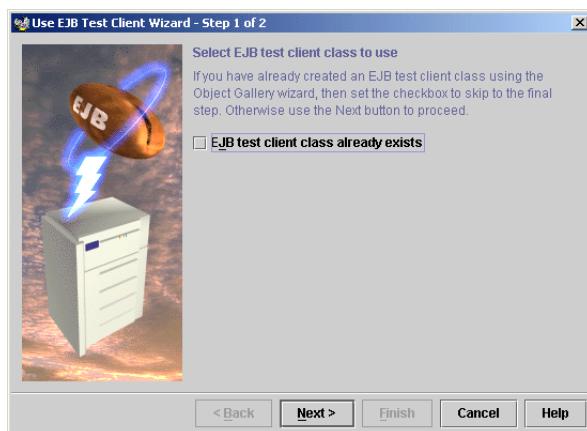
If you prefer to write the logic that calls each of the business methods from another class, you can choose to create and use an instance of the test client application. See "[Using the test client application](#)" on page 9-7.

Compile your test client application by right-clicking the test client node in the project pane and choosing Make.

Using the test client application

You can quickly add a declaration of a test client class to any class.

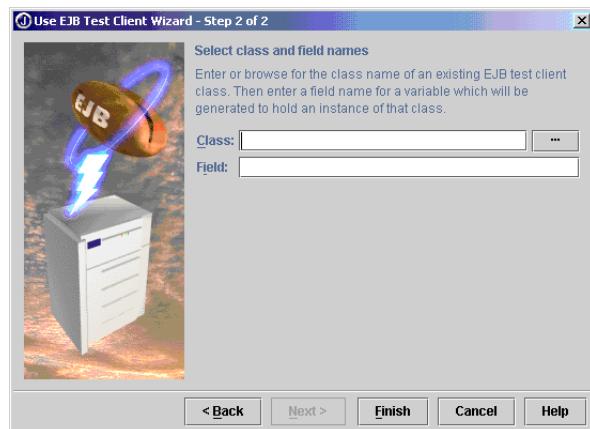
- 1 Display the class in which you want the declaration to appear in the editor.
- 2 Choose Wizards | EJB | Use EJB Test Client.



- 3** If the test client already exists, check the EJB Test Client Class Already Exists option.

If this option isn't checked, when you click Next, the EJB Test Client wizard starts. When you are through using it, the Use EJB Test Client wizard resumes.

- 4** Click Next to go to Step 2.



- 5** For the Class field, navigate to the test client class you want to use.
6 In the Field field, specify a name for the variable that will hold an instance of the test client class, or accept the default value the wizard suggests.
7 Choose Finish.

The wizard adds a declaration of the test client application you specified to the class like this, for example:

```
EmployeeTestClient1 employeeTestClient1 = new EmployeeTestClient1();
```

Now you're ready to call the methods declared in the test client application.

Using your test client application to test your enterprise bean

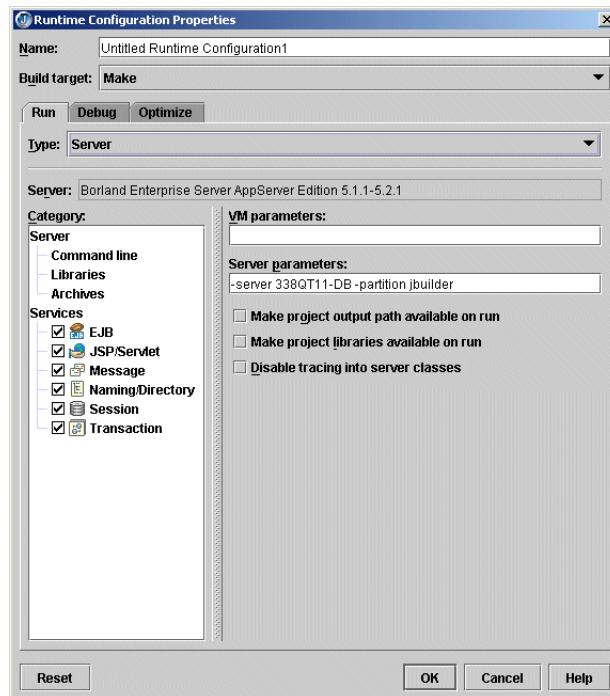
Once you've created a client test application, you're ready to start the container and run the client application. Create two runtime configurations: Server and Client.

Creating a Server runtime configuration

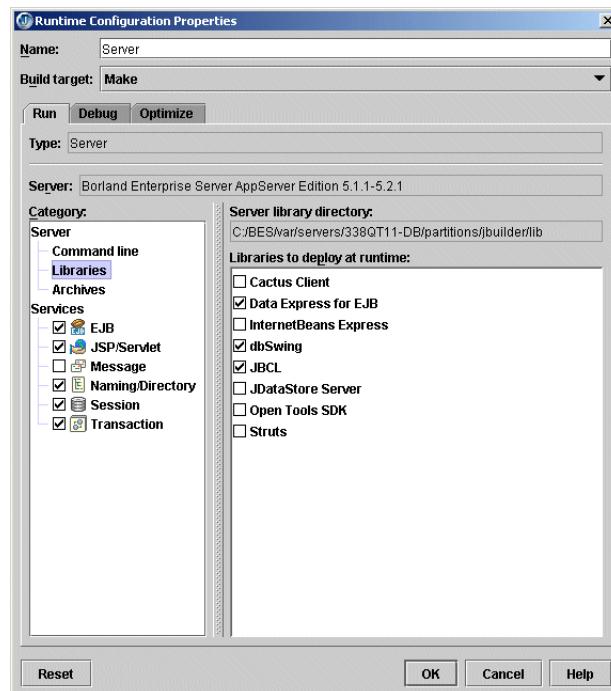
You will need to create a Server runtime configuration no matter which of the three types of EJB test clients you're using.

To create a Server configuration,

- 1 Choose Run | Configurations.
- 2 In the dialog box that appears, click the New button.
- 3 Select Server from the Type drop-down list.

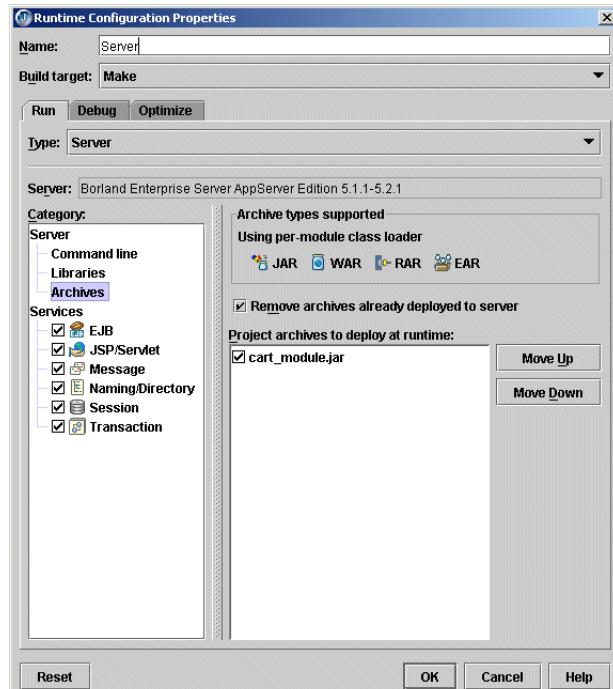


- 4 In the Name field, enter Server.
- 5 Fill in the VM Parameters and Server Parameters needed to run the server. If you've selected a target application server as described in "Selecting an application server" in *Developing J2EE Applications*, default values are already in place.
- 6 If Borland Enterprise Server is your target application server, click the Libraries node in the Category list. (This option is not available for other servers.)



Select those libraries you want deployed from the Libraries To Deploy At Runtime list.

- 7 If Borland Enterprise Server is your target application server, click the Archives node in the Category list. (This option is not available for servers that don't allow deployment on server startup.)



- 8 Select the JAR file containing the beans you want to test in the list of archives. The listed archives are retrieved from the EJB modules and EAR groups in the project.

If your EJB test client is a Cactus test client, you may also need to select a WAR file or an EAR file here. See "[Configuring your project for testing an EJB with Cactus](#)" on page 9-17 for more information.

- 9 Click OK two times.

Running your EJB test client application

For Borland Enterprise Server 5.1.1 - 5.2.1 users, you must now start the Borland Enterprise Server Management Agent. Choose Tools | Borland Enterprise Server Management Agent. (This step is optional as the server

startup process attempts to start the Enterprise Management Agent if it isn't already started.)

Now you're ready to start the container. Select the Server run configuration from the drop-down list next to the Run button on the JBuilder toolbar:



The container starts up. Be patient as the start-up process takes a while. You can view the progress of the start-up process in the message window. Any errors that occur will also appear there.

If you need to, deploy the archives you want to test to the server.

Next select the Client run configuration to run your client application. The messages that appear in the message pane report the success or failure of the client application's execution.

You can debug your enterprise beans or the client just as you would any other Java code with JBuilder. For information about debugging, see "Debugging Java programs" in *Building Applications with JBuilder*.

Working with JUnit test cases

This section explains how to use the EJB Test Client wizard to create a JUnit test case and how to run it.

Creating a JUnit test case

JUnit is an open source framework for unit testing written by Erich Gamma and Kent Beck. JUnit provides a variety of features which support unit testing, among them two classes, `junit.framework.TestCase` and `junit.framework.TestSuite`, which are used as base classes for writing unit tests. JUnit also provides three different kinds of test runners, `TextUI`, `SwingUI`, and `AwtUI`. Of these three test runners, two of them, `TextUI` and `SwingUI`, are available within the JBuilder IDE. The EJB Test Client wizard can generate a JUnit test case (a class that extends `TestCase`) for testing your EJB.

To create and run a JUnit test for an EJB, you need the following:

- A properly configured server that supports EJB services.
- A project which has the correct server selected on the Server page of the Project Properties dialog box.

- An EJB, contained in an EJB Module.
- A JUnit EJB test client, which you can create with the EJB Test Client wizard.
- A Server type runtime configuration that uses the correct server settings. See “[Creating a Server runtime configuration](#)” on page 9-9.
- A Test type runtime configuration. The simplest way to create this is to check Create A Runtime Configuration in the last step of the EJB Test Client wizard when you create your JUnit EJB test client. This helps ensure that the Test runtime configuration has the proper settings.

See also

- “Configuring the target server settings” in *Developing J2EE Applications*
- “[Creating a Server runtime configuration](#)” on page 9-9
- “Unit Testing” in *Building Applications with JBuilder*
- “Setting runtime configurations” in *Building Applications with JBuilder*

Creating a JUnit test case using the EJB Test Client wizard

The EJB Test Client wizard can generate three different types of test clients for your EJB. One of these is a standard JUnit test client. Another type of JUnit test client is a Cactus JUnit test client, but that’s discussed later.

To generate a standard JUnit test client for your EJB,

- 1 Select File | New.
- 2 Select EJB Test Client on the Enterprise page of the object gallery. Click OK. The EJB Test Client wizard opens.
- 3 Select the JUnit Test option.
- 4 Click Next to go to the next step of the EJB Test Client wizard.
- 5 Select the bean you want to create a client for using one of the Select EJB options and specifying the bean:
 - Select From Project if your bean is in the current project and specify which bean by selecting it from the drop-down list.
 - Select From JAR Or Directory if your bean is not in the current project, but exists elsewhere in a JAR file or a directory. Use the ... button to navigate to where the JAR is located and select the JAR, then use the drop-down list to select the bean you want.

Note

You will see only EJBs with remote interfaces in these lists because enterprise beans with local interfaces cannot be accessed by a client application, only by another bean or a web component.

- 6 Specify the Package, Class Name, and Base Class of the new JUnit test client. You don’t need to change the default values unless you want to.

The Base Class should not be changed unless you have another class that you want to extend in place of `junit.framework.TestCase`.

- 7 Select the optional code for the wizard to generate using the following checkboxes:

- Generate Method For Testing Remote Interface Calls With Default Arguments

Adds a `testRemoteCallsWithDefaultArguments()` method that tests the remote interface calls with default argument values. For example, the default argument for a String is "", the default argument for an int is 0, and so on.

- Generate Logging Messages

Adds code that displays messages reporting on the bean's status as the client runs. For example, a message is displayed when bean initialization is begun and another when it completes. This option also generates wrappers for all the methods declared in the home and remote interfaces and initialization functions. Finally, the messages report how long each method call takes to complete.

- Generate Header Comments

Adds JavaDoc header comments to the client you can use to fill in information such as title, author, and so on.

- 8 Click Next to go to the next step of the EJB Test Client wizard.

- 9 Check Create A Runtime Configuration if you wish to create a new runtime configuration. You need to do this if you don't already have a Test type runtime configuration for your project. If you are creating a new runtime configuration here, specify the Name. You may also specify a Base Configuration. This copies the settings from the specified configuration, which you can later modify using Runtime Configuration Properties dialog box.

- 10 Click Finish to generate the JUnit test client.

Now that you've used the EJB Test Client wizard to generate a JUnit test client, you must implement the test client. You'll have to do the following:

- 1 Create a remote interface reference to your EJB by calling the `create()` method that was generated by the wizard. There's a `@todo` comment in the generated code to show you where to add this (in the `setUp()` method). This can be as simple as adding the following line of code (where `<beanname>` is the private variable the wizard generated to refer to your EJB):

```
<beanname> = create();
```

- 2 Add any other test methods you wish to write.

Once you've created and implemented your JUnit test client, you're ready to run the test client and test your EJB.

See also

- “[Running your JUnit test case](#)” on page 9-15
- “[Creating a Server runtime configuration](#)” on page 9-9
- “[Setting runtime configurations](#)” in *Building Applications with JBuilder*

Running your JUnit test case

Running a JUnit test case for an EJB in the JBuilder IDE is in many ways similar to running other types of JUnit tests. The same test runners are available and you interact with them in the same ways. The main difference between testing an EJB and running any other JUnit test is that for an EJB, you need to start the server first.

To run your JUnit EJB test client,

- 1 Start the server using the Server runtime configuration.
- 2 If necessary, deploy the archives you want to test to the server.
- 3 Right-click the JUnit test file in the project pane.
- 4 Select Run Test Using <test configuration> from the context menu. The test runs in the test runner that's specified in your Test runtime configuration.

See also

- “[Creating a Server runtime configuration](#)” on page 9-9
- “[Running tests](#)” in *Building Applications With JBuilder*
- “[Setting runtime configurations](#)” in *Building Applications with JBuilder*

Working with Cactus JUnit test cases

This section explains how to configure your project for testing an enterprise bean with Cactus, how to use the EJB Test Client wizard to create a Cactus JUnit test case, and how to run it.

Creating a Cactus JUnit test case

Cactus extends JUnit to provide unit testing of server-side Java code. It does this by redirecting your test case to a server-side proxy. For example, it allows you to test an EJB with local interfaces. The EJB Test Client

wizard can generate a Cactus JUnit test case for server-side testing of your EJB.

To create and run a Cactus test for an EJB, you need the following,

- A properly configured server that supports EJB services.
- A project which has the correct server selected on the Server page of the Project Properties dialog box.
- A WebApp.
- An EJB, contained in an EJB Module.
- A Cactus EJB test client, which you can create with the EJB Test Client wizard.
- A Server type runtime configuration that uses the correct server settings. See “[Creating a Server runtime configuration](#)” on page 9-9 for more information.
- A Test type runtime configuration, which can optionally have the correct archives selected to be redeployed.
- A Cactus configuration for your project that specifies the correct Server type and Test type runtime configurations. You create this using the Cactus Setup wizard.

If you are running Cactus tests for any EJB using Borland Enterprise Server or running Cactus tests for an EJB with local references using WebLogic, you should consider the following:

- You need an Enterprise Archive (EAR) that contains both the EJB’s JAR file and the WebApp’s WAR file.
- Your Test type runtime configuration can optionally have the EAR file selected as an archive to redeploy.
- The Cactus configuration for your project can optionally specify the EAR file as an archive to redeploy.

See also

- “[Creating a Server runtime configuration](#)” on page 9-9
- “Unit Testing” in *Building Applications with JBuilder*
- “Working with WebApps and WAR files” in *Web Application Developer’s Guide*
- “Setting runtime configurations” in *Building Applications with JBuilder*

Configuring your project for testing an EJB with Cactus

JBuilder provides the Cactus Setup wizard to help configure your project for Cactus. This makes it possible to run Cactus tests within the JBuilder IDE. The wizard is available by selecting Wizards | Cactus Setup.

Cactus configuration prerequisites

Before you can successfully configure your project for Cactus testing of an EJB, you need the following:

- A properly configured server that supports EJB services. See “Working with servers” in *Server Configuration Guide*.
- A project which has the correct server selected on the Server page of the Project Properties dialog box.
- A WebApp. You create this using the Web Application wizard, available on the Web page of the object gallery (File | New). See “Working with WebApps and WAR files” in *Web Application Developer’s Guide*.

Tip You can run the Web Application wizard from within the Cactus Setup wizard by clicking the New button next to the WebApp drop-down list.

- An EJB that you want to test, contained in an EJB Module.
- A Server type runtime configuration that uses the correct server. See “[Creating a Server runtime configuration](#)” on page 9-9 for more information.

Tip You can create a new Server type runtime configuration from within the Cactus Setup wizard by clicking the New button next to the Server runtime configuration drop-down list.

- A Test type runtime configuration. See “Setting runtime configurations” in *Building Applications with JBuilder*.

Tip You can create a new Test type runtime configuration from within the Cactus Setup wizard by clicking the New button next to the Test runtime configuration drop-down list.

If you plan to run Cactus tests for any EJB using Borland Enterprise Server or run Cactus tests for an EJB with local references using Weblogic, before you can configure your project for Cactus you should consider the following:

- You need an Enterprise Archive (EAR) that contains both the EJB’s JAR file and the WebApp’s WAR file. You create this using the EAR wizard, available on the Enterprise page of the object gallery (File | New).
- Your Test type runtime configuration can optionally have the EAR file selected as an archive to redeploy.

Running the Cactus Setup wizard

The Cactus Setup wizard configures your project to run Cactus tests within the JBuilder IDE. It creates the properties files that Cactus requires and ensures that the archives and other files needed for your tests are properly deployed. The Cactus Setup wizard is available by selecting Wizards | Cactus Setup.

Before running the Cactus Setup wizard, take a look at the prerequisites for Cactus configuration that are described in the previous section. The necessary WebApp and runtime configurations can be created from within the Cactus Setup wizard. You must provide the rest of the prerequisites before the Cactus Setup wizard can successfully configure your project for EJB testing.

To run the Cactus Setup wizard and configure your project for Cactus,

- 1 Select Wizards | Cactus Setup. The Cactus Setup wizard opens.

Tip You can also run the Cactus Setup wizard from within the EJB Test Client wizard.

- 2 Select the WebApp to which the wizard will add Cactus test support. You may use the default WebApp, an existing WebApp, or click the New button to open the Web Application wizard and create a new WebApp.
 - 3 Choose the logging settings for the Cactus logs. Specify the locations for the Cactus server and client logs, or uncheck Enable Logging if you don't want any logs.
 - 4 Click Next.
 - 5 Select the archives to redeploy before each test. This keeps the archives in sync with the project. If any of the archives are shown in red with an exclamation point before the name, it means that the physical file does not yet exist. This is probably because the archive has not yet been built. It won't cause any problem to select one of these archives, as long as you remember to build the archive before attempting to run Cactus tests.
- Important** If you're using Cactus and Borland Enterprise Server to test an EJB that's accessed by a WebApp, make sure you have an Enterprise Archive (EAR) file that contains both the EJB's JAR file and the WebApp's WAR file. When selecting archives to deploy in the Cactus Setup wizard, select the EAR file and deselect the JAR file and WAR file.
- 6 Select a Server runtime configuration. You may create a new one using the New button.

- 7 Select a Test runtime configuration. You may create a new one using the New button.
- 8 Click Finish. Your project is now configured for use with Cactus.

Special server-specific setup is required when using Cactus to test EJBs that locally reference other EJBs. The following sections discuss the required setup for Borland Enterprise Server and Weblogic versions 6.x through 7.0.

Accessing EJBs with local interfaces using Cactus and Borland Enterprise Server

To test an EJB with local references using Cactus and Borland Enterprise Server, you must create an EJB local reference entry in both the standard (`web.xml`) and vendor-specific (`web-borland.xml`) web descriptors. You must also update the JNDI name reference in the Cactus test client to use the EJB local reference.

To add an EJB local reference to `web.xml`,

- 1 Expand the WebApp node in the project pane.
- 2 Expand the Deployment Descriptors node.
- 3 Double click on `web.xml` to open it in the WebApp DD Editor.
- 4 Click on Local EJB References in the structure pane.
- 5 Click on the Add button in the Local EJB References page of the WebApp DD Editor.
- 6 Enter a suitable Reference Name, such as `ejb/<local JNDI name>`.
- 7 Select the Type. The possible choices are Session or Entity.
- 8 Enter the class names for the local home and local interfaces.

To add an EJB local reference to `web-borland.xml`,

- 1 Expand the WebApp node in the project pane.
- 2 Expand the Deployment Descriptors node.
- 3 Double click on `web-borland.xml` to open it in the editor.
- 4 Add the following descriptor element to the XML source code:

```
<ejb-local-ref>
  <ejb-ref-name>ejb/Reference name</ejb-ref-name>
  <jndi-name>JNDI name of the EJB</jndi-name>
</ejb-local-ref>
```

The value of `ejb-ref-name` should match the reference name in `web.xml` and the value of `jndi-name` is the local JNDI name of the EJB.

To modify the Cactus test client,

- 1 Double-click the Cactus test client file in the project pane to open it in the editor.
- 2 Locate the following statement:

```
Object ref = ctx.lookup("<JNDI Local name>");
```

- 3 Modify the statement as follows:

```
Object ref = ctx.lookup("java:comp/env/<EJB Local reference name>");
```

Accessing EJBs with local interfaces using Cactus and Weblogic 6.x/7.x/8.x

To test an EJB with local references using Cactus and Weblogic 6.x/7.x/8.x, you must ensure your EJB's JAR file and your WebApp's WAR file are packaged inside an Enterprise Archive (EAR) and you must deploy the EAR.

To modify the server settings,

- 1 Select Project | Project Properties.
- 2 Select the Server page of the Project Properties dialog box.
- 3 Click on the JSP/Servlet service and uncheck Map Project WebApps At Runtime. This will turn off the default behavior of deploying the web application in exploded format at server startup.
- 4 Click OK to close the Project Properties dialog box.

Next you must create an EAR that contains the EJB's JAR file and the WebApp's WAR file. To create the EAR, use the EAR wizard, available on the Enterprise page of the object gallery (File | New). For more information on creating an EAR, see "[Creating an EAR file](#)" on page 10-6. After creating the EAR, you should rebuild the project.

If you are using Weblogic 7.x or 8.x,

- 1 Start the server.
- 2 Right-click on the EAR group and select Deploy Options | Deploy.

If you are using Weblogic 6.x, you need to create a new Server type runtime configuration. To create the runtime configuration,

- 1 Select Run | Configurations.
- 2 Click New. The Runtime Configuration Properties dialog box opens.
- 3 Select Server from the Type drop-down list.

- 4 Click on the Archives node of the Category tree.
- 5 Select the EAR and unselect the JAR and WAR files in the Project Archives To Deploy At Runtime list.
- 6 Click OK to close the Runtime Configuration Properties dialog box.
- 7 Click OK to close the Project Properties dialog box.

Start the server using the new runtime configuration.

- Note** To look up a bean with a local interface using Weblogic, the Cactus test client must refer to the bean using its local JNDI name.
- Note** For other servers, refer to the server documentation for instructions on accessing EJBs with local references from a web application.

Creating a Cactus JUnit test case using the EJB Test Client wizard

The EJB Test Client wizard can generate three different types of test clients for your EJB. One of these is a Cactus test client. Cactus is an extension of JUnit that provides server-side testing of Java code. To generate a Cactus test client for your EJB,

- 1 Select File | New.
- 2 Select EJB Test Client on the Enterprise page of the object gallery. Click OK. The EJB Test Client wizard opens.
- 3 Select the Cactus JUnit Test radio button.
- 4 Click Configure Project For Cactus if you have not already run the Cactus Setup wizard. Clicking this button opens the Cactus Setup wizard. See "[Running the Cactus Setup wizard](#)" on page 9-18 for more information.
- 5 Click Next to go to the next step of the EJB Test Client wizard.
- 6 Select the bean you want to create a client for using one of the Select EJB options and specifying the bean:
 - Select From Project if your bean is in the current project and specify which bean by selecting it from the drop-down list.
 - Select From JAR Or Directory if your bean is not in the current project, but exists elsewhere in a JAR file or a directory. Use the ... button to navigate to where the JAR is located and select the JAR, then use the drop-down list to select the bean you want.
- 7 Specify the Package, Class Name, and Base Class of the new Cactus test client. You don't need to change the default values unless you want to. The Base Class should not be changed unless you have another class that you want to extend in place of `org.apache.cactus.ServletTestCase`.

- 8 Select the optional code for the wizard to generate using the following checkboxes:
 - Generate Method For Testing Remote Interface Calls With Default Arguments
Adds a `testRemoteCallsWithDefaultArguments()` method that tests the remote interface calls with default argument values. For example, the default argument for a String is "", the default argument for an int is 0, and so on.
 - Generate Logging Messages
Adds code that displays messages reporting on the bean's status as the client runs. For example, a message is displayed when bean initialization is begun and another when it completes. This option also generates wrappers for all the methods declared in the home and remote interfaces and initialization functions. Finally, the messages report how long each method call takes to complete.
 - Generate Header Comments
Adds JavaDoc header comments to the client you can use to fill in information such as title, author, and so on.
- 9 Click Next to go to the next step of the EJB Test Client wizard.
- 10 Check Create A Runtime Configuration if you wish to create a new runtime configuration. This step is optional. If you already have a Test type runtime configuration that you specified when running the Cactus Setup wizard, it is unlikely that you need to do this. If you do decide to create a new runtime configuration here, specify the Name. You may also specify a Base Configuration. This copies the settings from the specified configuration, which you can later modify using Runtime Configuration Properties dialog box.
- 11 Click Finish to generate the Cactus test client.

Now that you've used the EJB Test Client wizard to generate a Cactus test client, you must implement the test client. You'll have to do the following:

- 1 Create a remote interface reference to your EJB by calling the `create()` method that was generated by the wizard. There's a `@todo` comment in the generated code to show you where to add this (in the `setUp()` method). This can be as simple as adding the following line of code (where `<beanname>` is the private variable the wizard generated to refer to your EJB):

```
<beanname> = create();
```
- 2 Add any other test methods you wish to write.

- 3 Update the JNDI name reference to use the EJB local reference if you are using Borland Enterprise Server as your server to test an EJB with local interfaces. See “[Accessing EJBs with local interfaces using Cactus and Borland Enterprise Server](#)” on page 9-19 for more information.
- 4 Look for any other `@todo` comments in the code.

Once you’ve configured your project for Cactus and created and implemented your Cactus test client, you’re ready to run the test client and test your EJB.

See also

- “[Configuring your project for testing an EJB with Cactus](#)” on page 9-17
- “[Running your Cactus JUnit test case](#)” on page 9-23

Running your Cactus JUnit test case

Running a Cactus JUnit test case in the JBuilder IDE is in many ways similar to running other types of JUnit tests. The same test runners are available and you interact with them in the same ways. The main difference between running a Cactus test and running any other JUnit test is that for a Cactus test, you need to start the server first. To run a Cactus test,

- 1 Start the server using the Server runtime configuration.
- 2 If necessary, deploy the archives you want to test to the server.
- 3 Right-click the Cactus test file in the project pane.
- 4 Select Run Test Using <test configuration> from the context menu. The test runs in the test runner that’s specified in your Test runtime configuration.

Note Changing the test case requires the WebApp or EAR file configured for Cactus to be redeployed.

See also

- “[Configuring your project for testing an EJB with Cactus](#)” on page 9-17
- “[Running tests](#)” in *Building Applications With JBuilder*

10

Deploying enterprise beans

The JBuilder WebLogic Edition provides support for WebLogic Servers only

Deploying an enterprise bean to an application server usually involves the following steps:

- 1 Creating a deployment descriptor XML-based file compliant with Sun's EJB 1.1 or 2.0 specification.

When you use JBuilder's EJB wizards to create your beans, the deployment descriptors are being created at the same time for you.

- 2 Editing the deployment descriptors, if necessary.

You can edit the deployment descriptors JBuilder creates using JBuilder's Deployment Descriptor editor.

- 3 Creating an EJB JAR file containing the deployment descriptor and all of the classes required to operate the enterprise bean (bean class, remote/local interface, home/local home interface, stubs and skeletons, primary key class if the enterprise bean is an entity bean, and any other associated classes).

When you compile your EJB module using the JBuilder development environment, the proper JAR file is created for you.

Note WebSphere 4.0 Advanced Edition only allows you to deploy an EAR group. For more information, see “[Creating an EAR file](#)” on page 10-6.

- 4 Deploying your EJB to an EJB container.

JBuilder has an Enterprise Deployment wizard that simplifies the deployment process for Borland enterprise beans. If an application server other than a Borland server is your target application server, choosing Tools | Enterprise Deployment displays a Deploy Settings dialog box that is specific to your server. Once you fill in those settings

to suit your needs and choose OK to close the dialog box, the enterprise bean is deployed as you specified.

You can also access server-specific deployment settings by right-clicking the EJB module node, selecting Properties, clicking the Deployment tab, and specifying your settings there. Then, to deploy your EJB module, right-click the EJB module, select Deploy Options For <jar name>.jar, and choose Deploy or Redeploy.

Creating a deployment descriptor file

As you create your enterprise beans using JBuilder's EJB tools, JBuilder is creating deployment descriptors at the same time. You can then use the Deployment Descriptor editor to add additional information and modify attributes in the deployment descriptors.

Each deployment descriptor that conforms to the EJB 1.1 or 2.0 specifications:

- Must be XML-based and conform to the rules of XML.
- Must be valid with respect to the DTD in the EJB 1.1 or 2.0 specification.
- Conforms to the semantics rules specified in the DTD.
- Refers to the DTD using one of the following statements:

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

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

When you use JBuilder's EJB tools to create and edit your deployment descriptors, you don't have to worry about learning XML or conforming to the semantics rules specified in Sun's DTD. The Deployment Descriptor editor imposes these rules on the data you enter and edit. As you fill in information using the Deployment Descriptor editor, it lets you know what data are required. JBuilder's tools automatically set up the Borland-specific extensions in an `ejb-inprise.xml` file for 1.1 deployment descriptors or `ejb-borland.xml` for 2.0 deployment descriptors. For more information about the Deployment Descriptor editor, see [Chapter 11, "Using the Deployment Descriptor editor."](#)

The role of the deployment descriptor

The role of the deployment descriptor is to provide information about each EJB that is to be bundled and deployed in a particular JAR file. It's intended to be used by the consumer of the EJB JAR file. As the bean developer, it's your responsibility to create the deployment descriptor.

The information in the deployment descriptor is used in setting enterprise bean attributes. These attributes define how the enterprise bean operates within a particular environment. For example, when you set the bean's transactional attributes, they define how the bean behaves with respect to transactions. The deployment descriptor keeps the following information:

- Type information, which defines the types, or names, of the classes for the home/local home and remote/local interfaces and the bean class.
- JNDI names, which set the name under which the home/local home interface of the enterprise bean is registered.
- Fields to enable container-managed persistence.
- Transactional policies that govern the transactional behavior of a bean.
- Security attributes that govern access to an enterprise bean.
- Borland-specific information, such as data source information used for connections to a database.

The types of information in the deployment descriptor

The information in the deployment descriptor can be divided into two basic kinds:

- Enterprise beans' structural information.

Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies. This information is required. The structural information usually can't be changed because doing so could break the bean's function.

- Application assembly information.

Application assembly information describes how the enterprise bean(s) included in the `ejb-jar.xml` file are composed into a larger application deployment unit. This information is optional. Assembly level information can be changed without breaking the bean's function, although doing so might alter the behavior of an assembled application.

Structural information

The bean developer must provide the following structural information for each bean in the EJB JAR file:

All enterprise beans

- Enterprise bean's name, a mnemonic used to refer to the bean in the deployment descriptor
- Enterprise bean's class
- Enterprise bean's type, either session, entity, or message-driven bean
- Environment entries, if the bean has configuration parameters
- Resource factory references
- EJB references, if an enterprise bean references another enterprise bean
- Security role references, if an enterprise bean needs to access specific roles
- Resource environment references, if the bean refers to an external resource

Session beans

- Session bean's home and/or local home interface
- Session bean's remote and/or local interface
- Session bean state management type, either stateful or stateless
- Session bean transaction demarcation type for stateful beans that have synchronization callbacks

Entity beans

- Entity bean's home and/or local home interface
- Entity bean's remote and/or local interface
- Entity bean's persistence management type
- Entity bean's primary key class
- Container-managed fields for container-managed beans and relationship information for EJB 2.0 components

Message-driven beans

- Message-driven bean's transaction management type
- Message-driven bean's destination and subscription durability

Application assembly information

You can specify any of the following application assembly information. During application assembly, this information is optional. This same information is not optional for the role of the deployer.

- Binding of enterprise bean references
- Security roles
- Method permissions
- Linking of security role references
- Security identity
- Transaction attributes

During the process of application assembly or deployment, you can modify the following structural information:

- The values of environment entries. The application assembler can change existing properties and/or define the values of environment properties.
- Description fields. The application assembler can change existing descriptions or create new description elements.

You can't modify any other types of structural information. You can modify any application assembly information at deployment time, however.

Security

The application assembler usually specifies the following information in the deployment descriptor:

- Security roles
- Method permissions
- Links between security role references and security roles
- Security identity

Security roles

Using the security role elements in the deployment descriptor, the developer can define one or more security roles. These define the required security roles for the clients of the enterprise beans.

Method permissions

Using the method-permission elements in the deployment descriptor, the developer can define method permissions. Method permissions are paired

relations between the security roles and the methods of the enterprise bean's remote/local and remote home/local home interfaces.

Linking of security role references

If security roles are defined, the developer must link them with security role references using the role-link element in the deployment descriptor.

Application server-specific properties

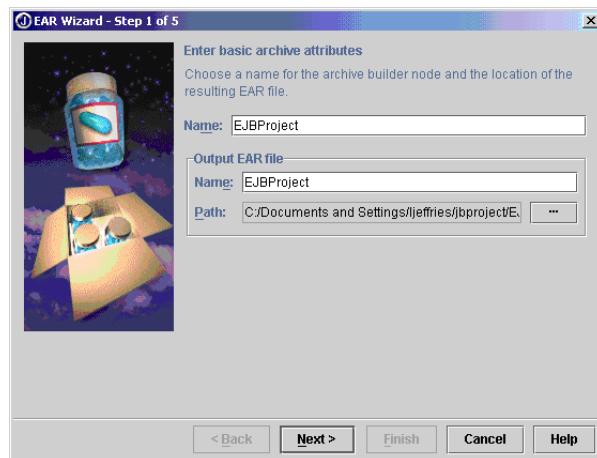
Deployment descriptors can also include properties that are specific to a particular application server.

Creating an EAR file

If you want to include your EJB JAR files in an EAR (Enterprise Archive) file, you can use JBuilder's EAR wizard to help you create the EAR.

Note You must create an EAR group to deploy your enterprise beans to WebSphere 4.0 Advanced Edition.

To access the EAR wizard, choose File | New, click the Enterprise tab, and double-click the EAR icon. The EAR wizard appears:



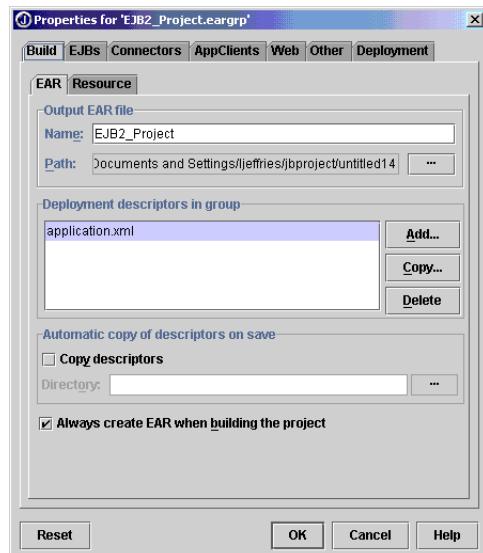
For information on using the EAR wizard, click the Help button.

When you finish using the wizard, it creates an .eargrp node in the project pane. Double-clicking this node displays an EAR DD Source tab in the content pane that displays the EAR's deployment descriptors.

Just as you can with an EJB module, you can modify the build properties of an .eargrp:

- 1 Right-click the .eargrp node in the project pane.
- 2 Choose Properties.

A Properties dialog box appears:



For information about using the Properties dialog box, click the Help button.

To create an EAR file from the .eargrp node, right-click it and choose Make. Now when you expand the .eargrp node, you'll see new EAR file in the project pane.

Deploying to an application server

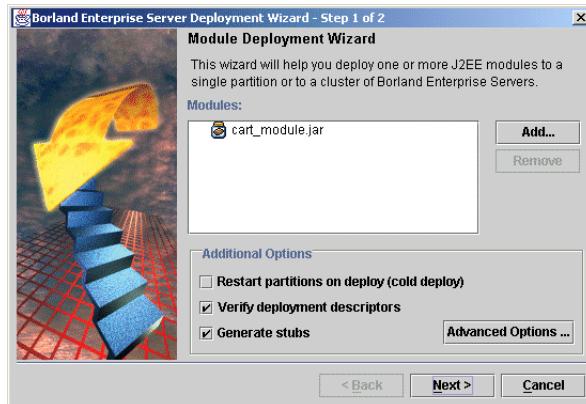
The JBuilder WebLogic Edition provides support for WebLogic Servers only

When your bean is working to your satisfaction and if you selected the Borland Enterprise Server as your application server for your current project, you can deploy your bean to the Borland Enterprise Server using the Borland's Enterprise Server Deployment wizard. If your target application server is from another vendor other than Borland, choosing Tools | Enterprise Deployment wizard displays a Deploy Settings dialog box you can use to deploy to those servers.

Deploying one or more JAR files

To deploy one or more J2EE modules to the Borland Enterprise Server 5.1.1 - 5.2.1,

- 1 Start a Management Agent if you have not already done so by choosing Tools | Borland Enterprise Server Management Agent. (This step is optional as the server startup process attempts to start the Enterprise Management Agent if it isn't already started.)
- 2 Choose Tools | Enterprise Deployment to display the Borland Enterprise Server Deployment wizard.

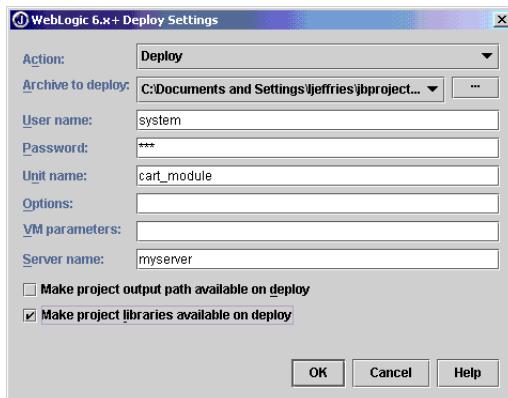


- 3 Click the Add button to navigate to the location of the J2EE modules (JAR, WAR, RAR, EAR, ZIP, and DAR files) you want to deploy and select them. Choose OK.
 - 4 If you want to verify that the deployment descriptors and the classes they reference are correctly formed before the modules are deployed, check the Verify Deployment Descriptors option.
 - 5 If you have not yet generated the stubs for your beans and want to do so, check the Generate Stubs option.
 - 6 The wizard has an Advanced Options button. It allows you to set additional Stub Generator and Verifier options. If you want to modify any of these options, click Advanced Options, and use the Advanced Options dialog box that appears to make your changes. Choose OK when you are done.
 - 7 Click Next to display the next page and use it to specify your deployment targets.
 - 8 Click Finish to close the wizard and begin the deployment process.
- The wizard attempts to deploy the J2EE module(s) and reports the results.

For more information about using the Borland Enterprise Server Deployment wizard, see the Borland Enterprise Server documentation.

Deploying to non-Borland servers

Developers for other servers, such as WebLogic, WebSphere, and iPlanet, can also deploy their EJBs using the Tools | Enterprise Deployment command. When one of these servers is the selected server for the current project, this command displays a Deploy Settings dialog box specific to the server. For example, here is the WebLogic Deploy 6.x+ Settings dialog box:



The deployment tool for WebSphere 4.0 differs depending on the version of the server you are using. For the Single Server, WebSphere's deployment tool is SEAppInstaller. For the Advanced Edition, the deployment tool is XmlConfig. Therefore, the appearance of the Deploy Settings dialog box will vary between these two WebSphere Server 4.0 editions.

In the iPlanet 6.x+ Deploy Settings dialog box, the List Deployments action works only when the Local Server option is the selected Target option even though the List Deployments action is still selectable in the Action drop-down list.

Fill in the fields you need and choose OK. For more information, click the Help button in the Deploy Settings dialog box.

Setting deployment options with the Properties dialog box

While you can use Tools | Enterprise Deployment to set options for deployment on the deployable nodes in your project, you can also use the Properties dialog box. These properties are saved for the specific node on which they are set. If you used the Enterprise Deployment dialog box

previously, the values you entered then become the default values for all deployable nodes in the project.

To set deployment options using the Properties dialog box,

- 1 Right-click the EJB module node, a child JAR of this node, or an EAR group to display the context menu. (You can also right-click an EAR or WAR node, if these are considered deployable for your selected server.)
- 2 Choose Properties to display the Properties dialog box. If you right-clicked the EJB module node, you must now click the Deployment tab of the Properties dialog box and then the page specific to your application server (such as the Borland Enterprise Server 5.1.1 - 5.2.1).
- 3 Set your options. The options available will vary depending your target application server. For example, for Borland Enterprise Server 5.1.1 - 5.2.1, you can set the host name, the container, and VM parameters. WebLogic users can set a unit name, deploy options, a password, and VM parameters. WebSphere users can set the primary node name, the application server name, the container name, VM parameters, and an option to generate XML. If multiple nodes are selected that have different deploy options, default values are used. Consult your application server's documentation for assistance in filling in these fields.
- 4 If your target application server is WebSphere 4.0 Advanced Edition and you want an XML file to be generated as input to the WebSphere XMLConfig utility, check the Generate XML check box. If this option isn't checked, the file won't be created. If you make your own modifications to the generated XML file (named `deploy_<selectednode>.xml` and appearing under the EJB module node or EAR group), uncheck this option to be sure you don't lose your changes. If you use the Deployment Options on the context menu (right-click the EJB module and choose Deployment Options <jar name>.jar to see the deployment commands), the generated XML file is `deploy.xml`. It appears under the project node.

Hot deploying to an application server

During your development cycle, you are likely to want to quickly deploy, redeploy, and undeploy your enterprise beans to an already running container. Right-click the EJB module node (or EAR or WAR node if it is considered deployable for your server) or its child nodes in the project pane and choose Deploy Options For <jar name>.jar to see a list of deployment commands:

- Deploy - Deploys a JAR to the currently running container of the project application server. If the Build Target option is Make for the current runtime configuration (Choose Run | Configurations, select the current runtime configuration and click Edit then click Server and

select Deployment in the tree of services to see the Build Target), this option will “make” the JAR’s contents before deploying it to the container.

- Redeploy - Deploys a JAR again to the currently running container. If the Build Target option is Make for the current runtime configuration (Choose Run | Configurations, select the current runtime configuration and click Edit then click Server and select Deployment in the tree of services to see the Build Target), this option will “make” the JAR’s contents before redeploying it to the container.
- Undeploy - Removes an already deployed JAR in the running container.
- List Deployments - Lists all JARs deployed in the running container.
- Stop Container - Stops the container. This option appears for WebSphere 4.0 Advanced Edition only. When a deployed EJB changes, the container must be stopped and then restarted for the changes to register.
- Start Container - Starts the container. This option appears for WebSphere 4.0 Advanced Edition only.
- Deploy Submodule - Deploys an individual submodule in an EAR. This option is available for WebLogic 7.0 and 8.1 only.
- Activate/Deactivate - This option is available for WebLogic 7.0 and 8.1 only. Deactivate suspends deployed components, leaving staged data in place for subsequent reactivation. Activate reactivates a deactivated component.

Note For WebSphere 4.0 Advanced Edition, you must right-click an EAR group or WAR node in the project pane instead of the EJB module to see the Deploy Options For <jar name>.jar menu option. The WebSphere Server 4.0 does not consider EJB JARs deployable.

For more information about deploying to specific servers, see these topics in *Developing J2EE Applications*:

- “Remote deploying” in “Using JBuilder with Borland servers.”
- “Remote deploying” in “Using JBuilder with BEA WebLogic servers.”
- “Deploying” in “Using JBuilder with IBM WebSphere servers.”
- “Deploying” in “Using JBuilder with Sybase servers.”
- “Remote deploying” in “Using JBuilder with iPlanet servers.”

11

Using the Deployment Descriptor editor

The JBuilder WebLogic Edition provides support for WebLogic Servers only

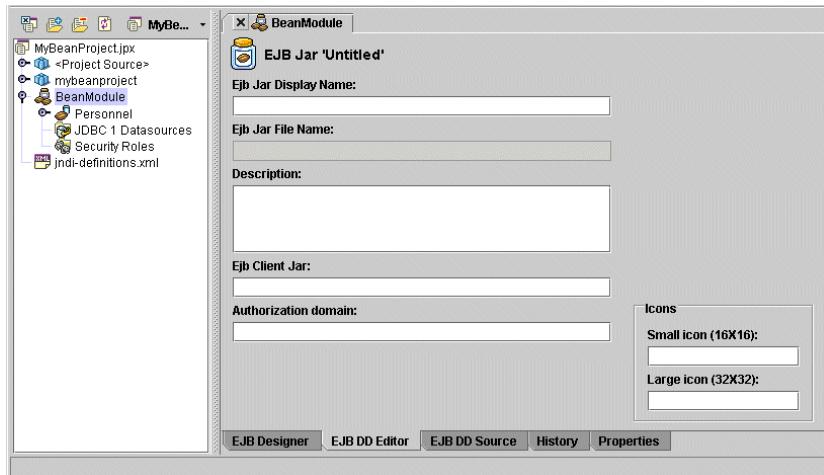
JBuilder includes a Deployment Descriptor editor you can use to change deployment information (such as transaction policies and security roles in an EJB deployment descriptor file). You can also alter the method of persisting an enterprise bean. For general information about deployment descriptors, see [Chapter 10, “Deploying enterprise beans.”](#)

You can also view and edit some of the properties specific to other application servers. For information, see [“Server-specific Properties panel” on page 11-20](#). There are also several WebLogic-specific panels in the Deployment Descriptor editor.

