WebLogic™ Server 6.1

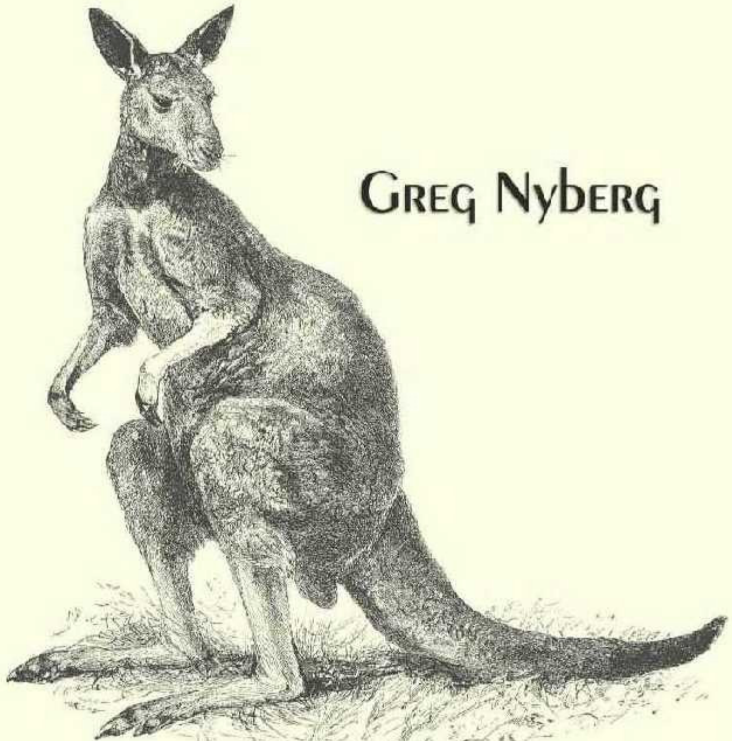
Workbook for Enterprise JavaBeans™, 3rd Edition

COVERS
EJB 2.0

Windows
2000

Windows
NT

Sun OS



Greg Nyberg

Titan®
Books

About the Series

Each of the books in this series is a server-specific companion to the third edition of Richard Monson-Haefel's best-selling and award-winning *Enterprise JavaBeans* (O'Reilly 2001), available at *http://www.titan-books.com/* and at all major retail outlets. It guides the reader step by step through the exercises called out in that work, explains how to build and deploy working solutions in a particular application server, and provides useful hints, tips, and warnings.

These workbooks are published by Titan Books in the context of a friendly agreement with O'Reilly and Associates, the publishers of *Enterprise JavaBeans*, to provide serious developers with the best possible foundation for success in EJB development on their chosen platforms.

WebLogic™ Server 6.1 Workbook for *Enterprise JavaBeans™* *3rd Edition*

Greg Nyberg

TITAN™
Books
Minneapolis

WebLogic Server 6.1 Workbook for *Enterprise JavaBeans*, *3rd Edition*, by Greg Nyberg

Published by Titan Books, Minneapolis, Minnesota.

Series Editor: Brian Christeson

Companion volume to *Enterprise JavaBeans*, *3rd Edition*, by Richard Monson-Haefel, published by O'Reilly & Associates, Inc., 2001, available at *http://www.titan-books.com/*.

Printed in the United States of America by Fidlar Doubleday, Inc.

Printing History:

October 2001 First Edition

ISBN 1-931822-46-8

for Abby, Matthew, and their Mommy

Table of Contents

Table of Figures

Preface

This workbook is designed to be a companion for O'Reilly's *Enterprise Java Beans, Third Edition,* by Richard Monson-Haefel, for users of BEA's WebLogic application server. It is one of a series of vendor-specific workbooks Titan Books is publishing to accompany the latest edition of that spectacularly best-selling work.

The goal of this workbook is to provide practical, step-by-step instructions for installing and configuring the WebLogic Server product, and for deploying the example programs from *Enterprise JavaBeans* in a WebLogic environment.

This workbook also discusses key WebLogic-specific requirements, best practices, and the use of WebLogic-specific tools such as the WebLogic Administration Console. You will build web-based versions of many of the example programs, gaining insight into the relationships between web applications and EJB applications in the WebLogic Server product.

This book is based on the production release of the WebLogic Server 6.1 product and includes all of the EJB 2.0 examples from the Enterprise JavaBeans book. All of the examples and techniques demonstrated in this book work properly with the 6.1 GA release of WebLogic, but not with the WebLogic 5.1, 6.0, or early 6.1 beta releases.

Contents of This Book

This workbook is divided into two kinds of sections:

- **Server Installation and Configuration** – The first section walks you through the process of downloading, installing, and configuring the WebLogic 6.1 product, and building the WebLogic "domain" and services required for the example programs. It also describes the working area used to build the example programs and the process for downloading and installing each exercise archive file.
- **Exercises** – These sections contain step-by-step instructions for downloading, building, configuring, and deploying the example programs for each exercise called out in *Enterprise JavaBeans, Third Edition* (which, for brevity, in this workbook will refer to as "the EJB book"). The text also examines many of the source files, descriptors, and other components used in each exercise to point out key WebLogic-specific issues and techniques.

Because WebLogic Server 6.1 is an EJB 2.0-compliant product, the EJB 1.1 exercises called out in *Enterprise JavaBeans* are not included in this workbook.

The workbook text for each exercise will depend on the amount of new material introduced in the exercise and the configuration tasks required for the example programs, but generally each exercise will contain text describing the following activities:

- Downloading and building the example code
- Configuring database tables or other system services
- Examining the standard EJB descriptor file, *ejb-jar.xml*
- Examining the WebLogic-specific descriptor files
- Deploying the EJB components to WebLogic
- Examining and running the example programs
- Examining and running the example web-application components

The exercises are designed to be built and run in order. Every effort was made to eliminate dependencies between the exercises by including all components for each exercise within each exercise archive file, but occasional dependencies exist nevertheless. The workbook text will highlight these dependencies when they occur and explain the prerequisites for each exercise. You should configure and clear database tables according to the instructions in each exercise, to avoid problems in subsequent exercises.