Displaying the Deployment Descriptor editor

To display the Deployment Descriptor editor, double-click the EJB module in the project pane and click the EJB DD Editor tab at the bottom of the content pane.

Viewing the deployment descriptor of an enterprise bean

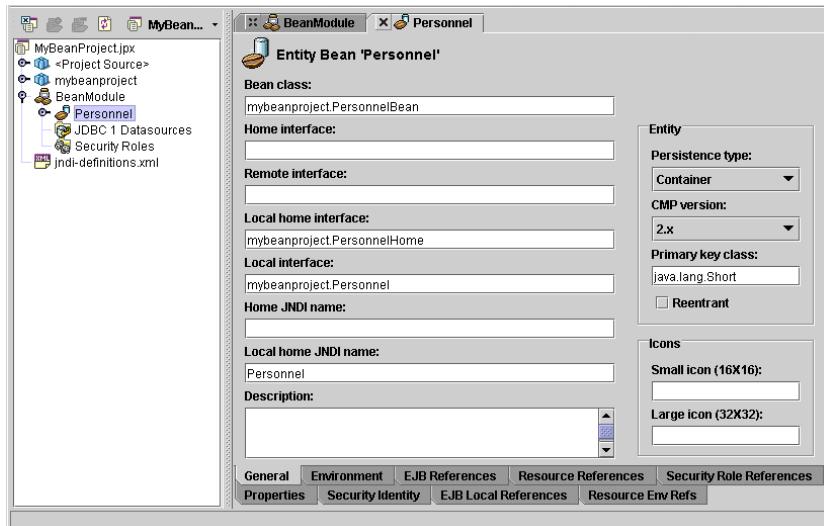


Viewing the deployment descriptor of an enterprise bean

To view information about an enterprise bean in the Deployment Descriptor editor,

- 1 Open the EJB module node (by clicking the icon to the far left of the EJB module node).
- 2 Double-click the bean in the project pane.

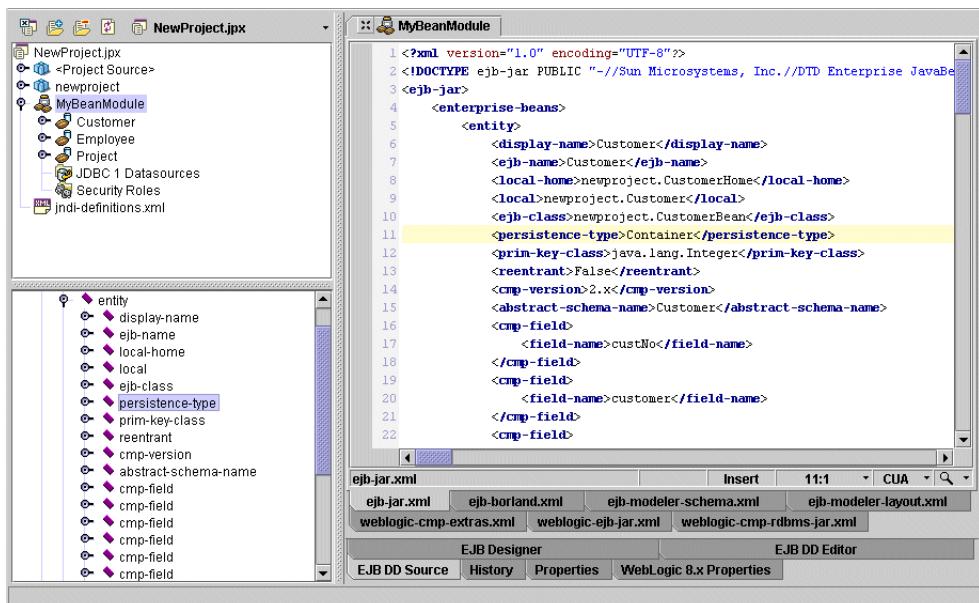
The Deployment Descriptor editor appears with the General panel selected.



The Deployment Descriptor editor has many other tabs. You can click any of these tabs at the bottom of the Deployment Descriptor editor to view other panels. Use the editor to make any changes you want to the deployment information for the bean.

You can view additional information about a deployment descriptor by opening a bean node in the project pane. By double-clicking these nodes, you can see additional panels. For example, double-clicking the Container Transactions node for a bean displays a Container Transactions panel in the Deployment Descriptor editor. The JDBC DataSources and Security Roles nodes can also be opened, if they contain data. The subnodes that appear can be used to display more information.

To view the source code of each descriptor, double-click the EJB module node in the project pane, then click the EJB DD Source tab at the bottom of the Deployment Descriptor editor. For each deployment descriptor in the EJB module, a tab appears with the name of the file on the tab. Select the tab of the file you want to view. While viewing the source code of a deployment descriptor, you can click on elements in the structure pane to move a highlight bar to the corresponding element in the source code. You can edit the source code directly.

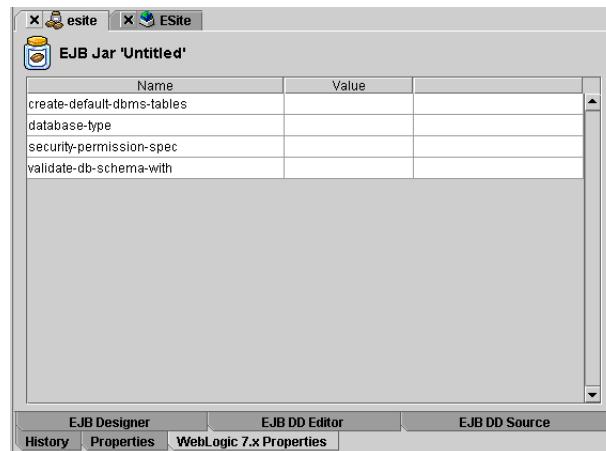


Viewing an EJB module-level WebLogic 6.x, 7.x, or 8.x Properties page

If your selected application server for your current project is WebLogic Server 6.x or later, the Deployment Descriptor editor displays an

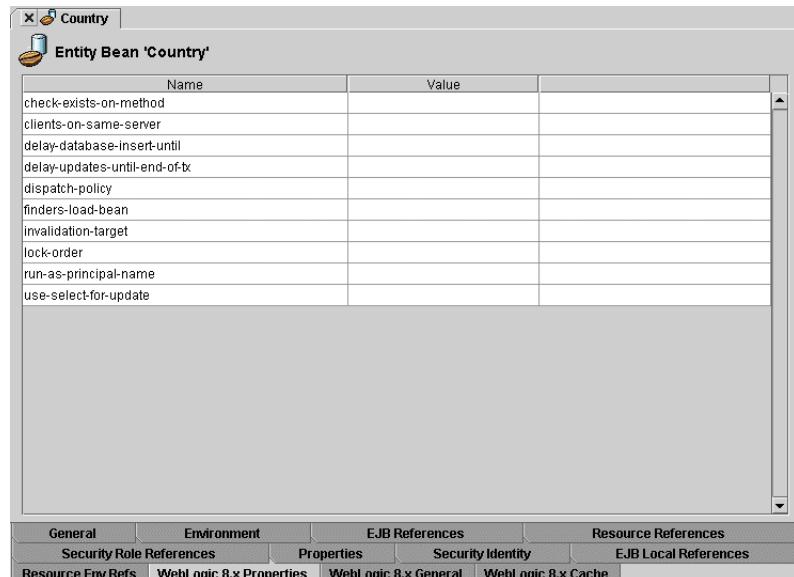
Viewing the deployment descriptor of an enterprise bean

additional page when the EJB module is the current node: the WebLogic 6.x, 7.x, or 8.x Properties page.



Use this page to set properties that are unique to the WebLogic server version configured for your project. Set the create-default-dbms-tables, database-type, and validate-db-schema-with properties by selecting your choice from the drop-down lists provided. For the security-permission-spec, type in the value and press *Enter*.

The page for WebLogic Platform 8.1 differs:



For information about these properties, see your WebLogic documentation.

Changing bean information

To change bean information in a deployment descriptor,

- 1 Open or expand the EJB module node in the project pane to see the beans contained in it.
- 2 Double-click the bean you are interested in.

The Deployment Descriptor editor appears with the General panel selected.

- 3 Use the tabs at the bottom to display the panel you want to use to modify the existing information about the new bean.
- 4 Enter your changes.

If you're comfortable working in XML and with deployment descriptor source code, you can select the EJB DD Source tab and make your changes directly in the source code.

Caution Although it's possible for you to edit properties directly in the deployment descriptor source code itself using the DD Source panel, you should always use the server-specific Properties panel instead to set properties that are unique to servers other than Borland's. If you don't, these changes will be overwritten when you change other property values using the Deployment Descriptor editor.

Caution All vendor-specific information is stored in `ejb-borland.xml` (for all servers). If you edit the deployment descriptor source directly, always make your changes to this file to be sure your changes remain.

Enterprise bean information

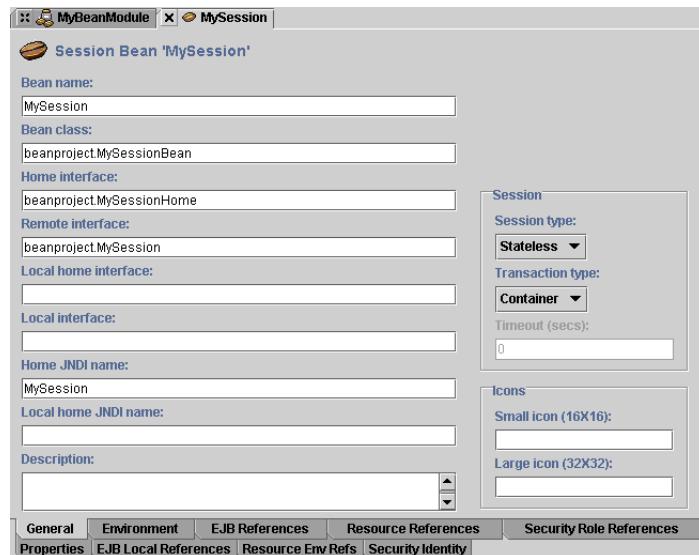
This section describes the type of information you can create and store for enterprise beans in the deployment descriptor.

General panel

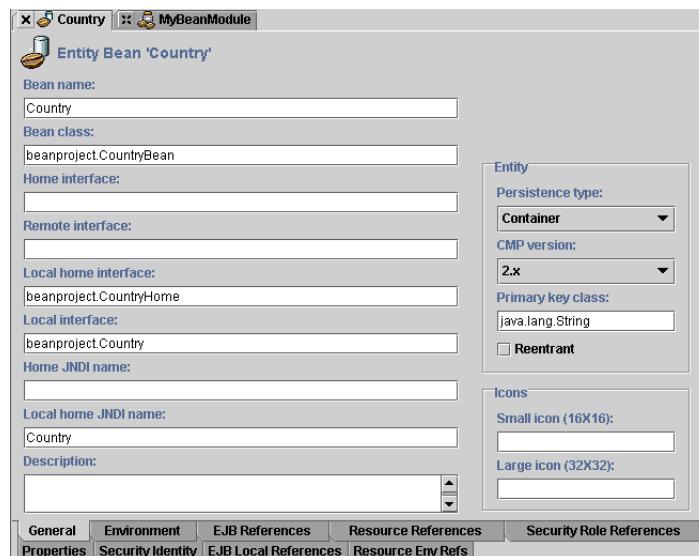
Use the General panel to enter or change general information about the enterprise bean.

Enterprise bean information

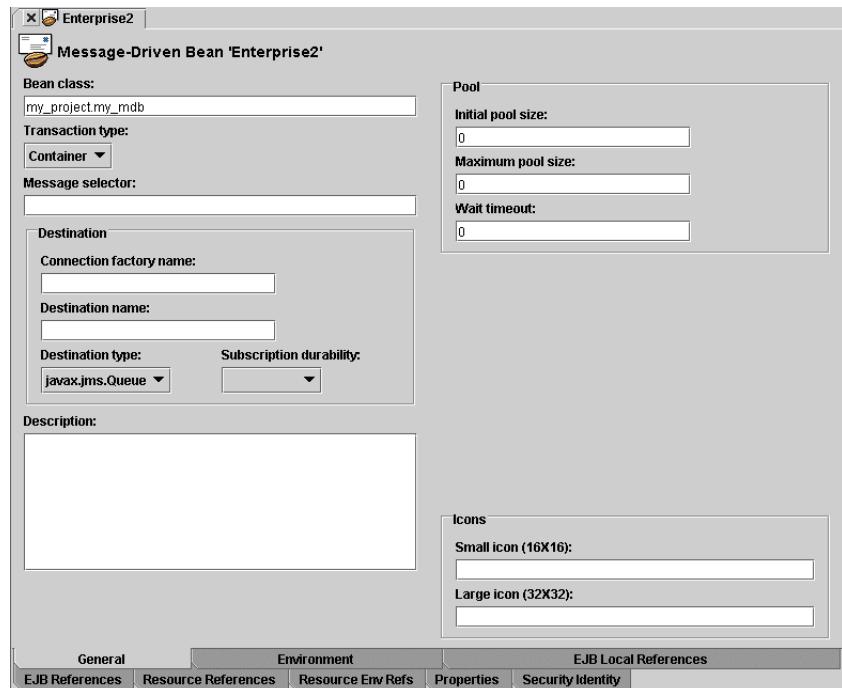
This is the General panel for a session bean:



This is the General panel for an entity bean:



This is the General panel for a message-driven bean:



The General panel includes this information:

- **Bean Class:** The fully-qualified name of the Java class that implements the bean's business methods. This information must be specified.
- **Transaction Type:** Specifies the bean's transaction management type. Choose Bean if the bean manages its own transactions; choose Container if the container manages the transaction handling.
- **Acknowledge Mode:** This field appears only if the Transaction Type value is Bean. The Auto-acknowledge option means that all messages the bean receives are acknowledged and a check is performed to prevent acting on duplicate messages. The Dups-ok-acknowledge means that all messages are acknowledged, including duplicate messages if they should occur.

- **Message Selector:** The message selector that determines which messages the message-driven bean should receive. Here is an example:

```
JMSType = 'chair' AND color = 'black' AND fabric = 'leather'
```

See the JMS specification on Sun Microsystems' web site at
<http://java.sun.com/products/jms/docs.html> for more information.

- **Connection Factory Name:** The JNDI name of the connection factory that is used to establish a connection to the message broker.
- **Destination Name:** The JNDI name of the queue or topic to which the message-driven bean listens. This is the JMS destination from which the message-driven bean instance consumes messages.
- **Destination Type:** Indicates whether a message-driven bean is intended for a Queue or a Topic. You can also select Not Specified. If you select Topic, the Subscription Durability field appears.
- **Subscription Durability:** Indicates whether the bean's subscription to a topic is durable or nondurable. This field is available only if you selected Topic as the value of the Message Driven Destination field.
- **Initial Pool Size:** The initial number of message-driven bean instances the container should create immediately after deployment.
- **Maximum Pool Size:** The maximum number of message-driven bean instances that can be created and kept in the message-driven bean instance pool.
- **Wait Timeout:** The amount of time in seconds a message can be queued while waiting for a message-driven bean instance to handle it. For example, you might limit your pool of message-driven bean instances to five for performance reasons. If the queue then becomes overloaded, you might want to set a limit on how long the message waits for service, especially in the context of a transaction in which you want to prevent a transaction from seeming to hang.

Environment panel

The Environment panel lists all the enterprise bean's environment entries. Environment entries allow you to customize the bean's business logic when the bean is assembled or deployed. The environment allows you to customize the bean without accessing or changing the bean's source code.

Each enterprise bean defines its own set of environment entries. All instances of an enterprise bean share the same environment entries.

Enterprise bean instances aren't allowed to modify the bean's environment at runtime.

Property	Value	Type
maxSalary	93600	java.lang.String

The 'Type' column dropdown menu is open, showing the following options:

- java.lang.String
- java.lang.Character
- java.lang.Integer
- java.lang.Double
- java.lang.Byte
- java.lang.Short
- java.lang.Long
- java.lang.Float

Below the table are buttons: Add, Remove, and Remove All. At the bottom are tabs: General, Properties, Environment, EJB References, Resource References, Security Role References, Security Identity, EJB Local References, Resource Env Refs.

To add an environment entry,

1 Click Add to create a new entry.

A new, blank row appears.

2 Enter a property in the Property column and a property value in the Value column.

3 Choose a property type from the Type drop-down list.

4 If you want to do so, enter a description of the added environment property in the Description field.

5 Continue to add environment entries as you desire.

To remove an environment entry,

1 Select the row.

2 Click the Remove button.

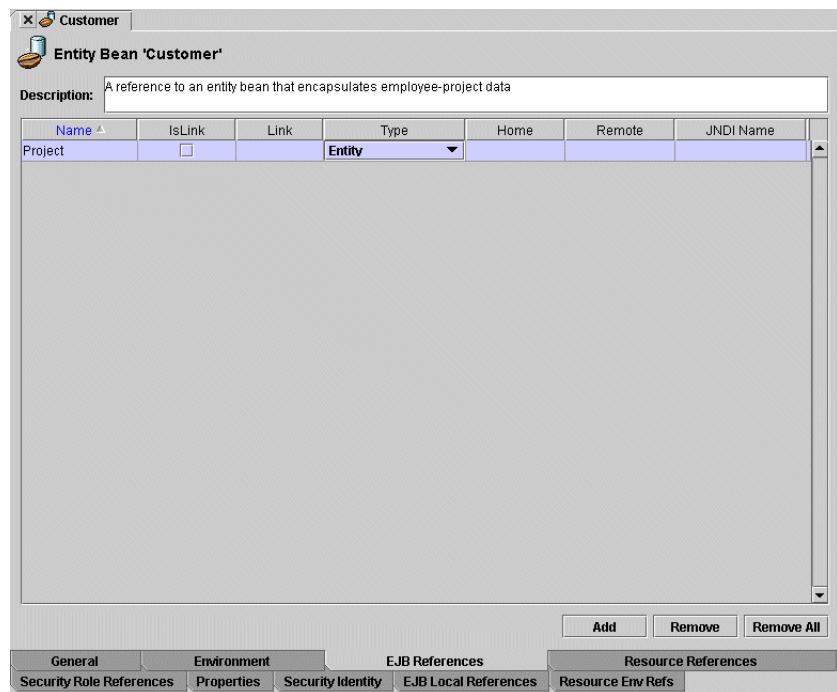
These are some things to keep in mind about the environment entries:

- The bean provider must declare all the environment entries accessed from the enterprise bean's code.
- If the bean provider includes a value for the environment entry, the value can be later changed during the assembly or deployment.

- The assembler can modify the values of the environment entries set by the bean provider.
- The deployer must ensure that the values of all environment entries are set to meaningful values.

EJB References panel

The EJB References panel lists all the enterprise bean references to the homes of other enterprise beans the bean requires. Use this panel for beans that reference remote beans only. For EJB 2.0 beans that reference local beans, use the EJB Local References panel.



Each EJB reference describes the interface requirements that the referencing enterprise bean has for the referenced bean. You can define references between beans within the same JAR file or from an external enterprise bean (one that is outside the JAR file), such as a session bean to an entity bean.

To add an EJB reference,

- 1 Click Add.
- 2 In the dialog box that appears, enter a name for the EJB reference and choose OK.

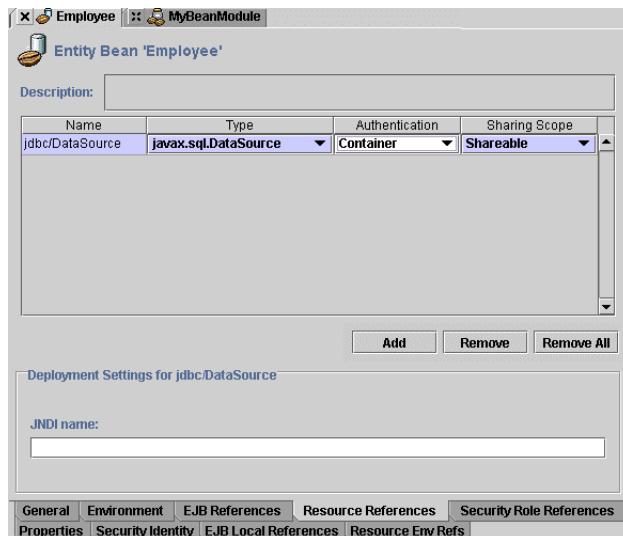
A new row is added to the panel.
- 3 Fill in the fields in the row with the following information:
 - **Description:** A brief description of the bean that is referenced. This information is optional. After you enter the Description, click the row or the description you entered will be lost.
 - **Name:** The name of the referenced bean.
 - **IsLink:** When IsLink is checked, the reference is to a bean within the JAR, so the JNDI Name value isn't relevant. If IsLink isn't checked, the JNDI name is used to find the bean. When you check this option, you then select the bean that is referenced from the Link drop-down list.
 - **Link:** Links the EJB reference to the target enterprise bean. The Link value is the name of the target bean. This information is optional.
 - **Type:** The expected type of the referenced bean.
 - **Home:** The expected Java type of the referenced bean's home interface. For an EJB 2.0 component, this field refers to the remote home interface.
 - **Remote:** The expected Java type of the referenced bean's remote interface.
 - **JNDI Name:** The JNDI name of the referenced bean.

These are important points to remember about EJB references:

- The target enterprise bean must be type-compatible with the declared EJB reference.
- All declared EJB references must be bound to the homes of enterprise beans that exist in the operating environment.
- If a Link value is specified, the enterprise bean reference must be bound to the home of the target enterprise bean.

Resource References panel

The Resource References panel lists all the enterprise bean's resource factory references. This enables the application assembler and/or the bean deployer to locate all references used by the enterprise bean.

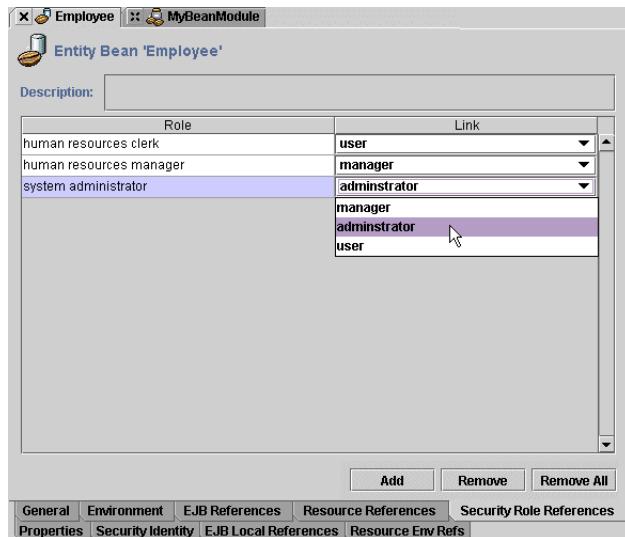


To add a resource reference, click the Add button and fill in the following fields:

- **Description:** A description of the resource reference. This information is optional. After you enter the Description, click the row or the description you entered will be lost.
- **Name:** The name of the environment entry used in the enterprise bean's code.
- **Type:** The Java type of the resource factory expected by the enterprise bean's code. (This is the Java type of the resource *factory*, not the Java type of the resource.)
- **Authentication:** An Application authentication indicates that the enterprise bean performs the resource sign-on programmatically. A Container authentication indicates that the container signs on to the resource based on the principal mapping information supplied by the deployer.
- **Sharing Scope:** Determines whether the resource can be shared. Your options are Shareable and Unshareable. This field is available for EJB 2.0 components only.
- **JNDI Name:** JNDI name for the resource reference.

Security Role References panel

The Security Role References panel lists all the enterprise bean's references to security roles. The panel links security role references used by the bean developer to specific security roles defined by the application assembler or deployer.



Before you can add a security role reference, one or more security roles must be already defined or the Add button on this panel is disabled. For information about creating and assigning security roles for application deployment, see [“Adding security roles and method permissions” on page 11-31](#).

To add a role, click the Add button and fill in the three fields:

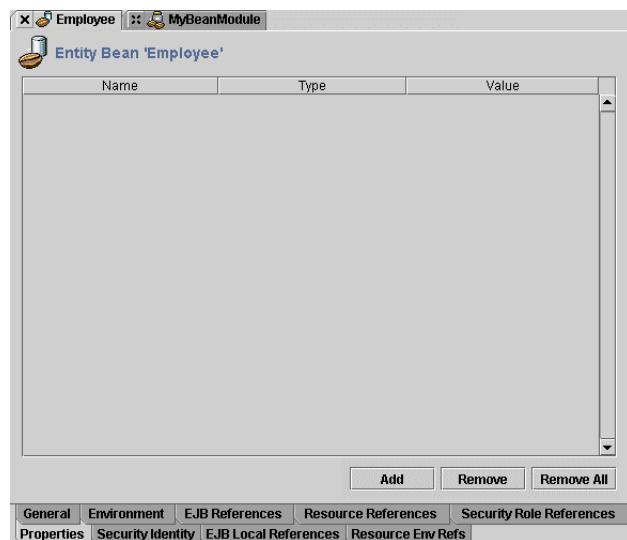
- **Description:** This is an optional field that describes the security role. After you enter the Description, click the row or the description you entered will be lost.
- **Role:** This is the name of the security role specified by the bean developer.
- **Link:** This is the name of the security role used when the application is deployed. Usually, this role is defined by the application assembler or deployer to work in a specific operating environment.

Properties panel

When you double-click an enterprise bean in the project pane and click the Properties tab in the Deployment Descriptor editor, the

Enterprise bean information

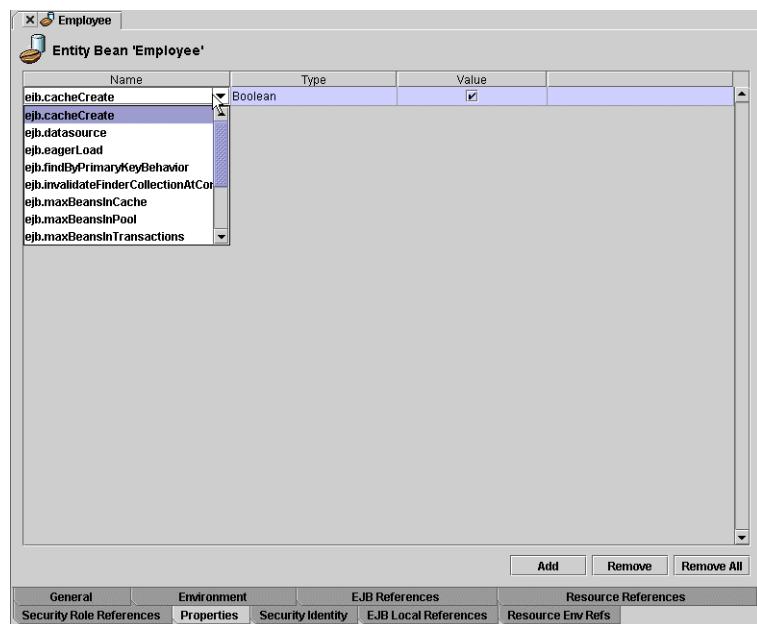
following Properties pane appears:



Use this panel only if your target application server is a Borland server.

To add a property to an enterprise bean selected in the project pane,

- 1 Click the Add button to add a row to the panel.
- 2 From the Name drop-down list, select the property you want to add.



3 Specify a value in the Value field.

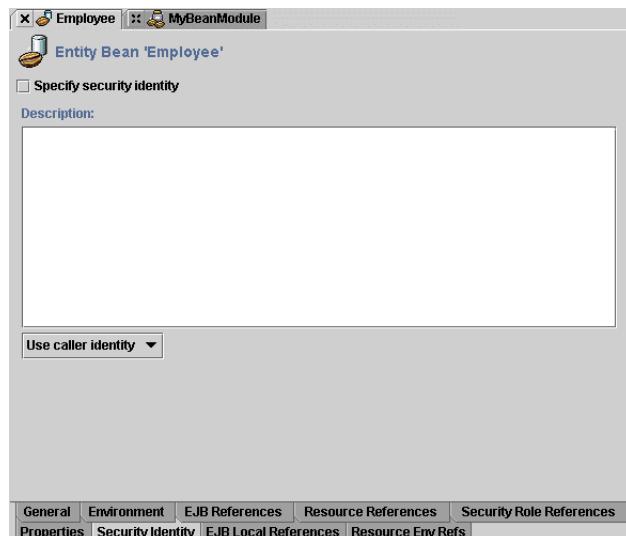
You specify some values by selecting a value from a drop-down list, others require you to enter an appropriate value such as a string or integer value, and one presents a check box for Boolean values; checking the check box indicates the value is true.

For information about these properties, see your Borland Enterprise Server documentation.

Security Identity panel

The Security Identify panel allows you to specify whether you want a security identify to be used when the methods of the bean are executed.

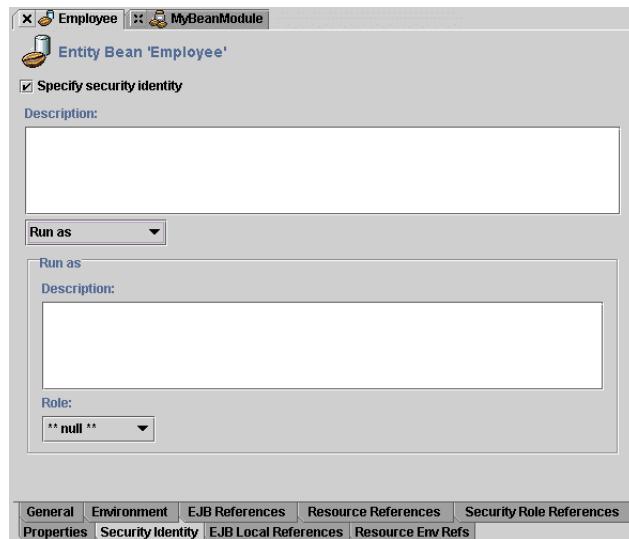
A Security Identity panel is available for an EJB 2.0 component only.



To specify that a security identify be used when methods of the bean are executed, check the Specify Security Identity check box and follow these steps:

- 1** If you choose to do so, enter a description of the security identity in the Description field.
- 2** Specify whether the caller's security identity should be used or whether a specific run-as security identity should be used with the drop-down

list below the Description field. If you select the Run As option, the panel changes so that it look like this:

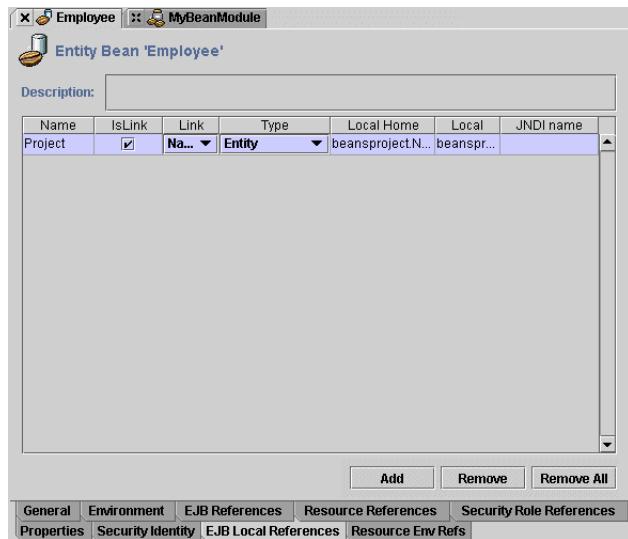


- 3 Fill in the Description field if you choose; it's not required.
- 4 From the Role drop-down list, select a role. The list displays all security roles that are defined for the current EJB module. For more information about creating security roles, see the ["Adding security roles and method permissions" on page 11-31](#).

EJB Local References panel

Each EJB local reference describes the interface requirements that the referencing enterprise bean has for the local referenced bean. Use the EJB Local References panel to define EJB local references only. To specify a reference to a remote bean, use the EJB References panel instead.

An EJB Local References panel is available for an EJB 2.0 component only.



To add a local EJB reference,

- 1 Click Add.
- 2 In the dialog box that appears, enter a name for the EJB local reference and choose OK.
A new row is added to the panel.
- 3 Fill in the fields in the row with the following information:
 - **Description:** A brief description of the bean that is referenced. This information is optional. After you enter the Description, click the row or the description you entered will be lost.
 - **Name:** The name of the referenced bean.
 - **IsLink:** When IsLink is checked, the reference is to a bean within the JAR, so the JNDI Name value isn't relevant. If IsLink isn't checked, the JNDI name is used to find the bean. When you check this option, you then select the bean that is referenced from the Link drop-down list.
 - **Link:** Links the EJB reference to the target enterprise bean. The Link value is the name of the target bean. This information is optional.

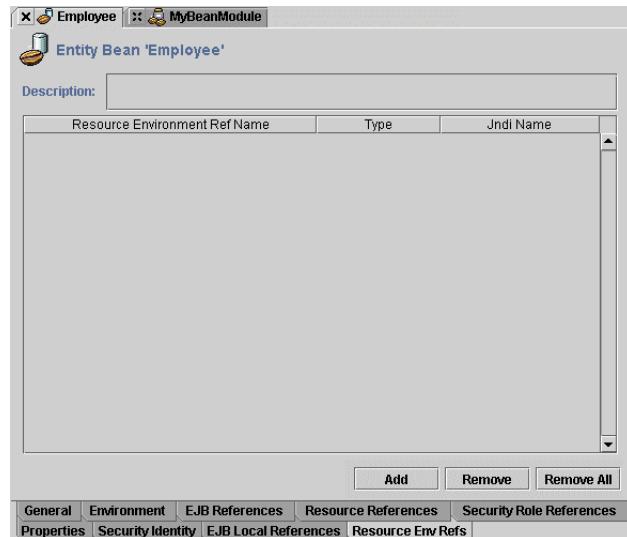
- **Type:** The expected type of the referenced bean.
- **Local Home:** The expected Java type of the referenced bean's local home interface.
- **Local:** The expected Java type of the referenced bean's local interface.
- **JNDI Name:** The JNDI name of the referenced bean.

These are important points to remember about EJB local references:

- The target enterprise bean must be type-compatible with the declared EJB local reference.
- All declared EJB references must be bound to the local homes of enterprise beans that exist in the operating environment.
- If a Link value is specified, the enterprise bean reference must be bound to the local home of the target enterprise bean.

Resource Env Refs panel

The Resource Env Refs panel let's you specify a reference to an external resource. It is available only for EJB 2.0 components. This is how the Resource Env Refs panel appears:

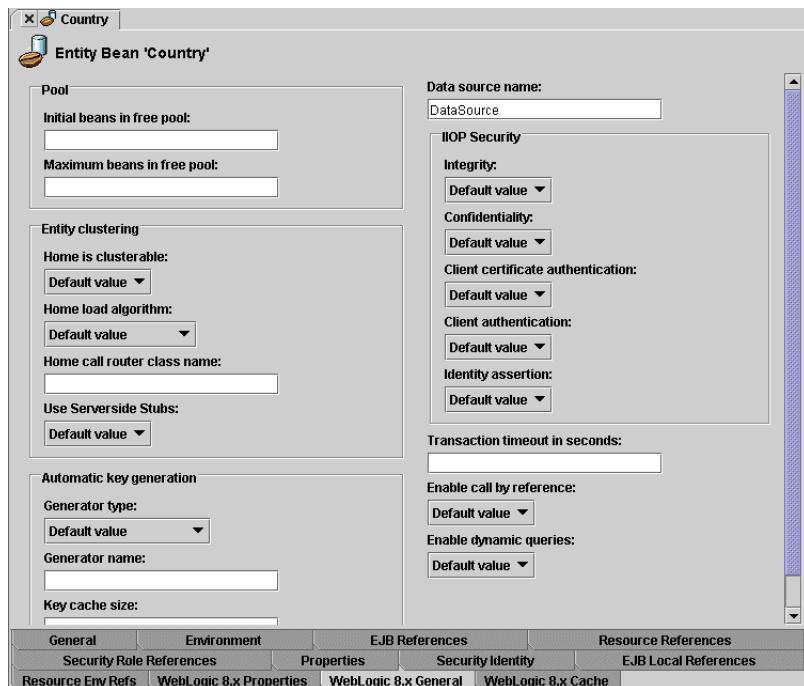


To add a new resource environment reference,

- 1 Click the Add button.
- 2 In the new row that was added, click in the Resource Environment Ref Name column.
- 3 Add a description of the reference in the Description field if you choose; it's not required.
- 4 Type in the name of your resource environment reference in the Resource Environment Ref Name column. The name you specify must be the unique within your enterprise bean.
- 5 Specify the resource environment reference type. It must be the fully-qualified name of a Java class or interface.
- 6 Specify the resource environment reference JNDI name.

WebLogic 6.x, 7.x, 8.x General panel

If a WebLogic 6.x, 7.x, or 8.x server is your targeted application server, a WebLogic General panel appears:

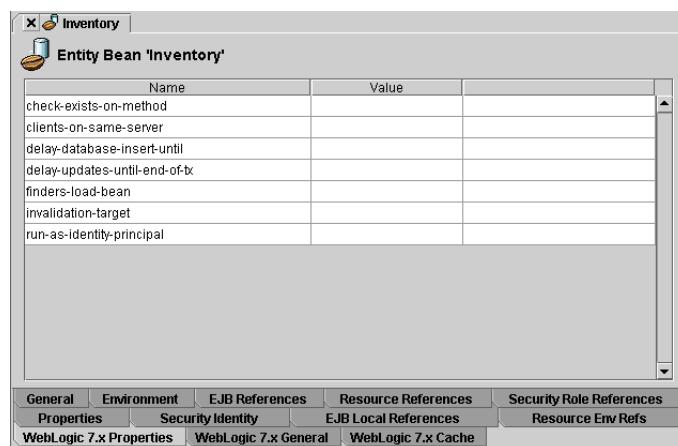


Which fields appear on this panel depend on the type of bean and which WebLogic Server is the target. For example, the fields shown above appear for an entity bean when WebLogic 8.x is the target application server.

For information about meaning of these fields, see your WebLogic documentation.

Server-specific Properties panel

If your target application server is some other than a Borland or Sybase server, a server-specific Properties panel is available for a selected bean. For example, here you see a tab at the bottom of the content pane for WebLogic 7.x Properties:



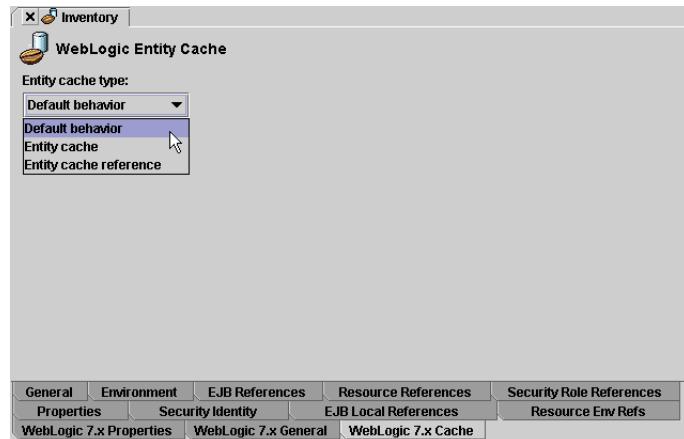
Use this panel if you want to view and edit some of the vendor-specific elements that are unique to your target application server. The panel displays a table of properties specific to the server. In the right column, enter values for properties you want to modify. The property values are stored in the server-specific deployment descriptors.

Important

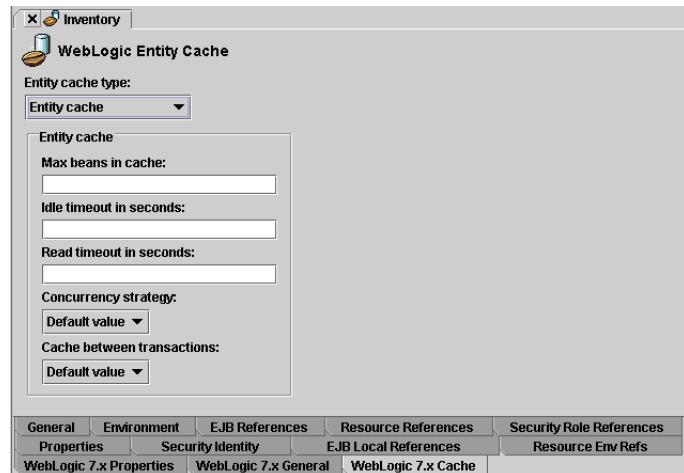
Although it's possible for you to edit properties directly in the deployment descriptor source code itself using the DD Source panel, you should always use the server-specific Properties panel to set the server-specific property values. If you don't, it's quite possible these values will be overwritten when you change other property values using the Deployment Descriptor editor.

WebLogic 6.x, 7.x, or 8.x Cache panel

If your target application server is a WebLogic 6.0 or later server and the selected bean is an entity bean, a WebLogic 6.x, 7.x, or 8.x Cache panel is available:



Specify the Entity Cache Type by selecting one of the values in the drop-down list. If you select either Entity Cache or entity Cache Reference, additional fields appear. For example, this shows how the panel appears when Entity Cache is selected:



For more information about entity caches, see your WebLogic documentation.

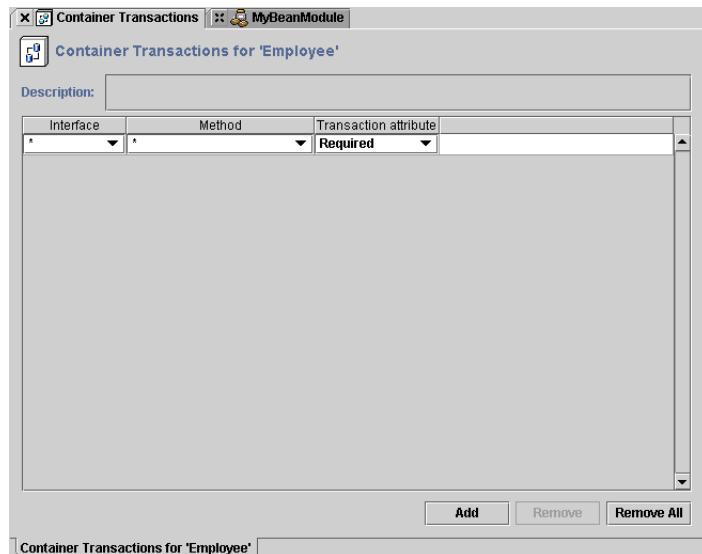
Container transactions

Enterprise beans that use container-managed transactions must have the transaction policies set by the container. The Deployment Descriptor editor enables you to set container-managed transaction policies and then associate these policies with methods in the enterprise bean's remote home, local home, remote, and local interfaces.

Setting container transaction policies

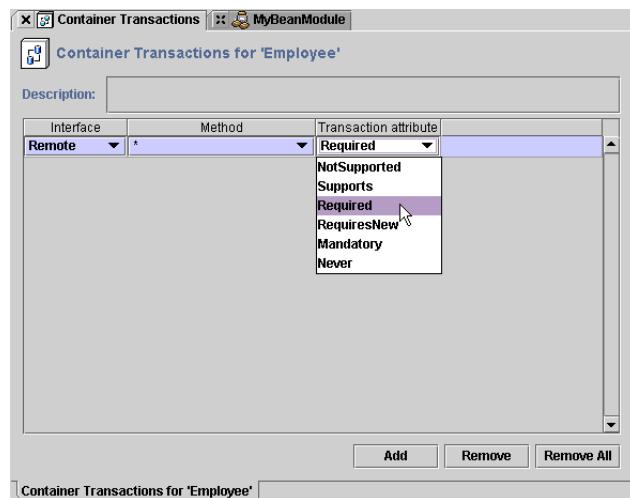
To set a container transaction policy for one or more methods,

- 1 Double-click the enterprise bean in the project pane to expand the bean's node.
- 2 Double-click Container Transactions in the project pane.



- 3 Click the Add button to add a row to the grid.
- 4 In the row, select the Interface that exposes the method or select * to indicate all interfaces. Your selection in this field determines which choices you will have in the Methods field.
- 5 From the drop-down list of Methods available, select the method you are setting the transaction policy for, or select * to select all the methods in the interface(s) you specified in the Interface field.

- 6 From the drop-down list of Transaction Attributes, select the attribute you want the transaction to have:



Enterprise beans that use container-managed transaction have transaction attributes associated with each method of the bean or with the bean as a whole. The attribute value tells the container how it must manage the transactions that involve the bean. There are several different transaction attributes that the application assembler or deployer can associate with each method of a bean:

- Required

Guarantees that the work performed by the associated method is within a global transaction context. If the caller already has a transaction context, the container uses the same context. If the caller doesn't have a transaction context, the container begins a new transaction automatically. Using this attribute makes it easy to compose multiple beans and coordinate the work of all the beans using the same global transaction.

- RequiresNew

Used when you don't want the method associated with an existing transaction. It ensures that the container always begins a new transaction.

- Supports

Permits the method to avoid using a global transaction. Use this attribute only when a bean's method accesses just one transaction resource or no transaction resources, and the method doesn't invoke another enterprise bean. Because this attribute avoids the cost associated with global transactions, using it optimizes your bean. If this attribute is set and a global transaction already exists, the

container invokes the method and it joins the existing global transaction. If there is no global transaction, the container starts a local transaction for the method and the transaction completes at the end of the method.

- **NotSupported**

Also permits the bean to avoid using a global transaction. If a client calls the method with a transaction context, the container suspends it. At the end of the method, the global transaction resumes.

- **Mandatory**

The client that calls a method with this transaction attribute must already have an associated transaction. If it doesn't, the container throws a `javax.transaction.TransactionRequiredException`. Using this attribute makes the bean less flexible for composition because it makes assumptions about the caller's transaction.

- **Never**

The client that calls a method with this transaction attribute must not have a transaction context. If it does, the container throws an exception.

- 7 Enter a description in the Description field to describe the transaction. Adding a description is optional.

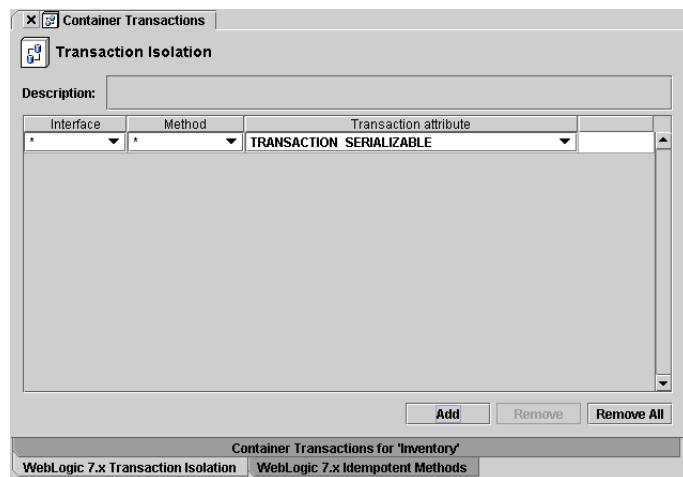
WebLogic 6.x, 7.x, or 8.x Transaction Isolation panel

If your target application server is a WebLogic Server 6.x, 7.x, or 8.x, a WebLogic Transaction Isolation panel is available to you to set the transaction isolation policy for methods.

To set transaction isolation policy for methods in your bean,

- 1 Double-click the enterprise bean in the project pane to expand the bean's node.
- 2 Double-click Container Transactions in the project pane.
- 3 Click the WebLogic 6.x, 7.x, or 8.x Transaction Isolation tab in the Deployment Descriptor editor.

- 4 Click the Add button to add a row to the grid.



- 5 In the row, select the Interface that exposes the method: the home, local home, remote, or local. Or select * to select all interfaces. Your selection in this field determines which choices you will have in the Methods field.
- 6 From the drop-down list of Methods available, select the method you are setting the transaction isolation policy for, or select * to select all the methods in the interface(s) you specified in the Interface field.
- 7 From the drop-down list, select the transaction attribute that describes the isolation policy you want for the methods you specified. The information found in ["Setting isolation levels" on page 11-28](#) may help you make your selection.
- 8 Enter a description in the Description field to describe the transaction. Adding a description is optional.

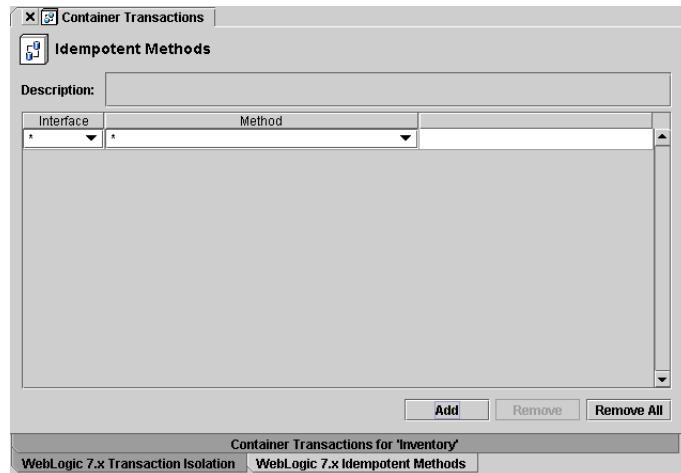
Refer to your WebLogic documentation for more assistance on the setting of isolation levels.

WebLogic 6.x, 7.x, or 8.x Idempotent Methods panel

If your target application server is a WebLogic 6.x, 7.x, or 8.x server, a WebLogic Idempotent Methods panel is available to you to specify which methods are to be treated as being idempotent.

To specify idempotent methods in your bean,

- 1 Double-click the enterprise bean in the project pane to expand the bean's node.
- 2 Double-click Container Transactions in the project pane.
- 3 Click the WebLogic 6.x, 7.x, or 8.x Idempotent tab in the Deployment Descriptor editor.
- 4 Click the Add button to add a row to the grid.



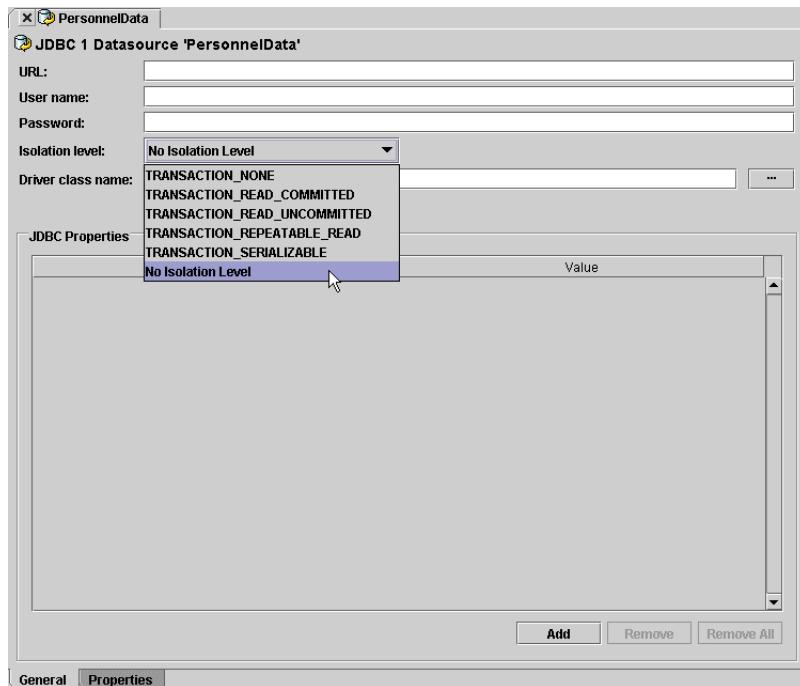
- 5 In the row, select the Interface that exposes the method: home or remote. Or select * to select all interfaces. Your selection in this field determines which choices you will have in the Methods field.
- 6 From the drop-down list of Methods available, select the method you specifying as idempotent, or select * to select all the methods in the interface(s) you specified in the Interface field.
- 7 Enter a description in the Description field to describe the transaction. Adding a description is optional.

Refer to your WebLogic documentation for information about idempotent methods.

Working with JDBC 1 data sources

To view information on a JDBC 1 data source in your deployment descriptor, expand the DataSources node in the project pane and double-click one of the data sources. (No DataSource nodes are available until you

add a new data source.) The Deployment Descriptor editor displays the General panel. You can use the panel to modify the information on the selected data source.



The Deployment Descriptor editor enables you to specify a new data source for entity beans and to set the isolation level for the data transactions.

To add a new data source to the deployment descriptor,

- 1 Right-click the JDBC node in the project pane and choose New Datasource on the menu that appears.
A New DataSource dialog box appears.
- 2 Enter the name of new data source and choose OK.
The new data source is added to the tree in the project pane.
- 3 Double-click the new data source in the project pane.

4 Enter the information for the new data source.

The data source is defined by a data source name, the URL location of the data source, and (if required) a user name and password to access the source. The panel also includes the class name of the JDBC driver and JDBC properties.

5 When you've specified the data source connection, you can choose the Test Connection button.

The Deployment Descriptor editor attempts to make the connection with the specified data source. The results are posted in the message log.

Setting isolation levels

The term *isolation level* refers to the degree to which multiple, interleaved transactions are prevented from interfering with each other in a multi-user database. These are possible transaction violations:

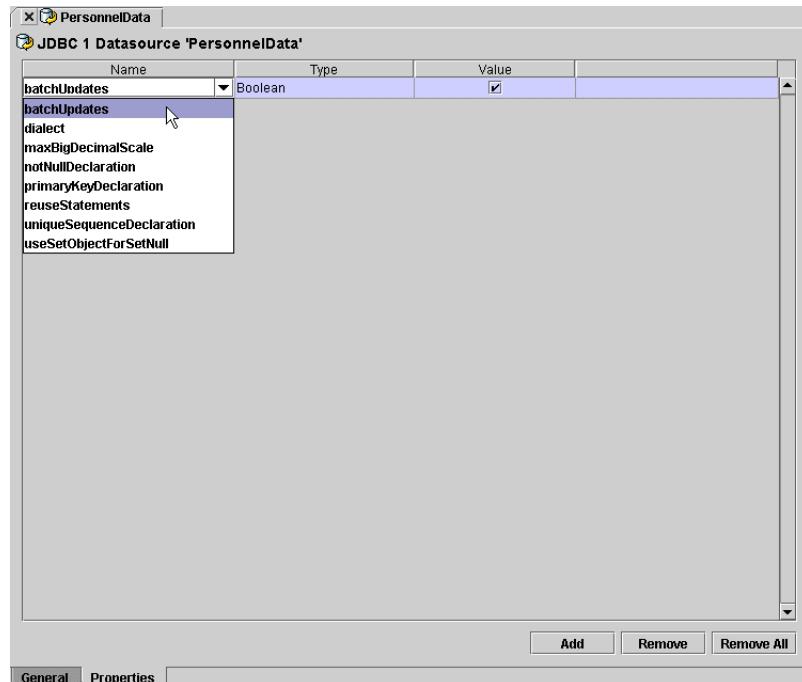
- **Dirty read:** Transaction t1 modifies a row; Transaction t2 then reads the row. Now t1 performs a rollback and t2 has seen a row that never really existed.
- **Non-repeatable read:** Transaction t1 retrieves a row. Then transaction t2 updates this row and t1 retrieves the same row again. Transaction t1 has now retrieved the same row twice and seen two different values for it.
- **Phantoms:** Transaction t1 reads a set of rows that satisfy certain search conditions. Then transaction t2 inserts one or more rows that satisfy the same search condition. If transaction t1 repeats the read, it will see rows that did not exist previously. These rows are called phantoms.

To set or change the transaction isolation level for a data source, choose one of these isolation levels from the Isolation Level drop-down list:

Attribute	Syntax	Description
Uncommitted	TRANSACTION_READ_UNCOMMITTED	Allows all three violations
Committed	TRANSACTION_READ_COMMITTED	Allows non-repeatable reads and phantoms, but doesn't allow a dirty read.
Repeatable	TRANSACTION_REPEATABLE_READ	Allows phantoms, but not the other two violations.
Serializable	TRANSACTION_SERIALIZABLE	Doesn't allow any of the three violations.

Setting data source properties

When a data source is selected in the Deployment Descriptor editor, a Properties tab appears as well as the General tab. The Properties panel allows you to set properties that affect the Borland container-managed persistence (CMP) engine.



To modify the properties of a data source,

- 1 Double-click the data source in the project pane.
- 2 Click the Properties tab.
- 3 On the Properties panel, select a property to set from the Name drop-down list.
The Type value is set automatically, depending on your selection from the Name list.
- 4 Select a value in the Value column for your property.
- 5 Add additional properties by clicking the Add button to add a new row, and then select the Name and Value entries for that new property.

These are the possible properties:

Property	Description
maxBigDecimalScale	If you are using JDBC 1.0, the value of this property determines the scale to use when this method is called: <pre>java.sql.BigDecimal java.sql.ResultSet.getBigDecimal(int columnIndex, int scale);</pre> If you are using JDBC 2.0, a scale value is not used when <code>getBigDecimal(int columnIndex)</code> is called.
uniqueSequenceDeclaration	Determines whether the CMP engine should declare a unique sequence for the primary key columns. Usually this is achieved by declaring the appropriate columns to be primary keys (see <code>primaryKeyDeclaration</code> in this table).
batchUpdates	Indicates whether the CMP engine should batch updates to the database. This can have a significant performance benefit for transactions that update a number of entities, and should be used if the driver supports it. Unfortunately, most don't support batch updates yet. The default value is false.
useSetObjectForSetNull	When a SQL column is set to null value, usually this method is used: <pre>void java.sql.PreparedStatement.setNull(int parameterIndex, int sqlType);</pre> Because some JDBC drivers do not support this, you can set this flag to make the CMP engine use the following method instead: <pre>void java.sql.PreparedStatement.setObject(int parameterIndex, Object x);</pre> "null" becomes the value of x.
reuseStatements	Determines whether the CMP engine should reuse prepared statements across transactions. Reusing prepared statements has a significant performance impact, and they should be used unless the JDBC driver exhibits are reused. The default value is true.
notNullDeclaration	Determines whether the Java fields that can't be null (such as <code>int</code> or <code>float</code>) should map to non-null columns. The default value is true.
dialect	Determines the type of the data source, such as whether its a JDataStore, Oracle, Informix, or other data source. Select the dialect value from the Value drop-down list. If you don't set this field, the CMP engine creates tables for JDataStore only. The default value is none.
primaryKeyDeclaration	Determines whether the CMP engine declares the primary key columns in the table to be primary keys. Some databases don't support primary key declarations. The default value is true.

Adding security roles and method permissions

The Deployment Descriptor editor enables you to create or edit security roles in the deployment descriptor. After you create security roles, you can then associate methods in an enterprise bean's home, remote, local, and local home interfaces with these roles, thereby defining the security view of the application.

This section describes how to use the Deployment Description editor to create the security roles and assign enterprise bean methods to the roles. [“Security Role References panel” on page 11-13](#) describes how to use the Roles panel to assign user groups and/or user accounts to the roles.

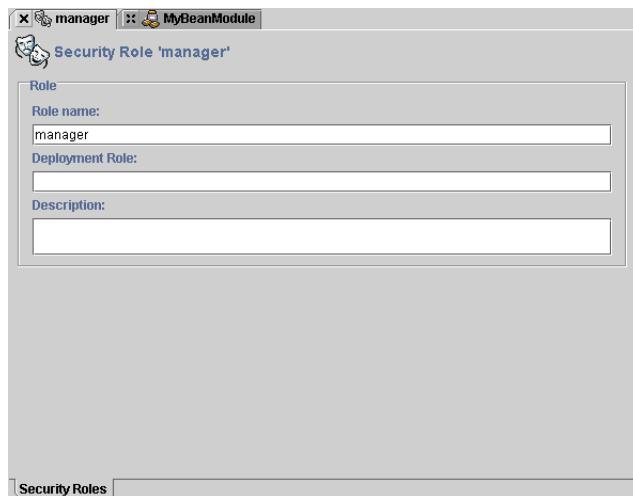
Defining security roles in the deployment descriptor is optional.

Creating a security role

To create a security role in the deployment descriptor,

- 1 Right-click the Security Roles folder node in the project pane. Choose the New Role command on the menu that appears.
- 2 In the dialog box that appears, enter the name of the new security role and choose OK.

The new role appears under the Security Roles node in the project pane. Expand the Security Roles node to see it. Double-click the new role in the project pane to view the Security Roles panel:



You can enter a description for the new role on the Security Roles panel. The description is optional.

Adding security roles and method permissions

If your selected application server is WebLogic Server 6.x or later, you'll see a WebLogic 6.x, WebLogic 7.x, or WebLogic 8.1 Security Roles panel instead. For example, this is the WebLogic 7.x Security Roles panel:

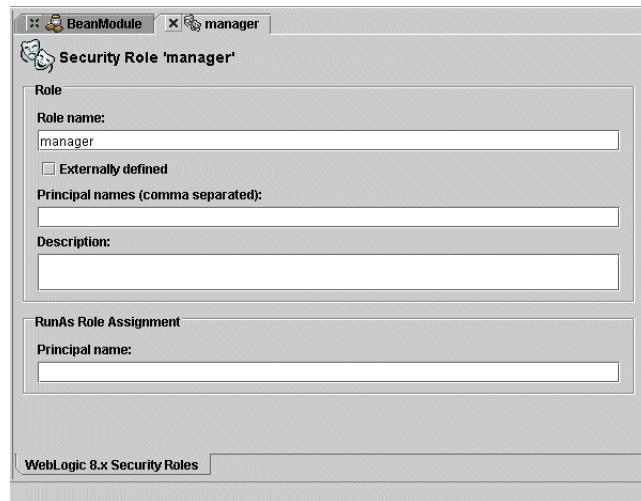


Enter principal names separated by commas in the Principal Names field. Filling in the Description field is optional.

WebLogic 7.0 with service pack 1

The panel also displays a Global check box. When this check box is checked, you can declare the edited role name based on the security realm role/principal mapping.

If you are using WebLogic Platform 8.1, the Security Roles panel will look like this:



Assigning method permissions

Once you've defined one or more security roles, you can specify which methods in the interfaces of an enterprise bean the security role is allowed to invoke.

You aren't required to associate a security role with methods in a bean's interfaces. In these cases, none of the security roles defined in the deployment descriptor are allowed to invoke these methods.

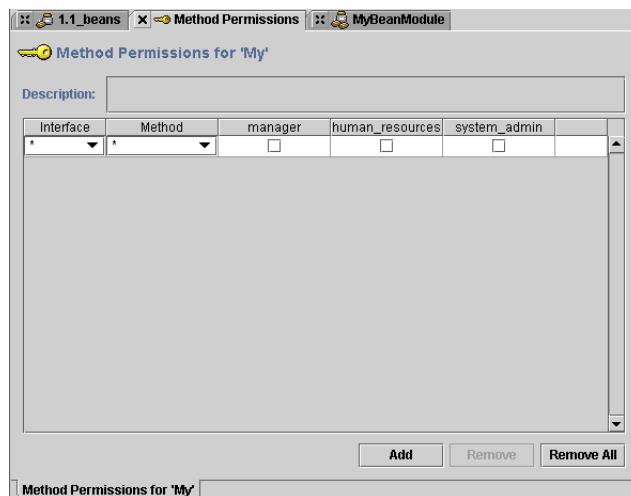
To assign method permissions,

- 1** Expand a bean's node in the project pane to reveal its Method Permissions sub-node.
- 2** Double-click the Method Permission node to display a Method Permissions panel.

Each defined security role appears as a column heading.

- 3** Click the Add button to add a row to the panel.

If your bean is an EJB 1.1 bean, the Method Permissions panel looks like this:

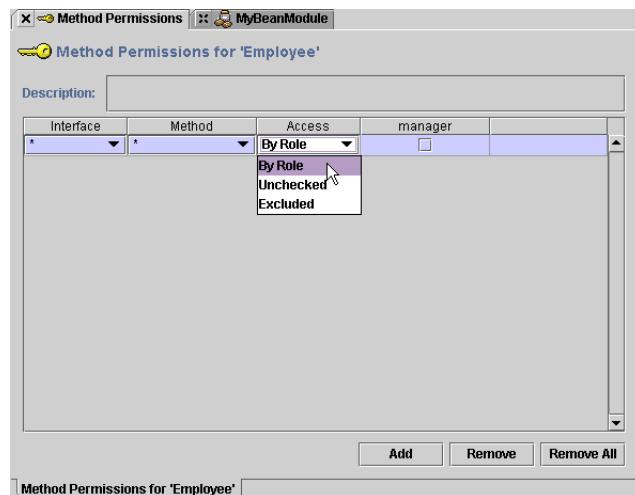


Follow these instructions for a 1.1 bean:

- 1** In the new row, choose Home or Remote to indicate which interface you are working with. Choosing * selects both interfaces. Your selection for the Interface column determines which methods are available to you in the Method column.
- 2** From the Method drop-down list, select the method you are granting permission to call, or select *, which indicates permission to call all the methods.

- 3 Check the check box for each security role you want to give permission to call the specified methods.
- 4 As a final step, you can enter an optional description in the Description field to describe the permission the row defines.

If your bean is an EJB 2.0 bean, the Method Permissions panel looks like this:



Follow these instructions for a 2.0 bean:

- 1 In the new row, choose Home, Remote, Local, or LocalHome to indicate which interface you are working with. Choosing * selects all interfaces. Your selection for the Interface column determines which methods are available to you in the Method column.
- 2 From the Method drop-down list, select the method you are granting permission to call, or select *, which indicates permission to call all the methods.
- 3 From the Access drop-down list, select how you want to establish access to the method:
 - By Role - Permission to call the specified methods is granted to each checked security role.
 - Unchecked - Indicates that principals in any role may access the method.
 - Excluded - The method cannot be called by any role.
- 4 If you selected By Role, check the check box for each security role you want to give permission to call the specified methods.
- 5 As a final step, you can enter an optional description in the Description field to describe the permission the row defines.

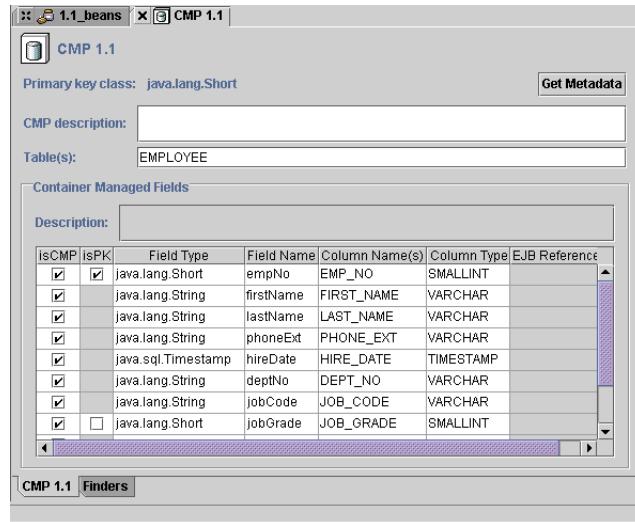
Adding container-managed persistence information for EJB 1.1 components

The CMP 1.1 specifies how the container manages persistence for the entity bean. Using the panel, you can map fields in the bean to columns in a database table.

To display the CMP 1.1 panel,

- 1 Expand an EJB 1.1 entity bean node in the project pane.
- 2 Double-click the CMP 1.1 node in the project pane.

The CMP 1.1 panel appears:



The CMP 1.1 panel includes these fields:

- **Primary Key Class:** Reports the fully-qualified name of the entity bean's primary key class. You specify the primary key for an entity bean on the General panel of the Deployment Descriptor editor.
- **Get Meta Data button:** Clicking this button retrieves the metadata for the table and populates a drop-down list for each Column Name(s) cell. Each element of the drop-down list is a column name/column type pair. Selecting from the drop-down list fills in both the column name and the column type cells. The drop-down list includes only the column names that have not already been used in the panel.
- **CMP Description:** An description of the container-managed persistence specified on this panel. Filling this field in is optional. After you enter the Description, click the row or the description you entered will be lost.

- **Table(s):** The name of the database table(s) referenced by the bean.
- **Description:** A description of a selected row in the table. The field is enabled only when a row is selected in the table. Filling this field in is optional.
- **isCMP:** A check mark indicates the field is container-managed.
- **isPK:** A check mark indicates the field is the primary key.
- **Field Type:** The data type of the field.
- **Field Name:** The name of the field. The Field Name column lists all fields in an entity bean.
- **Column Name(s):** You can map compound fields in the bean (for example, location.street) to columns in the database table. You can map either the root field (for example, location) or the subfields (for example, location.street), but not both.
- **Column Type:** The data type of the field.
- **EJB Reference:** If the field type is an EJB class, a menu appears with a list of EJB references to select. These references are set in the EJB references panel.

The panel displays the primary key class name, which you can't change. In the CMP Description field, you can enter optional text describing the bean.

Each field in your entity bean that will be container-managed has the CMP field checked. For each field, you enter the name of the column to which it maps. If you used the EJB 1.x Entity Modeler, JBuilder has already mapped these columns for you. You can edit the Column Name and Column Type if you choose. You can enter text describing each field in the Description field, but it's not required.

The Deployment Descriptor editor uses JDBC to obtain metadata on existing tables. You can conveniently hook up existing entity beans to existing tables. For example, you might purchase a third-party enterprise bean and want to use it with a table in your database. To populate both the Column Name and Column Type fields, click the Get Meta Data button and the metadata is retrieved and displayed.

Finders panel

The Finders panel specifies the “where” clauses used by the container-managed bean to execute finder methods defined by the bean. The Finders panel is available for EJB 1.1 entity beans with container-managed persistence.

To display the Finders panel,

- 1 Expand an EJB 1.1 entity bean node in the project pane.
- 2 Double-click the CMP 1.1 node in the project pane.
- 3 Click the Finders tab at the bottom of the Deployment Descriptor editor.

The Finders panel appears:



If you used the EJB 1.x Entity Bean Modeler to create your entity bean, you must check the Generate findAll() Method In Home Interface on the last page of the wizard or the Add button will be disabled on the Finders panel.

You'll find this information on the Finders panel:

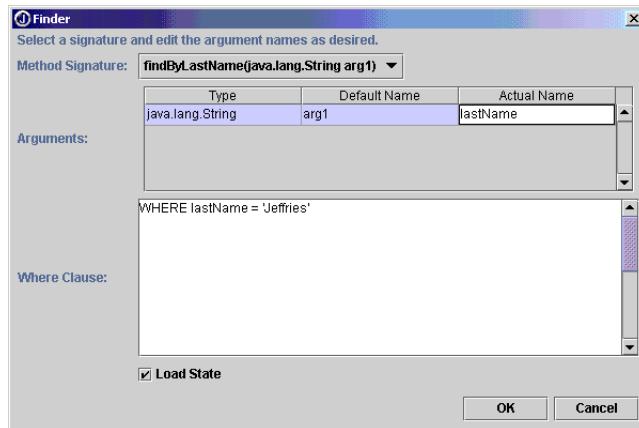
- **Method:** The finder method name and a list of all parameters.
- **Where Clause:** Specifies a SQL "where" clause used by the container to retrieve records from the database. Note that not all SQL statements can be converted to WebLogic query logic.
- **Load State:** When selected, this attribute enables the container to preload all container-managed fields whenever a find operation occurs.

To specify a finder method,

- 1 Click the Add button. The Add button is available only if you have defined a finder method in your bean.
A Finder dialog box appears.
- 2 Select the method signature for the finder method you want from the drop-down list.

- 3** Modify the argument names, if you wish, and specify the proper Where clause for the find operation.

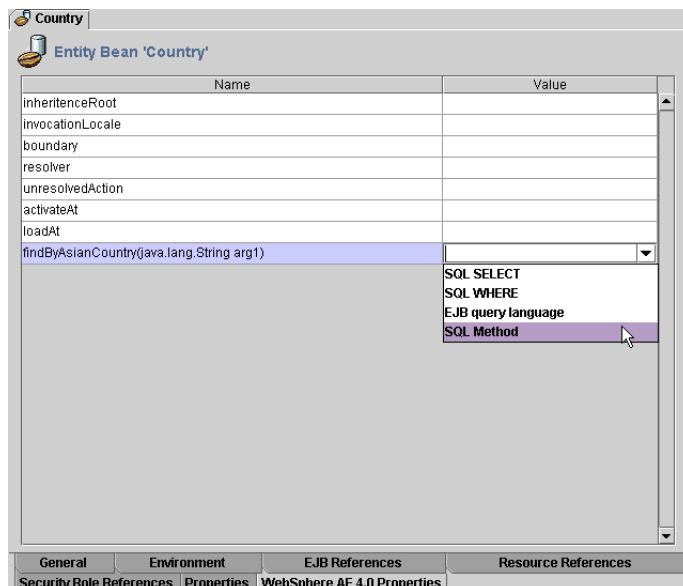
Here's an example:



Specifying WebSphere 4.0 finders

The JBuilder WebLogic Edition provides support for WebLogic Servers only

On the WebSphere 4.0 Properties panel, the finders in the bean appear at the bottom of the list of properties. You can choose the type of query you want to use for the finder: SQL SELECT, SQL WHERE, EJB Query Language, or SQL Method:



Select the finder type you want. The default value is a WHERE clause (SQL WHERE). The finder type you select is added to the WebSphere-specific deployment descriptor `ibm-ejb-jar-ext.xml`. You can then return to

the Finders panel and specify the query using the query type you selected on the WebSphere 4.0 Properties panel as the value of the Where Clause on the Finders panel, even if the query type is other than a WHERE clause.

Verifying descriptor information

After you've finished editing the descriptor file, you can verify that the descriptor information is in the correct format, the required bean class files are present, and so on.

To verify descriptor information, right-click the EJB module node and choose Verify on the menu that appears.

Verify does the following:

- Ensures that the descriptor conforms to the EJB specification.
- Ensures that the classes referenced by the deployment descriptors conform to the EJB specification.

Verify should be used with Borland application servers only.

12

Using the DataExpress for EJB components

The JBuilder WebLogic Edition provides support for WebLogic Servers only

JBuilder has several components that allow you to retrieve data from entity beans into DataExpress DataSets (providing), and to save data from DataExpress DataSets back to entity beans (resolving). These DataExpress for EJB components make it easy to implement a Session Bean wrap Entity Bean or Session Facade design pattern. Using this design pattern, clients usually don't access entity beans directly, but instead they access them using session beans. The session beans, which are co-located with the entity beans, make all the calls to the entity beans within a single transaction and then return all the data at once. DataSets provide a way of transporting the data from the session bean to the client and back. Because data is sent over the wire just once to provide the data to the client, and then just once again to resolve changes to the entity beans on the server, performance improves.

These components also make it easier for you to build client applications using DataExpress-aware visual components such as dbSwing or InternetBeans Express. For a full description of DataExpress, see the *Developing Database Applications*.

This chapter explains how to use these components to transfer data from entity beans deployed on a server to your client application and back again. The code is very similar to the /<jbuilder>/samples/Ejb/Ejb11/EjbDx.jpx sample project. The data the sample accesses is stored in an Employee data store. The sample creates an entity bean to hold Employee data. It also creates a Personnel session bean that retrieves data from Employee and then sends it to the client. The client sends the data back to the Personnel, which resolves the data to the Employee entity bean instances.

Note If you are using JDK 1.4, you won't be able to view live data with the DataExpress for EJB components in the UI Designer using the Borland Enterprise Server 5.1.1.

The DataExpress EJB components

Six of the DataExpress EJB components are on the EJB page of the component palette. You can work with these components in the UI designer, setting properties and events using the Inspector. There are additional classes that your code can call that you don't work with visually. For information about all of the classes, see the API reference.

Components for the server

The two components on the EJB page of the component palette that are used by the session bean deployed on the server are the `EntityBeanProvider` and `EntityBeanResolver` components. `EntityBeanProvider` provides data from the entity beans deployed on the server, and `EntityBeanResolver` resolves data to those entity beans. You add these components to the session bean you create to make the session bean capable of providing from and resolving to the entity beans.

If you are creating enterprise beans that run in an EJB 1.x container, you should continue using the `EntityBeanProvider` and `EntityBeanResolver` components. If your beans are going to be running in an EJB 2.0 container, you should use the `LocalEntityBeanProvider` and `LocalEntityBeanResolver` components instead. These components have an `ejbLocalHome` property instead of an `ejbHome` property. They also have an `ejbLocal` property, which takes the class of the entity bean's interface that implements `EJBLocalObject`. All the events and listeners in the `EntityBeanProvider` and `EntityBeanResolver` components have corresponding local versions in `LocalEntityBeanProvider` and `LocalEntityBeanResolver`.

Components for the client

Two of the components on the EJB page are used in the client side: `EjbClientDataSet` and `SessionBeanConnection`. The `EjbClientDataSet` provides data from and resolves changes to the session bean referenced in the `SessionBeanConnection`. A `SessionBeanConnection` holds the reference to a session bean on the server, and it contains the method names to provide datasets from and resolve datasets to that session bean.

Creating the entity beans

Begin by using the EJB 1.x Entity Modeler or the EJB Designer to create the entity beans that access the data you are interested in. The sample project creates Employee and Department entity beans, although this chapter refers only to Employee.

For information about creating EJB 1.1 entity beans, see [Chapter 7, “Creating EJB 1.x entity beans from an existing database table.”](#) For information about creating EJB 2.0 entity beans, see [Chapter 4, “Creating 2.0 entity beans with the EJB Designer.”](#)

Creating the server-side session bean

Create the session bean that will live on the server. Consider using the Enterprise JavaBean 1.x wizard or the EJB Designer to create a stateless session bean. Later in the next section you'll be adding EntityBeanProvider and EntityBeanResolver classes to this bean. Because these classes aren't serializable, it's easier to place them in a stateless session bean, which, for Borland application servers, is never passivated. If you require a stateful session bean for your application, you should either have the stateful session bean refer to a stateless session bean, or you must reinstantiate the EntityBeanProvider and EntityBeanResolver when the stateful session bean is activated.

Here is what a resulting bean class named PersonnelBean would look like:

```
public class PersonnelBean implements SessionBean {
    private SessionContext sessionContext;
    public void ejbCreate() {
    }
    public void ejbRemove() throws RemoteException {
    }
    public void ejbActivate() throws RemoteException {
    }
    public void ejbPassivate() throws RemoteException {
    }
    public void setSessionContext(SessionContext sessionContext) throws RemoteException {
        this.sessionContext = sessionContext;
    }
}
```

Click the Design tab to display the UI Designer.

Adding provider and resolver components to the session bean

From the EJB page of the component palette, add an EntityBeanProvider and an EntityBeanResolver to the session bean. If you are working with an 2.0 enterprise bean, add a LocalEntityBeanProvider and a LocalEntityBeanResolver instead. You must also add a dataset component to

hold the data gathered from the entity beans before it is sent to the client and the data that the client sends back. From the DataExpress page of the component palette, add a `TableDataSet` component and rename `TableDataSet` to some appropriate name.

This is how the top of `PersonnelBean` would look; the `TableDataSet` has been renamed to `employeeDataSet`:

```
public class PersonnelBean implements SessionBean {
    private SessionContext sessionContext;
    EntityBeanProvider entityBeanProvider = new EntityBeanProvider();
    EntityBeanResolver entityBeanResolver = new EntityBeanResolver();
    TableDataSet employeeDataSet = new TableDataSet();
    ...
}
```

Using the Inspector, set the provider and resolver properties of the `TableDataSet` to the newly added `EntityBeanProvider` and `EntityBeanResolver` components, respectively. The result is two new methods in the `jbInit()` method:

```
employeeDataSet.setProvider(entityBeanProvider);
employeeDataSet.setResolver(entityBeanResolver);
```

The sample project shows these methods in the `setSessionContext()` method instead. You can add the method calls yourself to `setSessionContext()` if you prefer to imitate the sample project exactly. Either approach is fine.

To the members of this class, add a reference to the home interface of the entity bean that contains the data you want to access. For this example, the reference is to the home interface of the `Employee` entity bean as shown here in bold.

```
public class PersonnelBean implements SessionBean {
    private SessionContext sessionContext;
    EntityBeanProvider entityBeanProvider = new EntityBeanProvider();
    EntityBeanResolver entityBeanResolver = new EntityBeanResolver();
    TableDataSet employeeDataSet = new TableDataSet();
    EmployeeHome employeeHome;
    ...
}
```

Writing the `setSessionContext()` method

In the session bean's `sessionContext()` method add a try block. Modify the method so that it looks like this:

```
public void setSessionContext(SessionContext sessionContext)
    throws RemoteException {
    this.sessionContext = sessionContext;
    try {
        Context context = new InitialContext();
        Object object = context.lookup("java:comp/env/ejb/Employee");
        employeeHome = (EmployeeHome) PortableRemoteObject.narrow(object,
            EmployeeHome.class);
```

```

        entityBeanProvider.setEjbHome(employeeHome);
        entityBeanResolver.setEjbHome(employeeHome);
    }
    catch (Exception ex) {
        throw new EJBException(ex);
    }
}

```

Note that `setSessionContext()` sets the value of the `ejbHome` properties in the `EntityBeanProvider` and `EntityBeanResolver` components to the name of the home interface of the `Employee` entity bean.

Adding an EJB reference or EJB local reference to the deployment descriptor

You must add an EJB reference to `Personnel` in the deployment descriptor for the lookup to work. For an entity bean with local interfaces, you must add an EJB local reference instead. You can use the Deployment Descriptor editor:

- 1 In the project pane, double-click the EJB module node. For the sample project, this is `personnel.ejbgrpx`.
- 2 Double-click the `Personnel` bean in the project pane.
The Deployment Descriptor editor appears.
- 3 Click the EJB References tab (or the EJB Local References tab) in the Deployment Descriptor editor.
- 4 Click the Add button to add a reference to the entity bean containing the data you are interested in.
- 5 Enter a reference name. For the sample project, the name is `ejb/Employee`.
- 6 Check the `IsLink` check box.
- 7 Specify the entity bean from the Link drop-down list.
The rest of the data should fill in for you automatically.

Adding the providing and resolving methods

You must add two methods to the session bean, a provider and a resolver. The names of these methods use the value you specified as the `methodName` property value in the `EjbClientDataSet` component. So the provider for `PersonnelBean` becomes `provideEmployee()` and the resolver becomes `resolveEmployee()`.

The provider must call the method of an `EntityBeanConnection` class that provides the data from an entity bean to a dataset that can be sent over the wire. This is what the `provideEmployee()` method must look like:

```
public DataSetData [] provideEmployee(RowData[] parameterArray,
    RowData[] masterArray) {
    return EntityBeanConnection.provideDataSets(new StorageDataSet []
        {employeeDataSet}, parameterArray, masterArray);
}
```

The resolver must call the method of an `EntityBeanConnection` class that resolves any updates to the entity beans. This is how `resolveEmployee()` should appear:

```
public DataSetData[] resolveEmployee(DataSetData[] dataSetdataArray) {
    return EntityBeanConnection.saveChanges(dataSetdataArray,
        new DataSet[] {employeeDataSet});
}
```

Next add these methods to the remote interface. The simplest way to do this for an EJB 1.x bean this is to use BeansExpress. With the bean source file open in the editor, click the Bean tab, click the Methods tab, and check the check boxes next names of the two methods you just added. For an EJB 2.0 bean, use the EJB Designer. In the bean representation in the EJB Designer, click a new method you just added and in the method inspector that appears, select Remote or Local from the Interfaces drop-down list.

You can now check the your session bean's remote interface (or local interface for a 2.0 bean if that is the interface you chose to define the methods in) to verify that the two methods are now defined. If you chose to define the methods in the remote interface, this is how it would look:

```
public interface Personnel extends EJBObject {
    public com.borland.dx.dataset.DataSetData[]
        providePersonnel(com.borland.dx.ejb.RowData[] parameterArray,
            com.borland.dx.ejb.RowData[] masterArray) throws RemoteException;
    public com.borland.dx.dataset.DataSetData[]
        resolvePersonnel(com.borland.dx.dataset.DataSetData[] dataSetdataArray)
            throws RemoteException;
}
```

Calling the finder method

You must tell the `EntityBeanProvider` which entity beans to provide. To do this, add an event to the `EntityBeanProvider`:

- 1 While in the UI Designer, select the `EntityBeanProvider` in the structure pane.

- 2** Click the Events tab of the Inspector and double-click the blank column next the `findEntityBeans` event. A new event is added.

Here is the resulting event:

```
entityBeanProvider1.addEntityBeanFindListener(new
    com.borland.dx.ejb.EntityBeanFindListener() {
        public void findEntityBeans(EntityBeanFindEvent e) {
            entityBeanProvider1_findEntityBeans(e);
        }
    });
    ...
void entityBeanProvider_findEntityBeans(EntityBeanFindEvent e) {
}
```

- 3** To the new event handler, add a finder method to return the entity beans you want. Here the added code appears in bold:

```
void entityBeanProvider_findEntityBeans(EntityBeanFindEvent e) {
    try {
        e.setEntityBeanCollection(employeeHome.findAll());
    }
    catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

In this example, the event handler calls a `findAll()` method to return all the entity beans. You can call any finder you want. You could use the `EntityBeanProvider`'s `parameterRow` property to dynamically determine which finder to call and/or which parameters to pass.

For resolving, the `EntityBeanResolver` can by default determine how to apply updates and deletes. But it can't automatically determine how to create new entity beans because there is no way it can determine which `create()` method to call and which parameters to pass to it. So, if you want to add a row to the data source, you must add the `create` event yourself and supply the necessary logic. You can use the Inspector to add the skeleton `create` event code to your session bean. You can see an example of a `create` event in the `EjbDx.jpx` sample project. You can also use the other events available in `EntityBeanResolver` to override the default behavior, if you choose.

Deploy the session and entity beans to the application server. For more information about deploying your beans, see “[Deploying to an application server](#)” on page 10-7.

Building the client side

Now that you've created the entity beans and the session bean that accesses them and deployed them to your target application server, you're ready to begin building the client.

Follow these steps:

- 1** Create a data module. Choose File | New | Data Module.
- 2** From the component palette, select the `EjbClientDataSet` and add it to the data module.
- 3** From the component palette, select the `SessionBeanConnection` and add it to the data module.
- 4** In the Inspector, set the `sessionBeanConnection` property of the `EjbClientDataSet` to the name of the `SessionBeanConnection` component.
- 5** In the Inspector, specify a name for the `methodName` property of the `EjbClientDataSet` component.

The `methodName` property determines how the methods that provide and resolve data are named. For example, if you specify a value of `Employee` for `methodName`, the session bean methods to provide and resolve data become `provideEmployee()` and `resolveEmployee()`. Later you will need to add these methods to the session bean you create.

- 6** In the Inspector or directly in the source code, set the `jndiName` property of the `SessionBeanConnection` component. Or you can specify the name of the remote interface of the session bean you will create instead as the value of the `sessionBeanRemote` property.

You can also use the Inspector to add a `creating` event to your `SessionBeanConnection`. Code you add to the event handler can control the creation of the session bean after the JNDI lookup occurs. Usually you must add a `creating` event if you want to invoke a `create()` method on the home interface that requires parameters. For example, look at this code:

```
import com.borland.dx.dataset.*;
import com.borland.dx.ejb.*;

public class PersonnelDataModule implements DataModule {
    private static PersonnelDataModule myDM;
    SessionBeanConnection sessionBeanConnection = new SessionBeanConnection();
    EjbClientDataSet personnelDataSet = new EjbClientDataSet();

    public PersonnelDataModule() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

private void jbInit() throws Exception {
    try {
        sessionBeanConnection.setJndiName("Personnel");
        sessionBeanConnection.addCreateSessionBeanListener(new
com.borland.dx.ejb.CreateSessionBeanListener() {
            public void creating(CreateSessionBeanEvent e) {
                sessionBeanConnection_creating(e);
            }
        });
        personnelDataSet.setSessionBeanConnection(sessionBeanConnection);
        personnelDataSet.setMethodName("Personnel");
    }
    catch (Exception ex) {

    }
}
public static PersonnelDataModule getDataModule() {
    if (myDM == null) {
        myDM = new PersonnelDataModule();
    }
    return myDM;
}
public com.borland.dx.ejb.SessionBeanConnection getSessionBeanConnection() {
    return sessionBeanConnection;
}
public com.borland.dx.ejb.EjbClientDataSet getPersonnelDataSet() {
    return personnelDataSet;
}

void sessionBeanConnection_creating(CreateSessionBeanEvent e) {

}
}

```

Handling relationships

The EntityBeanProvider automatically flattens relationships. For example, if you have any Employee entity bean that has a `getDept()` method that returns a Dept, where Dept is an entity bean remote, a DataSet is created that has all the fields in the Employee entity bean plus all the fields in the Dept entity bean (including any hidden columns containing the primary keys of each of the entity beans). Except for `Dept.ejbPrimaryKey`, the other Dept fields will be read-only.

To resolve changes when a one-to-one relationship is involved, you must add an event listener to the EntityBeanProvider because it can't dynamically determine the home of the related entity bean. The sample EjbDx.jpx project does not demonstrate handling relationships.

The sample project

So far you've seen how to efficiently transfer data back and forth between the client and the server. The sample `/<jbuilder>/samples/Ejb/Ejb11/EjbDx/EjbDx.jpx` project shows you how to use the described techniques with a Java client that uses dbSwing controls and with a Web client that uses JSP technology combined with InternetBeans Express. You will be able to work with live data. Check the project's `EjbDx.html` page to find complete instructions for running the sample project.

The `/<jbuilder>/samples/Ejb/Ejb11/EjbDxWL/EjbDxWL.jpx` project shows how to run the same project under a different application server. In this case, the server is the BEA WebLogic Server 6.x or 7.x. The `EjbDxWL.html` file describes the differences between the two projects and what you must do to run the project using WebLogic Server.

13

Developing session beans

JBuilder's EJB tools can greatly simplify the creation of enterprise beans and their supporting interfaces. You should understand what the requirements are for these classes and interfaces, however, so you can modify the files JBuilder produces and so you understand what JBuilder is doing for you. The next few chapters can help you gain that understanding.

A session bean usually exists for the lifetime of a single client session. The methods of a session bean perform a set of tasks or processes for the client that uses the bean. Session beans persist only for the life of the connection with the client. In a sense, the session bean represents the client in the EJB server. It usually provides a service to the client. Unless you need to work with persistent data that exists in a data store, you are usually working with session beans.

Types of session beans

There are two types of session beans: those that can maintain state information between method calls, which are called *stateful* beans, and those that can't, which are called *stateless* beans.

Stateful session beans

Stateful session beans are objects used by a single client and they maintain state on behalf of that client. For example, consider a shopping cart session bean. As the shopper in an online store selects items to purchase, the items are added to the "shopping cart" by storing the selected items in a list within the shopping cart session bean object. When the shopper is ready to purchase the items, the list is used to calculate the total cost.

Stateless session bean

Stateless session beans don't maintain state for any specific client. Therefore, they can be used by many clients. For example, consider a sort session bean that contains a `sortList()` business method. The client would invoke `sortList()`, passing it an unsorted list of items. `sortList()` would then pass back to the client a sorted list.

Writing the session bean class

To create a session bean class,

- Create a class that implements the `javax.ejb.SessionBean` interface.
- Implement one or more `ejbCreate()` methods. If you are creating a stateless session bean, the class implement just one parameterless `ejbCreate()` method. If you've already created the remote home or local home interface for the bean, the bean must have an `ejbCreate()` method with the same signature for each `create()` method in the remote home/local home interface.
- Define and implement the business methods you want your bean to have. If you've already created the remote or local interface for the bean, the methods must be defined exactly as they are in the remote/local interface.

JBuilder's EJB tools can start these tasks for you, including creating the home and remote interfaces. They create a class that extends the `SessionBean` interface and write empty implementations of the `SessionBean` methods. You fill in the implementations if your bean requires them. The next section explains what these methods are and how they are used.

Implementing the SessionBean interface

The `SessionBean` interface defines the methods all session beans must implement. It extends the `EnterpriseBean` interface.