On-Line Resources

This workbook is designed for use with the EJB book and with downloadable example code, both available from our web site:

http://www.titan-books.com/

The code and scripts are contained in the following archive files:

dbscripts.jar	Contains scripts for starting the Cloudscape tools, *Cloudview* and *ij*, and SQL scripts for building workbook database tables in multiple RDBMS technologies. Extracted to a convenient work directory.
bookdb.jar	Pre-built Cloudscape database for workbook exercises. Extracted to the Cloudscape *data* directory if Option #3 is chosen for database configuration during setup.
titanapp_empty.jar	Pre-built *titanapp* directory structure representing an "empty" exploded enterprise archive (.ear) file. Extracted to *applications* directory of new *ejbbook* domain.
work_ejbbook_win.jar	Contains common scripts or files required in the work root directory for NT/Win2K installations.
work_ejbbook_sol.jar	Contains common scripts or files required in the work root directory for SunOS installations.

ex04_1.jar – ex13_2.jar	Eighteen separate archive files containing code, build scripts, and descriptor files for individual exercises. Extracted to the work root directory to create separate *ex##_#* subdirectories.
everything.jar	Single file containing all archive files, as a convenience for downloading.

We will post errata at the download site, and any updates required to support specification or product changes. You will also find links to many popular EJB-related sites on the internet.

BEA Systems maintains a significant web presence with sites devoted to partners, developers, and product documentation. The following links will direct you to important areas in BEA's site:

http://www.bea.com/index.shtml

BEA home page containing links to all areas of their site.

http://developer.bea.com/index.jsp

Developer home page, requires (free) registration for access to all areas.

http://edocs.bea.com/index.html

General e-docs documentation home page. Links to documentation for all BEA products.

http://edocs.bea.com/wls/docs61/index.html

WebLogic Server 6.1 e-docs documentation home page. The place to start looking for most 6.1-related information.

http://newsgroups.bea.com/cgi-bin/dnewsweb

Web-based interface for newsgroups covering all aspects of BEA technology. Also available in NNTP format from *newsgroups.bea.com*. These are very active newsgroups monitored by representatives from BEA engineering. Searching the newsgroup archives is a very good way to find information not contained in the e-docs.

I hope you find this book useful in your study of Enterprise JavaBeans and WebLogic Server. Comments, suggestions, and error reports on the text of this workbook or the contents of the downloaded example code are welcome and appreciated. Please e-mail them to:

weblogic-workbook@yahoogroups.com

Conventions Used in This Book

Italics are used for:

- Filenames and pathnames
- Names of hosts, domains, applications

- URLs and email addresses
- New terms where they are defined

Boldface is used for:

- Emphasis
- Buttons, menu items, window and menu names, and other UI items you are asked to interact with

`Constant width` is used for:

- Code examples and fragments
- Sample program output
- Class, variable, and method names, and Java keywords used within the text
- SQL commands, table names, and column names
- XML elements and tags
- Commands you are to type at a prompt

`Constant width bold` is used for emphasis in some code examples.

`Constant width italic` is used to indicate text that is replaceable. For example, in *`BeanName`*`PK`, you would replace *`BeanName`* with a specific bean name.

An Enterprise JavaBean consists of many parts; it's not a single object, but a collection of objects and interfaces. To refer to an Enterprise JavaBean as a whole, we use the name of its business name in Roman type followed by "bean" or the acronym "EJB." For example, we will refer to the Customer EJB when we want to talk about the enterprise bean in general. If we put the name in a constant width font, we are referring explicitly to the bean's class name, and usually to its remote interface. Thus `CustomerRemote` is the remote interface that defines the business methods of the Customer bean.

Acknowledgements

Thank you to everyone who helped with the writing of this workbook, directly or indirectly. I was fortunate to be working with a consulting organization (Object Partners, Inc.) that supported me during the development of this workbook, and for that I am very grateful. My editor Brian Christeson helped to apply structure and polish to very rough piece of work, for which I thank him. My technical reviewers and colleagues at Object Partners were invaluable in finding and correcting errors in the example programs and workbook.

Thanks, Richard, for believing I could pull this off and giving me the opportunity to write my first book. It has been a rewarding experience.

Finally, thanks to Meredith for putting up with the late nights and my single-mindedness during this process, and for always supporting me in my pursuits.

Server Installation and Configuration

The EJB 2.0 examples from the O'Reilly book Enterprise JavaBeans 3rd Edition (referred to hereafter as the "EJB book") require the EJB 2.0-compliant version of the WebLogic Server product, version 6.1. This chapter will describe the steps required to download, install, and configure the WebLogic Server product to provide a platform for the subsequent exercises and examples.

BEA provides a very complete set of installation, administration, and programming documentation on their "edocs" web site, *http://edocs.bea.com/*, including limited search capability. Throughout this workbook references will be made to the online documentation including URLs where possible.

Installing WebLogic Server Software

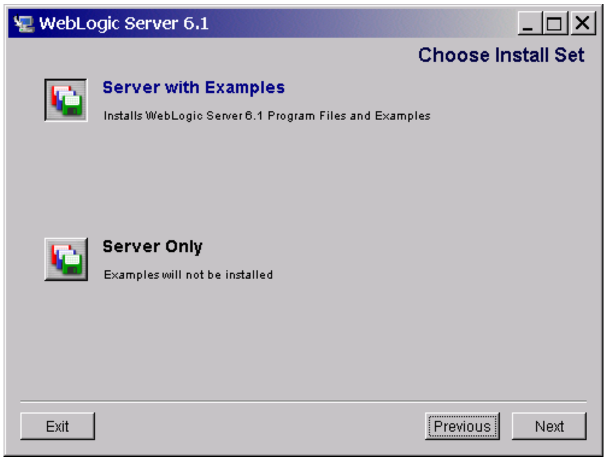
BEA provides a downloadable evaluation version of their WebLogic Server product directly from their download web site (*http://commerce.bea.com/downloads/products.jsp*). You will receive a license valid for approximately 30 days to allow you to evaluate the product.

Download WebLogic Server version 6.1 for your platform. You may need to register with the site if you have never done so. Be aware that the distribution archive is very large (75 MB).

Installing on NT/Win2K Machines

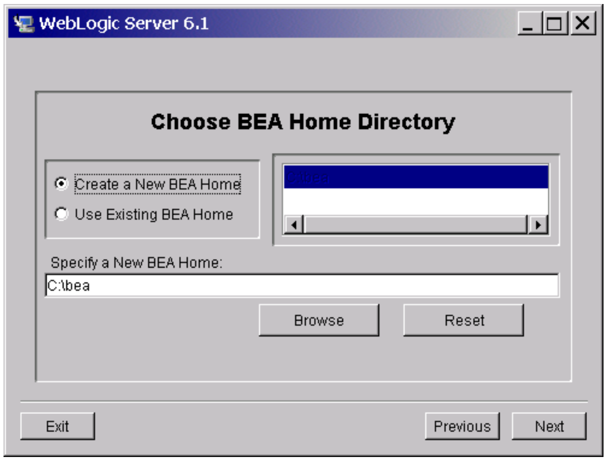
The Windows NT/Windows 2000 version is packaged as a self-extracting, self-installing application. Run the install application, choose your language, and accept the license agreement if any. The next dialog allows you to install only the server product, or the server product and the canned examples from BEA. We want both, so click on **Server with Examples** and click on **Next**.