```
package javax.ejb;
public interface SessionBean extends EnterpriseBean {
    void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException;
    void ejbRemove() throws EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
}
```

The methods of the `SessionBean` interface are closely associated with the life cycle of a session bean. This table explains their purpose:

Method	Description
<code>setSessionContext()</code>	Sets a session context. The bean's container calls this method to associate a session bean instance with its context. The session context interface provides methods to access the runtime properties of the context in which a session runs. Usually a session bean retains its context in a data field.
<code>ejbRemove()</code>	Notifies a session object that it is about to be removed. The container calls this method whenever it removes a stateful session bean as a result of the client calling a <code>remove()</code> method of the remote/local or remote home/local home interface.
<code>ejbActivate()</code>	Notifies a stateful session object that has been activated.
<code>ejbPassivate()</code>	Notifies a stateful session object that it is about to be deactivated by the container.

The `ejbActivate()` and `ejbPassivate()` methods allow a stateful session bean to manage resources. For more information, see “[Stateful beans](#)” on [page 13-7](#).

Writing the business methods

Within your enterprise bean class, write full implementations of the business methods your bean needs using JBuilder’s code editor. To make these methods available to a client, you must also declare them in the bean’s remote interface exactly as they are declared in the bean class. You can use JBuilder’s EJB tools to perform that task. If you are using the EJB Designer to create EJB 2.0 components, the methods are declared properly in the remote/local interface when you use the bean’s inspector to specify where the methods should be declared. See “[Creating session beans](#)” on [page 3-10](#). If you are creating EJB 1.1 components, use JBuilder’s Bean designer to ensure the methods are properly declared in the bean’s remote method. See the “[Exposing business methods through the remote interface](#)” on [page 6-11](#).

Adding one or more `ejbCreate()` methods

If you use the JBuilder’s EJB tools to begin your enterprise bean, you’ll see that an `ejbCreate()` method that takes no parameters is added to the bean class. You can add additional `ejbCreate()` methods that do include parameters. While stateless session beans never need more than a parameterless `ejbCreate()` method because they don’t retain any state, stateful session beans often need one or more `ejbCreate()` methods that

have parameters. As you write additional `ejbCreate()` methods with parameters, keep these rules in mind:

- Each `ejbCreate()` must be declared as public.
- Each must return void.
- The parameters of an `ejbCreate()` method must be of the same number and type as those in the corresponding `create()` method in the bean's remote interface. For stateless session beans, there can be only one parameterless `ejbCreate()`.

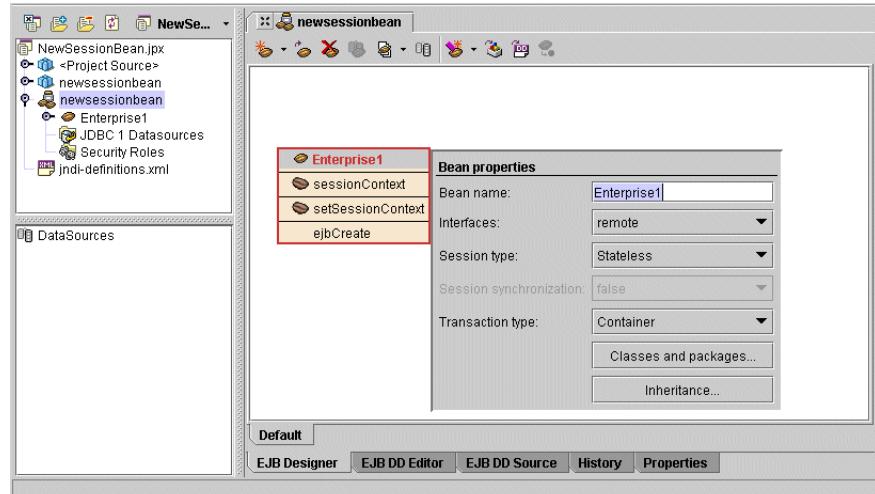
This is the signature for all `ejbCreate()` methods of a session bean:

```
public void ejbCreate( <zero or more parameters> ) {
    // implementation
}
```

The `ejbCreate()` method need not throw an exception, although it can throw application-specific exceptions and other exceptions, such as `javax.ejb.CreateException`. JBuilder's EJB tools generate an `ejbCreate()` method that throws `javax.ejb.CreateException`.

How JBuilder can help you create a session bean

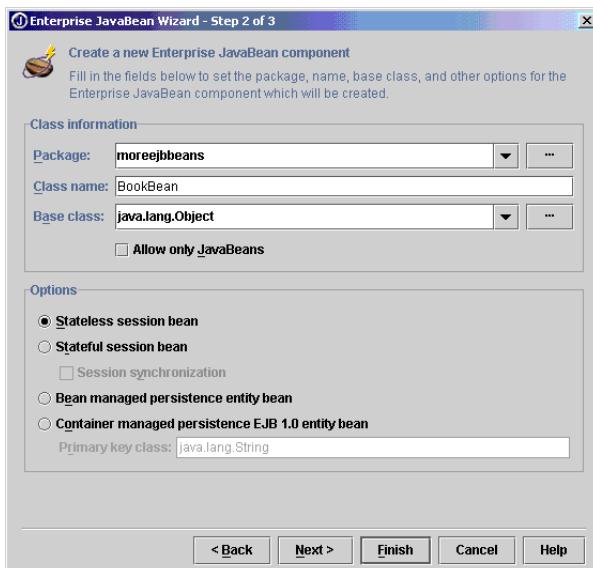
If you are creating a EJB 2.0 session bean, you begin by right-clicking the EJB Designer pane and choosing Create EJB | Session Bean from the context menu or by clicking the Create EJB icon on the EJB Designer toolbar and choosing Session Bean. A session bean representation appears in the EJB Designer with an inspector for modifying its attributes:



Use the inspector to choose whether the session bean is stateful or stateless and to set other attributes. For more information about creating session

beans with the EJB 2.0 Designer, see “[Creating session beans](#)” on page 3-10.

Using JBuilder’s Enterprise JavaBean 1.x wizard, you can begin creating a EJB 1.1 session bean by selecting either the Stateless Session Bean or Stateful Session Bean option on the wizard’s second page:



Not only do JBuilder’s EJB tools create your enterprise bean class, they also create the bean’s home/local home and remote/local interfaces as they create the bean class. This way, you are assured the `create()` method of the home interface returns the remote/local interface while the `ejbCreate()` method always returns void.

When you write the business methods in your EJB 2.0 bean class, you use the inspector to specify in which interface the methods are declared. The EJB Designer then adds the correct declaration for you to the proper interface. If you specify a home interface, the EJB Designer declares the method as a home business method. It adds the prefix `ejbHome` to the name of the method in your bean class and declares the method without the prefix in the home interface.

After you write the business methods in your EJB 1.x bean class, you can use the Bean designer to specify which of those you want defined in the bean’s remote interface for an EJB 1.1 session bean. A client application can access only those methods defined in the remote interface. Once you specify which methods you want a client to be able to call, the Bean designer defines the methods in the remote interface for you.

If you already have a complete enterprise bean class, but don’t have home and remote interfaces for it, you can use JBuilder’s EJB 1.x Interfaces wizard to create them. The method signatures will comply with EJB 1.1

specifications in the home and remote interfaces without you having to worry about making them correct.

For more information about using JBuilder's tools to develop EJB 1.1 session beans, see [“Creating a session bean” on page 6-6](#). For more information about using JBuilder's tools to develop EJB 2.0 session beans, see [Chapter 3, “Creating 2.0 session beans and message-driven beans with the EJB Designer.”](#)

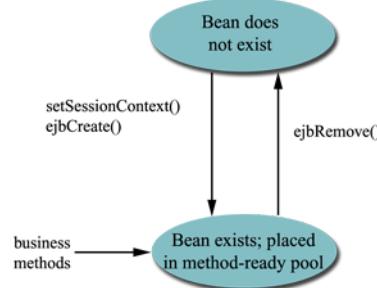
The life of a session bean

Stateful and stateless session beans have different life cycles. You should understand what happens during the life cycle of each.

Stateless beans

The life of a stateless session bean begins when the client calls the `create()` method of the session bean's home interface. The container creates a new instance of the session bean and returns an object reference to the client.

Figure 13.1 Stateless session bean life cycle



During the creation process, the container invokes the `setSessionContext()` method of the `SessionBean` interface and calls the `ejbCreate()` method of the session bean implementation. The new bean object joins a pool of stateless bean objects that are ready for use by clients. Because stateless session objects don't maintain client-specific state, the container can assign any bean object to handle an incoming method call. When the container removes an object from the session bean object pool, it calls the `ejbRemove()` method of the bean object.

Note that calling the `create()` or `remove()` methods of the home/local home and remote/local interfaces doesn't add or remove a stateless session bean object to or from the stateless session bean pool. The container controls the life cycle of stateless beans.

Stateful beans

The life of a stateful session bean begins when the client calls the `create()` method of the session bean's home interface. The container creates a new instance of the session bean, initializes it, and returns an object reference to the client.

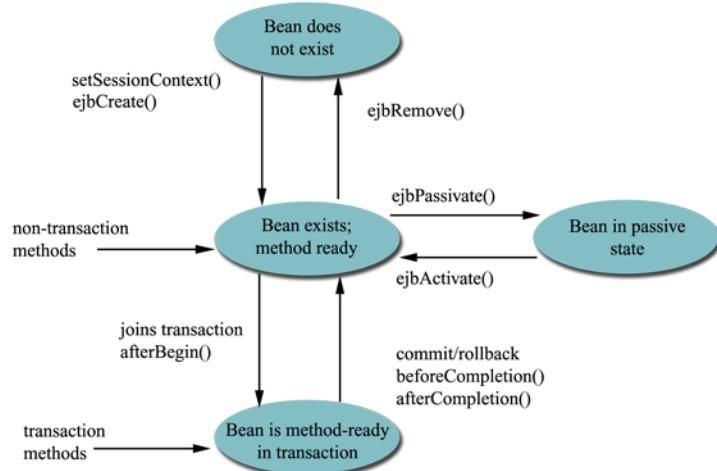
During the creation process, the container invokes the `setSessionContext()` method of the `SessionBean` interface and calls the `ejbCreate()` method of the session bean implementation. As a bean provider, you can use these methods to initialize the session bean.

The state of the session bean is now method ready, which means it can perform nontransaction operations or be included in a transaction for transaction operations. The bean remains in the method-ready state until one of three things happens:

- The bean enters a transaction.
- The bean is removed.
- The bean is passivated.

When a client calls the `remove()` method of the remote/local or home interface, the container invokes the corresponding `ejbRemove()` method on the session bean object. As a bean provider, you put any application-specific cleanup code in this method. After `ejbRemove()` completes, the bean object is no longer usable. If the client attempts to call a method in the bean object, the container throws the `java.rmi.NoSuchObjectException`.

Figure 13.2 Stateful session bean life cycle



The container can deactivate the session bean instance. Usually this occurs for resource management reasons such as when a session object is idle for

a while or if the container requires more memory. The container deactivates the bean by calling the bean's `ejbPassivate()` method. When a bean instance is deactivated, which is called passivation, the container stores reference information and the state of the session object on disk and frees the memory allocated to the bean. You can add code to `ejbPassivate()` if you have some task you want to execute just before passivation occurs.

The container activates the bean object again by calling the `ejbActivate()` method. This occurs when the client calls a method on the session bean that is passivated. During activation, the container recreates the session object in memory and restores its state. If you want something to happen immediately after the bean becomes active again, add your code to the `ejbActivate()` method.

The method-ready in transaction state

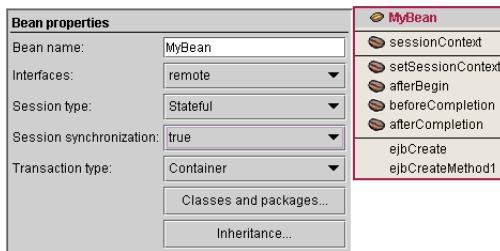
When a client calls a method on a session bean object in a transactional context, the container starts a new transaction or includes the bean object in an existing one. The bean enters the method-ready in transaction state. There are points in a transaction's life cycle, called transaction synchronization points, where a session bean object can be notified of upcoming transaction events and the object can take some action beforehand if necessary.

The SessionSynchronization interface

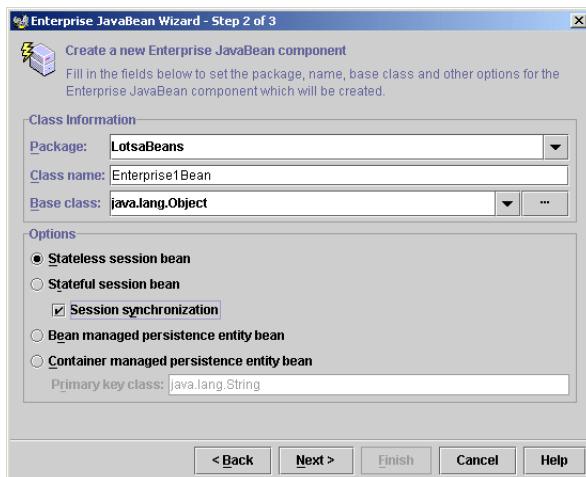
A session bean can implement the `SessionSynchronization` interface if it wants to be notified about the state of any transaction in which it is involved. Only stateful session beans using container-managed transactions can implement `SessionSynchronization`. Its use is optional. The methods of `SessionSynchronization` are callbacks made by the container into the bean, and they mark points within the transaction. Here is the `SessionSynchronization` interface:

```
public interface javax.ejb.SessionSynchronization
{
    public abstract void afterBegin() throws RemoteException;
    public abstract void beforeCompletion() throws RemoteException;
    public abstract void afterCompletion(boolean completionStatus) throws
        RemoteException;
}
```

For an EJB 2.0 session bean, use the bean's Inspector in the EJB Designer to set the `Session Synchronization` attribute to true to add the three `SessionSynchronization` interface methods to your bean class:



For an EJB 1.1 session bean, the Enterprise JavaBean 1.1 wizard can add these methods to your bean class. Using the wizard, check the Session Synchronization check box, and the wizard declares the three methods with empty bodies in your bean class:



The following table briefly describes each method:

Method	Description
afterBegin()	Notifies the bean instance that it is about to be used in a transaction. Any code you write within afterBegin() occurs within the scope of the transaction.
beforeCompletion()	Notifies the bean instance that the transaction is about to commit. If the bean has any cached values, use beforeCompletion() to write them to the database. If necessary, a session bean could use the beforeCompletion() method to force the transaction to roll back by calling the setRollbackOnly() method of the SessionContext interface.
afterCompletion()	Notifies the bean instance that the transaction has completed. If the transaction was committed, the parameter completionStatus is set to true. If the transaction was rolled back, the parameter is set to false.

This is how the `SessionSynchronization` methods are used: The client calls a transactional business method defined in the remote interface, which puts the bean object in the transaction-ready state. The container calls the `afterBegin()` method in the bean object. Later, if the transaction is committed, the container calls `beforeCompletion()`, and then, if the commit was successful, the `afterCompletion(true)` method. If the transaction was rolled back or otherwise failed to commit, the container calls `afterCompletion(false)`. The session bean object is now in method-ready state again.

For more information about using session beans in transactions, see [Chapter 18, “Managing transactions.”](#)

14

Developing entity beans

An entity bean directly represents data stored in persistent storage, such as a database. It maps to a row or rows within one or more tables in a relational database, or to an entity object in an object-oriented database. It can also map to one or more rows across multiple tables. In a database, a primary key uniquely identifies a row in a table. Similarly, a primary key identifies a specific entity bean instance. Each column in the relational database table maps to an instance variable in the entity bean.

Because an entity bean usually represents data stored in a database, it lives as long as the data. Regardless of how long an entity bean remains inactive, the container doesn't remove it from persistent storage.

The only way to remove an entity bean is to explicitly do so. An entity bean is removed by calling its `remove()` method, which removes the underlying data from the database. Or an existing enterprise application can remove data from the database.

Persistence and entity beans

All entity enterprise beans are persistent; that is, their state is stored between sessions and clients. As a bean provider, you can choose how your entity bean's persistence is implemented.

You can implement the bean's persistence directly in the entity bean class, making the bean itself responsible for maintaining its persistence. This is called *bean-managed persistence*.

Or you can delegate the handling of the entity bean's persistence to the EJB container. This is called *container-managed persistence*.

Bean-managed persistence

An entity bean with bean-managed persistence contains the code to access and update a database. That is, you, as the bean provider, write database access calls directly in the entity bean or its associated classes. Usually you write these calls using JDBC.

The database access calls can appear in the entity bean's business methods, or in one of the entity bean interface methods. (You'll read more about the entity bean interface soon.)

Usually a bean with bean-managed persistence is more difficult to write because you must write the additional data-access code. And, because you might choose to embed the data-access code in the bean's methods, it can also be more difficult to adapt the entity bean to different databases or to a different schema.

Container-managed persistence

You don't have to write code that accesses and updates databases for entity beans with container-managed persistence. Instead, the bean relies on the container to access and update the database.

Container-managed persistence has many advantages compared to bean-managed persistence:

- Such beans are easier to code.
- Persistence details can be changed without modifying and recompiling the entity bean. Instead the deployer or application assembler can modify the deployment descriptor.
- The complexity of the code is reduced, as is the possibility of errors.
- You, as the bean provider, can focus on the business logic of the bean and not on the underlying system issues.

Container-managed persistence has some limitations, however. For example, the container might load the entire state of the entity object into the bean instance's fields before it calls the `ejbLoad()` method. This could lead to performance problems if the bean has many fields.

Primary keys in entity beans

Each entity bean instance must have a primary key. A primary key is a value (or combination of values) that uniquely identifies the instance. For example, a database table that contains employee records might use the employee's social security number for its primary key. The entity bean

modeling this employee table would also use the social security number for its primary key.

For enterprise beans, the primary key is represented by a String or Integer type or a Java class containing the unique data. This primary key class can be any class as long as that class is a legal value type in RMI-IIOP. This means the class must extend the `java.io.Serializable` interface, and it must implement the `Object.equals(Other other)` and `Object.hashCode()` methods, which all Java classes inherit.

The primary key class can be specific to a particular entity bean class. That is, each entity bean can define its own primary key class. Or multiple entity beans can share the same primary key class.

Writing the entity bean class

To create an entity bean class,

- Create a class that implements the `javax.ejb.EntityBean` interface.
- Implement one or more `ejbCreate()` methods. If you've already created the home or local home interface for the bean, the bean must have an `ejbCreate()` method with the same signature for each `create()` method in that interface. If an `ejbCreate()` method has a parameter, you must also declare and implement an `ejbPostCreate()` method.
- Define and implement the business methods you want your bean to have. If you've already created the remote or local interface for the bean, the methods must be defined exactly as they are in the remote or local interface.
- For entity beans with bean-managed persistence, implement finder methods.

JBuilder's EJB tools can start these tasks for you. They create a class that extends the `EntityBean` interface and writes empty implementations of the `EntityBean` methods. You fill in the implementations if your bean requires it. The next section explains what these methods are and how they are used.

If you'd like to build EJB 2.0 entity beans using existing database tables, you import a database schema into the EJB Designer and use it to create your entity beans. For more information, see “[Creating CMP 2.0 entity beans from an imported data source](#)” on page 4-2

To use existing database tables for EJB 1.x entity beans, you use JBuilder's EJB 1.x Entity Modeler. For more information, see “[Creating entity beans with the EJB 1.x Entity Bean Modeler](#)” on page 7-1

Implementing the EntityBean interface

The `EntityBean` interface defines the methods all entity beans must implement. It extends the `EnterpriseBean` interface.

```
public void EntityBean extends EnterpriseBean {
    public void setEntityContext(EntityContext ctx) throws EJBException,
        RemoteException;
    public void unsetEntitycontext() throws EJBException, RemoteException;
    void ejbRemove() throws RemoveException, EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
    void ejbLoad() throws EJBException, RemoteException;
    public void ejbStore() throws EJBException, RemoteException;
}
```

The methods of the `EntityBean` interface are closely associated with the life cycle of an entity bean. This table explains their purpose:

Method	Description
<code>setEntityContext()</code>	Sets an entity context. The container uses this method to pass a reference to the <code>EntityContext</code> interface to the bean instance. The <code>EntityContext</code> interface provides methods to access properties of the runtime context for the entity bean. An entity bean instance that uses this context must store it in an instance variable.
<code>unsetEntityContext()</code>	Frees the resources that were allocated during the <code>setEntityContext()</code> method call. The container calls this method before it terminates the life of the current instance of the entity bean.
<code>ejbRemove()</code>	Removes the database entry or entries associated with this particular entity bean. The container calls this method when a client invokes a <code>remove()</code> method.
<code>ejbActivate</code>	Notifies an entity bean that it has been activated. The container invokes this method on the instance selected from the pool of available instances and assigned to a specific entity object identity. When the bean instance is activated, it has the opportunity to acquire additional resources that it might need.
<code>ejbPassivate()</code>	Notifies an entity bean that it is about to be deactivated—that is, the instance's association with an entity object identity is about to be broken and the instance returned to the pool of available instances. The instance can then release any resources allocated with the <code>ejbActivate()</code> method that it might not want to hold while in the pool.

Method	Description
ejbLoad()	Refreshes the data the entity object represents from the database. The container calls this method on the entity bean instance so that the instance synchronizes the entity state cached in its instance variables to the entity state in the database.
ejbStore()	Stores the data the entity object represents in the database. The container calls this method on the entity bean instance so that the instance synchronizes the database state to the entity state cached in its instance variables.

Declaring and implementing the entity bean methods

Entity beans can have three types of methods:

- Create methods
- Finder methods
- Business methods

Creating create methods

If you use JBuilder's EJB tools to begin your enterprise bean, you'll see that they add an `ejbCreate()` method and an `ejbPostCreate()` method to the bean class that takes no parameters. You can write additional create methods if your bean requires them.

Keep in mind that entity beans are not required to have create methods. Calling a create method of an entity bean inserts new data in the database. You can have entity beans without create methods if new instances of entity objects should be added to the database only through DBMS updates or through a legacy application.

`ejbCreate()` method

If you choose to add additional `ejbCreate()` methods that include parameters, remember these rules:

- Each `ejbCreate()` must be declared as public.
- For container-managed entity beans, an `ejbCreate()` method must return null.

The container has complete responsibility for creating container-managed entity beans.

- For bean-managed entity beans, an `ejbCreate()` method must return an instance of the primary key class for the new entity object.

The container uses this primary key to create the actual entity reference.

- The parameters of an `ejbCreate()` method must be of the same number and type as those in the corresponding `create()` method in the bean's home interface.
- Each `ejbCreate()` method must have a corresponding `ejbPostCreate()` method that matches the `ejbCreate()` in the same number of parameters.

The signature for an `ejbCreate()` method is the same, regardless of whether the bean uses container-managed or bean-managed persistence. This is the signature for all `ejbCreate()` methods of an entity bean:

```
public <PrimaryKeyClass> ejbCreate( <zero or more parameters> )
    // implementation
}
```

When the client calls the `create()` method, the container in response executes the `ejbCreate()` method and inserts a record representing the entity object into the database. The `ejbCreate()` methods usually initialize some entity state. Therefore, they often have one or more parameters and their implementations include code that sets the entity state to the parameter values. For example, the bank example discussed later in this chapter has a checking account entity bean whose `ejbCreate()` method takes two parameters, a string and a float value. The method initializes the name of the account to the string value and the account balance to the float value:

```
public AccountPK ejbCreate(String name, float balance) {
    this.name = name;
    this.balance = balance;
    return null;
}
```

ejbPostCreate() method

When an `ejbCreate()` method finishes executing, the container then calls a matching `ejbPostCreate()` method to allow the instance to complete its initialization. The `ejbPostCreate()` matches the `ejbCreate()` method in its parameters, but it returns void:

```
public void ejbPostCreate( <zero or more parameters> )
    // implementation
}
```

Follow these rules when defining an `ejbPostCreate()`:

- It must be declared as public.
- It can't be declared as final or static.
- Its return type must be void.
- Its parameter list must match that of the corresponding `ejbCreate()` method.

Use `ejbPostCreate()` to perform any special processing your bean needs to do before it becomes available to the client. If your bean doesn't need to

do any special processing, leave the method body empty, but remember to include one `ejbPostCreate()` for every `ejbCreate()` for an entity bean with bean-managed persistence.

Creating finder methods

Every entity bean must have one or more finder methods. Finder methods are used by clients to locate entity beans. Each bean-managed entity bean must have an `ejbFindByPrimaryKey()` method that has a corresponding `findByPrimaryKey()` in the bean's home interface. This is the `ejbFindByPrimaryKey()` method's signature:

```
public <PrimaryKeyClass> ejbFindByPrimaryKey(<PrimaryKeyClass primaryKey> {
    // implementation
}
```

You can define additional finder methods for your bean. For example, you might have an `ejbFindByLastName()` method. Each finder method must follow these rules:

- It must be declared as public.
- Its name must start with the prefix **ejbFind**.
- It can't be declared as static or final.
- For an EJB 1.1 bean, it must return either a primary key or a collection of primary keys or an Enumeration of primary keys. For an EJB 2.0 bean, it must return a `java.util.Collection` of `EJBObjects` (either the remote or local interfaces) or a single `EJBObject`.
- The parameters and return type of the method must be valid Java RMI types.

For entity beans with bean-managed persistence, each finder method declared in the bean class must have a corresponding finder method in the bean's home or local home interface that has the same parameters, but returns the entity bean's remote/local interface. The client locates the entity bean it wants by calling the finder method of the home interface and the container then invokes the corresponding finder method in the bean class. See “[Finder methods for entity beans](#)” on page 16-5.

Finder methods for an EJB 2.0 entity bean with container-managed persistence must have an EJB query language query defined in its deployment descriptor. Use the EJB Designer to add a finder method, then use the finder method's inspector to define the query. For information about writing EJB QL queries, see “[Enterprise JavaBeans Query Language](#)” on the Sun web site at http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html.

Writing the business methods

Within your enterprise bean class, write full implementations of the business methods your bean needs. To make these methods available to a client, you must also declare them in the bean's local or remote interface using the exact same signature.

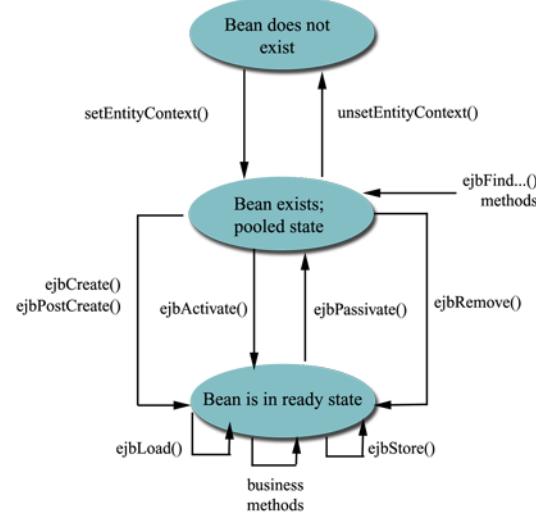
The life of an entity bean

There are three distinct states in the life cycle of an entity enterprise bean:

- Nonexistent
- Pooled
- Ready

The following diagram depicts the life cycle of an entity bean instance:

Figure 14.1 Entity bean life cycle



The nonexistent state

At first the entity bean instance doesn't exist. The EJB container creates an instance of an entity bean and then it calls the `setEntityContext()` method on the entity bean to pass the instance a reference to its context; that is, a reference to the `EntityContext` interface. The `EntityContext` interface gives the instance access to container-provided services and allows it to obtain information about its clients. The entity bean is now in the pooled state.

The pooled state

Each type of entity bean has its own pool. None of the instances in the pool are associated with data. Because none of their instance variables have been set, the instances have no identity and they are all equivalent. The container is free to assign any instance to a client that requests such an entity bean.

When a client application calls one of the entity bean's finder methods, the container executes the corresponding `ejbFind()` method on an arbitrary instance in the pool. The instance remains in the pooled state during the execution of a finder method.

When the container selects an instance to service a client's requests to an entity object, that instance moves from the pooled to the ready state. There are two ways that an entity instance moves from the pooled state to the ready state:

- Through the `ejbCreate()` and `ejbPostCreate()` methods.
- Through the `ejbActivate()` method.

The container selects the instance to handle a client's `create()` request on the bean's home interface. In response to the `create()` call, the container creates an entity object and calls the `ejbCreate()` and `ejbPostCreate()` methods when the instance is assigned to the entity object.

The container calls the `ejbActivate()` method to activate an instance so that it can respond to an invocation on an existing entity object. Usually the container calls `ejbActivate()` when there is no suitable instance in the ready state to handle the client's calls.

The ready state

When the instance is in the ready state, it is associated with a specific primary key. Clients can call the application-specific methods on the entity bean. The container calls the `ejbLoad()` and `ejbStore()` methods to tell the bean to load and store its data. They also enable the bean instance to synchronize its state with that of the underlying data entity.

Returning to the pooled state

When an entity bean instance moves back to the pooled state, the instance is decoupled from the data represented by the entity. The container can now assign the instance of any entity object within the same entity bean

home. There are two ways an entity bean instance moves from the ready state back to the pooled state:

- The container calls the `ejbPassivate()` method to disassociate the instance from its primary key without removing the underlying entity object.
- The container calls the `ejbRemove()` method to remove the entity object. It calls `ejbRemove()` when the client application calls the bean's home or `remote remove()` method.

To remove an unassociated instance from the pool, the container calls the instance's `unsetEntityContext()` method.

A bank entity bean example

The bank example shows you how to use entity beans. It includes two implementations of the same `Account` remote interface. One implementation uses bean-managed persistence, and the other uses container-managed persistence.

The `Savings` entity bean, which uses bean-managed persistence, models savings accounts. As you examine the entity bean code, you'll see that it includes direct JDBC calls.

The `Checking` entity bean, which uses container-managed persistence, models checking accounts. It relies on the container to implement persistence, not you, the bean developer.

A third enterprise bean called `Teller` transfers funds from one account to the other. It's a stateless session bean that shows you how calls to multiple entity beans can be grouped within a single container-managed transaction. Even if the credit occurs before the debit in the transfer operation, the container rolls back the transaction if the debit fails, and neither the debit nor the credit occurs.

You can find complete code for the bank example in the `/BorlandEnterpriseServer/examples/ejb/bank` directory.

The entity bean home interface

Multiple entity beans can share the same home and remote/local interfaces, even if one entity bean uses container-managed persistence and the other uses bean-managed persistence. Both `Savings` and `Checking` entity beans use the same home interface, `AccountHome`. They also use the same `Account` remote interface.

The home interface for an entity bean is very much like the home interface for a session bean. They extend the same `javax.ejb.EJBHome` or `javax.ejb.EJBLocalHome` interface. The home interface for entity beans must include at least one finder method. A `create()` method is optional.

Here is the code for the `AccountHome` interface:

```
public interface AccountHome extends javax.ejb.EJBHome {
    Account create(String name, float balance)
        throws java.rmi.RemoteException, javax.ejb.CreateException;
    Account findByPrimaryKey(AccountPK primaryKey)
        throws java.rmi.RemoteException, javax.ejb.FinderException;
    java.util.Enumeration findAccountsLargerThan(float balance)
        throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

The `AccountHome` home interface implements three methods. While the `create()` method is not required for entity beans, the bank example does implement one. The `create()` method inserts a new entity bean object into the underlying database. You could choose to defer the creation of new entity objects to the DBMS or to another application, in which case you would not define a `create()` method.

The `create()` method requires two parameters, an account name string and a balance amount. The implementation of this method in the entity bean class uses these two parameter values to initialize the entity object state—the account name and the starting balance—when it creates a new object. The `throws` clause of a `create()` method must include the `java.rmi.RemoteException` and the `java.ejb.CreateException`. It can also include additional application-specific exceptions.

Entity beans must have the `findByPrimaryKey()` method, so the `AccountHome` interface declares this method. It takes one parameter, the `AccountPK` primary key class, and returns a reference to the `Account` remote interface. This method finds one particular account entity and returns a reference to it.

Although it's not required, the home interface also declares a second finder method, `findAccountsLargerThan()`. This method returns a Java `Enumeration` containing all the accounts whose balance is greater than some amount.

The entity bean remote interface

More than one entity bean can use the same remote/local interface, even when the beans use different persistence management strategies. The bank example's two entity beans both use the same `Account` remote interface.

The remote interface extends the javax.ejb.EJBObject interface and exposes the business methods that are available to clients. Here is the code:

```
public interface Account extends javax.ejb.EJBObject {  
    public float getBalance() throws java.rmi.RemoteException;  
    public void credit(float amount) throws java.rmi.RemoteException;  
    public void debit(float amount) throws java.rmi.RemoteException;  
}
```

An entity bean with container-managed persistence

The bank example implements a `Checking` entity bean that illustrates the basics for using 1.1 container-managed persistence. In many ways, this implementation is like a session bean implementation. There are some key things to note in the implementation of an entity bean that uses container-managed persistence, however:

- The entity bean has no implementations for finder methods. The EJB container provides the finder method implementations for entity beans with container-managed persistence. Rather than providing the implementation for the finder methods in the bean's class, the deployment descriptor contains information that enables the container to implement these finder methods.
- The entity bean declares all fields `public` that are managed by the container for the bean. The `Checking` bean declares `name` and `balance` to be `public` fields.
- The entity bean class implements the methods declared in the `EntityBean` interface: `ejbActivate()`, `ejbPassivate()`, `ejbLoad()`, `ejbStore()`, `ejbRemove()`, `setEntityContext()`, and `unsetEntityContext()`. The entity bean is required to provide skeletal implementations of these methods only, however, although it is free to add application-specific code where it is appropriate. The `CheckingAccount` bean saves the context returned by `setEntityContext()` and releases the reference in `unsetEntityContext()`. Otherwise, it adds no additional code to the `EntityBean` interface methods.
- `Checking` includes an implementation of the `ejbCreate()` method because this enterprise bean allows callers to create new checking accounts. The implementation also initializes the instance's two variables, `name` and `balance`, to the parameter values. `ejbCreate()` returns a null value because, with container-managed persistence, the container creates the appropriate reference to return to the client.
- The `CheckingAccount` bean class provides the minimal implementation of the `ejbPostCreate()` method, although this method could have performed further initialization work if it was needed. For beans with container-managed persistence, you need just a minimal

implementation of ejbPostCreate() because it serves as a notification callback.

```

import javax.ejb.*;
import java.rmi.RemoteException;

public class CheckingAccount implements EntityBean {
    private javax.ejb.EntityContext _context;
    public String name;
    public float balance;

    public float getBalance() {
        return balance;
    }

    public void debit(float amount) {
        if(amount > balance) {
            // mark the current transaction for rollback ...
            _context.setRollbackOnly();
        }
        else {
            balance = balance - amount;
        }
    }

    public void credit(float amount) {
        balance = balance + amount;
    }

    public AccountPK ejbCreate(String name, float balance) {
        this.name = name;
        this.balance = balance;
        return null;
    }

    public void ejbPostCreate(String name, float balance) {}
    public void ejbRemove() {}
    public void setEntityContext(EntityContext context) {
        _context = context;
    }

    public void unsetEntityContext() {
        context = null;
    }

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public String toString() {
        return "CheckingAccount[name=" + name + ",balance=" + balance + "]";
    }
}

```

An entity bean with bean-managed persistence

The bank example also implements a `Savings` bean, an entity bean with bean-managed persistence. The `Savings` bean accesses a different account table than the `Checking` bean. Although these two entity beans use different persistence-management approaches, they can both use the same home and remote interfaces. There are differences between the two implementations, however.

An entity bean implementation with bean-managed persistence does the following:

- It can declare its instance variables to be private rather than public.

The bean includes code to access these variables, to load the database values into these instance variables, and to store their changes to the database. As such, the bean can limit access to these variables as it sees fit. This differs from a bean using container-managed persistence, which must declare all container-managed variables to be public so that the container can access them.

- The `ejbCreate()` method returns the primary key class.

In the `SavingsAccount` bean class the class is `AccountPK`. The container takes the returned primary key class and uses it to construct a remote reference to the entity bean instance.

- Just like beans with container-managed persistence, a bean with bean-managed persistence may optionally provide more than an empty implementation of the `ejbCreate()` method.

The `SavingsAccount` bean class doesn't need to include additional initialization code in this method.

- It has implementations for the `ejbLoad()` and `ejbStore()` methods.

A bean using container-managed persistence usually provides just an empty implementation of these methods because the container handles persistence. An entity bean with bean-managed persistence must provide its own code to read the database values into its instance variables in the `ejbLoad()` method, and to write to the database with changed values in the `ejbStore()` method.

- It has implementations for all finder methods.

The `SavingsAccount` entity bean class implements two finder methods, the required `ejbFindByPrimaryKey()` method, and the optional `ejbFindAccountsLargerThan()` method.

- An entity bean with bean-managed persistence must implement the `ejbRemove()` method.

Because the bean is managing the underlying database entity object, it must implement this method so that it can remove the entity object from the database. A bean with container-managed persistence will omit the implementation of this method because the container is responsible for the database management.

- Each method that accesses the underlying database object must include the correct database access code.

These methods are `ejbCreate()`, `ejbRemove()`, `ejbLoad()`, `ejbStore`, `ejbFindByPrimaryKey()`, all other finder methods, and the business methods. Each method contains code to connect to the database, followed by code to build and then execute SQL statements that accomplish the functionality encompassed by the method. When the SQL statements complete, the method closes the statements and the database connection before returning.

The following code sample shows the interesting code portions from the `SavingsAccount` implementation class. The example removes the code that is merely the empty implementations of the `EntityBean` interfaces methods, such as `ejbActivate()`, `ejbPassivate()`, and so on.

First look at the `ejbLoad()` method, which accesses the database entity object, to see how a bean with bean-managed persistence implements database access. Note that all of the methods implemented in the `SavingsAccount` class follow the same approach as `ejbLoad()` uses. The `ejbLoad()` method begins by establishing a connection to the database. It calls the internal `getConnection()` method, which uses a `DataSource` to obtain a JDBC connection to the database from a JDBC connection pool. Once the connection is established, `ejbLoad()` creates a `PreparedStatement` object and builds its SQL database access statement. Because `ejbLoad()` reads the entity object values into the entity bean's instance variables, it builds an SQL SELECT statement for a query that selects the balance value for the savings account whose name matches a pattern. The method then executes the query. If the query returns a result, it extracts the balance amount. The `ejbLoad()` method finished by closing the `PreparedStatement` objects and then closing the database connection. Note that the `ejbLoad()` method doesn't actually close the connection. Instead, it simply returns the connection to the connection pool.

```

import java.sql.*;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;

public class SavingsAccount implements EntityBean {
    private EntityContext _context;
    private String _name;
    private float _balance;
}

```

A bank entity bean example

```
public float getBalance() {
    return _balance;
}

public void debit(float amount) {
    if(amount > balance) {
        // mark the current transaction for rollback...
        _context.setRollbackOnly();
    } else {
        _balance = _balance - amount;
    }
}
public void credit(float amount) {
    _balance = _balance + amount;
}

// setEntitycontext(), unsetEntityContext(), ejbActivate(), ejbPassivate(),
// ejbPostCreate() skeleton implementations are not shown here
...

public AccountPK ejbCreate(String name, float balance)
    throws RemoteException, CreateException {
    _name = name;
    _balance = balance;
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement(
            ("INSERT INTO Savings_Accounts (name, balance) VALUES (?,?)"));
        statement.setString(1, _name);
        statement.setFloat(2, _balance);
        if(statement.executeUpdate() != 1) {
            throw new CreateException("Could not create: " + name);
        }
        statement.close();
        connection.close();
        return new AccountPK(name);
    } catch(SQLException e) {
        throw new RemoteException("Could not create: " + name, 3);
    }
}
...
public void ejbRemote() throws RemoteException, RemoveException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement(
            ("DELETE FROM Savings_Account WHERE name = ?"));
        statement.setString(1, _name);
        if(statement.executeUpdate() != 1) {
            throw new RemoteException("Could not remove: " + _name, e);
        }
        statement.close();
        connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not remove: " + _name, e);
    }
}
public AccountPK ejbFindByPrimaryKey(AccountPK key) throws RemoteException,
    FinderException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement(
            ("SELECT name FROM Savings_Accounts WHERE name = ?"));
        statement.setString(1, key.name);
        ResultSet resultSet = statement.executeQuery();
        if(!resultSet.next()) {
            throw new FinderException("Could not find: " + key

```

```

        statement.close();
        connection.close();
        return key;
    } catch(SQLException e) {
        throw new RemoteException("Could not find: " + key, e);
    }
}

public java.util.Enumeration ejbFindAccountsLargerThan(float balance)
    throws RemoteException, FinderException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
            ("SELECT name FROM Savings_Account WHERE balance > ?");
        statement.setFloat(1, balance);
        ResultSet resultSet = statement.executeQuery();
        Vector keys = new Vector();
        while(resultSet.next()) {
            String name = resultSet.getString(1);
            keys.addElement(new AccountPK(name));
        }
        statement.close();
        connection.close();
        return keys.elements();
    } catch(SQLException e) {
        throw new RemoteException("Could not findAccountsLargerThan: " + balance, e);
    }
}

public void ejbLoad() throws RemoteException {
    // get the name from the primary key
    _name = (AccountPK) _context.getPrimaryKey().name;
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
            ("SELECT balance FROM Savings_Account WHERE name = ?");
        statement.setString(1, _name);
        ResultSet resultSet = statement.executeQuery();
        if(!resultSet.next()) {
            throw new RemoteException("Account not found: " + _name);
        }
        _balance = resultSet.getFloat(1);
        statement.close();
        connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not load: " + _name, e);
    }
}

public void ejbStore() throw RemoteException {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement
            ("UPDATE Savings_Accounts SET balance = ? WHERE name = ?");
        statement.setFloat(1, _balance);
        statement.setString(2, _name);
        statement.executeUpdate();
        statement.close();
        connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not store: " + _name, e);
    }
}

private Connection getConnection() throws SQLException {
    Properties properties = _context.getEnvironment();

```

A bank entity bean example

```
String url = properties.getProperty("db.url");
String username = properties.getProperty("db.username");
String password = properties.getProperty("db.password");
if(username != null) {
    return DriverManager.getConnection(url, username, password);
} else {
    return DriverManager.getConnection(url);
}
}

public String toString() {
    return "SavingsAccount[name=" + _name + ",balance=" + _balance + "]";
}
```

The primary key class

Both the `CheckingAccount` and `SavingsAccount` bean classes use the same field to uniquely identify a particular account record. In this case, they both use the same primary key class, `AccountPK`, to represent the unique identifier for either type of account:

```
public class AccountPK implements java.io.Serializable {
    public String name;
    public AccountPK() {}
    public AccountPK(String name) {
        this.name = name;
    }
}
```

The deployment descriptor

The deployment descriptor for the bank example deploys three kinds of beans: the `Teller` session bean, the `Checking` entity bean with container-managed persistence, and the `Savings` entity bean with bean-managed persistence.

You use properties in the deployment descriptor to specify information about the entity bean's interfaces, transaction attributes, and so on, just as you do for session beans. But you also add additional information that is unique to entity beans.

The bean-managed XML code sample shows typical deployment descriptor property tags for an entity bean with bean-managed persistence. This container-managed XML code sample illustrates the typical deployment descriptor tags for an entity bean that uses container-managed persistence. When you compare the descriptor tags for the two types of entity beans, you'll notice that the deployment descriptor for an entity bean with container-managed persistence is more complex.

The bean's deployment descriptor type is set to `<entity>`. Notice that the first tags within the `<enterprise-beans>` section in both code samples specify that the bean is an entity bean.

An entity bean deployment descriptor specifies the following type of information:

- The names of the related interfaces (home and remote) and the bean implementation class.
- Each enterprise bean specifies its home interface using the <home> tag, its remote interface using the <remote> tag, and its implementation class name using the <ejb-class> tag.

- The JNDI names under which the entity bean is registered and by which clients locate the bean.

- The bean's transaction attributes and its transaction isolation level.

This usually appears in the <assembly-descriptor> section of the deployment descriptor.

- The name of the entity bean's primary key class.

In this example, the primary key class is `AccountPK` and it appears within the <prim-key-class> tag.

- The persistence used by the bean.

The `CheckingAccount` bean uses container-managed persistence, so the deployment descriptor sets the <persistence-type> tag to Container.

- Whether the bean class is reentrant.

Neither the `SavingsAccount` nor the `CheckingAccount` bean is reentrant, so the <reentrant> tag is set to False for both.

- The fields that the container manages, if the bean uses container-managed persistence.

A bean that uses bean-managed persistence doesn't specify any container-managed fields. Therefore, the deployment descriptor for the `SavingsAccount` bean doesn't specify any container-managed fields. An entity bean using container-managed persistence must specify the names of its fields or instance variables that the container must manage. Use a combination of the <cmp field> and <field name> tags for this. The first tag, <cmp field>, indicates that the field is container-managed. Within this tag, the <fields name> tag specifies the name of the field itself. For example, the `CheckingAccount` bean deployment descriptor indicates that the `balance` field is container-managed as follows:

```
<cmp field><field name>balance</field name></cmp field>
```

Information about the container-managed fields for container-managed beans. The container uses this information to generate the finder methods for these fields.

Deployment descriptor for an entity bean with bean-managed persistence

The following code sample shows the key parts of the deployment descriptor for an entity bean using bean-managed persistence. Because the bean, not the container, handles its own fetches from the database entity values and updates to these values, the descriptor doesn't specify fields for the container to manage. Nor does it tell the container how to implement its finder methods, because the bean's implementation provides those.

```
<enterprise-beans>
<entity>
    <description>This entity bean is an example of bean-managed
        persistence</description>
    <ejb-name>savings</ejb-name>
    <home>AccountHome</home>
    <remote>Account</remote>
    <ejb-class>SavingsAccount</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>AccountPK</prim-key-class>
    <reentrant>False</reentrant>
</entity>
...
</enterprise-beans>
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>savings</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

Deployment descriptor for an entity bean with container-managed persistence

The next code sample shows the key parts of the deployment descriptor for an entity bean using container-managed persistence. Because the bean lets the container handle loading database entity values and updating these values, the descriptor specifies the fields that the container will manage.

```
<enterprise-beans>
<entity>
    <description>This entity bean is an example of container-managed
        persistence</description>
    <ejb-name>checking</ejb-name>
    <home>AccountHome</home>
    <remote>Account</remote>
    <ejb-class>chkingAccount</ejb-class>
    <persistence-type>Container</persistence-type>
```

```
<prim-key-class>AccountPK</prim-key-class>
<reentrant>False</reentrant>
<cmp-field>
    <field-name>name</field-name>
<cmp-field>
</entity>
</enterprise-beans>
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>checking</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute-transaction>
        </container-transaction>
    </assembly-descriptor>
```


15

Developing message-driven beans

Message-driven beans, introduced in the EJB 2.0 specification, greatly simplify message-based programming. The bean provider's primary responsibility is implementing an `onMessage()` method that contains the logic that responds to a message. The EJB container handles all other messaging tasks. Message-driven beans use a Java Message Service (JMS) provider, such as SonicMQ Message Broker that is bundled with JBuilder Enterprise.

A message-driven bean is an enterprise bean that processes JMS messages. Such messages can come from any component that can send JMS messages, such as another enterprise bean, an application client, a web component, or a legacy system. A message-driven bean is a listener to JMS messages, responding when it detects a particular type of message. Much like stateless session beans, message-driven beans are not associated with a single client and they do not have conversational state.

Message-driven beans listen and respond to *asynchronous* messages. When a sender sends an asynchronous message, it does not wait for the receiver of the message to receive and process the message before it continues with its own work. In contrast, the sender of a *synchronous* message waits for the receiver to process the message before program control returns to the sender. Asynchronous messaging permits loose coupling between the message sender (the message producer) and the message receiver (the message consumer, the message-driven bean). This makes message-

driven beans particularly useful for business-to-business interactions and integrating an EJB system with legacy systems.

How message-driven beans work

The deployer of a message-driven bean sets the JMS destination that the bean listens to in the bean's deployment descriptor. The deployment descriptor can also include a message selector filter that associates a message-driven bean with a particular type of message. See the ["Message-driven bean deployment descriptor attributes" on page 15-6](#) for more specific information.

The container activates an instance of the proper type of message-driven bean. The bean instance consumes the message sent to its associated JMS destination. It responds to the message using the logic contained in its `onMessage()` method.

Unlike session and entity beans, message-driven beans have no home/local home or remote/local interfaces. Therefore, a client never directly accesses a message-driven bean. Instead a client sends a message to a JMS destination, which is either a queue or a topic.

The point-to-point message model uses a queue. Multiple message-driven beans can receive messages from the same queue, but only one bean can receive each message. Consumers *pull* messages from the queue in that no messages are received until a consumer requests it.

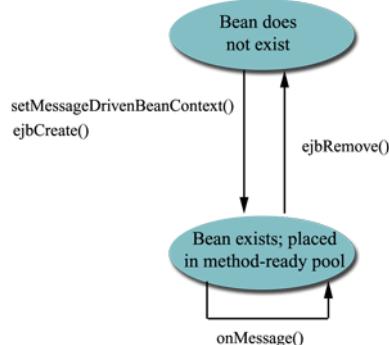
The publish-and-subscribe message model uses topics. One message producer can send a message to many consumers using a topic. Consumers, such as message-driven beans, subscribe to the topic. Messages sent to a topic are delivered to all the topic's subscribers. Therefore, each subscriber receives a copy of the message. Messages are *pushed* to the consumer.

The life of a management-driven bean instance

The life cycle of a message-driven bean is quite simple. A client sends a message to a JMS destination to which the bean is listening. The EJB container creates a new instance of the bean class. It then calls the bean instance's `setMessageDrivenContext()` and `ejbCreate()` methods, in that order. The message-driven bean instance can now consume a message sent to the bean's destination. The life of the bean instance ends when the

EJB container calls the `ejbRemove()` method. The following diagram depicts the life cycle of a message-driven bean instance:

Figure 15.1 Message-driven bean life cycle



Writing a message-driven bean class

Your task of writing a message-driven bean is simplified because you create a bean class only; message-driven beans don't have home/local home and remote/local interfaces.

To create a message-driven bean class,

- 1 Create a class that implements the `javax.ejb.MessageDrivenBean` interface. The class must be defined as `public` and it cannot be defined as `final` nor `abstract`.
- 2 In the bean class, implement the `javax.jms.MessageListener` interface.
- 3 Include a public constructor that takes no arguments. The container calls this constructor to create instances of the message-driven bean class.
- 4 Implement the `ejbCreate()` method that takes no arguments. The method must be declared as `public` and it cannot be `final` or `static`. Its return type must be `void` and it must not define any application exceptions.

Implementing the `MessageDrivenBean` interface

The `MessageDrivenBean` interface defines two methods all message-driven beans must implement. It extends the `EnterpriseBean` interface:

```

package javax.ejb;

public interface MessageDrivenBean extends javax.ejb.EnterpriseBean {
    public void setMessageDrivenContext(MessageDrivenContext context)
        throws EJBException;
    public void ejbRemove() throws EJBException;
}
  
```

The methods of the `MessageDrivenBean` interface are closely associated with the bean's life cycle:

- `setMessageDrivenContext()` - The container calls `setMessageDrivenContext()` to provide the message-driven bean instance with a reference to its `MessageDrivenContext`, which is passed into the method. The container calls this method at the beginning of the bean's life cycle.
- `ejbRemove()` - Called by the container at the end of the bean's life cycle. You can use this method to free any resources allocated in the `ejbCreate()` method.

Implementing the `MessageListener` interface

Only message-driven beans are allowed to implement the `MessageListener` interface. It defines just one method, `onMessage()`:

```
package javax.jms;

public interface MessageListener {
    public void onMessage(Message message);
}
```

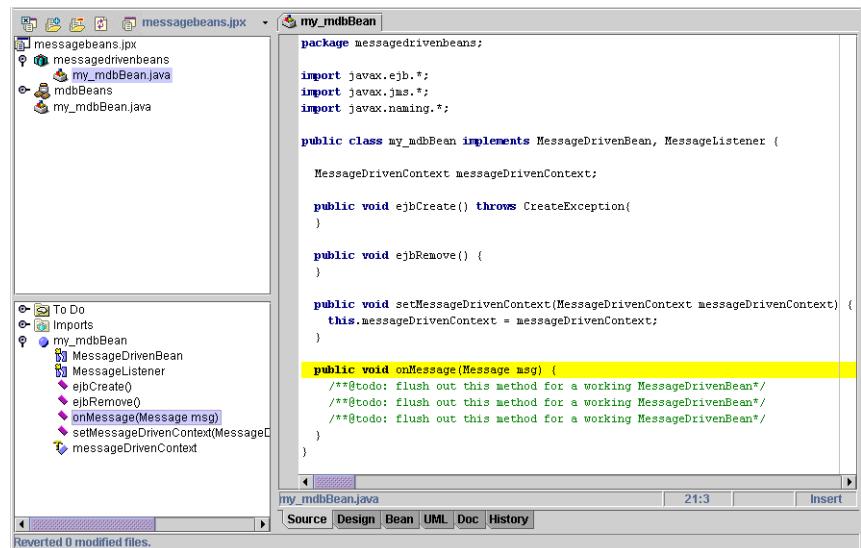
Writing the `onMessage()` method

The heart of a message-driven bean is its `onMessage()` method. Place all the logic that handles an incoming message in `onMessage()`. The message is passed to `onMessage()` in its sole argument. The logic can handle the incoming message itself, it can pass the message on to another bean, or it can send it to another JMS destination. You can implement helper methods in the bean's class that `onMessage()` can call.

How JBuilder can help you create a message-driven bean

You can begin creating a message-driven bean by right-clicking the EJB Designer pane and choosing New Message Bean from the context menu. A message-driven bean representation appears in the EJB Designer. Right-click the bean name in the bean representation, and in the Inspector that appears, set the properties of the bean.

Double-click the message-driven bean source code node in JBuilder's project pane to see the source code generated for you:



```

package messagedrivenbeans;

import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

public class my_mdbBean implements MessageDrivenBean, MessageListener {

    MessageDrivenContext messageDrivenContext;

    public void ejbCreate() throws CreateException {
    }

    public void ejbRemove() {
    }

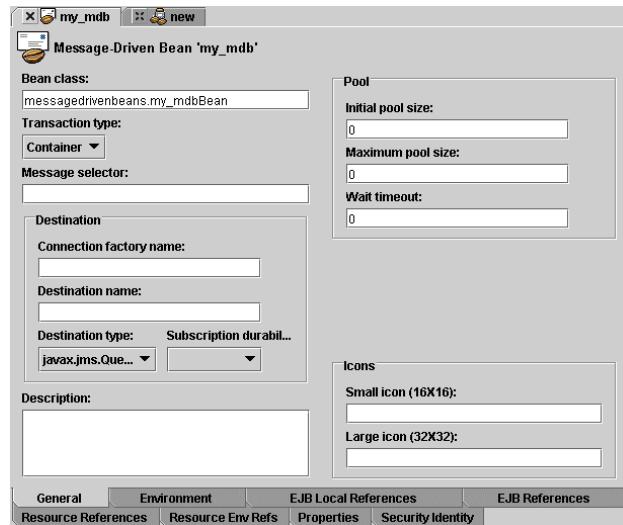
    public void setMessageDrivenContext(MessageDrivenContext messageDrivenContext) {
        this.messageDrivenContext = messageDrivenContext;
    }

    public void onMessage(Message msg) {
        /**@todo: flush out this method for a working MessageDrivenBean*/
        /**@todo: flush out this method for a working MessageDrivenBean*/
        /**@todo: flush out this method for a working MessageDrivenBean*/
    }
}

```

In the source code of the bean class, find the `onMessage()` method and write the logic that processes an incoming message as you see fit. You can add other methods to the bean class that are invoked by the `onMessage()` method.

You can use JBuilder's Deployment Descriptor editor to edit the bean's deployment descriptor. Double-click the message-driven bean's class node in the project pane and click the DD Editor tab at the bottom of the content pane. This is the General panel for a message-driven bean:



The next section explains some of the deployment attributes found on this panel.

Message-driven bean deployment descriptor attributes

Each message-driven bean must be assigned a JMS destination to which the bean listens and consumes messages from. This is a job for the deployer, but the bean provider can enter a JMS destination in the deployment descriptor as information for the deployer. The destination can be a queue or a topic. If it is a queue, then the queue should have a single message-driven bean as its consumer: don't assign a destination to more than one bean.

When the destination is a topic, the subscription-durability attribute must be declared as either durable or nondurable. If a connection is somehow lost between the EJB container and the JMS provider for a durable subscription, messages aren't lost. The JMS provider stores any messages the subscribing bean misses and sends the messages when the connection is established once again. Nondurable subscriptions would result in lost messages if the EJB container-JMS provider connection breaks. The advantage of nondurable subscriptions is improved performance, although using them makes message-driven beans less reliable.

When a message-driven bean executes bean-managed transactions, the acknowledge-mode attribute comes into play. It has two possible values: auto-acknowledge or dups-ok-acknowledge. When auto-acknowledge is specified, the EJB container sends acknowledgment to the JMS provider right after a message is delivered to a message-driven bean instance. The dups-ok-acknowledge value allows the EJB container to delay acknowledgment. In some cases the JMS provider thinks the message wasn't delivered and therefore resends the message, resulting in a duplicate message. If your message-driven beans uses dups-ok-acknowledge, it must be prepared to handle duplicate messages.

A message-driven bean deployment descriptor can include a message selector that allows a message-driven bean to be more selective about the messages it receives from a queue or topic. A message selector consists of an expression that uses Boolean logic. It uses a subset of SQL-92 conditional expression syntax. Here's an example:

```
Department = "547" AND Salary BETWEEN 56000.00 AND 85000.00
```

Remember when specifying a message selector that this information is kept in the bean's deployment descriptor, which is an XML file. In XML, the < and > symbols and other special characters have special meaning. Including them in the message selector means the resulting XML file will

create parsing errors. Therefore, logic which uses these symbols must be placed in a CDATA section. For example,

```
<message-selector>
<![CDATA[
    Total > 100.00
]]>
</message-selector>
```

See the JMS specification on Sun Microsystems' web site at <http://java.sun.com/products/jms/docs.html> for more information.

Using the SonicMQ Message Broker with message-driven beans

SonicMQ Message Broker, a JMS provider, is included with JBuilder Enterprise. To use SonicMQ with a message-driven bean you have developed, follow these steps:

- 1** Click the top of the message-driven bean in the EJB Designer to display the bean's inspector.
- 2** If your message-driven bean listens to a queue, follow these steps:
 - a** Specify the Transaction type as Bean or Container from the drop-down list.
 - b** Specify the Acknowledge Mode as Auto-Acknowledge or Dups-Ok-Acknowledge.
 - c** Specify the Message Selector.
 - d** Specify the Destination Name.
 - e** Choose javax.jms.Queue as the Destination Type from the drop-down list.
 - f** Specify the Connection Factory Name.
 - g** Set the Initial Pool size. For example, you might set it to 2.
 - h** Set the Maximum Pool Size. For example, you might set it to 20.
 - i** Specify the Connection Factory Name.
- 3** If your message-driven bean subscribes to a topic, follow these steps:
 - a** Specify the Transaction type as Bean or Container from the drop-down list.
 - b** Specify the Destination Name.
 - c** Specify the Message Selector.
 - d** Choose javax.jms.Topic as the Destination Type from the drop-down list.

- e Specify the Durability of the message as Durable or Nondurable from the drop-down list.
- f Set the Initial Pool size. For example, you might set it to 2.
- g Set the Maximum Pool Size. For example, you might set it to 20.
- h Specify the Connection Factory Name.

The names you specify for the connection factory name and the destination name must be unique names in the JNDI tree. For more information about filling in these fields in the bean's inspector, see the ["General panel" on page 11-5](#).

For more information about SonicMQ, see the SonicMQ documentation.

You can also use the Message Driven Bean panel of the Deployment Descriptor editor to specify your settings instead of the bean's inspector.

Compile your project. For information about compiling, see [Chapter 8, "Compiling enterprise beans and creating a deployment module."](#)

Once you've specified your settings using the bean's inspector or the Deployment Descriptor editor and compiled it, start SonicMQ by choosing Tools | Sonic MQ Broker. Also start Borland Enterprise Server Management Agent by choosing Tools | Borland Enterprise Server Management Agent. Now you are ready to run the bean. The quickest way to do that is to right-click its EJB module or the JAR file the module contains and select Run Using Defaults or Debug Using Defaults from the context menu. The EJB container starts up and runs your bean.

16

Creating the home and remote/local interfaces

An enterprise bean provider must create at least two interfaces for each session and entity bean. For a EJB 1.x bean you must create a home interface and a remote interface. An EJB 2.0 bean can have a remote home and a remote interface, and it can also have a local home and local interface instead of or in addition to the remote home and remote interfaces. The remote home and remote interfaces provide the client a remote view of the bean, while the local home and local interfaces provide the client a local view. The remote and/or local home interface defines the methods a client application uses to create, locate, and destroy instances of an enterprise bean. The remote/local interface defines the business methods implemented in the bean. A client accesses these methods through the remote/local interface.

Creating the home interface

The home interface of an enterprise bean, whether remote or local, controls the bean's life cycle. It contains the definition of the methods to create, find, and remove an instance of an enterprise bean.

As a bean provider, you must define the home interface, but you don't implement it. The EJB container does that, generating a home object that returns a reference to the bean.

An enterprise bean's client can have a remote view or a local view of the bean. A remote bean has a home interface that extends the `EJBHome` interface. A local bean has a home interface that extends the `EJBLocalHome` interface.

The EJBHome interface

Each remote home interface extends the `javax.ejb.EJBHome` interface. Here is the complete definition of `EJBHome`:

```
package javax.ejb
public interface EJBHome extends java.rmi.Remote {
    void remove(Handle handle) throws java.rmi.RemoteException, RemoveException;
    void remove(Object primaryKey) throws java.rmi.RemoteException, RemoveException;
    EJBMetaData getEJBMetaData() throws RemoteException;
    HomeHandle getHomeHandle() throws RemoteException;
}
```

`EJBHome` has two `remove()` methods to remove enterprise bean instances. The first `remove()` method identifies the instance by a handle; the second by a primary key.

A handle, a serializable bean object identifier, has the same lifetime as the enterprise bean object it's referencing. For a session bean, the handle exists only as long as the session does. For an entity bean, the handle can persist and a client can use a serialized handle to reestablish a reference to the entity object it identifies.

A client would use the second `remove()` method to remove an entity bean instance using its primary key.

The `getEJBMetaData()` returns the `EJBMetaData` interface of the enterprise bean object. This interface allows the client to obtain metadata information about the bean. Its purpose is to be used by development tools that build applications that use deployed enterprise beans.

Note that the `EJBHome` interface doesn't have any methods for creating or locating instances of an enterprise bean. Instead, you must add these methods to the home interfaces you develop for your beans. Because session beans and entity beans have different life cycles, the methods defined in their home interfaces differ.

The LocalHome interface

Each local home interface implements `javax.ejb.EJBLocalHome`. The local home interface has just one method, a single `remove()` method:

```
package javax.ejb
public interface EJBLocalHome {
    void remove(Object primaryKey) throws RemoveException, EJBException;
}
```

Creating a home or local home interface for a session bean

A session bean almost always has a single client (except occasionally for stateless session beans). When a client creates a session bean, that session bean instance exists for the use of that client only.

To create a remote home interface for a session bean,

- Declare a home interface that extends `javax.ejb.EJBHome`.
- Add a `create()` method signature for each `ejbCreate()` method in the bean, matching the number and type of arguments exactly.

To create a local home interface for a session bean,

- Declare a local home interface that extends `javax.ejb.EJBLocalHome`.
- Add a `create()` method signature for each `ejbCreate()` method in the bean, matching the number and type of arguments exactly.

When you use JBuilder's EJB tools, JBuilder creates a home interface with one defined `create()` method at the same time it creates the enterprise bean class. You must then add additional `create()` methods to the home interface if you add additional `ejbCreate()` methods to your bean.

If you are using the EJB Designer to create EJB 2.0 enterprise beans, an additional `create()` method is added as soon as you use the EJB Designer to add a new `ejbCreate()` method to the bean class. You don't have to take any extra steps.

If you have an existing 1.x enterprise bean class, use JBuilder's EJB 1.x Interfaces wizard to create a home and remote interface with signatures that match appropriately those in your bean class. For more information, see [Chapter 6, "Creating EJB 1.x components with JBuilder."](#)

If you choose to begin your EJB development for 1.x beans by creating a remote interface first, you can use the EJB 1.x Bean Generator to create a skeleton bean class and the home interface. For more information about using the EJB 1.x Bean Generator, see ["Generating the bean class from a remote interface" on page 6-12](#).

create() methods in session beans

A session bean remote home or local home interface functions as a session bean factory, because it must define one or more `create()` methods. When the client calls `create()`, a new bean instance is created. According to the EJB specification, each `create()` method defined in the remote home or local home interface must

- Return the bean's remote interface type if you are creating a home interface, and return the bean's local interface type if you are creating a local home interface.
- Be named `create()`.
- Match an `ejbCreate()` method in the session bean class. The number and types of arguments for each `create()` method must match its corresponding `ejbCreate()` method in the session bean class.

- Throw the exception `java.rmi.RemoteException` if a home interface is used. For a local home interface, the `create()` method must **not** throw `RemoteException`.
- Throw the exception `javax.ejb.CreateException`.
- Use its parameters, if there are any, to initialize the new session bean object.

You can use JBuilder's EJB tools to ensure that these rules are followed.

The following code sample shows two possible `create()` methods of a session home interface. The parts shown in bold are required:

```
public interface AtmHome extends javax.ejb.EJBHome {
    Atm create()
        throws java.rmi.RemoteException, javax.ejb.CreateException;
    Atm create(Profile preferredProfile)
        throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

Creating a remote home or local home interface for an entity bean

An entity bean is designed to serve multiple clients. When a client creates an entity bean instance, any other client can use it also.

To create a remote home interface for an entity bean,

- Declare an interface that extends `javax.ejb.EJBHome`.
- Add a `create()` method signature for each `ejbCreate()` method in the bean, matching the signatures exactly.
- Add a finder method signature for each finder method in the bean, matching the signatures exactly.

To create a local home interface for an entity bean,

- Declare an interface that extends `javax.ejb.EJBLocalHome`.
- Add a `create()` method signature for each `ejbCreate()` method in the bean, matching the signatures exactly.
- Add a finder method signature for each finder method in the bean, matching the signatures exactly.

When you use JBuilder's EJB tools, JBuilder creates a remote home and/or local home interface with one defined `create()` method at the same time it creates the enterprise bean class. You can then add additional `create()` methods to the home and/or local home interface if you add additional `ejbCreate()` methods to your bean. If you are using the EJB Designer to create 2.0 entity beans, adding a new `ejbCreate()` method automatically

adds a properly formed `create()` method in the home or local home interface.

If you have an existing 1.x enterprise bean class, use JBuilder's EJB 1.x Interfaces wizard to create a home and remote interface with signatures that match appropriately those in your bean class. For more information, see [Chapter 6, "Creating EJB 1.x components with JBuilder."](#)

If you choose to begin your EJB development for 1.x beans by creating a remote interface first, you can use the EJB 1.x Bean Generator to create a skeleton bean class and the home interface. For more information about using the EJB 1.x Bean Generator, see ["Generating the bean class from a remote interface" on page 6-12.](#)

create() methods for entity beans

Like a remote home or local home interface for a session bean, a remote home or local home interface for an entity bean must define one or more `create()` methods. According to the EJB specification, each `create()` method you define must

- Throw the exception `java.rmi.RemoteException` for a home interface. For a local home interface, the `create()` method must **not** throw `RemoteException`.
- Throw the exception `javax.ejb.CreateException`.
- Return the remote interface type of the entity bean if you are creating a home interface, and return the local interface if you are creating a local home interface.
- Be named `create()`.
- Match a `ejbCreate()` method in the session bean class. The number and types of arguments for each `create()` method must match its corresponding `ejbCreate()` method in the session bean class.
- Must include in the exceptions in the throws clause all the exceptions thrown by the corresponding `ejbCreate()` and `ejbPostCreate()` methods in the entity bean class. In other words, the set of exceptions for the `create()` method must be a superset of the union of exceptions for both the `ejbCreate()` and `ejbPostCreate()` methods. The return type of the `ejbCreate()` method is the primary key class.
- Use its parameters, if there are any, to initialize the new entity bean object.

Finder methods for entity beans

Because entity beans usually have long lives and can be used by multiple clients, an entity bean instance probably already exists when a client application needs it. In this case, the client doesn't need to create an entity bean instance, but it does need to locate the appropriate existing one.

That's why the remote home and/or local home interface of an entity bean defines one or more finder methods.

Session beans don't need finder methods because they serve one client, the application that created the bean. The client has no need to find the session bean instance—it already knows where the instance is.

Each entity bean remote home or local home interface must define the default finder method, `findByPrimaryKey()`. It allows a client to locate an entity object using a primary key. This is the `findByPrimaryKey()` method for a home interface:

```
<entity bean's remote interface> findByPrimaryKey(<primary key type> key)
    throws java.rmi.RemoteException, FinderException;
```

This is the `findByPrimaryKey()` method for a local home interface:

```
<entity bean's local interface> findByPrimaryKey(<primary key type> key)
    throws FinderException;
```

`findByPrimaryKey()` has a single argument, the primary key. Its return type is the entity bean's remote or local interface. In the bean's deployment descriptor, you tell the container the type of the primary key. `findByPrimaryKey()` always returns a single entity object.

You can define additional finder methods in the remote home and/or local home interface. Each finder method must have a corresponding implementation in the entity bean class for bean-managed persistence. For container-managed persistence, the container implements the finder methods. Each finder method must follow these conventions:

- In a remote home interface, the return type is the remote interface type, or for finder methods that return more than one entity object, a collection type that has the remote interface type as the content type. In a local home interface, the return type is the local interface type, or for finder methods that return more than one entity object, a collection type that has the local interface type as the content type. Valid Java collection types are `java.util.Enumeration` and, for EJB 2.0 beans, `java.util.Collection`.
- The finder method always starts with the prefix **find**. The corresponding finder method in the entity bean class with bean-managed persistence begins with the prefix **ejbFind**.
- The method must throw the exception `java.rmi.RemoteException` if it is defined in a home interface. Finders in a local home interface must **not** throw `RemoteException`.
- The method must throw the exception `javax.ejb.FinderException`.
- The throws clause of the finder method in the remote home/local home interface must match the throws clause of the corresponding `ejbFind<xxx>` method in the entity bean class.

The following sample home interface contains two `create()` methods and two finder methods. The parts shown in bold are required:

```
public interface AccountHome extends javax.ejb.EJBHome {
    Account create(String accountID)
        throws java.rmi.RemoteException, javax.ejb.CreateException;
    Account create(String accountID, float initialBalance)
        throws java.rmi.RemoteException, javax.ejb.CreateException;
    Account findByPrimaryKey(String key)
        throws java.rmi.RemoteException, javax.ejb.FinderException;
    Account findBySocialSecurity(String socialSecurityNumber)
        throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

Note that an EJB 2.0 component could also include home business methods. For more information, see “[Adding a home business method](#)” on page 4-22.

Creating the remote or local interface

The remote or local interface you create for your enterprise bean describes the business methods a client application can call. While you define the methods in the remote or local interface, you implement these same methods in the enterprise bean class. The clients of an enterprise bean never access the bean directly. They access its methods through its remote or local interface.

To create a remote interface,

- Declare an interface that extends `javax.ejb.EJBObject`.
- Declare in the remote interface every business method you want a client application to be able to call in the enterprise bean, matching the signatures exactly with those in the bean class.

To create a local interface,

- Declare an interface that extends `javax.ejb.EJBLocalObject`.
- Declare in the local interface every business method you want a client application to be able to call in the enterprise bean, matching the signatures exactly with those in the bean class.

When you use JBuilder’s EJB Designer, JBuilder creates a local interface that extends `EJBLocalObject` for you.

Each method defined in the remote or local interface must follow these rules, which are the same for both session and entity beans:

- It must be public.
- It must throw the exception `java.rmi.RemoteException` if its a method in a remote interface. It must not throw `RemoteException` if its a method in a local interface.
- A method must exist in the remote/local interface for each method in the enterprise bean's class you want a client to be able to call. The methods in the remote/local interface and in the bean itself must have the same name, the same number and types of arguments, the same return type, and they must throw the same exceptions, or a subset of the remote/local interface method's exceptions.

The following code sample shows the code for a sample remote interface called `Atm` for an ATM session bean. The `Atm` remote interface defines a business method called `transfer()`. The parts shown in bold are required:

```
public interface Atm extends javax.ejb.EJBObject{
    public void transfer(String source, String target, float amount)
        throws java.rmi.RemoteException, InsufficientFundsException;
}
```

The `transfer()` method declared in the `Atm` interface throws two exceptions: the required `java.rmi.RemoteException` and `InsufficientFundsException`, which is an exception specific to an application.

The EJBObject and EJBLocal Object interfaces

The remote interface extends the `javax.ejb.EJBObject` interface. Here is the source code for `EJBObject`:

```
package javax.ejb;
public interface EJBObject extends java.rmi.Remote {
    public EJBHome getEJBHome() throws java.rmi.RemoteException;
    public Object getPrimaryKey() throws java.rmi.RemoteException;
    public void remove() throws java.rmi.RemoteException, RemoveException;
    public Handle getHandle() throws java.rmi.RemoteException;
    boolean isIdentical (EJBObject other) throws java.rmi.RemoteException;
}
```

The `getEJBHome()` method allows an application to obtain the bean's home interface. If the bean is an entity bean, the `getPrimaryKey()` method returns the primary key for the bean. The `remove()` method deletes the enterprise bean. `getHandle()` returns a persistent handle to the bean instance. Use `isIdentical()` to compare two enterprise beans.

The EJB container creates an `EJBObject` for the enterprise bean. Because the remote interface extends the `EJBObject` interface, the `EJBObject` the container creates includes implementations for all the methods the `EJBObject`

interface as well as all the business methods you define in the remote interface. The instantiated EJBObject is visible over the network and it acts as a proxy for the bean. It has a stub and a skeleton. The bean itself is not visible over the network.

For completeness, here is the EJBLocalObject interface:

```
package javax.ejb;
public interface EJBLocalObject {
    public EJBHome getEJBLocalHome() throws EJBException;
    public Object getPrimaryKey() throws EJBException;
    public void remove() throws RemoveException, EJBException,;
    boolean isIdentical (EJBLocalObject other) throws EJBException;
}
```


Developing enterprise bean clients

A client of an enterprise bean is an application, a stand-alone application, a servlet, an applet, or another enterprise bean. In all cases, the client must do the following things to use an enterprise bean:

- Locate the bean's home interface.

The EJB specification states that the client should use the JNDI (Java Naming and Directory Interface) API to locate home interfaces.

- Get a reference to an enterprise bean object's remote/local interface.

This involves using methods defined on the bean's home interface. The methods allows you to you create a session bean, or to create or find an entity bean.

- Call one or more methods defined by the enterprise bean.

A client doesn't directly call the methods defined by the enterprise bean. Instead, the client calls the methods of the enterprise bean object's remote or local interface. The methods defined in the remote or local interface are the methods that the enterprise bean has exposed to clients.

The following sections describe the client application `SortClient.java`, that calls the sample `SortBean` session bean. `SortBean` is a stateless session bean that implements a merge/sort algorithm. Here is the code of `SortClient`:

```
// SortClient.java
...
public class SortClient {
    ...
    public static void main(String[] args) throws Exception {
        javax.naming.Context context;
        { // get a JNDI context using the Naming service
            context = new javax.naming.InitialContext();
        }
        Object objref = context.lookup("sort");
        SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow(objref,
            SortHome.class);
        Sort sort = home.create();
        ... //do the sort and merge work
        sort.remove();
    }
}
```

Locating the home interface

`SortClient` imports the required JNDI classes and the `SortBean` home and remote interfaces. The client uses the JNDI API to locate an enterprise bean's home interface. First the client must obtain a JNDI initial naming context. The code for `SortClient` instantiates a new `javax.naming.Context` object, which is called `InitialContext`. The client then uses the context `lookup()` method to resolve the name to a home interface.

The context's `lookup()` method returns an object of type `java.lang.Object`. Your code must cast this returned object to the expected type. The `SortClient` code shows a portion of the client code for the sort example. The `main()` routine begins by using the JNDI naming service and its context `lookup()` method to locate the home interface. You pass the name of the remote interface, which in this case is `sort`, to the `context.lookup()` method. Note that the program eventually casts the results of the `context.lookup()` method to `SortHome`, the type of the home interface.

Getting the remote/local interface

Once you have the home interface of an enterprise bean, you need a reference to the bean's remote or local interface. To do this, use the home interface's `create` or `finder` methods. Which method to use depends on the type of the enterprise bean and the methods the bean provider has defined in the home interface.

Session beans

If the enterprise bean is a session bean, the client uses a create method to return the remote or local interface. Session beans don't have finder methods. If the session bean is stateless, it will have just one `create()` method, so that is the one the client must call to obtain the remote or local interface. The default `create()` method has no parameters. So for the SortClient code sample, the call to the get the remote or local interface looks like this:

```
Sort sort = home.create();
```

The cart example discussed in [Chapter 21, “Tutorial: Developing a session bean with the EJB Designer”](#) uses a stateful session bean. Its `create()` method takes three parameters, which together identify the purchaser of the cart contents, and returns a reference to the Cart remote interface. The CartClient sets values for the three parameters: `cardHolderName`, `creditCardNumber`, and `expirationDate`. Then it calls the `create()` method. Here's the code:

```
...
String cardHolderName = "Suzy Programmer";
String creditCardNumber = "1234-5678-9012-3456";
Date expirationDate = new GregorianCalendar(2004, Calendar.JULY, 30).getTime();

Cart cart = null;

try {
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
} catch (Exception e) {
    System.out.println("Could not create Cart session bean\n" + e);
}
```

Entity beans

If your bean is an entity bean, use either a create or a finder method to obtain the remote or local interface. Because an entity object represents some underlying data stored in a database, and that data is persistent, entity beans usually exist for a long time. Therefore, the client most often needs to simply find the entity bean that represents that data rather than create a new entity object, which would create and store new data in the underlying database.

A client uses a find operation to locate an existing entity object, such as a specific row within a relational database table. That is, find operations locate data entities that have previously been inserted into data storage. The data might have been added to the data store by an entity bean, or it might have been added outside of the EJB context, such as directly from within the database management system (DBMS). Or, in the case of legacy systems, the data might have existed prior to the installation of the EJB container.

A client uses an entity bean object's `create()` method to create a new data entity that will be stored in the underlying database. An entity bean's `create()` method inserts the entity state into the database, initializing the entity's variables according to the values in the `create()` method's parameters.

Each entity bean instance must have a primary key that uniquely identifies it. An entity bean instance can also have secondary keys that can be used to locate a particular entity object.

Finder methods and the primary key class

The default finder method for an entity bean is `findByPrimaryKey()`, which locates the entity object using its primary key value. This is its signature:

```
public <remote interface> findByPrimaryKey(<key type> primaryKey)
```

Each entity bean must implement a `findByPrimaryKey()` method. The `primaryKey` parameter is a separate primary key class that is defined in the deployment descriptor. The key type is the type for the primary key, and it must be a legal value type in RMI-IIOP. The primary key class can be any class, such as a Java class or a class you've written yourself.

For example, suppose you have an `Account` entity bean for which you've defined the primary key class `AccountPK`. `AccountPK`, a `String` type, holds the identifier for the `Account` bean. To obtain a reference to a specific `Account` entity bean instance, set the `AccountPK` to the account identifier and call the `findByPrimaryKey()` method as shown here:

```
AccountPK accountPK = new AccountPK("1234-56-789");
Account source = accountHome.findByPrimaryKey(accountPK);
```

Bean providers can define additional finder methods that a client can use.

For more information about using the EJB Designer to add finder methods to an EJB 2.0 entity bean, see "[Adding a finder method](#)" on page 4-20.

Create and remove methods

A client can also create entity beans using create methods defined in the home interface. When a client invokes a create method for an entity bean, the new instance of the entity object is saved in the data store. The new entity object always has a primary key value that is its identifier. Its state can be initialized to values passed as parameters to the create method.

Keep in mind that an entity bean exists for as long as data is present in the database. The life of the entity bean isn't bound by the client's session. The entity bean can be removed by calling one of the bean's remove methods. These methods remove the bean and the underlying representation of the entity data from the database. It's also possible to directly delete an entity object, such as by deleting a database record using the DBMS or with a legacy application.

Calling methods

Once the client has a reference to the bean's remote or local interface, it can invoke the methods defined in the remote or local interface for the bean. The client is most interested in the methods that embody the bean's business logic.

For example, the following is some code from a client that accesses the `Cart` session bean. The code shown here begins from the point where it has created a new session bean instance for a card holder and retrieved a `Cart` reference to the remote interface. The client is ready to invoke the bean methods:

```

...
Cart cart = null;

// obtain a reference to the bean's remote interface
try {
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
} catch (Exception e) {
    System.out.println("Could not create Cart session bean\n" + e);
}

if (cart != null) {

    Item kassemBook = new Item("J2EE Blueprints", 39.99f, "Book");

    try {
        cart.addItem(kassemBook);
    } catch (Exception e) {
        System.out.println("Could not add the book to the cart\n" + e);
    }
    ...
}

...
// list the items currently in the cart
summarize(cart);
cart.removeItem(kassemBook);
...

```

First the client creates a new item object, setting its `title`, `price`, and `type` parameters. Next it invokes the enterprise bean business method `addItem()` to add the item object to a shopping cart. The `Cart` session bean defines the `addItem()` method, and the `Cart` remote interface makes it public. The client adds other items (these aren't shown here), then calls its own `summarize()` method to list the items in the shopping cart. This is followed by the `remove()` method to remove the bean instance. Note that a client calls the enterprise bean methods in the same way that it invokes any method, such as its own `summarize()` method.

To see a tutorial about creating this client application that accesses the `Cart` session bean, see [Chapter 22, “Tutorial: Creating a test client application.”](#)

Removing bean instances

The `remove()` method operates differently for session beans than it does for entity beans. Because a session object exists for one client and isn't persistent, a client of a session bean should call the `remove()` method when it's finished with a session object. Two `remove()` methods are available to the client: the client can remove the session object with the `javax.ejb.EJBObject.remove()` method, or the client can remove the session handle with the `javax.ejb.EJBHome.remove(Handle handle)` method. For more information on bean handles, see ["Referencing a bean with its handle" on page 17-6](#).

While it isn't required that a client remove a session object, it's good programming practice. If a client doesn't remove a stateful session bean object, the container will eventually remove the object after a certain time, specified by a timeout value. The timeout value is a deployment property. A client can also keep a handle to the session for future reference, however.

Clients of entity beans don't have this problem as entity beans are associated with a client only for the duration of a transaction and the container is in charge of their life cycles, including their activation and passivation. A client of an entity bean calls the bean's `remove()` method only when the entity object is to be deleted from the underlying database.

Referencing a bean with its handle

A handle is another way to reference an enterprise bean. A handle is a serializable reference to a bean. You can obtain a handle from the bean's remote interface. Once you have the handle, you can write it to a file (or other persistent storage). Later, you can retrieve the handle from storage and use it to reestablish a reference to the enterprise bean.

You can use the remote/local interface handle to recreate only the reference to the bean, however. You can't use it to recreate the bean itself. If another process has removed the bean, or the system removed the bean instance, then an exception is thrown when the client tries to use the handle to reestablish its reference to the bean.

When you aren't sure that the bean instance will still be in existence, rather than using a handle to the remote/local interface, you can store the bean's home handle and recreate the bean object later by invoking the bean's create or finder method.

After the client creates a bean instance, it can use the `getHandle()` method to obtain a handle to this instance. Once it has the handle, it can write it to a serialized file. Later, the client program can read the serialized file, casting the object that it reads in to a `Handle` type. Then, it calls the `getEJBObject()` method on the handle to obtain the bean reference, narrowing the results of `getEJBObject()` to the correct type for the bean.

For example, the CartClient program might do the following to use a handle to the Cart session bean:

```

import java.io;
import javax.ejb.Handle;
...
Cart cart;
...
cart = home.create(cardHolderName, creditCardNumber, expirationDate);

// call getHandle() on the cart object to get its handle
cartHandle = cart.getHandle();

// write the handle to serialized file
FileOutputStream f = new FileOutputStream ("carthandle.ser");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(myHandle);
o.flush();
o.close();
...

// read handle from file at later time
FileInputStream fi = new FileInputStream ("carthandle.ser");
ObjectInputStream oi = new ObjectInputStream(fi);

//read the object from the file and cast it to a Handle
cartHandle = (Handle)oi.readObject();
oi.close();
...

// Use the handle to reference the bean instance
Cart cart = (Cart) javax.rmi.PortableRemoteObject.narrow(cartHandle.getEJBObject(),
    Cart.class);
...

```

When it's finished with the session bean handle, the client can remove it by calling the `javax.ejb.EJBHome.remove(Handle handle)` method.

Managing transactions

A client program can manage its own transactions rather than letting the enterprise bean (or container) manage the transactions. A client that manages its own transactions does so in exactly the same manner as a session bean that manages its own transactions.

When a client manages its own transactions, it's responsible for delimiting the transaction boundaries. That is, it must explicitly start the transaction and end (commit or roll back) the transaction.

A client uses the `javax.transaction.UserTransaction` interface to manage its own transactions. It must first obtain a reference to the `UserTransaction` interface, using JNDI to do so. Once it has the `UserTransaction` context, the client uses the `UserTransaction.begin()` method to start the transaction, followed later by the `UserTransaction.commit()` method to commit and end the transaction (or `UserTransaction.rollback()` to rollback and end the transaction). In between, the client accesses EJB objects and so on.

The code shown here demonstrates how a client would manage its own transactions; the code that pertains specifically to client-managed transactions are highlighted in bold:

```

...
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
...
public class clientTransaction {
    public static void main (String[] args) {
        InitialContext initContext = new InitialContext();
        UserTransaction ut = null;
        ut = (UserTransaction)initContext.lookup("java:comp/UserTransaction");

        // start a transaction
        ut.begin();

        // do some transaction work
        ...

        // commit or rollback the transaction
        ut.commit(); // or ut.rollback();
        ...
    }
}

```

For more information about transactions, see [Chapter 18, “Managing transactions.”](#)

Discovering bean information

Information about an enterprise bean is referred to as metadata. A client can obtain metadata about a bean using the enterprise bean’s home interface `getMetaData()` method.

The `getMetaData()` method is most often used by development environments and tool builders that need to discover information about an enterprise bean, such as for linking together beans that have already been installed. Scripting clients might also want to obtain metadata on the bean.

Once the client retrieves the home interface reference, it can call its `getEJBMetaData()` method. Then, the client can call the `EJBMetaData` interface methods to extract such information as this:

- The bean’s `EJBHome` home interface, using the `EJBMetaData.getEJBHome()` method.
- The bean’s home interface class object, including its interfaces, classes, fields, and methods, using the `EJBMetaData.getHomeInterfaceClass()` method.
- The bean’s remote interface class object, including all class information, using the `EJBMetaData.getRemoteInterfaceClass()` method.

- The bean's primary key class object, using the `EJBMetaData.getPrimaryKeyClass()` method.
- Whether the bean is a session bean or an entity bean, using the `EJBMetaData.isSession()` method. The method returns `true` if this is a session bean.
- Whether a session bean is stateless or stateful, using the `EJBMetaData.isStatelessSession()` method. The method returns `true` if the session bean is stateless.

Here's the `EJBMetaData` interface in its entirety:

```
package javax.ejb;

public interface EJBMetaData {
    EJBHome getEJBHome();
    Class getHome InterfaceClass();
    Class getRemoteInterfaceClass();
    Class getPrimaryKeyClass();
    boolean isSession();
    boolean isStatelessSession();
}
```

Creating a client with JBuilder

You can use JBuilder to give you a head start on creating your client. JBuilder has a EJB Test Client wizard that is intended to create a simple client application to test your enterprise bean. You can also use it to get started building your actual client application. Inform the wizard of the name of one of the enterprise beans the client will access, and the wizard writes the code that gets a naming context, locates the bean's home interface and secures a reference to its remote/local interface.

Your client is likely to call multiple beans, however, so you'll have to perform these steps in your client code for other beans it accesses. And you'll add the calls that access the business logic of the enterprise beans to your client code yourself.

For more information on using the EJB Test Client wizard, see “[Creating a test client application](#)” on page 9-3.

18

Managing transactions

The JBuilder WebLogic Edition provides support for WebLogic Servers only

You can benefit from developing applications on platforms such as Java 2 Enterprise Edition (J2EE) that support transactions. A transaction-based system simplifies application development because it frees you, the developer, from the complex issues of failure recovery and multi-user programming. Transactions aren't limited to single databases or single sites. Distributed transactions can simultaneously update multiple databases across multiple sites.

A developer usually divides the total work of an application into a series of units. Each unit of work is a separate transaction. As the application progresses, the underlying system ensures that each unit of work, each transaction, fully completes without interference from other processes. If it doesn't, the system rolls back the transaction and completely reverses the work the transaction had performed so that the application is back to the same state before the transaction began.

Characteristics of transactions

Usually transactions refer to operations that access a database. All access to the database occurs in the context of a transaction. All transactions share these characteristics, denoted by the acronym ACID:

- Atomicity

Usually a transaction consists of more than a single operation. Atomicity requires that all of the operations of a transaction are performed successfully for the transaction to be considered complete. If all of a transaction's operations can't be performed, that none of them are allowed to be performed.

- Consistency

Consistency refers to database consistency. A transaction must move the database from one consistent state to another, and it must preserve the database's semantic and physical integrity.

- Isolation

Isolation requires that each transaction appear to be the only transaction currently manipulating the data in the database. Although other transactions can run concurrently, a transaction shouldn't see these manipulations until and unless they complete successfully and commit their work. Because of interdependencies among updates, a transaction might get an inconsistent view of the data were it to see just a subset of another transaction's updates. Isolation protects a transaction from this sort of data inconsistency.

Isolation is related to transaction concurrency. There are levels of isolation. Higher degrees of isolation limit the extent of concurrency. The highest level of isolation occurs when all transactions can be serialized. That is, the database contents appear as if each transaction ran by itself to completion before the next transaction began. Some applications, however, might be able to tolerate a reduced level of isolation for a higher degree of concurrency. Usually these applications run a greater number of concurrent transactions, even if transactions are reading data that might be partially updated and possibly inconsistent.

- Durability

Durability means that updates made by committed transactions persist in the database regardless of failure conditions. Durability guarantees that committed updates remain in the database despite failures that occur after the commit operation, and that the databases can be recovered after a system or media failure.

Transaction support in the container

An EJB container supports flat transactions, but not nested ones. It also propagates transactions implicitly. This means that you don't have to explicitly pass the transaction context as a parameter, because the container handles this task for the client transparently.

You should keep in mind that JSPs and servlets, while they can act as clients, aren't designed to be transactional components. Use enterprise beans to perform transactional work. When you invoke an enterprise bean to perform the transactional work, the bean and container set up the transaction properly.

Enterprise beans and transactions

Enterprise beans and the EJB container greatly simplify transaction management. Enterprise beans make it possible for an application to update data in multiple databases in a single transaction, and these databases can reside on multiple EJB servers.

Traditionally, an application responsible for managing transactions had to perform these tasks:

- Creating the transaction object
- Explicitly starting the transaction
- Keeping track of the transaction context
- Committing the transaction when all updates completed

Such an application demanded a very skilled developer and it was easy for errors to creep in.

Using enterprise beans, the container manages most if not all aspects of the transaction for you. It starts and ends the transaction and maintains its context throughout the life of the transaction object. Your responsibilities are greatly reduced, especially for transactions in distributed environments.

An enterprise bean's transaction attributes are declared at deployment time. These transaction attributes indicate whether the container manages the bean's transactions, or whether the bean manages its own transactions and to what extent.

Bean-managed versus container-managed transactions

When an enterprise bean programmatically performs its own transaction demarcation as part of its business methods, that bean is considered to be using bean-managed transaction. (To demarcate a transaction means to indicate where a transaction begins and where it ends.) When a bean defers all transaction demarcation to its EJB container, the container performs the transaction demarcation based on the application assembler's deployment instructions. This is called using container-managed transaction.

Both stateful and stateless session beans can use either type of transaction. A bean can't use both types of transaction management at the same time, however. The bean provider decides which type the session bean will use. Entity beans can use container-managed transactions only.

You might want a bean to manage its own transaction if you want to start a transaction as part of one operation, and then finish the transaction as part of another operation. You might encounter problems, however, if one

operation calls the transaction starting method, but no operation calls the transaction ending method.

Whenever possible, you should write enterprise beans that use container-managed transactions. They require less work on your part and are less prone to errors. Also, it's easier to customize a bean with a container-managed transaction and to use it to compose other beans.

Local and global transactions

When a single connection to a database exists, the enterprise bean can directly control the transaction by calling `commit()` or `rollback()` on the connection. This type of transaction is a local transaction. Using global transactions, all database connections are registered with the global transaction service, which handles the transaction. For a global transaction, the enterprise bean never makes calls directly on a database connection itself.

A bean that uses bean-managed transaction demarcation uses the `javax.transaction.UserTransaction` interface to identify the boundaries of a global transaction. When a bean uses container-managed demarcation, the container interrupts each client call to control the transaction demarcation, using the `transaction` attribute set in the bean's deployment descriptor by the application assembler. The `transaction` attribute also determines whether the transaction is local or global.

For container-managed transactions, the container follows certain rules to determine when it should do a local versus a global transaction. Usually a container calls the method within a local transaction after verifying that no global transaction already exists. It also verifies that it isn't expected to start a new global transaction and that the transaction attributes are set for container-managed transactions. The container automatically wraps a method call within a local transaction if one of the follow conditions is true:

- The `transaction` attribute is set to `NotSupported` and the container detects that the database resources were accessed.
- The `transaction` attribute is set to `Supports` and the container detects that the method wasn't invoked from within a global transaction.
- The `transaction` attribute is set to `Never` and the container detects that database resources are accessed.

Using the transaction API

All transactions use the Java Transaction API (JTA). When transactions are container managed, the platform handles the demarcation of transaction

boundaries and the container uses the JTA API. You never need to use this API in your bean code.

If your bean manages its own transactions, it must use the JTA `javax.transaction.UserTransaction` interface. This interface allows a client or component to demarcate transaction boundaries. Enterprise beans that use bean-managed transactions use the `EJBContext.getUserTransaction()` method.

Also, all transactional clients use JNDI to look up the `UserTransaction` interface. Do this by constructing a JNDI `InitialContext` using the JNDI naming service, such as shown here:

```
javax.naming.Context context = new javax.naming.InitialContext();
```

Once the bean has the `InitialContext`, it can then use the JNDI `lookup()` operation to obtain the `UserTransaction` interface:

```
javax.transaction.UserTransaction utx = (javax.transaction.UserTransaction)
    context.lookup("java:comp/UserTransaction")
```

Note than an enterprise bean can obtain a reference to the `UserTransaction` interface from the `EJBContext` object. The bean can simply use the `EJBContext.getUserTransaction()` method rather than having to obtain an `InitialContext` object and then using the `lookup()` method. A transactional client that isn't an enterprise bean, however, must use the JNDI `lookup` approach.

When the bean or client has the reference to the `UserTransaction` interface, it can then initiate its own transactions and manage them. That is, you can use the `UserTransaction` interface methods to begin and commit (or rollback) transactions. You use the `begin()` method to start the transaction, then the `commit()` method to commit the changes to the database. Or, you use the `rollback()` method to abort all changes made within the transaction and restore the database to the state it was in prior to the start of the transaction. Between the `begin()` and `commit()` methods, you include the code to carry out the business of the transaction. Here's an example:

```
public class NewSessionBean implements SessionBean {
    EJBContext ejbContext;

    public void doSomething(...) {
        javax.transaction.UserTransaction utx;
        javax.sql.DataSource dataSource1;
        javax.sql.DataSource dataSource2;
        java.sql.Connection firstConnection;
        java.sql.Connection secondConnection;
        java.sql.Statement firstStatement;
        java.sql Statement secondStatement;

        java.naming.Context context = new javax.naming.InitialContext();
```

Handling transaction exceptions

```
dataSource1 = (javax.sql.DataSource)
context.lookup("java:comp/env/jdbcDatabase1");
firstConnection = dataSource1.getConnection();

firstStatement = firstConnection.createStatement();

dataSource2 = (javax.sql.DataSource)
context.lookup("java:comp/env/jdbcDatabase2");
secondConnection = dataSource2.getConnection();

secondStatement = secondConnection.createStatement();

utx = ejbContext.getUserTransaction();

utx.begin();

firstStatement.executeQuery(...);
firstStatement.executeUpdate(...);
secondStatement.executeQuery(...);
secondStatement.executeUpdate(...);

utx.commit();

firstStatement.close;
secondStatement.close
firstConnection.close();
secondConnection.close();
}

...
...
```

Handling transaction exceptions

Enterprise beans can throw application and/or system-level exceptions if they encounter errors while handling transactions. Application-level exceptions arise from errors in the business logic. The calling application must handle them. System-level exceptions, such as runtime errors, transcend the application itself and can be handled by the application, the enterprise bean, or the bean container.

The enterprise bean declares application-level exceptions and system-level exceptions in the `throws` clauses of its home and remote/local interfaces. You must check for checked exceptions in your client application's try/catch block when calling bean methods.

System-level exceptions

An enterprise bean throws a system-level exception (usually a `java.ejb.EJBException`, but possibly a `java.rmi.RemoteException`) to indicate an unexpected system-level failure. For example, it throws an exception if

it can't open a database connection. The `java.ejb.EJBException` is a runtime exception and it isn't required to be listed in the `throws` clause of the bean's business methods.

System-level exceptions usually require the transaction to be rolled back. Often the container managing the bean does the rollback. Sometimes the client must roll back the transaction, though, especially if transactions are bean-managed.

Application-level exceptions

The bean throws an application-level exception to indicate application-specific error conditions. These are business logic errors, not system problems. Application-level exceptions are exceptions other than `java.ejb.EJBException`. Application-level exceptions are checked exceptions, which means you must check for them when you call a method that potentially can throw this exception.

The bean's business methods use application exceptions to report abnormal application conditions, such as unacceptable input values or amounts beyond acceptable limits. For example, a bean method that debits an account balance might throw an application exception to report that the account balance isn't sufficient to permit a particular debit operation. A client can often recover from these application-level errors without having to roll back the entire transaction.

The application or calling program gets back the same exception that was thrown, allowing the calling program to know the precise nature of the problem. When an application-level exception occurs, the enterprise bean instance doesn't automatically roll back the client's transaction. The client now has the knowledge and the opportunity to evaluate the error message, take the necessary steps to correct the situation, and recover the transaction. Or the client can abort the transaction.

Handling application exceptions

Because the application-level exceptions report business logic errors, your client must handle these exceptions. While these exceptions might require transaction rollback, they don't automatically mark the transaction for rollback. The client can retry the transaction, although often it must abort and roll back the transaction.

You, as the bean provider, must ensure that the state of the bean is such that if the client continues with the transaction, there is no loss of data integrity. If you can't ensure this, you must mark the transaction for rollback.

Transaction rollback

When your client gets an application exception, first check if the current transaction has been marked for rollback only. For example, a client might receive a `javax.transaction.TransactionRolledbackException`. This exception indicates that the helper enterprise bean failed and the transaction has been aborted or marked “rollback only”. Usually the client doesn’t know the transaction context within which the enterprise bean operated. The bean might have operated in its own transaction context separate from the calling program’s transaction context, or it might have operated in the calling program’s context.

If the enterprise bean operated in the same transaction context as the calling program, then the bean itself (or its container) has already marked the transaction for rollback. When an EJB container marks a transaction for rollback, the client should stop all work on the transaction. Usually a client using declarative transactions gets an appropriate exception, such as `javax.transaction.TransactionRolledbackException`. Note that declarative transactions are those transactions where the container manages the transaction details.

A client that is itself an enterprise bean should call the `javax.ejb.EJBContext.getRollbackOnly()` method to determine if its own transaction has been marked for rollback.

For bean-managed transactions, which are those transactions managed explicitly by the client, the client should roll back the transaction by calling the `rollback()` method from the `java.transaction.userTransaction` interface.

Options for continuing a transaction

When a transaction isn’t marked for rollback, the client has these options:

- Roll back the transaction.

When a client receives a checked exception for a transaction not marked for rollback, its safest course is to roll back the transaction. The client does this by either marking the transaction as rollback only or, if the client has actually started the transaction, calling the `rollback()` method to actually roll back the transaction.

- Pass the responsibility by throwing a checked exception or re-throwing the original exception.

The client can also throw its own checked exception or re-throw the original exception. By throwing an exception, the client lets other programs further up the transaction chain decide whether to abort the transaction. Usually, however, it’s preferable for the code or program closest to the occurrence of the problem to make the decision about continuing the transaction or not.

- Retry and continue the transaction. This might entail retrying portions of the transaction.

The client can continue with the transaction. The client can evaluate the exception message and decide if calling the method again with different parameters is likely to succeed. You must remember, however, that retrying a transaction is potentially dangerous. Your code doesn't know if the enterprise bean properly cleaned up its state.

Clients that are calling stateless session beans, however, can retry the transaction if they can determine the problem from the thrown exception. Because the called bean is stateless, there is no improper state to worry about.

If you are using the Borland Enterprise Server, see the Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1 documentation for additional information about transactions and the Borland container.

19

Creating JMS producers and consumers

This is a feature of
JBuilder Enterprise

The Java™ Message Service (JMS), which is part of the Java™ 2 Enterprise Edition (J2EE), supplies you with the APIs you need to create applications that use an enterprise message system. You can use this message system to develop scalable, reliable, and very flexible distributed applications. Message systems allow separate applications to communicate asynchronously.

An application or class that sends a message is a message producer. One that can receive a message is a message consumer. JBuilder has a JMS wizard that can help you build both message producers and consumers.

JMS messages can follow either of these two models:

- Publish/subscribe

A publish/subscribe message system follows an event-driven model in which producers of messages send out or publish messages and consumers of messages subscribe to or receive messages in which they are interested. Each published message is on a specific topic. Message consumers specify which topics they want to receive.

- Point to point

A point to point message system requires that message producers send a message to a particular message consumer. The message arrives at the consumer's incoming message queue.

To find complete information about JMS, see Sun's Java Message Service documentation at <http://java.sun.com/products/jms/docs.html>.

The JBuilder WebLogic
Edition does not include
the SonicMQ Message
Broker

JBuilder Enterprise ships with SonicMQ Message Broker, a Java Message Service provider. See the SonicMQ documentation for more information.

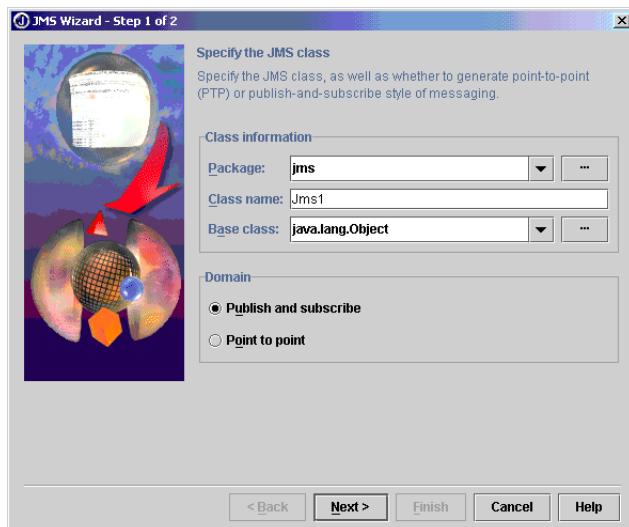
Using the JMS wizard

The JMS wizard generates a Java class that includes all the supporting code for a class to produce and consume messages. You then write very simple code to actually publish or send the message if the class is a message producer. If the class is a message consumer, you add the code to receive the message and implement the `onMessage()` method to handle messages subscribed to or received, depending on the type of message model you are using.

To begin using the JMS wizard,

- 1 Choose File | New, click the Enterprise tab, and double-click the JMS icon.

The JMS wizard appears:



- 2 Specify the package and class name for your class or accept the default values.
- 3 Enter the class you want this class to extend or accept the default value of `java.lang.Object` in the Base Class field.
- 4 Select Publish And Subscribe if you are creating a class for a publish\subscribe message system, or select Point To Point if you are creating a class for a point to point message system.
- 5 Choose Next.

The next page that appears in the JMS message depends on the Domain type you selected.

Publish\subscribe message systems

If you selected Publish\Subscribe, this page appears in the JMS wizard:



To finish the class,

1 Specify a Connection Factory Name.

A topic connection factory is used to set up a connection and topic session.

2 Specify the name of the message topic as the Topic Name.

3 If you want the session to use a durable subscriber,

- Check the Durable check box.
- Specify a unique Durable Name for the durable subscription.
- Specify a Client ID.

4 If the session is to be transacted, check the Transacted check box.

Transacted sessions use the session's `commit()` and `rollback()` methods to demarcate a local transaction.

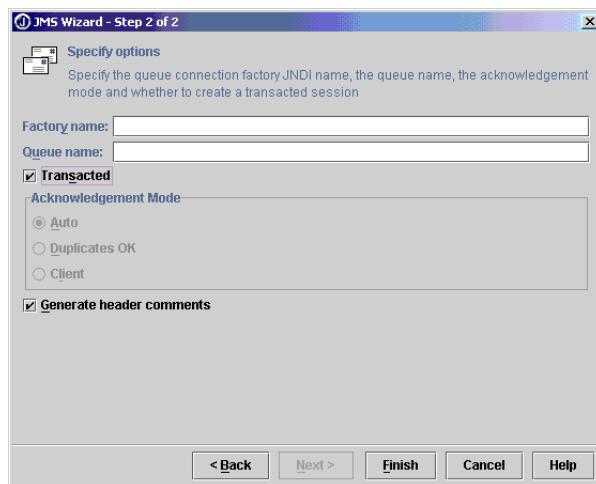
5 If the session is not transacted, select an acknowledgement mode:

- Auto - The session automatically acknowledges the receipt of a message.
- Duplicates OK - The session acknowledges all messages, performing no checks to prevent duplicates. This can improve processing time.
- Client - The client acknowledges a message by calling the message's `acknowledge()` method.

- 6 Leave the Generate Header Comments check box if you want the comments listing the title, description, and so on included in the generated code.
- 7 Choose Finish.

Point to point message systems

If you selected Point to Point, this page appears in the JMS wizard:

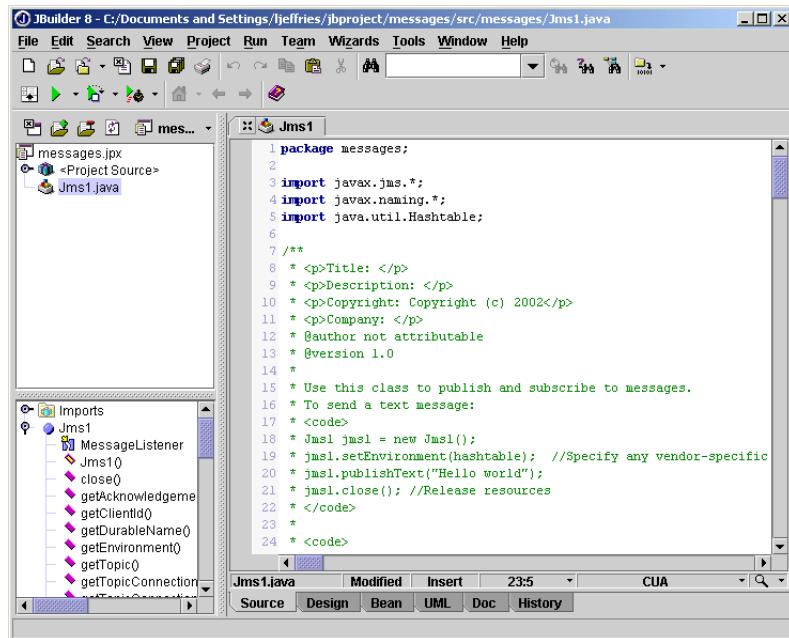


To finish the class,

- 1 Specify a Connection Factory Name.
- 2 Specify the name of the message queue as the Queue Name.
- 3 If the session is to be transacted, check the Transacted check box.
Transacted sessions use the session's `commit()` and `rollback()` methods to demarcate a local transaction.
- 4 If the session is not transacted, select an acknowledgement mode:
 - Auto - The session automatically acknowledges the receipt of a message.
 - Duplicates OK - The session acknowledges all messages, performing no checks to prevent duplicates. This can improve processing time.
 - Client - The client acknowledges a message by calling the message's `acknowledge()` method.
- 5 Uncheck the Header Comments check box if you want the comments listing the title, description, and so on omitted.
- 6 Choose Finish.

Completing the code

Once the JMS wizard finishes, you can view the source code that was generated in the Source pane:



```

1 package messages;
2
3 import javax.jms.*;
4 import javax.naming.*;
5 import java.util.Hashtable;
6
7 /**
8 * <p>Title: </p>
9 * <p>Description: </p>
10 * <p>Copyright: Copyright (c) 2002</p>
11 * <p>Company: </p>
12 * @author not attributable
13 * @version 1.0
14 *
15 * Use this class to publish and subscribe to messages.
16 * To send a text message:
17 * <code>
18 * Jms1 jms1 = new Jms1();
19 * jms1.setEnvironment(hashtable); //Specify any vendor-specific
20 * jms1.publishText("Hello world");
21 * jms1.close(); //Release resources
22 * </code>
23 *
24 * <code>

```

The commented code at the top of the generated class file demonstrates how to send a text message and also how to receive a message. Follow this example and add the code you need to send or receive messages. If your class is a message consumer, locate the `onMessage()` method in the source code and complete its implementation so that it handles the received message.

Exploring CORBA-based distributed applications

This is a feature of
JBuilder Enterprise

This chapter discusses creating a distributed application using JBuilder, the VisiBroker ORB (part of the Borland Enterprise Server), and the Common Object Request Broker Architecture (CORBA).

What is CORBA?

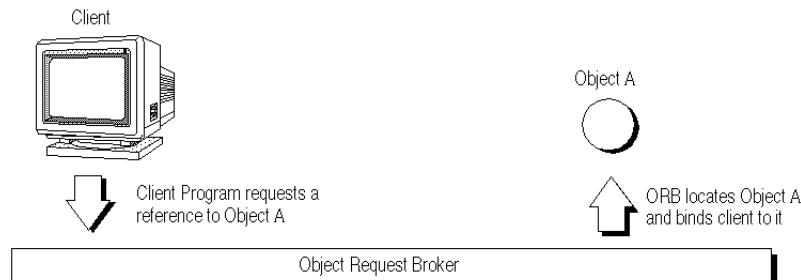
The Common Object Request Broker Architecture (CORBA) allows distributed applications to interoperate (application to application communication), regardless of what language they are written in or where these applications reside.

The CORBA specification was adopted by the Object Management Group to address the complexity and high cost of developing distributed object applications. CORBA uses an object-oriented approach for creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well defined interface, which reduces application complexity. The cost of developing applications is reduced, because once an object is implemented and tested, it can be used over and over again.

The Object Request Broker (ORB) connects a client application with the objects it wants to use. The client program does not need to know whether the object implementation it is in communication with resides on the same computer or is located on a remote computer somewhere on the network. The client program only needs to know the object's name and understand

how to use the object's interface. The ORB takes care of the details of locating the object, routing the request, and returning the result.

Figure 20.1 Client program acting on an object



Note The ORB itself is not a separate process. It is a collection of libraries and network resources that integrates within end-user applications, and allows your client applications to locate and use objects.

What is the VisiBroker ORB?

The VisiBroker ORB provides a complete CORBA ORB runtime and supporting development environment for building, deploying, and managing distributed Java applications that are open, flexible, and interoperable. The VisiBroker ORB is part of the Borland Enterprise Server. Objects built with the VisiBroker ORB are easily accessed by Web-based applications that communicate using OMG's Internet Inter-ORB Protocol (IIOP) standard for communication between distributed objects through the Internet or through local intranets. The VisiBroker ORB has a native implementation of IIOP to ensure high-performance and inter-operability.

How JBuilder and the VisiBroker ORB work together

JBuilder provides a set of wizards and other tools to make development of CORBA applications straight-forward.

[Chapter 23, "Tutorial: Creating a CORBA application"](#) will help you become acquainted with some of the JBuilder and VisiBroker ORB development tools. A sample application is used to illustrate each step of the process.

When you develop distributed applications with JBuilder and the VisiBroker ORB, you must first identify the objects required by the application. You will then usually follow these steps:

- 1** Write a specification for each object using the IDL wizard to generate the Interface Definition Language (IDL) file.

IDL is the language an implementer uses to specify the operations an object will provide and how they should be invoked. In this example, we define, in IDL, the `Account` interface with a `balance()` method and the `AccountManager` interface with an `open()` method.

- 2** Use the IDL compiler to generate the client stub code and server Portable Object Adapter (POA) servant code. For more about the POA, see "[What is the POA?](#)" on page 23-9

Using the `idl2java` compiler, we'll produce client-side stubs (which provide the interface to the `Account` and the `AccountManager` objects' methods) and server-side classes (which provides classes for the implementation of the remote objects).

- 3** Write the client program code.

To complete the implementation of the client program, initialize the ORB, bind to the `Account` and the `AccountManager` objects, invoke the methods on these objects, and print out the balance.

- 4** Write the server object code.

To complete the implementation of the server object code, we must derive from the `AccountPOA` and `AccountManagerPOA` classes, provide implementations of the interface's method, and implement the server's main routine.

- 5** Compile the client and server code.

To create the client program, compile the client program code with the client stub. To create the `Account` server, compile the server object code with the server servants.

- 6** Start the server.

- 7** Run the client program.

For more advanced information on using the VisiBroker ORB to create distributed applications, see the Borland Enterprise Server documentation in the `doc` directory of your Borland Enterprise Server installation.

For more information about Common Object Request Broker Architecture (CORBA), some useful links are:

- <http://www.borland.com/books> - Books on Borland products and related technologies
- “*A White Paper: Java, RMI, and CORBA*” at <http://www.omg.org/news/whitepapers/wpjava.htm>
- *CORBA/IOP Specification at Introduction to OMG’s Specifications* at <http://www.omg.org/gettingstarted/specintro.htm>
- *CORBA Basics* at <http://www.omg.org/gettingstarted/corbafaq.htm>

Setting up JBuilder for CORBA applications

This section explains how to set up JBuilder with the VisiBroker ORB or OrbixWeb so that you can create, run, and deploy CORBA applications. The CORBA tutorial in this book uses the VisiBroker ORB because it is included with JBuilder Enterprise, as part of the Borland Enterprise Server. Consult the OrbixWeb documentation for information on particular features of that ORB.

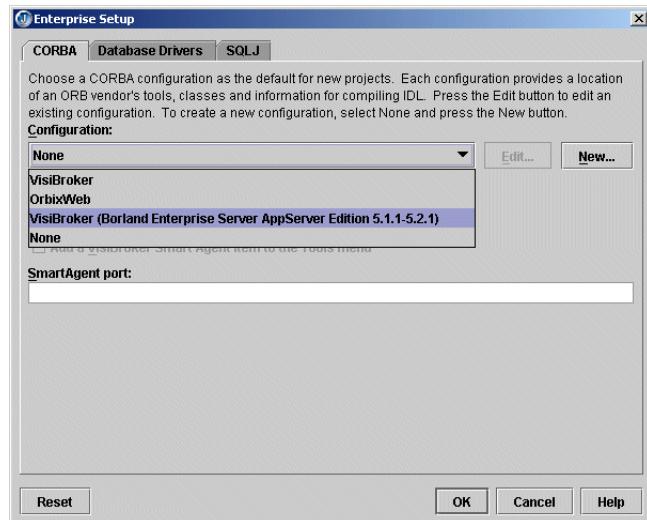
To set up these pieces on your system:

- 1 Install JBuilder Enterprise.
- 2 Install an application server that contains an ORB, such as the VisiBroker ORB or OrbixWeb. For more information see, “Configuring the target application server settings” in *Developing J2EE Applications*.
- 3 From JBuilder, run Tools | Enterprise Setup to display the Enterprise Setup dialog box. Select the CORBA tab. The parameters in this dialog box enable JBuilder to see the ORB.
- 4 Select a configuration from the Configuration drop-down list. The choices are:
 - VisiBroker
 - OrbixWeb
 - VisiBroker (Borland Enterprise Server AppServer Edition 5.1.1-5.2.1)
 - None

Note

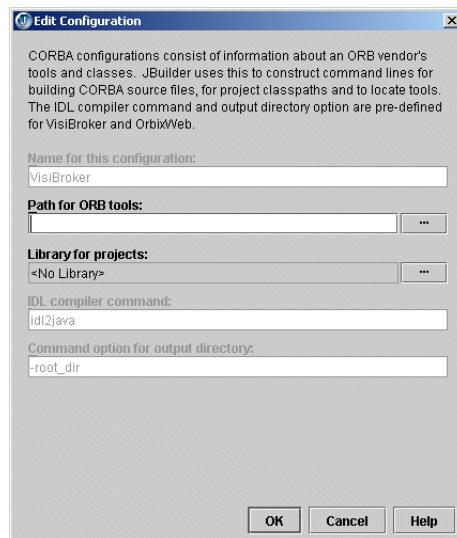
The VisiBroker ORB is included with JBuilder Enterprise, as part of the Borland Enterprise Server. If you have Borland Enterprise Server AppServer Edition 5.1.1-5.2.1, the server creates its own VisiBroker configuration.

The Enterprise Setup dialog box looks like this:



- If you do not have Borland Enterprise Server AppServer Edition 5.1.1-5.2.1 and would like to use VisiBroker, choose the VisiBroker configuration, and continue with the steps in this section. Then, follow the instructions in the section called “Making the ORB available to JBuilder” in “Configuring the target application server settings” of *Developing J2EE Applications*.
 - If you’re using the OrbixWeb, choose the OrbixWeb configuration and continue with the steps in this section.
 - If you’re using the Borland Enterprise Server, choose the VisiBroker (Borland Enterprise Server AppServer Edition 5.1.1-5.2.1) configuration and click OK to close the dialog box. JBuilder is configured for use with VisiBroker.
 - If you would like to create a new configuration, select None.
- 5 If you selected the VisiBroker or OrbixWeb configuration, click the Edit button to display the Edit Configuration dialog box. If you selected None, click the New button to display the New Configuration dialog box. (Follow the instructions in this section to create a new configuration; the fields are the same as those on the Edit Configuration

dialog box.) For the VisiBroker configuration, the Edit Configuration dialog box looks like this:



- 6** On the Edit Configuration dialog box, click the ellipsis button next to the Path For ORB Tools field to enable JBuilder to access the ORB tools. When using the VisiBroker ORB, point to the directory that contains the osagent.exe file. This is the bin directory of your Borland Enterprise Server installation.

Note If you're creating a new configuration, you also need to fill in the Name For This Configuration field.

- 7** To define the library location, click the ellipsis button next to the Library For Projects field. This displays the Select A Different Library dialog box. The library is necessary for compiling the generated stubs and skeletons and for executing an application. For VisiBroker, browse to the Borland Enterprise Server Client library.
- a** To add a library that is not displayed in the Select A Different Library dialog box, click New.
 - b** In the New Library wizard, enter a name for the new library in the Name field.
 - c** Select a location for the library: the JBuilder directory, the project directory, or your home directory.
 - d** Click Add to browse to the library (JAR) file.

e Click OK until you return to the Edit Configuration dialog.

f Click OK to close the Edit Configuration dialog box.

Note If you're creating a new configuration, you also need to fill in the IDL Compiler Command and Command Option For Output Directory fields.

8 In the Enterprise Setup dialog box, leave the Apply This Configuration To The Current Project option selected if the current project should be set to the same ORB configuration as the default project.

9 Set the Make This Configuration's ORB The Default For The Java VM option to use this ORB as the default ORB for the Java VM. If you receive an error message, you may not have sufficient rights to modify the `orb.properties` file, located in the `<jdk>/jre/lib` directory of your JBuilder installation. You can manually create and/or edit this file, specifying the following information. This sample file references the VisiBroker ORB.

```
# Make the VisiBroker ORB the default ORB
org.omg.CORBA.ORBClass=com.inprise.vbroker.orb.ORB
org.omg.CORBA.ORBSingletonClass=com.inprise.vbroker.orb.ORB
```

10 For the VisiBroker ORB, select the Add A VisiBroker Smart Agent Item To The Tools Menu option to enable the Smart Agent to run as a JBuilder service during development. This option will be disabled if the Smart Agent is not necessary. Select a port on which to run the Smart Agent. The default port of 14000 will work on most systems.

11 Click OK. These settings will be written to the `orb.properties` file.

12 For VisiBroker, choose Tools | VisiBroker Smart Agent to start the Smart Agent.

You've now completed setting up your system to use the JBuilder CORBA features. For a tutorial that uses these features to create a CORBA application, see [Chapter 23, "Tutorial: Creating a CORBA application."](#) For more information on using the VisiBroker ORB features with JBuilder, see the following section, ["Defining interfaces in Java" on page 20-7](#).

Important If you are using the Borland Enterprise Server, make sure your project uses JDK 1.3.1. Check the JDK setting on the Paths page of the Project Properties dialog box.

Defining interfaces in Java

This section explains how to use the `java2iiop` and `java2idl` compilers to generate client stubs and servants from interface definitions written in Java instead of IDL. These compilers are part of the Borland Enterprise Server, which comes "in-the-box" with JBuilder.

This section contains the following topics:

- “[Working with the java2iiop compiler.](#)” Explains how to create distributed applications in an all-Java, IIOP-compliant environment.
- “[Working with the java2idl compiler.](#)” Explains how to turn your Java code into IDL interfaces, allowing you to generate client stubs in the programming language of your choice.

You can find more information about these compilers in the Borland Enterprise Server documentation. This documentation is located in the `doc` directory of your Borland Enterprise Server installation.

About the `java2iiop` and `java2idl` compilers

The Borland Enterprise Server incorporates the following compilers that make it easy to work in a Java environment. These features include:

- **`java2iiop`**

The `java2iiop` compiler allows you to stay in an all-Java environment, if that's what you want. The `java2iiop` compiler takes your Java interfaces and generates IIOP-compliant stubs and skeletons. Part of the advantage of using the `java2iiop` compiler is that, through the use of extensible structs, you can pass Java serialized objects by value.

- **`java2idl`**

The `java2idl` compiler turns your Java code into IDL, allowing you to generate client stubs in the programming language of your choice. In addition, because this compiler maps your Java interfaces to IDL, you can re-implement Java objects in another programming language that supports the same IDL.

- **Web Naming**

Web Naming allows you to associate a URL with objects, allowing an object reference to be obtained by specifying a URL.

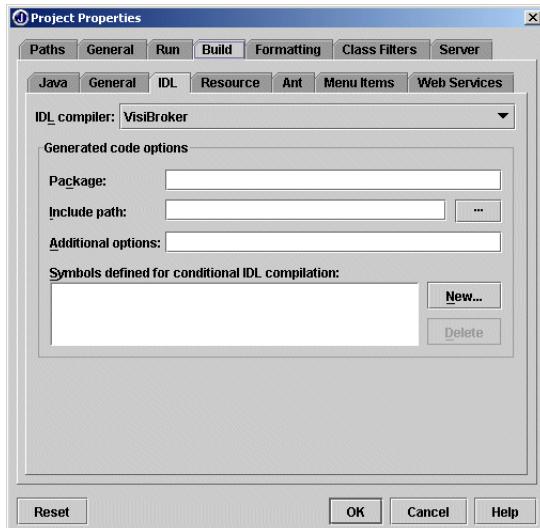
If you are going to design CORBA interfaces in Java, you must use `java2iiop` as the compiler to produce the client stubs and servants. When designing CORBA interfaces in Java, you do not need to create an IDL file. In fact, the generated IDL file cannot be modified or compiled. You would generate an IDL file from a Java interface only to:

- Populate the interface repository.
- Build cross-language applications.

Accessing the java2iiop and java2idl compilers in JBuilder

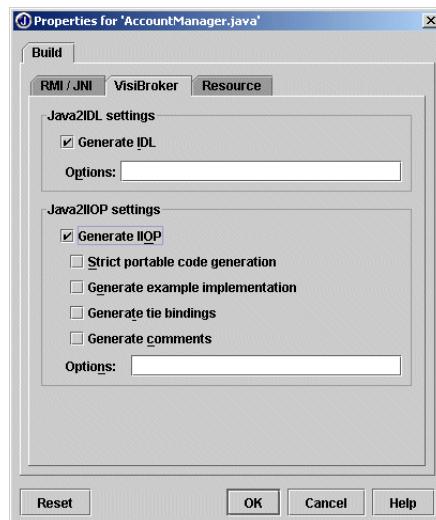
To access the `java2iiop` and `java2idl` compilers in the JBuilder development environment,

- 1 Make sure JBuilder is set up correctly by following the steps outlined in “[Setting up JBuilder for CORBA applications](#)” on page 20-4.
- 2 Choose Project | Project Properties to display the Project Properties dialog box. On the IDL tab of the Build page, make sure the IDL Compiler field is set to VisiBroker.



- 3 On the Paths page of the Project Properties dialog box, make sure the Required Libraries list includes the Borland Enterprise Server 5.1.1 - 5.2.1 Client.
- 4 Click OK to close the Project Properties dialog box.
- 5 Once you've created a Java interface file in your JBuilder project, right-click the file in the project pane. Choose Properties.

- 6 On the VisiBroker tab of the Build page, select the Generate IDL and Generate IIOP options. The dialog box should look like this:



- 7 Click OK to close the dialog box.
- 8 Right-click the Java interface file in the project pane and select Make to compile the interface definition and generate IIOP or IDL interface files.

RMI

Another way to invoke methods on remote Java objects is to use Remote Method Invocation (RMI). The Borland Enterprise Server provides capabilities equivalent to RMI, but, in addition, allows you to create Java objects that communicate with all other objects that are CORBA compliant even if they are not written in Java. For an example of a Java interface that describes an RMI remote interface, see the sample file `Account.java` in the `examples/vbe/rmi-iiop/Bank` directory of your Borland Enterprise Server installation.

Working with the `java2iio` compiler

The `java2iio` compiler lets you define interfaces and data types that can then be used as interfaces and data types in CORBA. The advantage is that you can define them in Java rather than IDL. The compiler reads Java byte code—it does not read source code, but reads class files. The compiler then generates IIOP-compliant stubs and skeletons needed to do all the marshalling and communication required for CORBA.

When you run the `java2iio` compiler, it generates the same files as if you had written the interface in IDL. All primitive data types, including numeric types (short, int, long, float, and double), string, CORBA objects

or interface objects, Any's, and type codes are understood by the `java2iiop` compiler and mapped to corresponding types in IDL.

When using the `java2iiop` compiler on Java interfaces, you define and mark them as interfaces to be used with remote calls. You mark them by having them extend the `org.omg.CORBA.Object` interface. (The interface must also define methods that you can use in remote calls.) When you run the compiler, it searches for these special CORBA interfaces. When one is found, it generates all of the marshalling elements, the readers and writers which enable you to use the interface for remote calls. An example of a Java interface definition file, `AccountManager.java`, is located in the `examples/vbe/rmi-iiop/Bank` directory of your Borland Enterprise Server installation.

Note For developers who are familiar with RMI (Remote Method Invocation), this is equivalent to a class extending the `java.rmi.Remote` interface.

For classes, the compiler follows other rules and maps the classes either to IDL structs or extensible structs. For more information about complex data types, see “[Mapping complex data types](#)” on page 20-13.

Generating IIOP interfaces running `java2iiop`

The example files located in the `examples/vbe/rmi-iiop` directory of your Borland Enterprise Server installation demonstrate how to define an interface in Java and compile it into IIOP-compliant stubs and servants using the `java2iiop` compiler.

The files in this example are:

- `Bank/Account.java`: The interface that describes the Account RMI remote interface.
- `Bank/AccountData.java`: The class that is used to carry data to create an account.
- `Bank/AccountManager.java`: The interface that describes the AccountManager CORBA interface.
- `Bank/AccountManagerOperations.java`: This interface declares the method signatures defined in `AccountManager.java`.
- `AccountImpl.java`: The servant that implements the Account interface.
- `AccountManagerImpl.java`: The servant that implements the AccountManager interface.
- `Client.java`: The client mainline.
- `Server.java`: The server mainline.

The `AccountManager` interface creates an account, (re)opens an account, and gets a list of accounts. The account that is created contains operations to get the name on the `Account`, and to set and get the balance in the account. `AccountData` represents the information that is used to create accounts.

This example illustrates the use of RMI over IIOP, and how RMI over IIOP can be used to pass native Java classes (those not mapped from IDL) across the wire in remote method calls. It also illustrates the use of an RMI-style remote interface with CORBA servers, but uses RMI clients to access the CORBA servers. To run and view the example, refer to the file `rmi-iiop_java.html` in the `examples/vbe/rmi-iiop` directory of your Borland Enterprise Server installation.

To generate the IIOP interfaces in JBuilder, you would make sure that the `java2iiop` compiler was selected the Properties dialog box. To do this,

- 1 In the project pane, right-click the Java interface file, `AccountManager`. Select Properties from the context menu.
- 2 Select the Generate IIOP option on the VisiBroker tab of the Build page, then click OK.
- 3 Right-click the Java interface file, and select Make from the context menu.

The `java2iiop` compiler will then generate all of the usual auxiliary files, like Helper and Holder classes. For more information about the generated files, see the Borland Enterprise Server documentation.

Note The files generated by the `java2iiop` compiler are the same as those generated by the `idl2java` compiler.

The Borland Enterprise Server documentation provides more information, including sample code and sample applications, for completing the development process with the VisiBroker ORB. In general, after generating stubs and skeletons, create your client and server classes. You would then follow these steps:

- 1 Use the skeleton class to create an implementation of server object; for an example, see `Server.java` in the `examples/vbe/rmi-iiop` directory of your Borland Enterprise Server installation.
- 2 Compile your server class by right-clicking the server file in the project pane and selecting Make.

Note If you're using `Server.java`, you may have to rename `Makefile` and `Makefile.java` in the `examples/vbe/rmi-iiop` directory. They may interfere with JBuilder's compiler.

- 3 Write the code for the client. For an example, see `Client.java` in the `examples/vbe/rmi-iiop` directory of your Borland Enterprise Server installation.
- 4 Compile the client code by right-clicking the client file in the project pane and selecting Make.

Note If you're using `Client.java`, you may have to rename `Makefile` and `Makefile.java` in the `examples/vbe/rmi-iiop` directory. They may interfere with JBuilder's compiler.

- 5 Start the VisiBroker ORB Smart Agent by selecting it from the Tools menu. For this option to be available, you must have set up JBuilder as described in “[Setting up JBuilder for CORBA applications](#)” on page 20-4.
- 6 Start the server object by right-clicking the server file in the project pane and selecting Run.
- 7 Start the client by right-clicking the client file in the project pane and selecting Run.

Important The examples shown in this chapter assume that a VisiBroker ORB Smart Agent is running. Make sure the Smart Agent is running by selecting Tools | VisiBroker Smart Agent. For more information on the Smart Agent, see the Borland Enterprise Server documentation.

Mapping primitive data types to IDL

Client stubs generated by `java2iiop` handle the marshalling of the Java primitive data types that represent an operation request so that they may be transmitted to the object server. When a Java primitive data type is marshalled, it must be converted into an IIOP-compatible format. For more information, see the Borland Enterprise Server documentation.

Mapping complex data types

This section shows how the `java2iiop` compiler can be used to handle complex data types, such as interfaces, arrays, Java classes, and extensible structs. For more information, see the Borland Enterprise Server documentation.

Interfaces

Java interfaces are represented in IDL as CORBA interfaces. They must inherit from the `org.omg.CORBA.Object` interface. When passing objects that implement these interfaces, they are passed by reference.

Note The `java2iiop` compiler does not support overloaded methods on CORBA interfaces.

Interfaces which do not extend `org.omg.CORBA.Object` are mapped to an extensible struct.

Arrays

Another complex data type that may be defined in classes is an array. If you have an interface or definitions that use arrays, the arrays map to CORBA unbounded sequences.

Mapping Java classes

In Java classes, you can define arbitrary data types. Some arbitrary data types are analogous to IDL structures (also called *structs*). If you define a

Java class so that it conforms to certain requirements, then the `java2iiop` compiler maps it to an IDL struct. You can map Java classes to IDL structs if the class fits all of these requirements:

- The class is final.
- The class is public.
- The class does not use implementation inheritance.
- The data members of the class are public.

The `java2iiop` compiler looks at a class, and checks to see if it meets all of these requirements. If the class does meet all of the requirements, the compiler maps it to an IDL struct; if it does not meet all of the requirements, it maps it to an extensible struct.

Extensible structs

Any Java class that does not meet all of the requirements listed above is mapped to an extensible struct. An extensible struct is an upwardly-compatible extension of CORBA structs. When you use extensible structs, objects are passed by value.

Pass-by-value is the ability to pass object state to another Java program. Assuming that a class definition of the Java object is present on the server side, the Java program can invoke methods on the cloned object that has the same state as the original object.

The use of extensible structs is a Borland Enterprise Server extension to the OMG IDL. It provides an additional keyword: `extensible`. If you want to stay within pure CORBA or if you are going to port your code to other ORBs, you should use IDL structs and not extensible structs. Extensible structs allow you to use classes that can be defined in Java but cannot be defined in IDL because of CORBA limitations.

The VisiBroker ORB uses Java serialization to pass classes in the form of extensible structs. Java serialization compresses a Java object's state into a serial stream of octets that can be passed on-the-wire as part of the request.

Note Because of Java serialization, all the data types that are passed must be serializable; that is, they must implement `java.io.Serializable`.

Extensible structs allow data that use pointer semantics to be passed successfully. For example, defining a linked list in your application code is standard practice, yet there is no way to define a linked list in standard IDL. The solution to this problem is that you can define it by using extensible structs. When you pass extensible structs across address spaces, pointers are maintained.

Working with the java2idl compiler

IDL is the foundation of CORBA applications. IDL interfaces define a component's boundaries. You can use IDL to define a component's attributes, the exceptions it raises, and the methods its interface supports.

You can generate IDL interfaces from Java interfaces defined in a `.java` file, using the `java2idl` compiler. Using a Java interface to describe a CORBA object has some limitations. All Java primitive types can be used. Java objects, however, can be used to define the interface only if the object implements `java.io.Serializable`.

The following steps show you how to generate IDL interfaces from a Java interface definition.

- 1** In the project pane, right-click a Java interface file. Select Properties from the context menu.
- 2** Select the Generate IDL option on the VisiBroker tab of the Build page, then click OK.
- 3** Right-click the Java interface file, and select Make from the context menu.

The `java2idl` compiler will then generate all of the usual auxiliary files, like Helper and Holder classes. Click the plus sign to the left of the Java interface file in the project pane to display the generated files. For more information about the generated files, see the Borland Enterprise Server documentation.

21

Tutorial: Developing a session bean with the EJB Designer

This tutorial creates a shopping cart session bean that can hold the items a customer buys at an online store. In this case, the items for sale are books and CDs (compact discs). The bean you create by following this tutorial is a very simple one. The goal of the tutorial is to show you how to use the features of JBuilder to create session beans.

During this tutorial, you will develop the session bean and start it running in a local container. In a later tutorial, you will create a client application that calls the methods of the bean, testing its logic.

To see the code this tutorial creates, see “[Code for cart session bean](#)” on [page 21-19](#).

If you would like to see a more complex online shopping cart bean that also uses additional enterprise beans, JavaServer Pages, servlets, and DataExpress, see the `JBuilder/samples/Ejb/Ejb11/ESite/ESite.jpx` sample that uses EJB 1.1 beans and the Borland Enterprise Server 5.1.1 - 5.2.1. You can also find the same sample using the WebLogic Server 6.x, 7.x, or 8.x and EJB 2.0 beans in `JBuilder/samples/Ejb/Ejb20/ESite/esite.jpx`.

The Accessibility section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see “[Documentation conventions](#)” on [page 1-1](#).

Creating a new project

Begin by creating a new project for the session bean you are going to develop:

- 1 Choose File | New Project.

The Project wizard appears.

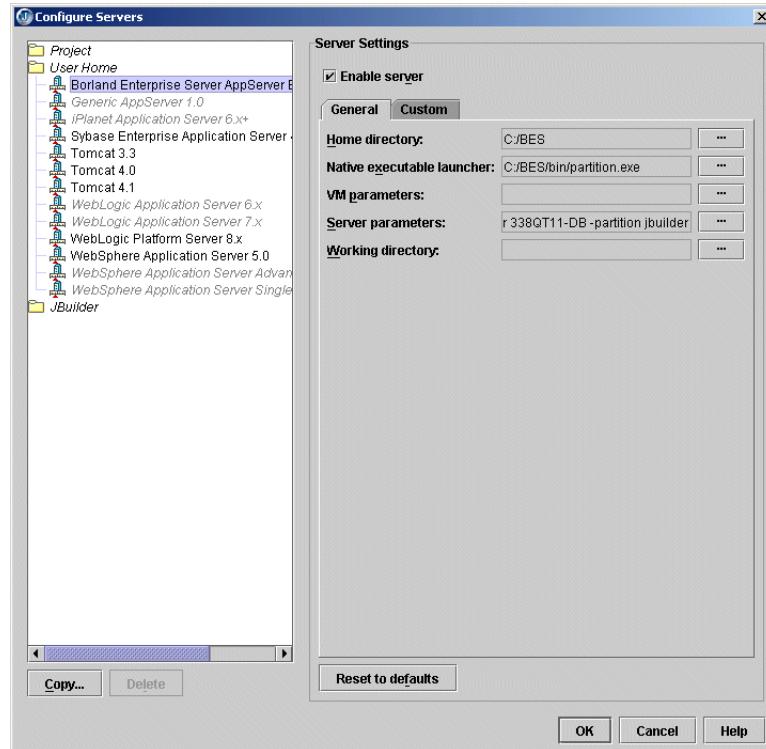
- 2 Specify the Name of the project as `cart_session` and click Finish.

A `cart_session.jpx` node appears in the project pane.

Specifying the target application server

This tutorial builds a session bean that will be deployed to the Borland Enterprise Server. Therefore, if you haven't already done so, you must set up JBuilder to target that server. Follow these steps:

- 1 Choose Tools | Configure Servers to display the Configure Servers dialog box:



2 Click the Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1 in the list of servers on the left side of the dialog box.

3 Check the Enable Servers check box.

Use the ... button next to the Home Directory field to specify the location of the Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1. The remaining fields contain default values which should be sufficient for your needs. You can make changes to the default values if you need to.

4 Click the Custom button and specify the location of the JDK used by the Borland Enterprise Server in the JDK Installation Directory if it isn't already filled in for you.

5 Choose OK.

If you have installed and already set up more than one application server using the Enterprise Setup dialog box, you must be sure the correct application server is selected for your project:

1 Choose Project | Project Properties.

2 Click the Server tab.

3 Select the Single Server For All Services In Project option and select Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1 from the drop-down list.

4 Click OK.

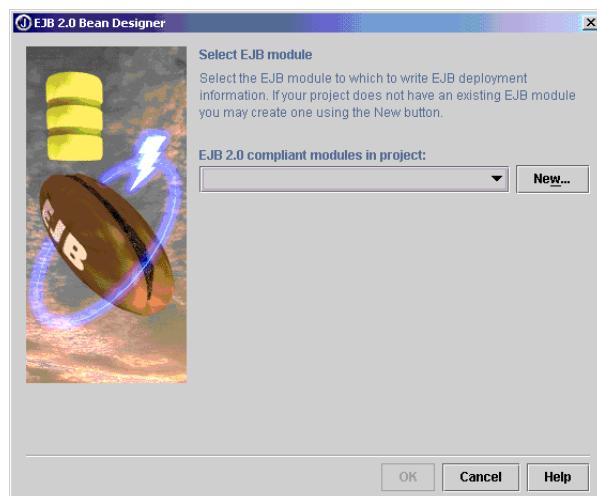
To find more details about configuring and selecting an application server as the target, see "Configuring the target server settings" in *Developing J2EE Applications*.

Creating an EJB module

Each enterprise bean must belong to an EJB module. To create an EJB module node and display the EJB Designer, follow these steps:

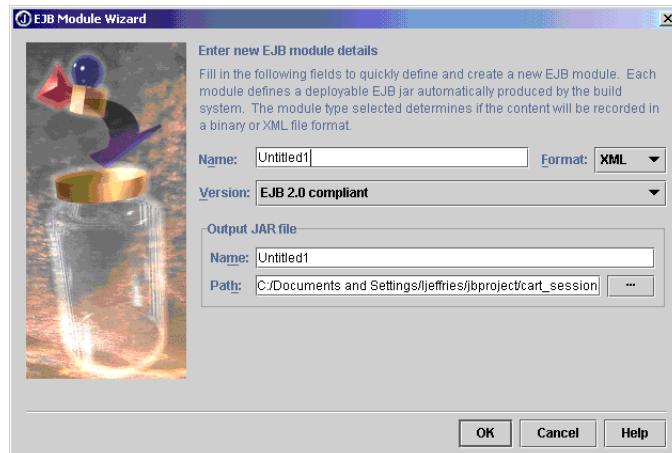
1 Choose File | New to display the object gallery and click the Enterprise tab.

- 2 Double-click the EJB 2.0 Designer icon. The EJB 2.0 Bean designer wizard appears.



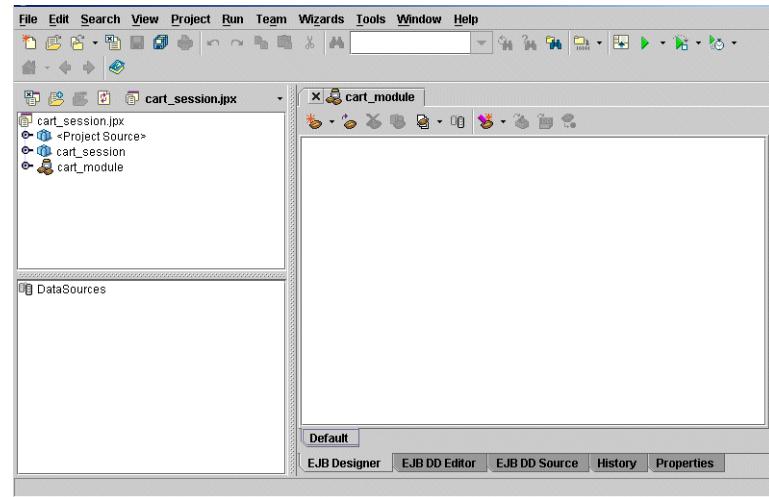
No EJB 2.0 compliant modules will be listed as you have yet to create any.

- 3 Click the New button to display the EJB Module wizard.



- 4 In the Name field type in `cart_module`. Note that as you enter the EJB module name, the Name Of The Output JAR field is automatically filled in with `cart_module.jar`.
- 5 Accept all other default values and click OK.

- 6 Click OK once more and the EJB Designer appears:

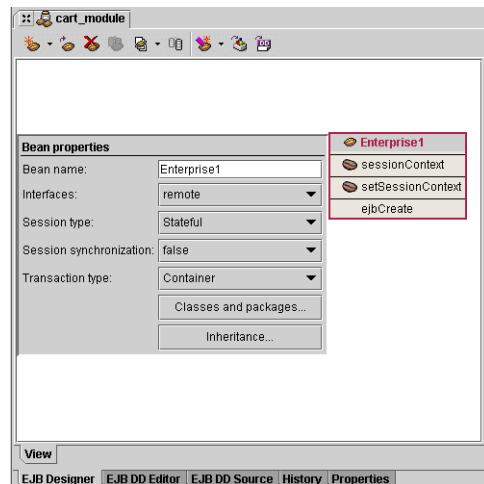


Note that an EJB module node titled `cart_module` now appears in the project pane. If you click the icon to the far-left of the node to open it, you'll see it already contains other nodes, although at this point they contain nothing.

Building the bean

To begin building a shopping cart session bean, right click the EJB Designer pane and choose Create EJB | Session Bean from the menu, or click the Create EJB icon in the EJB Designer toolbar and choose Session Bean.

A bean representation named `Enterprise1` appears in the EJB Designer and a bean inspector opens.



Setting bean properties

You use the bean inspector to set property values for the bean. (If the inspector closes, you can open it again by clicking the name of the bean.)

Set the property values that affect the whole bean:

- 1 Type `Cart` in the Bean Name field to name the bean.
- 2 Select Remote from the Interfaces drop-down list. (It probably is selected by default.) This selection determines which interfaces the EJB Designer creates for you: remote, local, or both.
- 3 Select Stateful from the Session Type drop-down list. We want to design the bean so that a shopper could theoretically stop a shopping session and later return to the online store and still access the same items in the shopping cart. Because the bean must be able to maintain state between sessions, the bean must be declared as a stateful bean. (The simple bean we develop here doesn't actually do this.)
- 4 Leave the values of the other fields unchanged.

Look in the project pane and notice that a `cart_session` package node now exists. Open the node to see the files it contains. You'll find these three:

- `Cart.java` - the bean's remote interface
- `CartBean.java` - the bean class itself
- `CartHome.java` - the bean's home interface

You can double-click the names of the files to view their source code and see what the EJB Designer has generated so far. As you work in the EJB Designer, it generates additional lines of code. To return to the EJB Designer, double-click the `cart_module` node in the project pane or click the `cart_module` tab at the top of the content pane.

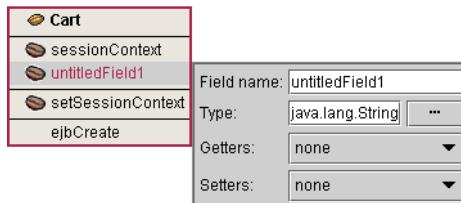
Adding fields to the Cart bean

A shopping cart bean must be associated with a particular user. It might hold the user's name, the user's credit card number, and the expiration date of that credit card. It must have a way to maintain the list of items the user wants to buy. Follow these steps to add the necessary fields to the bean:

- 1 Right-click the `Cart` bean representation in the EJB Designer to display a context menu.

2 Choose Add | Field.

A new field appears in the bean representation and an associated inspector opens.



3 In the inspector, enter _cardHolderName as the Field Name.

4 Accept the default Type value as java.lang.String and the Getters and Setters values as None.

You can edit a field's property value any time by clicking the field in the bean representation.

Add the remaining three fields to the Cart bean:

1 Right-click the bean and choose Add | Field.

2 In the inspector, enter _creditCardNumber as the Field Name.

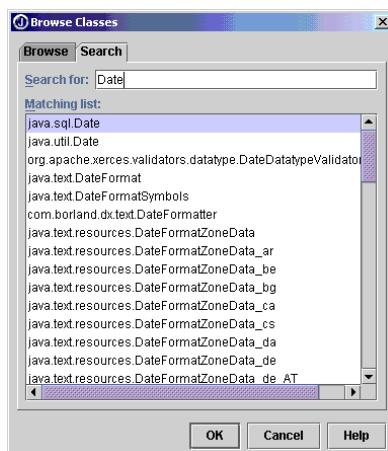
3 Accept the default Type value as java.lang.String and the Getters and Setters values as None.

4 Right-click the bean and choose Add | Field.

5 In the inspector, enter _expirationDate as the Field Name.

6 Click the ... button next to the Type field to display the Browse Classes dialog box. Click the Search tab and type Date in the Search For field.

You'll see a list of all possible matches. Select java.util.Date.



Select `java.util.Date` from the Matching List and click OK. Or you can simply type `java.util.Date` in the Type field of the inspector.

- 7 Right-click the bean and choose Add | Field.
- 8 In the inspector, type `_items` as the Field name.
- 9 Click the ... button next to the Type field and use the Browse Classes dialog box again to specify the `java.util.List` class, or simply type in `java.util.List` in the Type field of the inspector.

Look at the source code for bean to see the new fields. A quick way to go from the EJB Designer to the bean's source code is to right-click the bean representation in the EJB Designer and choose View Bean Source. So far, your bean class should look like this:

```
package cart_session;

import javax.ejb.*;

public class CartBean implements SessionBean {
    SessionContext sessionContext;
    java.lang.String _cardHolderName;
    java.lang.String _creditCardNumber;
    java.util.Date _expirationDate;
    java.util.List _items;
    public void ejbCreate() throws CreateException {
        /**@todo: complete this method*/
    }
    public void ejbRemove() {
        /**@todo: complete this method*/
    }
    public void ejbActivate() {
        /**@todo: complete this method*/
    }
    public void ejbPassivate() {
        /**@todo: complete this method*/
    }
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }
}
```

A session bean class must be defined as public. It can't be defined as final or abstract. The bean class must implement the `SessionBean` interface. The session bean class generated for you meets all these requirements.

The generated code also includes four methods that are defined by the `SessionBean` interface. The EJB container invokes these methods on the bean instance at specific points in a session bean's life cycle. A bean must include these methods, although their method bodies can be empty:

```
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext context) {}
```

As you can see, the EJB Designer correctly added these required methods to the bean class. In the `setSessionContext()` method, the EJB Designer assigned the value of the `context` parameter to the `sessionContext` instance variable.

The container calls the `setSessionContext()` method to associate the bean instance with its session context. The bean can choose to retain this session context reference as part of its conversational state, but it's not required to do so. The session bean can use the session context to get information about itself, such as environment variables or its home interface.

The container calls the `ejbPassivate()` method on the bean instance when it needs to place the bean instance into a passive state. The container writes the bean's current state to secondary storage when it passivates the bean. It restores this state when it later activates the bean. Because the container calls the `ejbPassivate()` method just before it actually passivates the bean instance, you, as the bean provider, can add code to this method to do any special variable caching you want. Similarly, the container calls the `ejbActivate()` method on the bean instance just prior to returning the bean instance to an active from a passive state. When it activates the bean, it restores all persisted state values. You can choose to add code to the `ejbActivate()` method. `CartBean` leaves these implementations empty.

While a session bean isn't required to implement a constructor, it must implement at least one `ejbCreate()` method. This method serves as a constructor to create a new bean instance. A stateful session bean can implement more than one `ejbCreate()` method. Multiple `ejbCreate()` methods would differ from others only in their parameters. Later you'll add parameters to the single `ejbCreate()` method of `CartBean`.

The container calls the `ejbRemove()` method just prior to removing the bean instance. You can add some application-specific code that would execute before the bean is removed, but it isn't required. The `CartBean` example leaves the body of the `ejbRemove()` method empty.

Adding business methods to the Cart bean

A shopping cart bean must have methods that add and remove items from the cart as the shopper browses the online store. When the shopper is ready to buy the items, the bean needs a method that can list the contents of the cart and another to calculate the total price. It would also need one or more methods that complete the purchase transaction.

Each business method added to a bean must follow a few rules:

- None of the method names can start with the prefix `ejb` to avoid conflict with names reserved by the EJB architecture.
- Each method must be declared as public.
- No method can be declared as final or static.

- The parameters and return types must be legal RMI-IIOP types.
- The **throws** clause may include the `javax.ejb.EJBException` exception and it may define arbitrary application-specific exception.

Adding items to and removing items from the cart

Focus first on the methods that maintain the list of items the bean maintains, an `addItem()` and a `removeItem()` method.

Double-click the EJB module node in the project pane to return to the EJB Designer and follow these steps to add an `addItem()` method to the bean:

- 1 Right-click the `Cart` bean representation in the EJB Designer and choose Add | Method.
- 2 In the method inspector that appears, enter in the Method Name as `addItem`. (If the method inspector is not present whenever you want to use it to edit the method, simply click the method in the bean representation. This might be necessary, for example, if you switch to the Source pane to look at the source code, then return to the EJB Designer.)
- 3 Leave the Return Type default value of `void` unchanged, but specify the Input Parameters as `Item item`. Later you'll create an `Item` class that holds the data you pass to the method.
- 4 In the Interfaces drop-down list, specify Remote. Selecting Remote means the method will be declared in the bean's remote interface.

To add a `removeItem()` method to the bean,

- 1 Right-click the `Cart` bean representation and choose Add | Method.
- 2 In the method inspector, type in the Method Name as `removeItem`.
- 3 Leave the Return Type default value of `void` unchanged, but specify the Input Parameters as `Item item`.
- 4 In the Interfaces drop-down list, specify Remote.

Retrieving the items held by the bean and their cost

Add a method to the `Cart` bean that reports the items the user currently has in the cart:

- 1 Right-click the `Cart` bean representation and choose Add | Method.
- 2 In the method inspector, type in the Method Name as `getContents`.
- 3 Specify the Return Type as `java.util.List` and leave the Input Parameters field blank.
- 4 In the Interfaces drop-down list, specify Remote.

Add a method to the Cart bean the totals the price of the items in the cart:

- 1 Right-click the Cart bean representation and choose Add | Method.
- 2 In the method inspector, type in the Method Name as getTotalPrice.
- 3 Specify the Return Type as float and leave the Input Parameters field blank.
- 4 In the Interfaces drop-down list, specify Remote.

Adding a purchase() method

Add one final method that is called when the user decides to buy the items in the cart:

- 1 Right-click the Cart bean representation and choose Add | Method.
- 2 In the method inspector, type in the Method Name as purchase.
- 3 Leave the Return Type and Input Parameters unchanged.
- 4 In the Interfaces drop-down list, specify Remote.

Working in the source code

Right-click the Cart bean representation and choose View Bean Source.

The code for CartBean should now look like this:

```
package cart_session;

import javax.ejb.*;

public class CartBean implements SessionBean {
    SessionContext sessionContext;
    java.lang.String _cardHolderName;
    java.lang.String _creditCardNumber;
    java.util.Date _expirationDate;
    java.util.List _items;
    public void ejbCreate() throws CreateException {
        /**@todo: complete this method*/
    }
    public void ejbRemove() {
        /**@todo: complete this method*/
    }
    public void ejbActivate() {
        /**@todo: complete this method*/
    }
    public void ejbPassivate() {
        /**@todo: complete this method*/
    }
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }
}
```

Working in the source code

```
public void addItem(Item item) {  
    /**@todo: complete this method*/  
}  
public void removeItem(Item item) {  
    /**@todo: complete this method*/  
}  
public java.util.List getContents() {  
    /**@todo: complete this method*/  
    return null;  
}  
public float getTotalPrice() {  
    /**@todo: complete this method*/  
}  
public void purchase() {  
    /**@todo: complete this method*/  
}  
}
```

The EJB Designer adds comments to the bodies of the methods you still must implement.

In the structure pane, you'll see two error messages in the Errors folder. Both point out that no `Item` class exists in the `CartBean`. Click an error message in the structure pane and the line of code that contains the problem is highlighted in the code. You'll address the missing `Item` class problem later.

While the EJB Designer generated the code you see in the `CartBean.java` class, it also declared the methods you added to the bean's remote interface, `Cart.java`. To see the method declarations, double-click `Cart.java` in the `cart_session` package in the project pane. It should look like this:

```
package cart_session;  
  
import javax.ejb.*;  
import java.util.*;  
import java.rmi.*;  
  
public interface Cart extends javax.ejb.EJBObject {  
    public void addItem(Item item) throws RemoteException;  
    public void removeItem(Item item) throws RemoteException;  
    public java.util.List getContents() throws RemoteException;  
    public float getTotalPrice() throws RemoteException;  
    public void purchase() throws RemoteException;  
}
```

The EJB Designer also generated code in the `CartHome.java` interface. `CartHome` contains a single `create()` method that is called when a new `Cart` bean instance is needed.

```
package cart_session;  
  
import javax.ejb.*;  
import java.util.*;  
import java.rmi.*;
```

```
public interface CartHome extends javax.ejb.EJBHome {
    public Cart create() throws CreateException, RemoteException;
}
```

Initializing the list of items

In this sample, the shopping cart list will be limited to ten items. So initialize the `_items` list by extending this `_items` variable declaration:

```
java.util.List _items;
```

so that it looks like this:

```
java.util.List _items = new ArrayList(10);
```

Adding the import statements

The `_items` and `_expirationDate` fields are defined as classes in the `java.util.*` package. Add these lines of code to the top of the bean class that import these classes right after the `import javax.ejb.*` package statement that the EJB Designer already generated for you:

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

Implementing ejbCreate()

So far the `Cart` bean doesn't really do anything. It is up to you to fill in the empty method bodies to supply the logic to the methods. Start by filling in the `ejbCreate()` method. You want to pass three parameters, the shopper's name, the shopper's credit card number, and the expiration date of the credit card to the bean's `ejbCreate()` method.

To add the parameters to the `ejbCreate()` method:

- 1 Click `ejbCreate` at the bottom of the `Cart` bean representation in the EJB Designer.
- 2 In the Input Parameters field, type this line, which declares the three parameters:

```
String cardHolderName, String creditCardNumber, Date expirationDate
```

- 3 Select Home (the default) as the Home Interface.
- 4 Right-click the bean representation and choose Regenerate Interfaces.
- 5 Return to the bean's class source code and add the code that instantiates the list that holds the shopping items and assigns the parameter values to the fields in the bean within the `ejbCreate()`

method. When you are done, your `ejbCreate()` method should look like this:

```
public void ejbCreate(String cardHolderName, String creditCardNumber,
    Date expirationDate) throws CreateException {
    _cardHolderName = cardHolderName;
    _creditCardNumber = creditCardNumber;
    _expirationDate = expirationDate;
}
```

Each `ejbCreate()` method in a bean class has a corresponding `create()` method in the bean's home interface. Because you've added the three parameters to the `ejbCreate()` method, the EJB Designer added the same parameters to the `create()` method in `CartHome.java`. Double-click `CartHome.java` in the `cart_session` package or click the `CartHome.java` tab at the top of the content pane to see the code added:

```
public interface CartHome extends javax.ejb.EJBHome {
    public Cart create(String cardHolderName, String creditCardNumber,
        Date expirationDate)
        throws CreateException, RemoteException;
}
```

Of course, you could have added the parameters to the `ejbCreate()` method directly in your source code, but you would have to remember to add it to the home interface also. By using the EJB Designer to add the parameters, you only have to enter the information once.

Implementing addItem() and removeItem()

Once you've taken care of `ejbCreate()` and its corresponding `create()` method in the bean's home interface, return to the `CartBean` class source code and implement the methods you added to the bean. Go to the `addItem()` method in the source code and fill it in so it looks like this:

```
public void addItem(Item item) {
    System.out.println("\taddItem(" + item.getTitle() + "): " + this);
    _items.add(item);
}
```

The `addItem()` method adds the specified item to the list of items held by the cart and prints out the title of the added item to the console.

Now go the `removeItem()` method in the source code and add the code that makes it look like this:

```
public void removeItem(Item item) {
    System.out.println("\tremoveItem(" + item.getTitle() + "): " + this);
    if (! _items.remove(item)) {
        throw new EJBException("The item " + item.getTitle() +
            " is not in your cart.");
    }
}
```

The `removeItem()` method removes the specified item from the list of items held by the cart and prints out the title of the removed item. If an attempt to remove the item is made when the item is not in the cart, an exception is thrown that prints an error message.

Creating an Item class

While the `Cart` bean now has methods to add and remove an item, exactly what an item is hasn't been defined. You need an `Item` class.

- 1** Choose File | New Class to display the Class wizard.
- 2** Type `Item` as the Class Name.
- 3** Check the Public and Generate Default Constructor options, unchecking all others.
- 4** Click OK.

An `Item` class is added to the `cart_session` package. Modify the resulting class so that looks like this:

```
package cart_session;

import java.io.Serializable;

public class Item implements Serializable {
    private static final long serialVersionUID = -560245896319031239L;
    private String _title;
    private float _price;
    private String _type;

    public Item(String title, float price, String type) {
        _title = title;
        _price = price;
        _type = type;
    }

    public String getTitle() {
        return _title;
    }

    public float getPrice() {
        return _price;
    }

    public String getType() {
        return _type;
    }
}
```

```

public final boolean equals(Object o) {
    // two items are equal if they have the same class and title
    if (!(o instanceof Item)) {
        return false;
    }
    Item i = (Item) o;
    return (getClass() == i.getClass()) &&
        (_title == null ? i._title == null : _title.equals(i._title));
}
}

```

The `Item` class contains three fields that hold the relevant information about an item, `_title`, `_price`, and `_type`, and three accessor methods to retrieve information, `getTitle()`, `getPrice()`, and `getType()` from the fields. It also contains an `equals()` method to determine if one instance of an item is the same as another.

Notice that the `Item` class implements the `Serializable` interface, making it possible to write and read an item to a stream. If you want to know more about serialization, consider “*Serialization*” in *Getting Started with Java* and <http://java.sun.com/j2se/1.3/docs/guide/serialization/> as places to start your research. You can search the Sun web site for additional sources of information about object serialization.

Implementing the remaining methods

Return to the source code of the bean class. Note that the structure pane no longer reports errors because the package now contains an `Item` class.

Now that you’ve implemented methods to maintain the list of items in a `Cart` instance, add the code necessary to implement the remaining methods you defined. Here is the `getContents()` method that retrieves the items in the list:

```

public java.util.List getContents() {
    System.out.println("\tgetContents(): " + this);
    return _items;
}

```

This is `getTotalPrice()`, which calculates the total cost of all the items in the cart:

```

public float getTotalPrice() {
    System.out.println("\tgetTotalPrice(): " + this);
    float totalPrice = 0f;
    for (int i = 0, n = _items.size(); i < n; i++) {
        Item current = (Item) _items.get(i);
        totalPrice += current.getPrice();
    }
    return ((long) (totalPrice * 100))/100f;
}

```

Finally, here is the `purchase()` method, which really doesn't do much in this example except to check the expiration date on the customer's credit card:

```
public void purchase() throws EJBException {
    System.out.println("\tpurchase(): " + this);
    Date today = new Date();
    if (_expirationDate.before(today)) {
        throw new EJBException("Expiration date: " + _expirationDate);
    }
    System.out.println("\tPurchasing not implemented yet!");
}
```

Save your project. Choose File | Save Project “cart_sessionbean.jpx”.

Working with the bean's deployment descriptors

While you were building the `Cart` bean in JBuilder, a deployment descriptor was being generated for you. To learn about deployment descriptors, see [Chapter 10, “Deploying enterprise beans.”](#) The deployment descriptor for this bean is very simple.

To see the source code of the `Cart` bean's deployment descriptor,

- 1 Return to the EJB Designer by clicking the `cart_module` tab at the top of the content pane or by double-clicking the `cart_module` node in the project pane.
- 2 Click the EJB DD Source tab.
- 3 Click the `ejb-jar.xml` tab, if it isn't already selected.

The `ejb-jar.xml` file is an XML file. It should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>Cart</display-name>
            <ejb-name>Cart</ejb-name>
            <home>cart_session.CartHome</home>
            <remote>cart_session.Cart</remote>
            <ejb-class>cart_session.CartBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
```

```
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>Cart</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

You can edit the fields in the XML document directly, if you like. Or you can use the Deployment Descriptor editor to make changes and additions to generated deployment descriptors.

To edit the `Cart` deployment descriptor using the Deployment Descriptor editor,

- 1 Click the icon to the left of the `cart_module` node in the project pane to see the contents of the module.
- 2 Double-click the `Cart` node in the EJB module to display the Deployment Descriptor editor.

As you can see, there are many panels available in the Deployment Descriptor editor for the `Cart` bean. To learn about using the Deployment Descriptor editor, see [Chapter 11, “Using the Deployment Descriptor editor.”](#)

Compiling your project

To compile your project, choose Project | Make Project “`cart_session.jpx`”. All the files in the project will compile and a `cart_module.jar` is created. If any compilation errors appear, fix the errors and compile the project again.

To read more about compiling enterprise beans, see [Chapter 8, “Compiling enterprise beans and creating a deployment module.”](#)

Running the Cart bean

To quickly start up a local container and deploy the `cart_module.jar` file to it that contains the `Cart` bean,

- 1 Choose Tools | Borland Enterprise Server Management Agent.
- 2 Right-click the `cart_module` node in the project pane and choose “Run Using Defaults” on the context menu.

Within JBuilder's message pane you'll see several lines that document the EJB container start up process. Eventually you will see a message that indicates `cart_module.jar` has been deployed and that the container is ready. Some EJB container statistics follow. Look carefully and you'll see that as of yet, no instances of `Cart` are in memory. To do that, you need a client program that calls the `create()` method in the home interface, `CartHome`. See [Chapter 22, "Tutorial: Creating a test client application"](#) that explains how to create a Java client application that calls the methods of `Cart`, testing their logic.

Code for cart session bean

```

package cart_session;

import javax.ejb.*;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class CartBean implements SessionBean {
    SessionContext sessionContext;
    java.lang.String _cardHolderName;
    java.lang.String _creditCardNumber;
    java.util.Date _expirationDate;
    java.util.List _items = new ArrayList(10);

    public void ejbCreate(String cardHolderName, String creditCardNumber,
        Date expirationDate) throws CreateException {
        _cardHolderName = cardHolderName;
        _creditCardNumber = creditCardNumber;
        _expirationDate = expirationDate;
    }

    public void ejbRemove() {
        /**@todo Complete this method*/
    }
    public void ejbActivate() {
        /**@todo Complete this method*/
    }
    public void ejbPassivate() {
        /**@todo Complete this method*/
    }
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }
    public void addItem(Item item) {
        System.out.println("\taddItem(" + item.getTitle() + "): " + this);
        _items.add(item);
    }
}

```

Code for cart session bean

```
public void removeItem(Item item) {
    System.out.println("\removeItem(" + item.getTitle() + "): " + this);
    if (! _items.remove(item)) {
        throw new EJBException("The item " + item.getTitle() +
                               " is not in your cart.");
    }
}
public java.util.List getContents() {
    System.out.println("\getContents(): " + this);
    return _items;
}
public float getTotalPrice() {
    System.out.println("\getTotalPrice(): " + this);
    float totalPrice = 0f;
    for (int i = 0, n = _items.size(); i < n; i++) {
        Item current = (Item) _items.get(i);
        totalPrice += current.getPrice();
    }
    return ((long) (totalPrice * 100))/100f;;
}
public void purchase() throws EJBException {
    System.out.println("\tpurchase(): " + this);
    Date today = new Date();
    if (_expirationDate.before(today)) {
        throw new EJBException("Expiration date: " + _expirationDate);
    }
    System.out.println("\tPurchasing not implemented yet!");
}
}
```

Tutorial: Creating a test client application

This tutorial creates a client application that makes test calls to the methods of the `Cart` session bean that you created in [Chapter 21, “Tutorial: Developing a session bean with the EJB Designer.”](#) It intends to show you how you might create a client application to test an enterprise bean you created.

To see the code this tutorial creates, see [“Code for the test client application” on page 22-12.](#)

The Accessibility section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 1-1](#)

Opening the `cart_session` project

Open the `cart_session` project you created in the `Cart` tutorial:

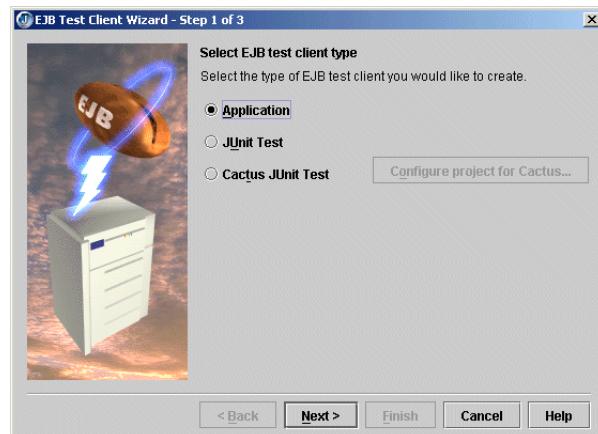
- 1 Choose File | Open Project to display the Open Project dialog box.
- 2 Navigate to the `cart_session` directory, which is likely to be in your `jbproject` directory, and double-click it.
- 3 In the right pane of the dialog box, double-click `cart_session.jpx`, or select `cart_session.jpx` and choose OK.

Using the EJB Test Client wizard

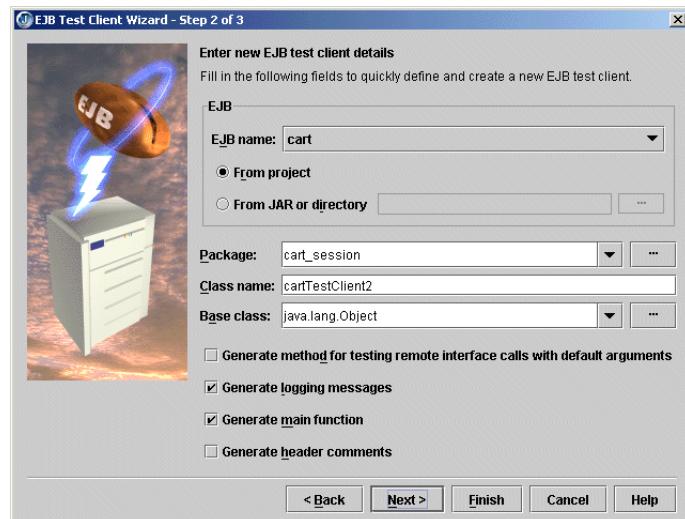
The EJB Test Client wizard can start a test client application for you. For additional information about using the EJB Test Client wizard, see “[Creating a test client application](#)” on page 9-3.

To use the EJB Test Client wizard to begin an application,

- 1 Choose File | New to display the object gallery and click the Enterprise tab.
- 2 Double-click the EJB Test Client icon.



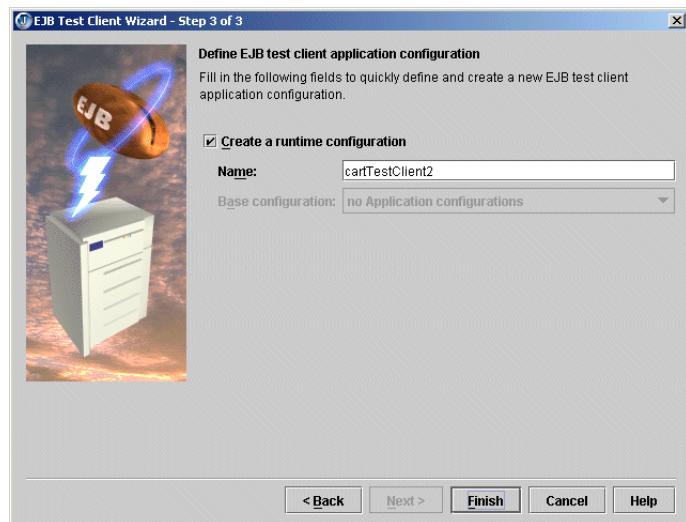
- 3 Select the Application option and click Next.



- 4 From the EJB Name drop-down list, select Cart. (Because Cart is the only enterprise bean in this project, it will be selected for you already.)

5 Uncheck the Generate Logging Messages option.

6 Choose Next.



The page that appears gives you the opportunity to create a runtime configuration for your application. If you want to, you can choose to do that now by checking the Create A Runtime Configuration check box and specifying a name you want to use as the name of your runtime configuration. This tutorial takes you through the steps of creating a runtime configuration for this application later, so you can skip this step. JBuilder gives you the option of creating a runtime configuration as you use the wizard to create a test client, or of creating a runtime configuration at any time of your choosing.

7 Choose Finish.

The EJB Test Client wizard generates this code for you:

```
package cart_session;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class CartTestClient1 extends Object {
    private CartHome cartHome = null;

    //Construct the EJB test client
    public CartTestClient1() {
        initialize();
    }

    public void initialize() {
        try {
            //get naming context
            Context context = new InitialContext();
```

```

        //look up jndi name
        Object ref = context.lookup("Cart");
        //look up jndi name and cast to Home interface
        cartHome = (CartHome) PortableRemoteObject.narrow(ref, CartHome.class);
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

//-----
// Utility Methods
//-----
public CartHome getHome() {
    return cartHome;
}
//Main method

public static void main(String[] args) {
    CartTestClient1 client = new CartTestClient1();
    // Use the getHome() method of the client object to call Home interface
    // methods that will return a Remote interface reference. Then
    // use that Remote interface reference to access the EJB.
}
}

```

Examining the generated code

The test client constructor generated by the EJB Test Client wizard accomplishes these critical tasks for you:

- 1 It obtains a reference to the JNDI server. It does this by getting a reference to an `InitialContext` object, which serves as the root of the JNDI tree.
- 2 It uses the obtained `InitialContext` object to call the `lookup()` method to get a reference to the `Cart` home object.
- 3 The obtained reference, which is an `Object`, is typecast to the bean's home interface and narrowed using the `narrow()` method of `PortableRemoteObject`. (`PortableRemoteObject.narrow()` is not required if the client is local to the bean; that is, if the bean has a local home interface instead of a remote home interface.)

The three statements that perform these tasks appear in a try/catch block to capture any exceptions thrown if any of these steps fail.

The `main()` method creates a test client instance named `client`. The remaining lines in `main()` are comments that suggest how you might proceed to create or find a bean instance and invoke its methods. The

comments to the `getHome()` method refer to the method declared immediately above the `main()`.

Adding your code to the test client

Follow the comments' suggestion and continue developing the `main()` method by calling the `getHome()` method to obtain a reference to the `Cart` bean's home interface. Place this line of code immediately after first statement in the `main()` method:

```
CartHome home = client.getHome();
```

Now you are almost ready to call the `create()` method of the home interface to create a bean instance. But before you do, declare the values you want the bean instance to have in the `main()` method:

```
String cardHolderName = "Suzy Programmer";
String creditCardNumber = "1234-5678-9012-3456";
Date expirationDate = new GregorianCalendar(2004, Calendar.JULY, 30).getTime();
```

Because `Date`, `GregorianCalendar`, and `Calendar` are in the `java.util` package, add this import statement to the import statements at the top of the file:

```
import java.util.*;
```

Creating a Cart bean instance

It's usually a good idea to place the `create()` method within a try/catch block. First declare the `cart` variable outside of a try/catch block in the `main()` method, assign the `cart` variable the value of null, then write the try/catch block that attempts to create the bean instance and prints out a message if it fails:

```
Cart cart = null;

try {
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
} catch (Exception e) {
    System.out.println("Could not create Cart session bean\n" + e);
}
```

Adding items to and removing items from the cart

Once you have a `cart` instance, you can call the methods of the bean. For this tutorial, you will attempt to add one or more items to the `cart` instance, try removing an item, and finally print out the items in the `cart`, calculating and printing the total price of `cart`'s contents.

Just in case the call to create the `cart` instance fails, you should place all calls to the bean's methods within an if statement that checks to see if a `cart` instance exists:

```
if (cart != null) {  
}  
}
```

To review, the `main()` method should now look like this:

```
public static void main(String[] args) {  
    CartTestClient1 client = new CartTestClient1();  
    CartHome home = client.getHome();  
  
    String cardHolderName = "Suzy Programmer";  
    String creditCardNumber = "1234-5678-9012-3456";  
    Date expirationDate = new GregorianCalendar(2004, Calendar.JULY,  
        30).getTime();  
  
    Cart cart = null;  
  
    try {  
        cart = home.create(cardHolderName, creditCardNumber, expirationDate);  
    } catch (Exception e) {  
        System.out.println("Could not create Cart session bean\n" + e);  
    }  
  
    if (cart != null) {  
    }  
}
```

Place the remaining code you write within the if statement. First create an `Item` instance to add to the cart:

```
Item kassemBook = new Item("J2EE Blueprints", 39.99f, "Book");
```

Then write the try/catch block that adds the new item to the cart:

```
try {  
    cart.addItem(kassemBook);  
} catch (Exception e) {  
    System.out.println("Could not add the book to the cart\n" + e);  
}
```

Add a second item to the cart:

```
Item milesAlbum = new Item("Kind of Blue", 11.97f, "CD");  
  
try {  
    cart.addItem(milesAlbum);  
} catch (Exception e) {  
    System.out.println("Could not add the CD to the cart\n" + e);  
}
```

Call a `summarize()` method that you have yet to write. `summarize()` will print out the contents of the cart including the price of each item, then it will

calculate the total cost of the items and print it too. The call to `summarize()` should look like this:

```
try {
    summarize(cart);
} catch (Exception e) {
    System.out.println("Could not summarize the items in the cart\n" + e);
}
```

Now try removing an item from the cart. Add this code to the `main()` method:

```
try {
    cart.removeItem(kassemBook);
} catch (Exception e) {
    System.out.println("Could not remove the book from the cart\n" + e);
}
```

Create another item, add it to the cart, and call the `summarize()` method to total up all the items and print the total once again:

```
Item calvertBook = new Item("Charles Calvert's Learn JBuilder 7", 49.95f,
    Book);

try {
    cart.addItem(calvertBook);
} catch (Exception e) {
    System.out.println("Could not add book to the cart\n" + e);
}

try {
    summarize(cart);
} catch (Exception e) {
    System.out.println("Could not summarize the items in the cart\n" + e);
}
```

Completing the purchase

Now that you've written methods that test the adding and removing of items from the cart, call the actual `purchase()` method to complete the transaction. Once again, make the call within a try/catch block:

```
try {
    cart.purchase();
} catch(Exception e) {
    System.out.println("Could not purchase the items:\n" + e);
}
```

Removing the bean instance

It's not absolutely necessary to remove a session bean instance when the client application is done with it, but it is good programming practice. If

the client doesn't remove it, the EJB server is likely to remove it after a period of time passes, which can be set in the bean's deployment descriptor. For this tutorial, add a `remove()` method within a try/catch block to complete the if statement and the `main()` method:

```
try {
    cart.remove();
} catch(Exception e) {
    System.out.println("Could not remove the Cart bean\n" + e);
}
```

The completed `main()` method should look like this:

```
public static void main(String[] args) {
    CartTestClient client = new CartTestClient();
    CartHome home = client.getHome();

    String cardHolderName = "Suzy Programmer";
    String creditCardNumber = "1234-5678-9012-3456";
    Date expirationDate = new GregorianCalendar(2004, Calendar.JULY,
        30).getTime();

    Cart cart = null;

    try {
        cart = home.create(cardHolderName, creditCardNumber, expirationDate);
    } catch (Exception e) {
        System.out.println("Could not create Cart session bean\n" + e);
    }

    if (cart != null) {

        Item kassemBook = new Item("J2EE Blueprints", 39.99f, "Book");

        try {
            cart.addItem(kassemBook);
        } catch (Exception e) {
            System.out.println("Could not add the book to the cart\n" + e);
        }

        Item milesAlbum = new Item("Kind of Blue", 11.97f, "CD");

        try {
            cart.addItem(milesAlbum);
        } catch (Exception e) {
            System.out.println("Could not add the CD to the cart\n" + e);
        }

        try {
            summarize(cart);
        } catch (Exception e) {
            System.out.println("Could not summarize the items in the cart\n" + e);
        }
    }
}
```

```

try {
    cart.removeItem(kassemBook);
} catch (Exception e) {
    System.out.println("Could not remove the book from the cart\n" + e);
}

Item calvertBook = new Item("Charles Calvert's Learn JBuilder 7", 49.95f,
    "Book");

try {
    cart.addItem(calvertBook);
} catch (Exception e) {
    System.out.println("Could not add book to the cart\n" + e);
}

try {
    summarize(cart);
} catch (Exception e) {
    System.out.println("Could not summarize the items in the cart\n" + e);
}

try {
    cart.purchase();
} catch(Exception e) {
    System.out.println("Could not purchase the items:\n" + e);
}

try {
    cart.remove();
} catch(Exception e) {
    System.out.println("Could not remove the Cart bean\n" + e);
}
}
}

```

Summarizing the items in the cart

The `main()` method you just completed calls a `summarize()` method that you have yet to define. In fact, if you look at the structure pane, you'll see two error messages in the Errors folder—one for each time you call `summarize()`—telling you `summarize()` is not in the test client. So begin the declaration of the `summarize()` method outside the `main()` method:

```

private static void summarize(Cart cart) throws Exception {
}

```

Compiling the test client

The first line you add to the method prints the title of the report that will be printed on the console:

```
System.out.println("===== Cart Summary =====");
```

The second line you add to the method returns all the items held within the cart:

```
List items = cart.getContents();
```

Now add a for loop that prints out the price, title, and type of item for each item in the cart:

```
for (int i = 0, n = items.size(); i < n; i++) {  
    Item current = (Item) items.get(i);  
    System.out.println("Price: $" + current.getPrice() + "\t" +  
        current.getType() + "\t" + current.getTitle());  
}
```

Finally add the lines that calculate and print the total price for all the items in the cart and print the final line of the report:

```
System.out.println("Total: $" + cart.getTotalPrice());  
System.out.println("=====");
```

Compiling the test client

To compile your test client application, right-click the `CartTestClient1` node in the project pane and choose Make.

Running the test client

To run the test client and test its logic as well as that of the Cart session bean, you must start the EJB container and deploy the bean. For information about how to do that, refer to “[Running the Cart bean](#)” on page 21-18.

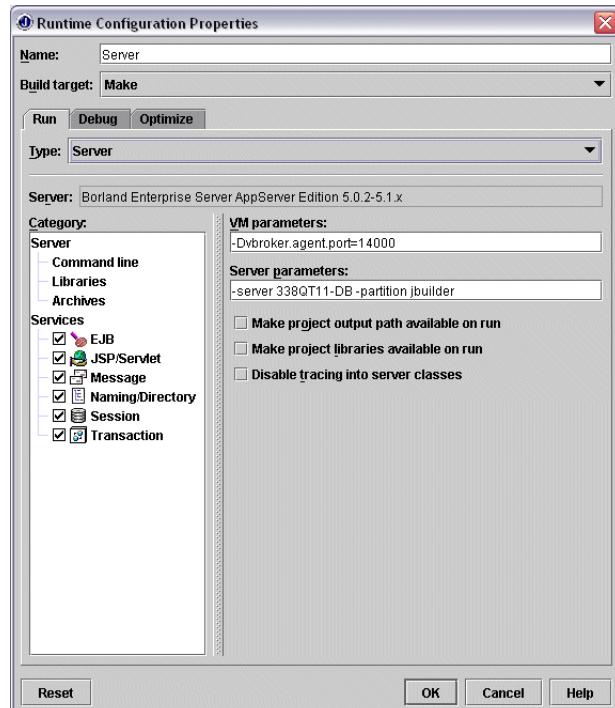
You can also create runtime configurations for both the EJB container and the client and then launch them by selecting the configuration you want from the drop-down arrow next to the Run button on the toolbar.

To create a Server configuration that starts the server and deploys Cart bean to it,

- 1 Choose Run | Configurations and click the Run tab.
- 2 Click the New button.
- 3 In the dialog box that appears, type `Server` as the value of the Name field.

- 4** Choose Server from the Type drop-down list.

Your screen should look like this:



- 5** The values shown on this page are default values that were set when you selected the Borland Enterprise Server AppServer Edition 5.1.1 - 5.2.1 as your target application server. If you need to change the default values you can do so now.
- 6** Click OK twice to close all dialog boxes.

You must now start Borland Enterprise Server Management Agent. Choose Tools | Borland Enterprise Server Management Agent.

Now you're ready to start the container. Select the Server run configuration from the drop-down list next to the Run button on the JBuilder toolbar:



The container starts up. Be patient as the start-up process takes a while. You can view the progress of the start-up process in the message window. Any errors that occur will also appear there.

Next select the Client run configuration to run your client application. The messages that appear in the message pane report the success or failure of the client application's execution.

Code for the test client application

```
package cart_session;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.util.*;

public class CartTestClient1 extends Object {
    private CartHome cartHome = null;

    //Construct the EJB test client
    public CartTestClient1() {
        initialize();
    }

    public void initialize() {
        try {
            //get naming context
            Context context = new InitialContext();

            //look up jndi name
            Object ref = context.lookup("Cart");
            //look up jndi name and cast to Home interface
            cartHome = (CartHome) PortableRemoteObject.narrow(ref, CartHome.class);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    //-----
    // Utility Methods
    //-----

    public CartHome getHome() {
        return cartHome;
    }

    //Main method

    public static void main(String[] args) {
        CartTestClient1 client = new CartTestClient1();
        CartHome home = client.getHome();
        String cardHolderName = "Suzy Programmer";
        String creditCardNumber = "1234-5678-9012-3456";
        Date expirationDate = new GregorianCalendar(2004, Calendar.JULY,
            30).getTime();
    }
}
```

```
Cart cart = null;

try {
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
} catch (Exception e) {
    System.out.println("Could not create Cart session bean\n" + e);
}

if (cart != null) {

    Item kassemBook = new Item("J2EE Blueprints", 39.99f, "Book");
    try {
        cart.addItem(kassemBook);
    } catch (Exception e) {
        System.out.println("Could not add the book to the cart\n" + e);
    }

    Item milesAlbum = new Item("Kind of Blue", 11.97f, "CD");

    try {
        cart.addItem(milesAlbum);
    } catch (Exception e) {
        System.out.println("Could not add the CD to the cart\n" + e);
    }

    try {
        summarize(cart);
    } catch (Exception e) {
        System.out.println("Could not summarize the items in the cart\n" + e);
    }

    try {
        cart.removeItem(kassemBook);
    } catch (Exception e) {
        System.out.println("Could not remove book from cart\n" + e);
    }

    Item calvertBook = new Item("Charles Calvert's Learn JBuilder 6.0",
        49.95f, "Book");

    try {
        cart.addItem(calvertBook);
    } catch (Exception e) {
        System.out.println("Could not add book to the cart\n" + e);
    }

    try {
        summarize(cart);
    } catch (Exception e) {
        System.out.println("Could not summarize the items in the cart\n" + e);
    }
}
```

Code for the test client application

```
try {
    cart.purchase();
} catch (Exception e) {
    System.out.println("Could not purchase the items: \n" + e);
}

try {
    cart.remove();
} catch (Exception e) {
    System.out.println("could not remove the Cart bean\n" + e);
}
}

private static void summarize(Cart cart) throws Exception {
    System.out.println("===== Cart Summary =====");
    List items = cart.getContents();
    for (int i = 0, n = items.size(); i < n; i++) {
        Item current = (Item) items.get(i);
        System.out.println("Price: $" + current.getPrice() + "\t" +
                           current.getType() + "\t" + current.getTitle());
    }

    System.out.println("Total: $" + cart.getTotalPrice());
    System.out.println("=====");
}
```

23

Tutorial: Creating a CORBA application

This tutorial is a feature of
JBuilder Enterprise

In this tutorial, you will create a sample client application that can query the balance in a bank account. This tutorial provides the steps to create the basic files needed to run and deploy a simple, CORBA-based distributed application.

From this example, you will learn how to:

- Define object interfaces in IDL.
- Generate client stub routines and server servant code.
- Implement the client program: initialize the ORB, bind to the `AccountManager` object, obtain the balance, print the balance, and handle exceptions.
- Implement the server object: initialize the ORB, create the Portable Object Adapter (POA), create the account manager servant object, activate the servant object, activate the POA manager and the POA, and prepare to receive requests.
- Build the example.
- Run the example with a Java client: start the VisiBroker ORB Smart Agent, start the `AccountManager` server, and run the client program.

For more information about CORBA, see [Chapter 20, “Exploring CORBA-based distributed applications.”](#)

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see “[Documentation conventions](#)” on [page 1-1](#).

Step 1: Setting up the project

This tutorial assumes that you are familiar with the JBuilder development environment. If you are new to JBuilder, refer to “The JBuilder environment” in *Introducing JBuilder* for an introduction.

- 1 Set up JBuilder and the VisiBroker ORB to “see” one another, as described in “[Setting up JBuilder for CORBA applications](#)” on [page 20-4](#).
- 2 Select File | New Project to create a new project.
- 3 Change the name in the Name field to `BankTutorial`.
- 4 Leave the name in the Directory field set to `BankTutorial`. Make sure the Generate Project Notes File option is checked.
- 5 Click Next to go to Step 2 of the wizard.
- 6 Set the JDK to JDK 1.3.1. For information on setting the JDK, see “[Setting the JDK](#)” in the “Creating and managing projects” chapter of *Building Applications with JBuilder*.
- 7 Make sure the Required Libraries tab lists the Borland Enterprise Server 5.1.1 - 5.2.1 Client library.
- 8 On Step 2, click the Finish button.
- 9 Check to ensure that the VisiBroker ORB is set as the default IDL compiler. To do this,
 - a Select Project | Project Properties to display the Project Properties dialog box.
 - b Select the Build tab. Choose the IDL tab on the Build page.
 - c Make sure that the VisiBroker ORB is selected in the IDL Compiler drop-down list.
 - d Click OK to close the dialog box.

The `BankTutorial` project appears in the project pane. Next, we will define the object interfaces by writing the account interface in IDL.

Step 2: Defining the interfaces for the CORBA objects in IDL

The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). IDL has a syntax similar to C++ and can be used to define modules, interfaces, data structures, and more. The IDL can be mapped to a variety of programming languages. The IDL mapping for Java is summarized in the Borland Enterprise Server documentation.

This topic shows how to create the `BankTutorial.idl` file for this example. The `Account` interface provides a single method for obtaining the current balance. The `AccountManager` interface creates an account for the user if one does not already exist.

To create the IDL file and define the object interfaces,

- 1 Select File | New, then select Sample IDL from the CORBA page of the object gallery.
- 2 Enter `BankTutorial.idl` in the File Name field. Click OK. A sample IDL file is created and added to the project.
- 3 Remove the generated code and insert the following IDL code.