Figure 1: Choosing the installation set


WebLogic Server 6.1
Choose Install Set
Server with Examples
Installs WebLogic Server 6.1 Program Files and Examples
Server Only
Examples will not be installed
Exit
Previous
Next

The next dialog (Figure 2) asks you to select the BEA "home" directory for all BEA products, including WebLogic Server 6.1 and the JDK 1.3.1 installation. The online documentation provides guidelines for selecting a BEA home directory. The default for a typical NT/Win2K machine is *c:\bea*.

Figure 2: Choosing a home directory



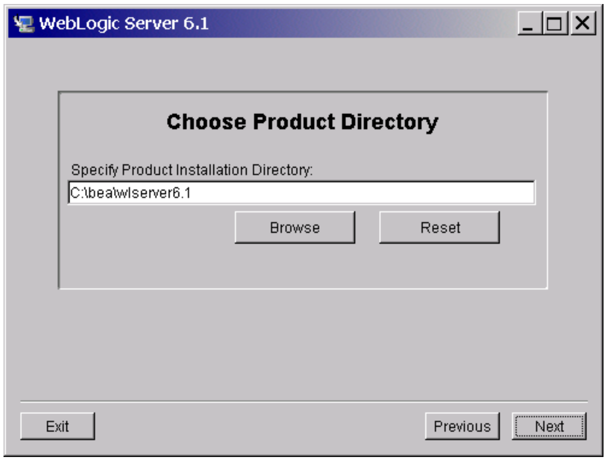
[● Create a New BEA Home] Use Existing BEA Home

Specify a New BEA Home: C:\bea

If you already have a BEA home directory, select **Use Existing BEA Home** and use the same directory. The new version of the server product will be installed alongside any existing version in the BEA home directory. Click on **Next** to continue.

The next dialog asks you to choose the product installation directory. It should be the directory below the BEA home directory, and is assumed in this workbook to be *c:\bea\wlserver6.1* as shown in Figure 3.

Figure 3: Choosing a product directory

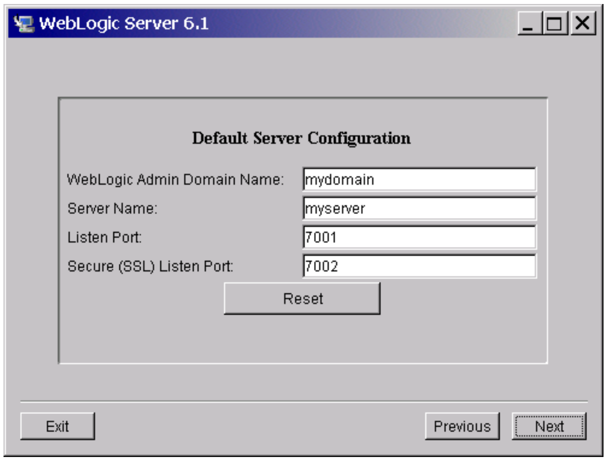


Choose Product Directory

Specify Product Installation Directory: C:\bea\wlserver6.1

WebLogic creates a series of "domains" during the installation, including an empty domain and an *examples* domain containing many example components and applications. The concept of a domain is described in detail in the online documentation and will be discussed briefly in the section on configuring the domain you will use in exercises. The next dialog (Figure 4) allows you to define the name of this empty domain (*mydomain*), the name of the default server (*myserver*) and the default listen ports for normal and secure socket communication.

Figure 4: Configuring the default server



Default Server Configuration

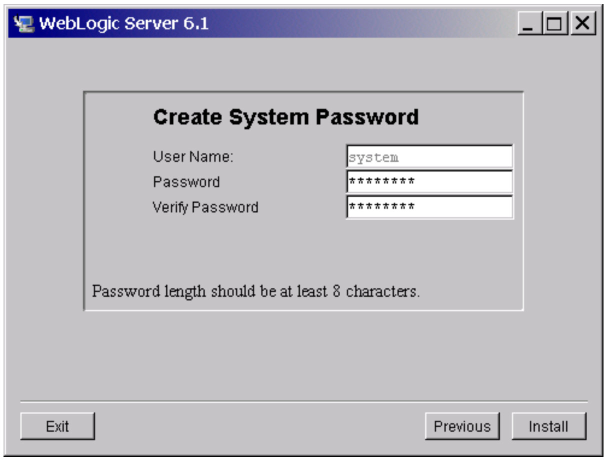
WebLogic Admin Domain Name:	mydomain
Server Name:	myserver
Listen Port:	7001
Secure (SSL) Listen Port:	7002

Leave all of these values as shown and click on **Next** to continue.

The next dialog allows you to configure the default WebLogic server instance as a service in NT/Win2K. Select **No** and click on **Next** to continue.

In the next dialog (Figure 5), choose an easy-to-remember password. This password is stored in encrypted form in a file in each domain directory. Because it is not possible to reset it or otherwise recover if you forget the password, be sure you write it down. Many people use the password “weblogic” on their local machines. Enter your chosen password twice and click on **Install** to continue.

Figure 5: Setting your password


WebLogic Server 6.1

Create System Password

User Name: system
Password ********
Verify Password ********

Password length should be at least 8 characters.

The installation process should now commence and take approximately 3-5 minutes.

The install process will create a directory structure under the BEA home directory similar to this:

```
C:\bea\
       jdk131\
       wlserver6.1\
                   bin\
                   config\
                   ext\
                   lib\
                   samples\
                   uninstaller\
```

The following is a brief description of the contents of each directory.

♦ *jdk131* – Contains a full copy of the Java 1.3.1 SDK. All of the build scripts provided by BEA and those provided as examples for this workbook assume this directory is present.

♦ *wlserver6.1\bin* – Because WebLogic is actually a Java application run via a standard JVM, the only files in the *\bin* directory are DLLs and a few scripts.

♦ *wlserver6.1\config* – This is a root directory for all of the "domains" defined on this machine. Under this directory you should have an *examples* directory, a *mydomain* directory, and a *petstore* directory. Each of these directories represents a separate bootable domain installed on this machine. These individual domain directories are called *domain root* directories throughout this workbook.

♦ *wlserver6.1\ext* – Extensions directory, not important for our purposes.

♦ *wlserver6.1\lib* – Directory containing all runtime library .jar files used by WebLogic. The most important file in this directory is the *weblogic.jar* file which must be in the Java classpath when WebLogic server is booted.

♦ *wlserver6.1\samples* – Contains source code and build scripts for the canned examples provided with the product.

♦ *wlserver6.1\uninstaller* – Self-explanatory, not important for our purposes.

Proceed to the "Final Installation Steps" section to verify your license file and complete the installation process.

Installing on SunOS Machines

Download the *weblogic610_sol.bin* installation file from the BEA web site and place it somewhere convenient on the SunOS machine. It is best to use a *weblogic* user account rather than installing and running the product as *root*. You may need the system administrator to create an */opt/bea* directory owned by the *weblogic* user account prior to beginning the installation, since that is the default location for the BEA Home directory.

Console mode installation is available on UNIX platforms. Additional information is available in the online documentation (*http://e-docs.bea.com/wls/docs61/install/instcon.html*). Begin the installation using:

```
sh weblogic610_sol.bin -i console
```

Choose your Locale in the first text-based menu that appears. Press **<ENTER>** to page through any license agreement information or notes displayed by the installation program, then agree to the terms of the license to continue.

Next, you will be asked to specify the type of installation:

```
Choose Install Set
------------------
Please Choose the Install Set to be installed by this installer.
  ->1- Server and Examples
    2- Server Only
    3- Customize...
ENTER THE NUMBER FOR THE INSTALL SET, OR <ENTER> TO ACCEPT THE
DEFAULT
   : 1
```

Choose **Server and Examples** to include the *examples* domain and all sample code in the installation, allowing you to configure and test the *examples* domain to test the installation.

The next menu will be:

```
Choose BEA Home Directory
-------------------------
    1- Create a New BEA Home
    2- Use Existing BEA Home
Enter a number: 1
```

Choose **Use Existing BEA Home** if you have a previous version of WebLogic 6.0 or any other recent WebLogic product which created the BEA Home directory structure (WebLogic 5.1 did not use this). Otherwise choose **Create a New BEA Home**.

If you select **Use an Existing BEA Home**, choose it (by number) from the displayed menu:

```
1- /opt/bea
Existing BEA Home: 1
```

Alternately, specify the new BEA Home directory if you chose to create a new one:

```
Specify a New BEA Home: /opt/bea
```

The next step is to choose the product directory. The workbook examples and documentation will assume the product is installed in the */opt/bea/wlserver6.1* directory:

```
Choose Product Directory
------------------------
    1- Modify Current Selection (/opt/bea/wlserver6.1)
    2- Use Current Selection (/opt/bea/wlserver6.1)
Enter a number: 2
```

When you are ready to continue, choose the **Use Current Selection** option to proceed to the next step.

WebLogic creates a series of “domains” during the installation, including an empty domain and an *examples* domain containing many example components and applications. The concept of a domain is described in detail in the online documentation.

The next menu allows you to define the name of this empty domain (*mydomain*), the name of the default server (*myserver*) and the default ports for normal and secure socket communication:

```
Default Server Configuration
----------------------------
    1- Modify WebLogic Admin Domain Name (mydomain)
    2- Modify Server Name (myserver)
    3- Modify Listen Port (7001)
    4- Modify Secure (SSL) Listen Port (7002)
    5- Done Configuration
Enter a number: 5
```

Accept the defaults and continue.

The next step is to set the system password. This password is stored in encrypted form in a file in each domain root directory. Many people use the password "weblogic" on development machines. Enter your chosen password twice.

```
Create System Password
----------------------
Password:
Verify Password:
```

💣 Warning: Because it is not possible to reset or otherwise recover the system password if you forget it, be sure to write it down!

The installation process should now commence and take approximately 3-5 minutes.

When the process is complete you should see the following message:

```
Install Complete
----------------
Congratulations. 'WebLogic Server' has been successfully installed
to:
   /opt/bea/wlserver6.1
PRESS <ENTER> TO EXIT THE INSTALLER:
```

That's it. Press `<ENTER>` and proceed to the final installation steps, below.

Final Installation Steps

Examine the contents of the BEA home directory, *\bea* (NT/Win2K) or */opt/bea* (SunOS) assuming you followed the installation guidelines. If WebLogic 6.1 is the first BEA product installed in this BEA home directory, the directory will contain a *license.bea* file created by the installation process with your 30-day evaluation license for the WebLogic Server 6.1 product. Examine the *license.bea* file and look for a `<license-group>` element for release 6.1:

```
<license-group format="1.0" product="WebLogic Server" release="6.1">
  <license
    component="WebLogic"
    expiration="2001-12-25"
    ip="any"
    licensee="BEA Evaluation Customer"
    type="EVAL"
    units="5"
    signature="..."
  />
  ...
</license-group>
```

If your *license.bea* file includes a `<license-group>` element like this, you're all set to go.

Unfortunately, if you previously installed WebLogic 6.0 or any other BEA product in this BEA home directory, the WebLogic 6.1 evaluation license was not placed in the *license.bea* file. There should be a new file in the directory called *license_new.bea* containing the WebLogic 6.1 license information. You need to place this license information in the *license.bea* file using one of the following techniques:

1. Edit the *license_new.bea* file, copy the entire WebLogic 6.1 `<license-group>` element to the clipboard, and paste it in the *license.bea* file within the `<bea-licenses>` section.
2. Rename the old *license.bea* file to something else and rename *license_new.bea* to *license.bea*.
3. Use the UpdateLicense utility provided by BEA to merge the old and new license files. This utility essentially performs the copy/paste outlined in the first option. Open a command prompt or telnet window, navigate to the home directory, and run the *UpdateLicense.cmd* or *.sh* script supplying the new license file as a command-line parameter:

   ```
   C:\bea>UpdateLicense license_new.bea
   ```

The first option is the recommended one. You might as well get used to editing the *license.bea* file, because if you continue to work with BEA products there will be many opportunities to edit and merge license files as new products are installed and old licenses expire.

Celebrate! You are now ready to enable EJB 2.0 functionality and verify the installation with the *examples* domain.

Enabling EJB 2.0 Capability in WebLogic

At the current time, the standard install of WebLogic does not, by default, enable the EJB 2.0 capability of the product. If you examine the contents of the *wlserver6.1/lib* directory, you may or may not see a file called *ejb20.jar*. If this file is absent you will encounter runtime errors when you try to deploy EJB 2.0 beans or use other EJB 2.0 features.

The *ejb20.jar* file is available for download from the standard WebLogic download site *(http://commerce.bea.com/downloads/products.jsp)*. In the WebLogic Server products section, choose the *EJB 2.0 Upgrade* from the download page, download the file, and place it in the *wlserver6.1/lib* directory. The presence of this file will enable EJB 2.0 capability in the product.