```
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
}
```

- 4 Select File | Save All. The IDL file must be saved before it will compile.

In Step 3 you will use the Borland Enterprise Server IDL compiler to generate stub routines and servant code from the IDL specification.

Step 3: Generating client stubs and server servants

In this step, you will use the Borland Enterprise Server IDL compiler, also called the `idl2java` compiler, to generate stub routines and servant code from the IDL specification. The stub routines are used by your client program to invoke operations on an object. The servant code, along with the code you write, creates a server that implements the object. The code for the client program and server object is used as input to your Java compiler to produce the client and server executable classes.

To generate the client stubs and server servants,

- 1 Right-click the BankTutorial.idl file in the project pane.
- 2 Select Make to invoke the idl2java compiler.

If the file requires special handling, you can set IDL properties by right-clicking the IDL file in the project pane and selecting Properties. This displays the IDL tab of the Build page (Project Properties dialog box), where you can specify options for compiling remote interfaces defined in IDL.

Generated files

Because Java allows only one public interface or class per file, compiling the IDL file will generate many .java files. These files are stored in a generated directory called Generated Source, which is the module name specified in the IDL and is the package to which the generated files belong. You can view the generated files by clicking the expand glyph beside BankTutorial.idl.

The following files are generated:

- `_AccountManagerStub.java` - Stub code for the AccountManager object on the client side.
- `_AccountStub.java` - Stub code for the Account object on the client side.
- `Account.java` - The Account interface declaration.
- `AccountHelper.java` - Declares the AccountHelper class, which defines helpful utility methods.
- `AccountHolder.java` - Declares the AccountHolder class, which provides a holder for passing Account objects.
- `AccountManager.java` - The AccountManager interface declaration.
- `AccountManagerHelper.java` - Declares the AccountManagerHelper class, which defines helpful utility methods.
- `AccountManagerHolder.java` - Declares the AccountManagerHolder class, which provides a holder for passing AccountManager objects.
- `AccountManagerOperations.java` - This interface declares the method signatures defined in the AccountManager interface in the Bank.idl file.
- `AccountManagerPOA.java` - POA servant code (implementation base code) for the AccountManager object implementation on the server side.
- `AccountManagerPOATie.java` - Class used to implement the AccountManager object on the server side using the tie mechanism. For more information about the tie mechanism, see the Borland Enterprise Server documentation.

- AccountOperations.java - This interface declares the method signatures defined in the Account interface in the Bank.idl file.
- AccountPOA.java - POA servant code (implementation base code) for the Account object implementation on the server side.
- AccountPOATie.java - Class used to implement the Account object on the server side using the tie mechanism. For more information about the tie mechanism, see the Borland Enterprise Server documentation.

For more information about the generated files, see the Borland Enterprise Server documentation.

In the next step, you will implement the client program.

Step 4: Implementing the client

The client for the CORBA server application can be another application that runs locally, as outlined in this tutorial. It could also be an HTML client that runs in a web browser. You could create an applet, a JavaServer Page (JSP), or a servlet. You could also create a JSP or servlet that uses InternetBeans. For more information on developing web applications in JBuilder, see the book *Web Application Developer's Guide*.

Many of the classes used in implementing the bank client are contained in the Bank package generated by the idl2java compiler. This next set of steps creates a Java client to initialize the ORB, bind to the AccountManager object, obtain a balance and print it, and handle exceptions.

To create the client,

- 1 Create a new application that will be used for the user interface. To do this,
 - a Select File | New, and then Application from the General page of the object gallery to start the Application wizard.
 - b Accept all defaults in both Steps 1 and 2. Click Next to move to the next steps.
 - c On Step 3, make sure the Create A Runtime Configuration option is checked.
 - d Click Finish to create the application file.
- 2 Create the AccountManager interface.
 - a Select Wizards | Use CORBA Interface. Do not select the option on Step 1 of the wizard.
 - b Click Next to go to Step 2.
 - c Select BankTutorial.idl as the IDL file.

Step 4: Implementing the client

- d** Do not change the class name from `AccountManagerClientImpl1`.
 - e** Make sure `banktutorial.Bank.AccountManager` is selected in the Interface list.
 - f** Click Next to go to Step 3.
 - g** Accept the default field name of `myAccountManager`. Click Finish. The file `AccountManagerClientImpl1.java` is created and displayed in the project pane.
- 3** Repeat the same steps to create the `Account` interface:
 - a** Select Wizards | Use CORBA Interface. Do not select the option on Step 1 of the wizard.
 - b** Click Next to go to Step 2.
 - c** Select `BankTutorial.idl` as the IDL file.
 - d** Change the class name to `AccountClientImpl1`.
 - e** Select `banktutorial.Bank.Account` in the Interface list.
 - f** Click Next to go to Step 3.
 - g** Accept the default field name of `myAccount`. Click Finish. The file `AccountClientImpl1.java` is created and displayed in the project pane.
 - 4** Select File | Save All.

Binding to the AccountManager object

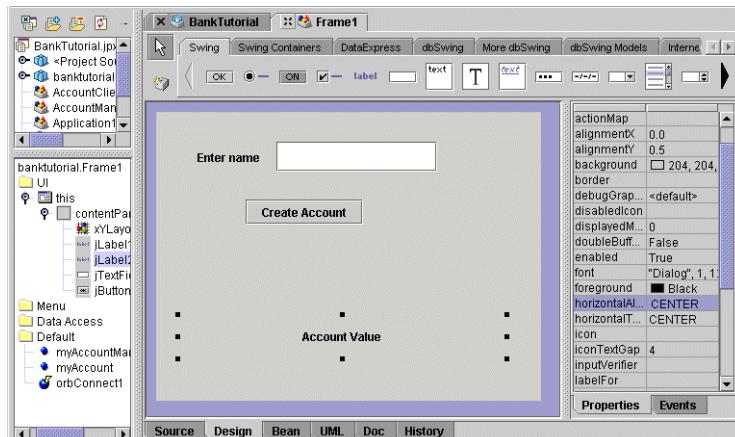
The next step is to connect the factory interface (`AccountManager`) to the Object Request Broker (ORB). To do this,

- 1** Double-click `Frame1.java` in the project pane.
- 2** Click the Design tab to open the file in the UI designer.
- 3** Select the CORBA tab of the component palette.
- 4** Select the `OrbConnect` object.
- 5** Click in the component tree to add the `OrbConnect` object to the Frame file.
- 6** Select the `myAccountManager` object in the component tree.
- 7** In the Inspector, select the `ORBConnect` property. Choose `orbConnect1` from the drop-down list.
- 8** Select the `OrbConnect1` object in the component tree.
- 9** In the Inspector, set its `initialize` property to `true`.
- 10** Choose File | Save All.

Binding the wrapper class at runtime

To bind the wrapper class at runtime,

- 1 Select the `contentPane` object in the component tree.
- 2 Change the `layout` property to `XYLayout`.
- 3 Select a `JButton` control from the Swing tab of the component palette. Click the middle of the frame to add the `JButton` control to the design surface. This button will be used to create a new account and bind it to the `AccountManager` object.
- 4 In the Inspector, select the `text` property of the `JButton`. Enter `Create account`. You may need to resize the control.
- 5 Select a `JLabel` control from the Swing tab of the component palette. Drop it in the upper-left side of the design surface.
- 6 In the Inspector, change the `text` property to `Enter name`. You may need to resize the control in order to fit the text.
- 7 Select a `JTextField` control from the Swing tab. Place it to the right of the `JLabel` control you just added. This control will be used to enter the name of the new account.
- 8 In the Inspector, remove `jTextField1` from the `text` property and leave it blank.
- 9 Select another `JLabel` control from the Swing tab. Place this control below the `JButton` control you added in Step 3. This label will display the balance in the account. Resize this control so it is almost as wide as the frame.
- 10 In the Inspector, change the `text` property to `Account value`. You can also change the horizontal alignment properties so the text is aligned in the center of the control. The UI designer should look similar to this:



Step 4: Implementing the client

Important

If you were to deploy this program to a production environment, you would not use `XYLayout`, as this layout is for design purposes only. Instead, you would switch to another layout, such as `GridBagLayout`, that will adjust the display for different platforms. To switch layouts, select the `contentPane` object in the Inspector and change the `layout` property.

- 11** Select the `JButton` control. Click the Events tab of the Inspector. Select the `actionPerformed` event, then double-click its value box to create the event. The focus moves to the editor.

- 12** Enter the following code in the `jButton1ActionPerformed` event:

```
myAccount.setCorbaInterface(myAccountManager.open(jTextField1.getText()));  
jLabel2.setText("The account balance is " + myAccount.balance());
```

- 13** Select File | Save All.

The client classes—`Application1.java` and `Frame1.java`—do the following:

- 1** Initialize the ORB, using the ORB specified in the `OrbConnect` properties.
- 2** Bind to an `AccountManager` object.

Before your client program can invoke the `balance()` method, it must first use the `bind()` method to establish a connection to the server that implements the `AccountManager` object. The implementation of the `bind()` method is generated automatically by the `idl2java` compiler. The `bind()` method requests the ORB to locate and establish a connection to the server. If the server is successfully located and a connection is established, a proxy object is created to represent the server's `AccountManagerPOA` object. An object reference to the `AccountManager` object is returned to your client program.

Your client class then needs to call the `open()` method on the `AccountManager` object to get an object reference to the `Account` object for the specified customer name.

- 3** Obtain the balance of the account using the object reference returned by `bind()`.

Once your client program has established a connection with an `Account` object, the `balance()` method can be used to obtain the balance. The `balance()` method on the client side is actually a stub generated by the `idl2java` compiler that gathers all the data required for the request and sends it to the server object.

- 4** Display the balance.

In the next step, you will implement the server.

Step 5: Implementing the server

This step shows you how to implement the server. You will initialize the ORB, create the Portable Object Adapter (POA), create the account manager servant object, activate the POA manager and the POA, and prepare to receive requests. Many of the files used in implementing the bank server are contained in the `BankTutorial` package generated when the IDL file is compiled.

To create the server,

- 1 Select File | New, then select CORBA Server Application from the CORBA page of the object gallery. Click OK to display the CORBA Server Application wizard.
- 2 Make sure that `BankTutorial.idl` is selected in the IDL File field.
- 3 Select Generate Visible Application With Monitor to create a server monitor.
- 4 Click Next to go to the next step of the wizard.
- 5 Make sure the Create A Runtime Configuration option is selected.
- 6 Click Finish to create the server and close the wizard.

The package `banktutorial.Bank.server` and the file `BankServerApp.java` are generated. The file `BankAppGenFileList.html` is also added to the project, and contains the list of generated files.

The server program, named `BankServerApp.java`, does the following:

- Initializes the Object Request Broker.
- Creates a Portable Object Adaptor (POA) with the required policies.
- Creates the account manager servant object.
- Activates the servant object.
- Activates the POA manager (and the POA).
- Waits for incoming requests.

Refer to the Borland Enterprise Server documentation for a description of the generated files.

What is the POA?

The Portable Object Adaptor, or POA, intercepts a client request and identifies the object that satisfies the client request. The object is then invoked and the response is returned to the client. The POA is a replacement for the Basic Object Adaptor (BOA) and was developed to offer portability on the server side.

The POA is designed to:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated Object Ids.
- Allow object implementations to be maximally responsible for an object's behavior.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers, or a DSI implementation.

For more information on the POA, see the Borland Enterprise Server documentation.

Step 6: Providing an implementation for the CORBA interface

The application now contains a user interface that will create `Account` objects, each with a zero balance. To make the example more interesting, we could add a database of customers, account numbers, and balances that could be looked up and used to debit and credit the account with withdrawals and deposits. However, in this simple example, we will simply add an implementation to generate a balance for a given name based on the number of characters in the name.

To implement the CORBA interface,

- 1 Expand the `banktutorial.Bank.server` package in the project pane to expose all files.
- 2 Double-click the file `AccountImpl.java` to open it in the content pane.

- 3** Find the `balance()` method (you can use *Ctrl+F*). Replace `return 0` with:

```
return _name.length()*100;
```

- 4** Select File | Save All.

In the next step, you will compile the application using the Project | Make command.

Step 7: Compiling the application

To compile the project, select Project | Make Project “BankTutorial.jpx.”

Any errors will be noted in the message pane. Correct any syntax errors before continuing to the next step.

Before compiling

If you've set up your Smart Agent to run on a port other than 14000, you need to enter the following VM parameter into the VM Parameters field on the Application tab of the Run page in the Project Properties dialog box (Project | Project Properties):

```
-Dvbroker.agent.port=port number
```

Step 8: Running the Java application

To run the tutorial application,

- 1** Start the VisiBroker ORB Smart Agent.
- 2** Start the server implementation.
- 3** Run the client application.
- 4** Test and deploy the application.

This steps are discussed in more detail in the following sections.

Starting the VisiBroker ORB Smart Agent

The VisiBroker ORB Smart Agent, or `OsAgent`, is the object location service. The client finds the server by using `OsAgent` and the server advertises services by using `OsAgent` - it's how they find each other. Ordinarily, you will want the VisiBroker ORB Smart Agent to be running on at least one host in your local network. The VisiBroker ORB Smart Agent is described in the Borland Enterprise Server documentation.

To start the VisiBroker ORB Smart Agent, choose Tools | VisiBroker Smart Agent. The VisiBroker ORB Smart Agent is toggled on and off by selecting and unselecting it from the Tools menu.

Step 8: Running the Java application

If you're running Windows NT and you want to start the VisiBroker ORB Smart Agent as an NT Service, you need to register the ORB Services as NT Services during installation. See the *Borland Enterprise Server Installation* guide for instructions. If the Services were registered, you then are able to start the VisiBroker ORB Smart Agent as an NT Service through the Services Control Panel. You do not have to start it from the Tools menu.

Alternatively, you could run the VisiBroker ORB Smart Agent from the command line by running `osagent.exe` from the `bin` directory of your Borland Enterprise Server installation.

Starting the server

To start the server implementation right-click the file `BankServerApp.java` in the project pane. Choose Run. From the drop-down menu, choose Use "Server Application." The Bank server window displays. It looks like this:

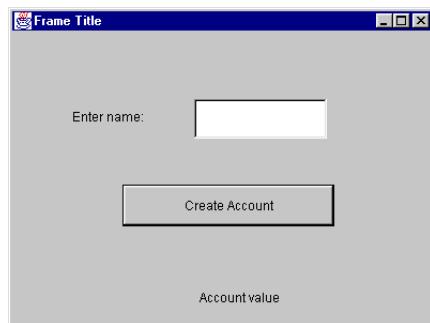


Running `BankServerApp.java` invokes the virtual machine and offers other special features. For command-line options and more information, see the Borland Enterprise Server documentation.

Running the client

To run the Java client, right-click `Application1.java` in the project pane. Choose Run. From the drop-down menu, choose Use "Application1."

The running application looks like this:

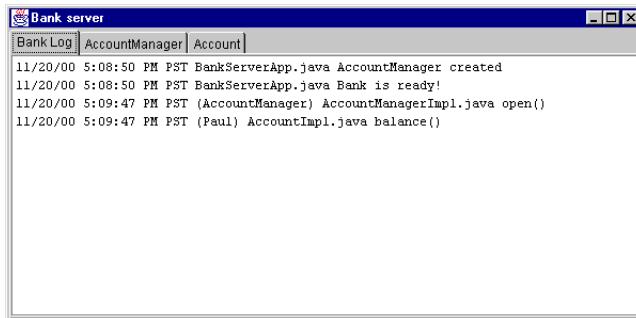


Enter a name (for example, Paul) in the text field and click the Create Account button. The generated balance is displayed in the label, as shown below:



The account balance is based on the number of characters in the name multiplied by 100.

The server monitor prints out a message every time it creates a new Account. You will see the following output on the server side:



The AccountManager page shows the number of `AccountManager` objects created. This interface creates an account for the user if one does not already exist. In this example, only one `AccountManager` interface object is created. The Account page shows the number of `Account` objects created. An `Account` object is created each time the `balance()` method is called.

Deploying the application

The VisiBroker ORB is also used in the deployment phase. This phase occurs when a developer has created client programs or server applications that have been tested and are ready for production. At this point a system administrator is ready to deploy the client programs on end-users' desktops or server applications on server-class machines.

For deployment, the VisiBroker ORB supports client programs on the front end. You must install the ORB on each machine that runs the client program. Clients that make use of the ORB on the same host share the

ORB. The VisiBroker ORB also supports server applications on the middle tier. You must install the full ORB on each machine that runs the server application. Server applications or objects that make use of the ORB on the same server machine share the ORB. Clients may be GUI front ends, applets, or client programs. Server implementations contain the business logic on the middle tier.

JBuilder includes a single DEVELOPMENT LICENSE for the Borland Enterprise Server. This license must be used in conjunction with JBuilder. Additional developers will require their own Borland Enterprise Server development licenses. Deployed programs require a separate deployment license for each server machine.

Other sample applications

Some sample CORBA applications are provided with the Borland Enterprise Server, in the `examples` directory of your Borland Enterprise Server installation. Each sample attempts to incorporate functionality described in the Borland Enterprise Server documentation.

Index

A

acknowledgement modes
 JMS messages 19-3
 message-driven beans 15-6
activating
 entity beans 14-4
 session beans 13-7
Add | ejbCreate command 3-14
adding business methods to EJB 6-8
adding properties to EJB 6-8
afterBegin() 13-9
afterCompletion() 13-9
Always Create JAR When Building option 8-1
Always Regenerate Interfaces option 3-15
Always Wrap Primary Key 4-7
application assembler
 EJB role 2-2
application EJB roles 2-2
architecture
 Enterprise JavaBeans 2-4
Arrange EJBs command 3-20
arranging beans 3-20
arrays
 map to CORBA unbounded sequences 20-13
attributes
 transaction 11-22, 18-3
Automatic Copy Of Descriptors On Save 8-1
automatic EJB interface regeneration 3-15

B

Bean designer
 Methods page 6-11
 Properties page 6-8
bean developer
 EJB role 2-2
 tasks 2-6
bean provider
 EJB role 2-2
 tasks 2-6
bean-managed persistence 14-1, 14-2, 14-20
 disadvantages 14-2
 finder methods 14-3
bean-managed transactions 18-3, 18-4
BeansExpress
 exposing EJB methods 6-11
beforeCompletion() 13-9
bidirectional relationships 4-16
binary format
 EJB modules 3-2, 6-1

Borland

 contacting 1-3
 developer support 1-3
 e-mail 1-5
 newsgroups 1-4
 online resources 1-4
 reporting bugs 1-5
 technical support 1-3
 World Wide Web 1-4

Borland Enterprise Server Deployment
 wizard 10-7

build properties

 Automatic Copy Of Descriptors On Save 8-1
 bean 8-4
 changing 8-1
 deployment module 8-1
 EAR group 8-1
 EJB module 8-4
 EJB modules 8-1

Build Properties dialog box 8-5

building

 enterprise beans 8-4
business delegate classes 5-2
business methods 14-8
 adding to enterprise bean 6-8
 entity beans 14-5
 exposing EJB 6-11
 invoked by client 17-1, 17-5
 local interface 16-7
 remote interface 6-11, 16-7
 writing 13-3

C

caches

 WebLogic entity 11-21

Cactus

 configuring 9-17
 EJB Test Client wizard 9-21
 running tests 9-23
 setup for EJB testing 9-17
 testing an enterprise bean 9-15

Cactus Setup wizard 9-17

classes

 mapping to IDL 20-13

client applications
 tutorial 22-1

client stubs

 generating 20-7
 generation 8-1

clients

- creating with JBuilder 17-9
- enterprise bean 17-1
- invoking business methods 17-5
- locate home interface 17-2
- managing transactions 17-7
- obtain remote interface 17-2
- removing bean instances 17-6
- CMP Properties 4-7
- CMP properties
 - editing Borland 4-4
- CMP Properties dialog box 4-4
- column mapping
 - multi-table WebLogic 4-11
- Common Object Request Broker Architecture *See* CORBA
- comparing
 - two EJBs 16-8
- complex data types 20-13
- container provider
 - EJB role 2-3
- container transactions 11-22
- container-managed persistence 11-35, 14-1, 14-2, 14-20
 - advantages 14-2
 - editing Borland property settings 4-4
 - limitations 14-2
 - no primary key class 14-6
 - properties 4-7
- container-managed transactions 18-3
 - adding 11-22
- containers, EJB *See* EJB container
- converting Java to IDL 20-15
- copying EJBs between views 3-19
- CORBA 20-1
- CORBA applications
 - sample 23-14
 - setting up JBuilder for 20-4
- CORBA interfaces 20-7
- CORBA tutorial 23-1
- create methods
 - entity bean 14-5, 17-3
 - session bean 17-3
- create()
 - entity beans 16-5
 - exceptions 16-3, 16-5
 - session beans 16-3
- CreateException exception 16-3, 16-5
- creating
 - EJB 2.0 session beans 3-1

D

-
- data source properties
 - editing schema 4-4
 - data sources
 - EJB 11-26
 - exporting from EJB Designer 4-23
 - importing into EJB Designer 4-2
 - modifying imported schema 4-4
 - properties 11-29
 - transaction isolation levels 11-28
 - data transfer objects(DTOs) 5-2
 - data types
 - complex mappings 20-13
 - primitive mappings 20-13
 - Database Schema Provider dialog box 4-2
 - DataExpress for EJB components 12-1
 - DataSource Properties dialog box 4-4
 - DD Source panel 11-2
 - deactivating
 - session bean instances 13-6
 - default EJB view 3-18
 - Delete View command 3-19
 - deleting EJB instances 16-2
 - deleting EJB views 3-19
 - deleting EJBs from views 3-19
 - deleting enterprise beans 16-8
 - Deploy Settings dialog boxes 10-1, 10-9
 - deployed EJB JARS
 - listing 10-10
 - deployer
 - EJB role 2-4
 - deploying
 - EJB JAR to running container 10-10
 - EJB JARs 10-8
 - enterprise beans 10-1, 10-7
 - enterprise beans to iPlanet 10-1, 10-9
 - enterprise beans to WebLogic 10-1, 10-9
 - enterprise beans to WebSphere 10-1, 10-9
 - Deployment Descriptor editor 11-1
 - CMP 1.1 panel 11-35
 - container transactions 11-22
 - Data Source panel 11-26
 - data source properties 11-29
 - displaying 3-23, 11-1
 - EJB Local References panel 11-16
 - EJB References panel 11-10
 - Environment panel 11-8
 - Finders panel 11-36
 - General panel 11-5

method permissions 11-33
Properties panel 11-13
Resource Env Refs panel 11-18
Resource References panel 11-12
Security Identity panel 11-15
Security Role References panel 11-13
security roles 11-31
server-specific panel 11-20
server-specific Properties panel 11-20
transaction isolation levels 11-28
verifying descriptors 11-39
WebLogic Cache panel 11-21
WebLogic General panel 11-19
WebLogic Idempotent Methods panel 11-25
WebLogic Properties page 11-3
WebLogic Transaction Isolation panel 11-24

deployment descriptors
application assembly information 10-5
automatically copying 8-1
changing bean information 11-5
container transactions 11-22
creating 10-2, 11-1
data source properties 11-29
data sources 11-26
editing 11-1
editing source directly 11-5
EJB 1.1 persistence 11-35
EJB local references 11-16
EJB references 11-10
EJB security identity 11-15
entity bean sample 14-20
environment properties 11-8
finder methods 11-36
information in 10-3
inserting into EAR groups 8-1
inserting into EJB modules 8-1
method permissions 11-33
properties 11-13
purpose 10-3
resource environment references 11-18
resource references 11-12
security roles 11-13, 11-31
server-specific properties 11-20
structural information 10-4
transaction isolation levels 11-28
transaction policies 11-22
verifying 11-39
viewing 11-2
viewing EJB 2.0 source code 3-23
WebLogic 11-19

WebLogic cache 11-21
WebLogic idempotent methods 11-25
WebLogic transaction isolation 11-24
WebSphere 4.0 finder methods 11-38
XML file 8-5

deployment EJB role 2-4
deployment module
 build properties 8-1
deployment options
 EJB 10-9
design patterns
 Session Bean wrap Entity Bean 12-1
Dirty Module On Layout Changes option 3-9
distributed applications
 CORBA 20-1
 EJB 2-1
 examples 20-11
 VisiBroker ORB 20-1, 20-2

documentation conventions 1-1
 platform conventions 1-3

DTO and Session Facade wizard 5-2
 generated classes 5-7

DtoAssembler classes 5-2

DTOs 5-2

durable subscriber 19-3

E

EAR files
 creating 10-6

EAR groups
 copy deployment descriptors from 8-1
 inserting deployment descriptors into 8-1

EAR wizard 10-6

Edit Column CMP Properties command 4-4

Edit Column Properties command 4-4

Edit RDBMS Relation button 4-18

Edit Table Properties command 4-4

Edit Table Reference command 4-16

editing deployment descriptors 11-1

EJB
 See also Enterprise JavaBeans
 running a Cactus test 9-23

EJB 1.x Bean Generator wizard 6-12

EJB 1.x Entity Bean Modeler wizard 7-1

EJB 1.x Interface Generator wizard 6-14

EJB 2.0 Designer wizard 3-7

EJB 2.0 modules
 creating 3-2

EJB client components 12-2
EJB components 12-1
EJB container 2-5
activating session beans 13-7
container-managed persistence 14-2
creates EJBObject 16-8
deactivating session beans 13-6
defined 2-5
implements home interface 16-1
life cycle of entity bean 14-8
life cycle of stateful beans 13-7
life cycle of stateless beans 13-6
provider 2-3
transaction support 18-2
EJB deployment options
 setting 10-9
EJB Designer
 arranging beans 3-20
 copying EJBs between views 3-19
 default view 3-18
 displaying 3-7
 drawing of relationship lines 4-20
 file naming conventions 3-9
 importing data source 4-2
 importing EJBs 3-16
 improving performance 4-20
 layout changes 3-9
 lazy loading 3-24
 loading beans into memory 3-8
 moving EJBs between views 3-18
 organizing beans in views 3-18
 quicken the display 3-8
 removing EJBs 3-21
 returning to 3-23
 setting IDE options 3-24
 speeding display 3-24
 views 3-18
EJB Designer Errors 3-22
EJB Enterprise wizard 10-8
EJB Local References panel 11-16
EJB Module From Descriptors wizard 3-4, 6-3
EJB Module wizard 3-2, 6-2
EJB modules 6-2
 automatic copy of descriptors 8-1
 build properties 8-1, 8-4
 copy deployment descriptors from 8-1
 creating 6-2
 defined 3-2, 6-1, 6-2
 file extensions 3-2, 6-1
 formats 3-2, 6-1
 from deployment descriptors 3-4, 6-3
 importing beans 3-16
 inserting deployment descriptors into 8-1
 types 3-2, 6-1
EJB QL queries 4-20
EJB roles 2-2
EJB server
 defined 2-5
 provider 2-3
EJB server components 12-2
EJB test client application
 running 9-11
EJB Test Client wizard 9-1
 Cactus Test 9-21
 JUnit Test 9-13
 test client types 9-3
EJB wizards
 EJB 1.x Bean Generator 6-12
 EJB 1.x Entity Bean Modeler 7-1
 EJB 1.x Interface Generator 6-14
 EJB Module 3-2, 6-2
 EJB Module From Descriptors 3-4, 6-3
 Enterprise JavaBean 6-5
 Project For Existing Code 3-4
 Test Client 9-1
 Use Test Client 9-7
ejbActivate() 13-3, 14-4
ejbCreate methods
 creating in EJB Designer 3-14
 passing parameters 3-14
ejbCreate() 3-14, 14-5
 called by container 14-6
 requirements 13-3, 14-5
 sample 14-6
ejbFindByPrimaryKey() 14-7
EJBHome base class
 extended by home interface 16-2
ejbHome()
 adding to entity beans 4-22
ejb-jar.xml 8-5
ejbLoad() 14-4
EJBLocalObject interface
 extended by local interface 16-7
EJBMetaData interface 17-8
EJBObject interface 16-8
 extended by remote interface 16-7
ejbPassivate() 13-3, 14-4
ejbPostCreate() 14-6
ejbRemove() 13-3, 14-4
ejbSelect()
 adding to entity beans 4-21
ejbStore() 14-5
enterprise beans 2-1
 adding fields 3-12
 adding methods 3-13
 arranging in EJB Designer 3-20
 build properties 8-1, 8-4
 business methods 14-8

Cactus testing setup 9-17
CMP properties 4-7
collapsed view in EJB Designer 3-24
comparing two 16-8
compiling 8-4
copying between views 3-19
creating EJB 2.0 session beans 3-1
creating with wizards 6-5
deleting fields 3-13
deploying 10-1, 10-7
deploying to iPlanet server 10-9
deploying to WebLogic servers 10-9
deploying to WebSphere servers 10-9
deployment options 10-9
developing with JBuilder 2-8
entity 2-7
errors in EJB Designer 3-22
expanding in EJB Designer 3-24
generating from remote interface 6-12
getting information 17-8
hot deploying 10-10
how they work 2-6
importing 3-16
layout 3-20
local and remote access 2-8
managing resources 13-3
message-driven 2-7, 15-1
modifying 2.0 3-12
modifying attributes 3-12
moving between views 3-18
organizing in EJB Designer 3-18
referencing tables 4-8
regenerating interfaces 3-15
removing from EJB Designer 3-21
removing from views 3-19
removing instances 16-8
removing methods 3-14
running 9-9
session 2-7
test client 9-1
testing 9-9
testing remote methods 9-3
testing with Cactus 9-15, 9-21
testing with JUnit 9-12
transactions 18-3
types 2-7
viewing source code 3-11
enterprise beans (CMP 2.0)
generating bean classes 4-6
Enterprise JavaBean 1.x wizard 6-5
Enterprise JavaBeans 2-1
architecture 2-4
developing 2-1
roles 2-2
specification 2-1
types 2-7
why needed 2-1
EnterpriseBean interface
extended by EntityBean 14-4
extended by SessionBean 13-2
entity bean classes
requirements 14-3
writing 14-3
entity beans 2-7
create() 16-5, 17-3
creating BMP 4-23
data sources 11-26
defined 14-1
finder methods 11-36, 14-7, 16-5, 17-3
home interface 14-10, 16-4
local by default in EJB Designer 3-9
local home interface 16-4
methods 14-5
nonexistent state 14-8
persistence 14-1
pooled state 14-9
primary keys 14-2, 17-4
providing data from 12-1
ready state 14-9
referencing 17-3
referencing tables 4-8
remote interface 14-11
removing 14-1
resolving data to 12-1
sample 14-10, 14-12, 14-14
session facades 5-1
sharing home and remote interfaces 14-10
states 14-8
writing 14-3
entity beans (2.0)
adding home methods 4-22
creating 4-1
creating relationships 4-15
creating WebLogic relationships 4-18
ejbHome() 4-22
ejbSelect() 4-21
WebSphere 4.0 finder methods 11-38
entity beans (CMP 2.0)
adding finder methods 4-20
creating from imported data source 4-2
editing properties 4-7
removing relationships 4-20
entity caches
WebLogic 11-21
EntityBean interface
entity bean implements 14-3
implementing 14-4
methods 14-4

environment properties
EJB 11-8
errors
in EJB Designer 3-22
exceptions
application transaction 18-7
application-level 18-7
system-level transaction 18-6
transaction 18-6
exporting schema
to SQL DDL 4-23
extensible structs 20-14

F

field groups
WebLogic 4-11
fields
adding and removing EJB 3-12
Find EJB command 3-19
findByPrimaryKey() 16-5, 17-4
finder methods 11-36, 14-5, 16-5, 17-2, 17-3
1.1 entity beans 14-3
adding to entity beans (2.0) 4-20
creating 14-7
default 17-4
prefix 16-5
requirements 14-7
WebSphere 4.0 11-38
FinderException exception 16-5
finding beans
views 3-19
finding entity objects
primary key 16-8
fonts
JBuilder documentation conventions 1-1

G

generated classes
DTO and Session Facade wizard 5-7
generating EJB interfaces 3-15
getEJBHome()
 EJBObject 16-8
getEJBMetaData() 16-2
 EJBHome 16-2
getHandle()
 EJBObject 16-8
getHomeHandle()
 EJBHome 16-2
getMetaData()
 enterprise bean 17-8
getPrimaryKey()
 EJBObject 16-8
getRollbackOnly() 18-8

H

handles
EJB 16-2
getting enterprise bean 16-8
home and remote interfaces
entity beans sharing 14-10
home business methods 4-22
home interface
client locates 17-1, 17-2
create methods 17-2, 17-4
creating 16-1, 16-2
creating for existing enterprise bean 6-14
defined 16-1
entity bean requirements 16-5
entity bean sample 16-7
entity beans 14-10, 16-4
extends EJBHome 16-2
finder methods 16-5, 17-2
naming 16-1
remove methods 17-4
session bean requirements 16-3
session bean sample 16-3
session beans 16-2
home methods 4-22
hot deploying
 enterprise beans 10-10

I

IDE options
 setting EJB Designer 3-24
idempotent methods 11-25
IDL files 20-7
 generating 20-15
IDL interfaces
 creating from Java interfaces 20-9
 from Java interfaces 20-15
IIOP interfaces
 creating from Java interfaces 20-9
 defining 20-10
 generating 20-11
IIOP-compliant skeletons 20-7
 generating 20-7
IIOP-compliant stubs 20-7
 generating 20-7
Import EJB dialog box 3-16
importing EJB deployment descriptors 3-6
importing enterprise beans 3-16
infrastructure EJB roles 2-3
initial naming context
 obtaining 17-2
InitialContext 18-4

inspectors
EJB 3-12, 4-7
entity bean field 4-14
entity bean method 4-14
relationship 4-16
session bean 3-10
session bean field 3-12
session bean method 3-13

interfaces
CORBA 20-13
for IIOP-compliant clients 20-7
home 16-1
Java 20-13
local home 16-2
remote 16-7
iPlanet server
deploying to 10-9
isIdentical()
EJBObject 16-8
isolation levels 11-28
WebLogic 11-24

J

Java interface definitions 20-7
Java interfaces
converting to IDL 20-7, 20-15
Java Message Service *See* JMS
Java Transaction API (JTA) 18-4
java2idl compiler 20-7, 20-9, 20-15
java2iop compiler 20-7, 20-9, 20-10, 20-11
JMS 19-1
JMS consumers 19-1
JMS destination 15-2, 15-6
JMS messages 15-1, 19-1
point to point model 19-1
publish/subscribe model 19-1
JMS producers 19-1
JMS provider
Sonic MQ Message Broker 15-7
JMS providers 15-1
JMS wizard 19-2
JNDI API 17-2
JNDI naming services 17-2, 18-4
jndi-definitions.xml file 4-3
JUnit
running EJB test case 9-15
testing an EJB 9-12
JUnit Test
EJB Test Client wizard 9-13

L

layout changes
EJB Designer 3-9
Lazily Load EJBs option 3-8
listing deployed JARS 10-10
local access
enterprise beans 2-8
local home interface 16-2
creating 16-2
entity beans 3-9, 16-4
finder methods 16-5
session beans 16-2
local interface
creating 16-7
EJBLocalObject 16-7
reference to entity bean 17-3
reference to session bean 17-3
LocalHome interface 16-2
locating
enterprise beans with primary key 16-8
locating beans
views 3-19
lookup() 17-2

M

Mandatory transaction attribute 11-22
many-to-many relationships
relationships
many-to-many 4-16
mapping
complex data types to IDL 20-13
Java classes to IDL 20-13
primitive data types to IDL 20-13
message consumers 15-1
message models
point-to-point 15-2
subscribe-and-publish 15-2
message producers 15-1
message selector 15-2, 15-6
message systems
point to point 19-1
publish/subscribe 19-1
message-driven beans 2-7, 15-1
creating 3-20
creating with JBuilder 15-4
deployment descriptors 15-6
life cycle 15-2
writing 15-3

MessageDrivenBean interface 15-3
MessageListener interface 15-4
messages
 JMS 19-1
metadata
 defined 17-8
method permissions 10-5, 11-33
method-ready state
 defined 13-7
 in transaction 13-8
methods
 adding EJB 3-13
 removing EJB 3-14
moving EJBs between views 3-18
multiplicity
 many to many 4-18
 one to one 4-18
multi-tier applications
 for distributed systems 2-1

N

naming context
 obtaining 17-2
naming EJB files
 EJB Designer 3-9
Never transaction attribute 11-22
newsgroups
 Borland 1-4
 public 1-4
nonexistent state
 entity beans 14-8
NotSupported transaction attribute 11-22

O

Only Expand Selected EJBs option 3-8
onMessage() 15-2, 15-4
 writing 15-4
operation EJB role 2-4
ORB (Object Request Broker)
 default 20-4
 properties 20-4
 setting up 20-4
OrbixWeb
 setting up for JBuilder 20-4

P

packages
 moving beans to other 3-16
 setting for enterprise beans 3-16
parsing errors
 in EJB Designer 3-22

passivation 13-6, 14-4
persistence 2-7
 1.1 entity beans 11-35
 bean-managed 14-1, 14-2
 bean-managed disadvantages 14-2
 container-managed 14-1, 14-2
 container-managed advantages 14-2
 container-managed disadvantages 14-2
 container-managed EJB 1.1 11-35
 container-managed versus bean-managed 14-2
 entity beans 14-1
persisting
 1.1 entity beans 11-35
point to point message model 19-1, 19-4
point-to-point messaging 15-2
polled state
 entity beans return to 14-9
pooled state
 entity beans 14-9
primary key
 entity bean 17-4
 locating entity objects 16-5
 obtaining 16-8
primary key class 14-2, 17-4
 entity bean requirements 14-3
 JBuilder wraps 4-7
primitive data type mappings 20-13
project
 Cactus configuration 9-17
Project For Existing Code wizard 3-4, 3-6
projects
 creating from existing EJBs 3-4
properties
 container-managed persistence 4-7
 data source 11-29
 deployment descriptor 11-13
 EJB 11-13
Properties panel
 EJB server-specific 11-20
providing data
 from entity beans 12-1
publish/subscribe message model 19-1, 19-3

Q

queue JMS destination 15-2

R

ready state
 entity beans 14-9
redeploying EJB JAR 10-10
references
 EJB 11-10

referencing enterprise beans
 create methods 17-2, 17-3
 finder methods 17-2, 17-4
 using handles 17-6
Regenerate Interfaces command 3-15
regenerating EJB interfaces 3-15
relationship lines
 EJB Designer 4-20
relationships
 between EJBs 4-16
 bidirectional 4-16
 EJB 2.0 4-15
 EJB 2.0 removing 4-20
 EJB 2.0 WebLogic 4-18
 unidirectional 4-16
remote access
 enterprise beans 2-8
remote and home interfaces
 entity beans sharing 14-10
remote home interface
 session beans 3-9
remote interface 16-7
 business methods 6-11
 client obtains reference 17-1
 creating 16-7
 creating for existing bean 6-14
 defined 16-1
 entity beans 14-11
 extends EJBObject 16-8
 generating EJB from 6-12
 reference 17-2
 reference to entity bean 17-3
 reference to session bean 17-3
 requirements 16-7
 session bean sample 16-8
remote methods
 testing 9-3
RemoteException exception 16-3, 16-5, 16-7
Remove Stub Files On Application Server Change
 option 8-1
remove()
 EJBHome 16-2
 EJBObject 16-8
removing deployed EJB JAR 10-10
removing EJB instances 16-2
removing EJB views 3-19
removing EJBs between views 3-19
removing enterprise bean instances 16-8
removing entity bean instances 14-1, 17-4
removing session bean instances 17-6
Rename View command 3-18
Required transaction attributes 11-22
RequiresNew transaction attribute 11-22
resolving data
 to entity beans 12-1
Resource Env Refs panel 11-18
resource environment references
 EJB 11-18
resource references
 EJB 11-12
RMI
 and VisiBroker ORB 20-10
rmi-iiop example 20-11
roles
 application EJB 2-2
 deployment 2-4
 EJB 2-2
 infrastructure EJB 2-3
 operation 2-4
rollback() 18-8
rolling back transactions 18-8
Route Relationship Lines option 4-20
running
 Cactus tests 9-23
runtime configurations
 Client and Server 9-9
 Server 9-9

S

sample EJB client applications
 SortClient 17-2
sample entity bean
 CheckAccount 14-12
 SavingsAccount 14-14
schema
 creating from EJB 4-23
 editing data source properties 4-4
 exporting to SQL DDL 4-23
 importing into EJB project 4-2
 modifying imported data source 4-4
Security Identity panel 11-15
security role references 10-6
security roles 10-5
 creating 11-31
 creating WebLogic 11-31
 EJB 11-13, 11-31
Security Roles panel
 Deployment Descriptor editor 11-31
server provider
 EJB 2-3
 EJB role 2-3
servers
 EJB 2-5
 running EJB 9-9

server-specific EJB properties
 editing 11-5, 11-20
ServiceLocator classes 5-2
session bean classes
 requirements 13-2
Session Bean wrap Entity Bean design pattern 12-1
session beans 2-7, 13-1
 create methods 17-3
 create() 16-3
 creating 2.0 3-10
 creating with JBuilder 13-4
 home interface 16-2
 life cycle 13-6
 local home interface 16-2
 method-ready state 13-7
 referencing 17-3
 remote by default in EJB Designer 3-9
 remove() 17-6
SessionSynchronization interface 13-8
 stateful 13-1, 17-3
 stateless 13-2, 17-3
 stateless pool 13-6
 tutorial 21-1
 types 13-1
 writing 13-2
session facades for entity beans 5-1
SessionBean interface
 extending 13-2
 methods 13-2
SessionContext interface 13-9
SessionFacade classes 5-2
SessionSynchronization interface 3-10
 methods 13-9
 stateful session beans 13-8
Set Packages For Selection command 3-16
setEntityContext() 14-4
setRollbackOnly() 13-9
setSessionContext() 13-3
sharing home and remote interfaces 14-10
Sonic MQ Message Broker 15-7
source code
 viewing 2.0 deployment descriptors 3-23
stateful beans
 life cycle 13-7
stateless beans
 life cycle 13-6
Struts client
 for EJBs 5-2
stubs
 generating client 8-1
subscription durability 15-6
Supports transaction attribute 11-22
system administrator
 EJB role 2-4

T

Table Reference editor 4-8
table references
 editing 4-16
 entity beans 4-8
test client applications
 tutorial 22-1
test clients
 creating EJB 9-1
 declaring instance of 9-7
 running EJB 9-9
 types 9-2
 using EJB 9-7
testing enterprise bean methods 9-3, 9-12, 9-15
testing enterprise beans 9-9
topic connection factories 19-3
topic JMS destination 15-2
topic subscription durability 15-6
transacted sessions 19-3
transaction
 attributes 11-22
 boundaries 17-7, 18-4
 demarcation 18-3
transaction isolation levels
 WebLogic 11-24
transaction isolation policy
 WebLogic 11-24
transaction-ready state 13-10
transactions
 and enterprise beans 18-3
 atomicity characteristic 18-1
 attributes 18-3
 bean-managed 18-3
 characteristics 18-1
 concurrency characteristic 18-1
 consistency 18-1
 container-managed 11-22, 18-3, 18-4
 demarcation 18-4
 durability characteristic 18-1
 exceptions 18-6
 global 18-4
 isolation characteristic 18-1
 isolation levels 11-28
 local 18-4
 managed by client 17-7
 policies 11-22
 rolling back 18-8
 WebLogic isolation policies 11-24
tutorials
 CORBA tutorial 23-1
 creating a test client application 22-1
 designing a session bean with the EJB
 Designer 21-1
 java2iop 20-11

U

undeploying an EJB JAR 10-10
unidirectional relationships 4-16
unit testing
 an EJB with Cactus 9-15, 9-21
 an EJB with JUnit 9-12
 Cactus setup for EJB 9-17
 running Cactus tests 9-23
 running EJB JUnit test 9-15
unsetEntityContext() 14-4
Use EJB Test Client wizard 9-7
Usenet newsgroups 1-4
UserTransaction interface 17-7, 18-4

V

value objects 5-2
verifying
 deployment descriptors 11-39
views
 arranging beans 3-20
 copying EJBs between 3-19
 creating 3-18
 default EJB 3-19
 finding EJBs 3-19
 moving EJBs between 3-18
 removing EJB 3-19
 removing EJBs from 3-19
 renaming default EJB 3-18
Views | Copy Selection command 3-19
Views | Delete View command 3-19
Views | Move Selection command 3-18
Views | Remove Selection command 3-19
Views | Rename View command 3-18
VisiBroker ORB 20-2
 and RMI 20-10
IDL compiler 20-7, 20-15
IIOP compiler 20-7, 20-10
rmi-iiop example 20-11
sample applications 23-14
setting up for JBuilder 20-4
with JBuilder 20-2

W

Web Naming 20-7
WebLogic
 deployment descriptors 8-5
 field groups 4-11
 multi-table mapping 4-11
WebLogic Cache panel 11-21
WebLogic General panel 11-19
WebLogic Idempotent Method panel 11-25
WebLogic Properties page
 Deployment Descriptor editor 11-3
WebLogic RDBMS Relation editor 4-18
WebLogic Security Roles panel
 Deployment Descriptor editor 11-31
WebLogic servers
 deploying to 10-9
WebLogic Transaction Isolation panel 11-24
weblogic-ejb-jar.xml 8-5
WebSphere servers
 deploying to 10-9
wizards
 Cactus Setup 9-17
 EJB 1.x Bean Generator 6-12
 EJB 1.x Entity Bean Modeler 7-1
 EJB 1.x Interface Generator 6-14
 EJB Module 3-2, 6-2
 EJB Module From Descriptors 3-4, 6-3
 EJB Test Client 9-1
 Enterprise Deployment 10-7
 Enterprise JavaBean 6-5
 JMS 19-2
 Use Test Client 9-7
wrapped primary key class 4-7

X

XML format
 EJB modules 3-2, 6-1