Verifying Installation using WebLogic Examples

The canned examples provided with WebLogic Server demonstrate a wide range of features and capabilities of the product. They are also an excellent way to verify a product installation and troubleshoot configuration problems, since BEA is normally willing to provide support to evaluation users attempting to configure and run the examples.

Reviewing the Examples Domain

First, examine the contents of the *examples* domain root directory. Recall that each domain in the installation is a separate directory under the *wlserver6.1/config* directory. The *examples* domain root directory is therefore *bea/wlserver6.1/config/examples*, where you should see a set of files and directories something like this:

NT/Win2K

```
applications\
clientclasses\
logs\
serverclasses\
xml\
ca.pem
config.xml
democert.pem
demokey.pem
fileRealm.properties
SerializedSystemIni.dat
setExamplesEnv.cmd
startExamplesServer.cmd
```

SunOS

```
applications\
clientclasses\
logs\
serverclasses\
xml\
ca.pem
config.xml
democert.pem
demokey.pem
fileRealm.properties
SerializedSystemIni.dat
setExamplesEnv.sh
startExamplesServer.sh
```

Key files/directories beneath this domain root directory include:

- *applications* – This is a “magic” directory in WebLogic Server. Any web-application (.war) files, EJB component (.jar) files, or enterprise-application (.ear) files placed in the applications directory are automatically deployed by the server during boot. In addition, components in this directory are automatically deployed and/or redeployed at runtime should the file appear in the directory or have its timestamp change.
- *config.xml* – The mother of all configuration files, this XML file contains all of the configuration information for all servers, clusters, resources, and applications defined in this domain. It can be edited by hand while the server is not running if you know what you are doing.

- *fileRealm.properties* – File used by the default file-based security realm in WebLogic. Contains users, groups, passwords (encrypted), and ACLs.
- *SerializedSystemIni.dat* – A mysterious but important file used in the encryption process for the *fileRealm.properties* passwords. The trick is that whenever you create a new domain, you must manually copy *fileRealm.properties* and *SerializedSystemIni.dat* to the new domain root directory before attempting to boot the domain. If you do this, the system password will be the same as the source domain you copied from. If you do not, there is no way to determine the correct system password for the new domain or indeed to boot the domain to add/modify user information.
- *startExamplesServer.cmd* or *.sh* – Script used to start the *examples* domain server process.
- *setExamplesEnv.cmd* or *.sh* – Script used to modify the current shell's CLASSPATH to allow the running of example client applications, etc.

Booting the Examples Domain

The first step is to boot the *examples* domain and check that the basic installation and configuration was a success. To ensure that environment variables are controlled and consistent and shortcut problems are avoided, this workbook will direct you to perform the majority of server-related operations for the NT/Win2K platform using a Command Prompt window rather than Start Menu items.

Booting the Examples Server – NT/Win2K:

1. Open a Command Prompt window
2. Change to *\bea\wlserver6.1\config\examples* directory
3. Execute the *startExamplesServer.cmd* script
4. Provide the system password when asked during the boot process.
5. Watch the output during the boot process. Because the logging severity threshold is set to *Error* or higher, there will be very few messages unless there are errors.

   ➢ Tip: There are two ways to avoid Step 4 each time you boot:

   - Create a small text file called *password.ini* in the domain root directory containing the system password in clear text (with no newline).
   - Add `-Dweblogic.management.password=`*`thepassword`* to the *java* command in the start script, replacing *`thepassword`* with your system password.

Booting the Examples Server – SunOS:

1. Change to */opt/bea/wlserver6.1/config/examples* directory
2. Execute the *startExamplesServer.sh* script

3. Provide the system password when asked during the boot process.
4. Watch the output during the boot process. Because the logging severity threshold is set to *Error* or higher, there will be very few messages unless there are errors.

➢ Tip: There are two ways to avoid Step 3 each time you boot:

- Create a small text file called *password.ini* in the domain root directory containing the system password in clear text (with no newline).
- Add `-Dweblogic.management.password=`*`thepassword`* to the *java* command in the start script, replacing *`thepassword`* with your system password.

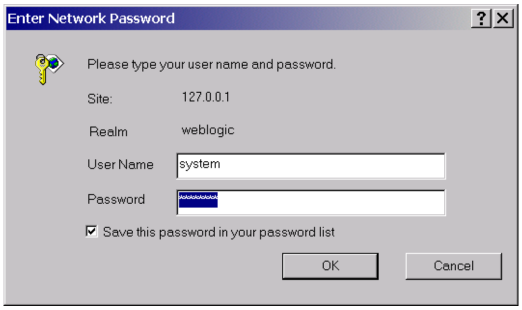
Opening the WebLogic Management Console

Assuming the server has booted without errors, it is time to connect to the domain using the management console. Essentially the management console is nothing more than a web application installed and deployed in the domain. You start the console by opening a browser window and connecting to the */console* web application. Assuming you've used the default listen ports, the URL would be:

http://localhost:7001/console or *http://servername:7001/console*

The console is protected via Access Control Lists (ACLs), so an HTTP challenge window (Figure 6) will pop up asking for the system username and password:

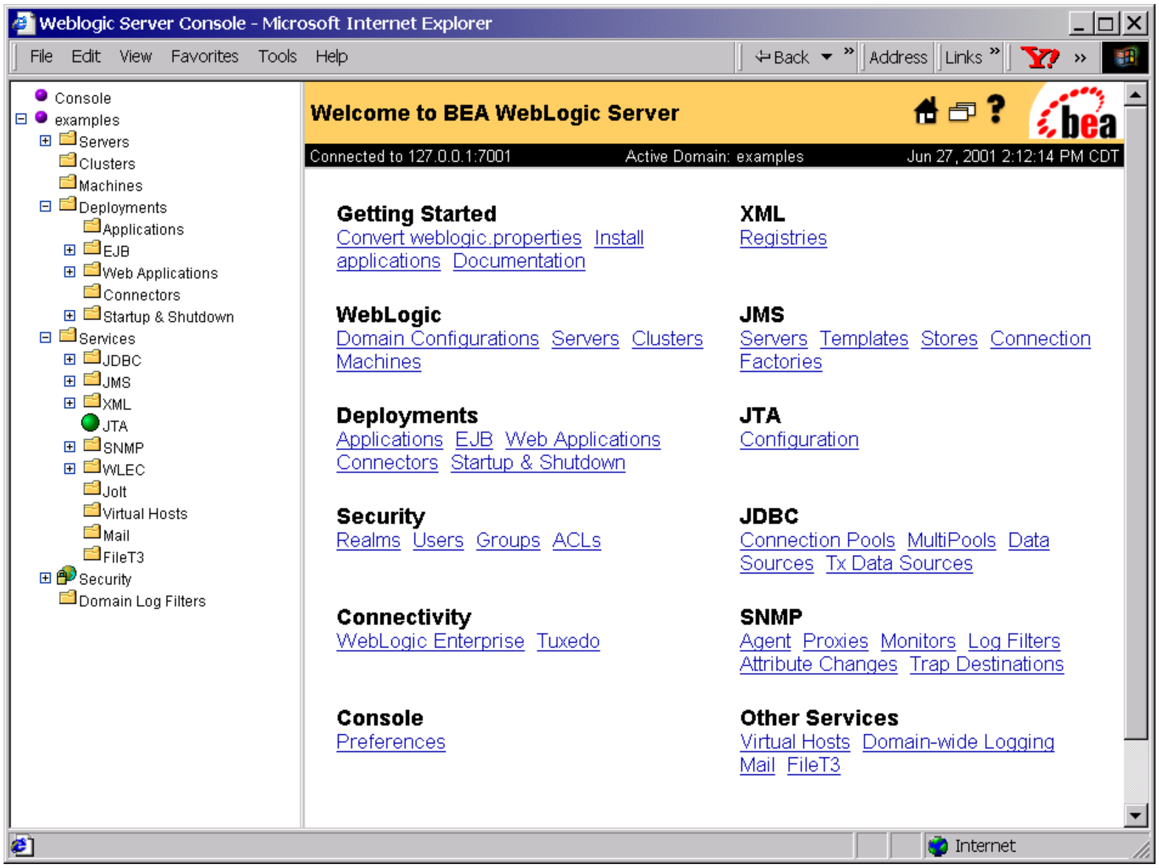
Figure 6: Entering your username and password



Site: 127.0.0.1
Realm weblogic
User Name system
Password ********
☑ Save this password in your password list

Enter the system username and password and click **OK** to continue. You should see the main home page for the *examples* domain (Figure 7):

Figure 7: WebLogic's examples domain main home page



Welcome to BEA WebLogic Server

Connected to 127.0.0.1:7001 Active Domain: examples Jun 27, 2001 2:12:14 PM CDT

Getting Started
Convert weblogic.properties Install applications Documentation

XML
Registries

WebLogic
Domain Configurations Servers Clusters Machines

JMS
Servers Templates Stores Connection Factories

Deployments
Applications EJB Web Applications Connectors Startup & Shutdown

JTA
Configuration

Security
Realms Users Groups ACLs

JDBC
Connection Pools MultiPools Data Sources Tx Data Sources

Connectivity
WebLogic Enterprise Tuxedo

SNMP
Agent Proxies Monitors Log Filters Attribute Changes Trap Destinations

Console
Preferences

Other Services
Virtual Hosts Domain-wide Logging Mail FileT3

- Tip: If you get lost in the console at any point, clicking on the little black house icon on the top of the right pane should bring you back to this page. It can take 5-10 seconds to refresh the navigation pane when you do this, so be patient.

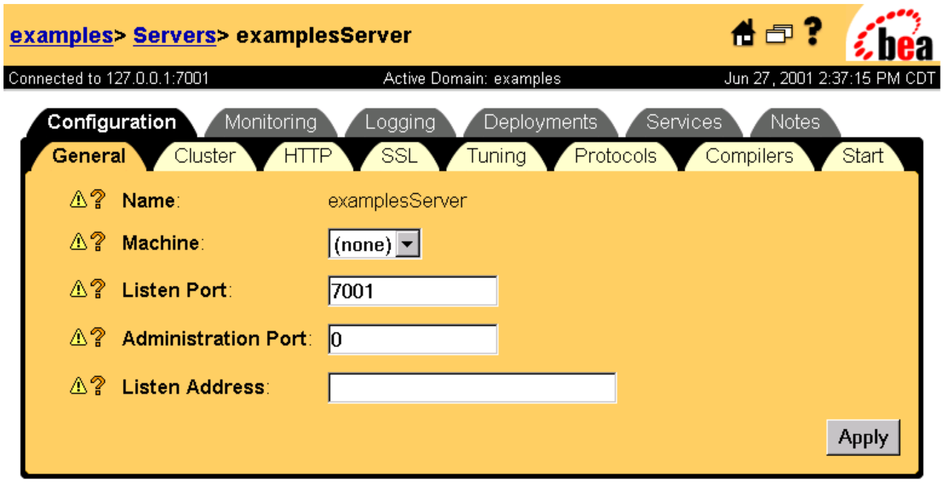
A detailed walkthrough of everything available in the management console is beyond the scope of this workbook. Our discussion will focus on the specific features necessary to configure, deploy, and execute the examples from the EJB book (remember: "the EJB book" is our shorthand for O'Reilly's *Enterprise JavaBeans, 3rd Edition*).

Changing the Logging Severity Threshold

The first step is to change the logging verbosity to include all messages of level *Info* or higher, rather than just *Error* or higher. Click on the **Servers** folder to open it and click on the **examplesServer** item within it to view the configuration details for the *examplesServer*.

The right pane on the screen should look something like Figure 8:

Figure 8: Reviewing the server configuration


examples> Servers> examplesServer

Connected to 127.0.0.1:7001 Active Domain: examples Jun 27, 2001 2:37:15 PM CDT

Name: examplesServer

Machine: (none)

Listen Port: 7001

Administration Port: 0

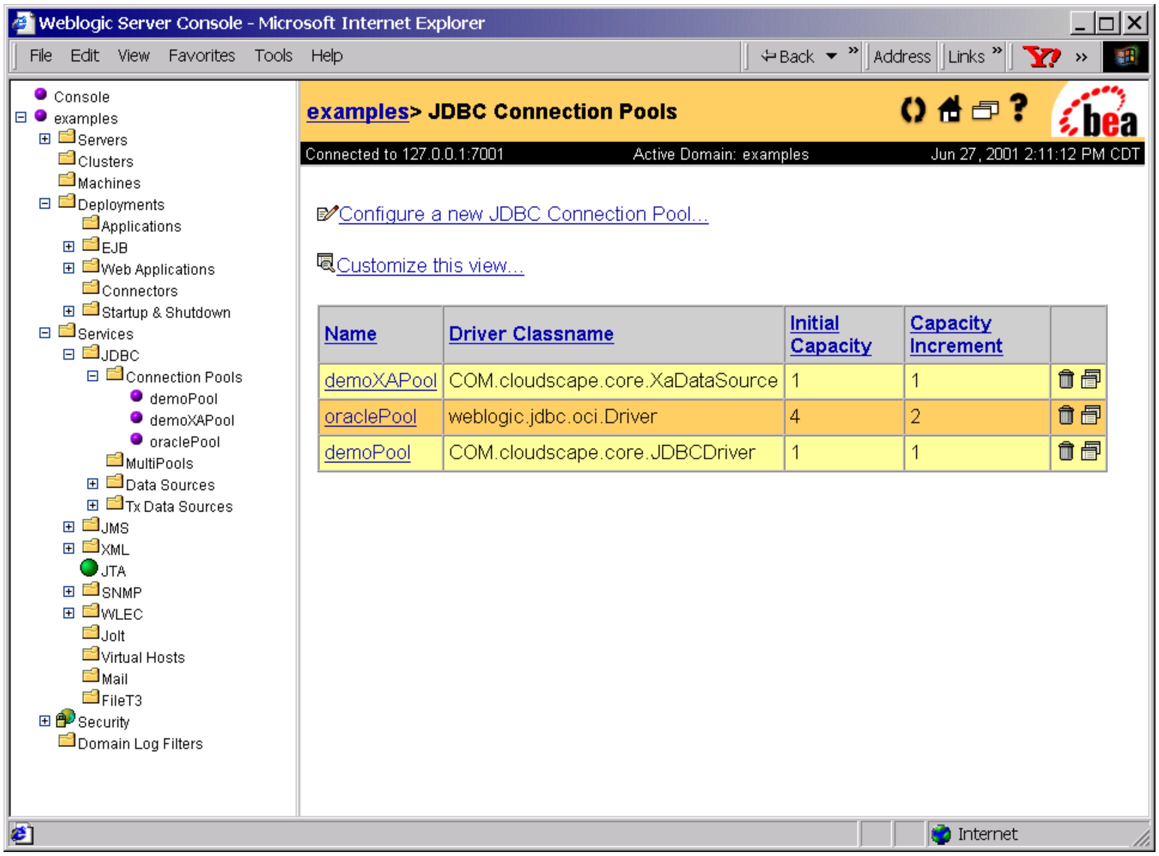
Listen Address:

Click on the **Logging** tab to view details on the logging service for the *examplesServer*. Change the severity threshold to *Info* and apply the changes. This logging change will be saved in the configuration file (*config.xml*) and will cause informational messages to be displayed to the log during the boot process, during domain configuration changes, and as a result of bean deployments.

Configuring the JDBC Connection Pools

The next step is to configure the JDBC connection pools in the *examples* domain to use whichever JDBC-compliant driver and database you have available. Click on the **JDBC** folder in the **Services** section of the navigation pane on the left. The folder will open to reveal additional subfolders. Click on the **Connection Pools** subfolder to see the pools that were pre-configured during the installation process:

Figure 9: Configuring JDBC connection pools



examples> JDBC Connection Pools

Connected to 127.0.0.1:7001 Active Domain: examples Jun 27, 2001 2:11:12 PM CDT

Configure a new JDBC Connection Pool...

Customize this view...

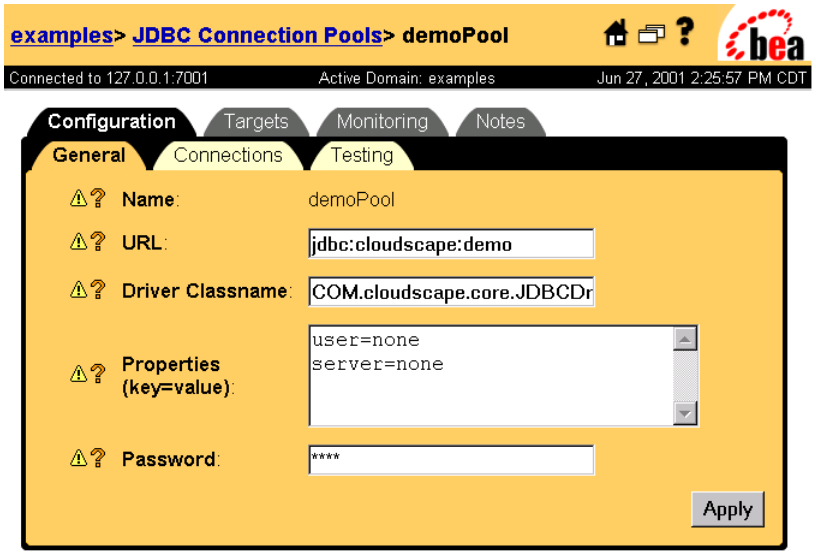
Name	Driver Classname	Initial Capacity	Capacity Increment
demoXAPool	COM.cloudscape.core.XaDataSource	1	1
oraclePool	weblogic.jdbc.oci.Driver	4	2
demoPool	COM.cloudscape.core.JDBCDriver	1	1

The installation process automatically configures the JDBC pools in the *examples* domain to use the Cloudscape database and JDBC drivers. An evaluation version of the Cloudscape database management system is included in your WebLogic distribution and can be used for the WebLogic examples with no additional effort. If you choose to use the Cloudscape database, you may skip the next section and proceed to "Building and Deploying the EJB 2.0 Examples."

Configuring the JDBC pools for alternate JDBC drivers/databases:

1. Click on the **demoPool** in the list on the right side of the screen to display the configuration details for this pool (Figure 10).

Figure 10: Configuring the connection pools for an alternate driver/database


examples> JDBC Connection Pools> demoPool
Connected to 127.0.0.1:7001
Active Domain: examples
Jun 27, 2001 2:25:57 PM CDT
Configuration
Targets
Monitoring
Notes
General
Connections
Testing
Apply

Name: demoPool
URL: jdbc:cloudscape:demo
Driver Classname: COM.cloudscape.core.JDBCDr
Properties (key=value):
```
user=none
server=none
```
Password: ****

2. Enter the URL, Driver Classname, properties, and password for the JDBC driver/database you wish to use. See the list of example entries below or consult the online documentation for details.
3. Apply the changes. The change may not take effect until the next reboot if there are already active connections in the pool using the old parameters. Reboot to be safe.

Here is a list of example entries for common JDBC drivers:

WebLogic's Oracle OCI Driver

```
URL:           jdbc:weblogic:oracle
Driver:        weblogic.jdbc.oci.Driver
Properties:    user=<username>
               server=<servername>
Password:      <password for username>
```

DB/2 UDB JDBC Driver

```
URL:           jdbc:db2:dbname   (where dbname is database name)
Driver:        COM.ibm.db2.jdbc.app.DB2Driver
Properties:    user=<username>
Password:      <password for username>

Note: The db2java.zip file must be included in the server classpath,
requiring a minor change to the startExamplesServer script.
```

Sun's JDBC-ODBC Bridge Driver

```
URL:            jdbc:odbc:dsname   (where dsname is ODBC source name)
Driver:         sun.jdbc.odbc.JdbcOdbcDriver
Properties:     dummy=xxx          (not used but can't be blank)
Password:       anything           (not used but can't be blank)
```

WebLogic's SQL*Server Driver

```
URL:            jdbc:weblogic:mssqlserver4:master@localhost:1433
Driver:         weblogic.jdbc.mssqlserver4.Driver
Properties:     sql7=true
                user=sa
Password:       <password for sa>
```

The WebLogic EJB 2.0 examples require a number of tables in the database to deploy and operate properly. Because you will test just a single example program, *ejb20/basic/containerManaged*, you need only create a single table in your chosen database technology:

```
CREATE TABLE ejbAccounts
(
  id varchar(15),
  bal float,
  type varchar(15)
);
```

Create this table in your database and proceed to the next section to build and deploy the EJB20 examples.

Building and Deploying the EJB20 Examples

The *bea/wlserver6.1/samples/examples/ejb20* directory contains a number of example EJBs we can use to test the functionality of the EJB 2.0 container and JDBC connections. In the first GA version of WebLogic 6.1, at least, these EJB 2.0 examples were not pre-built and deployed to the server by the installation process, probably because the plain-vanilla installation does not support EJB 2.0 without the magic *ejb20.jar* file in the *wlserver6.1/lib* directory.

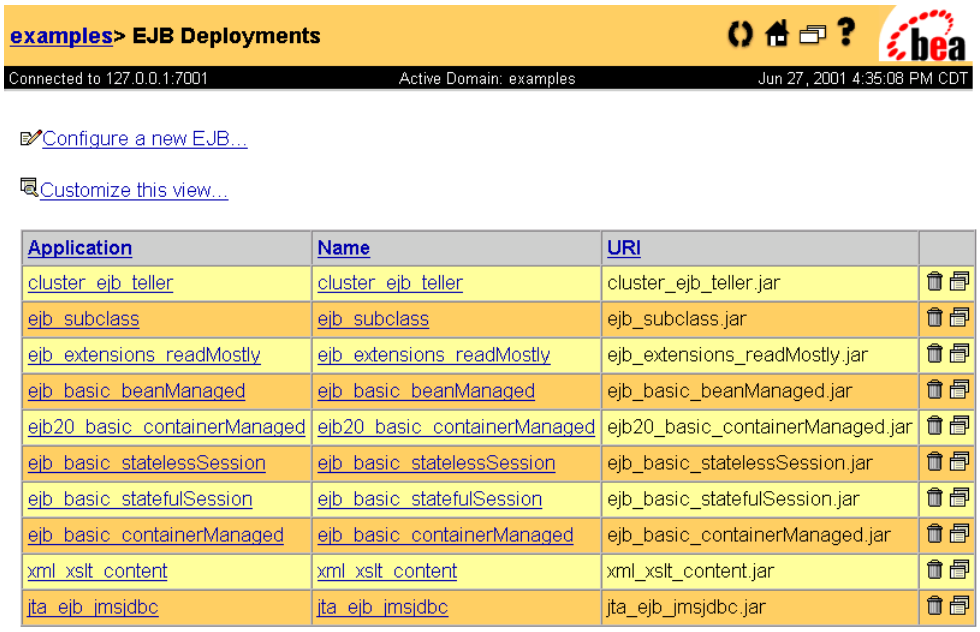
Building the EJB 2.0 examples:

1. Open a command prompt (or new telnet window if you're using SunOS)
2. Change to the */wlserver6.1/config/examples* directory
3. Execute the *setExamplesEnv* script to set PATH and CLASSPATH variables
   - SunOS users: Be sure to "source" the script using a command like:
     ```
     ../setExamplesEnv.sh
     ```
4. Change to the */wlserver6.1/samples/examples/ejb20/basic/containerManaged* directory. In this directory you will see a *build.xml* file used by the "ant" build process for the examples.

5. Type `ant all` to compile and deploy all components in this example. You can also do one step at a time by specifying *ant* targets individually (see the *all* target in the *build.xml* file to see the list of individual targets).
6. If the process succeeds, a new *ejb20_basic_containerManaged.jar* file will be placed in the magic *applications* directory in the *examples* domain.
7. Assuming auto-deployment is working, the new EJB should be sensed by the running copy of WebLogic Server and deployed automatically.
8. Verify that the new *ejb20_basic_containerManaged.jar* was placed in the *config/examples/applications* directory.

Next, use the management console to verify that the EJB was properly deployed by the server automatically. Click on the **EJB** folder in the pane on the left side of the console a few times to force it to refresh. You should see a list similar to the following:

Figure 11:Verifying ejb20 example bean deployment


examples> EJB Deployments

Connected to 127.0.0.1:7001 Active Domain: examples Jun 27, 2001 4:35:08 PM CDT

Configure a new EJB...

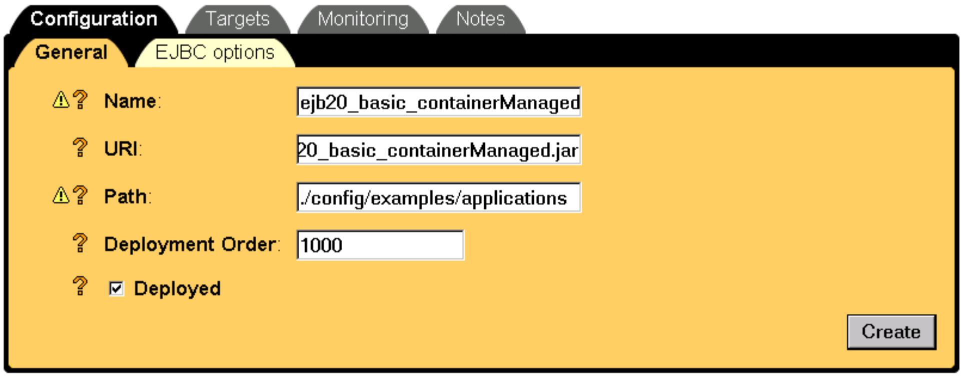
Customize this view...

Application	Name	URI	
cluster_ejb_teller	cluster_ejb_teller	cluster_ejb_teller.jar	
ejb_subclass	ejb_subclass	ejb_subclass.jar	
ejb_extensions_readMostly	ejb_extensions_readMostly	ejb_extensions_readMostly.jar	
ejb_basic_beanManaged	ejb_basic_beanManaged	ejb_basic_beanManaged.jar	
ejb20_basic_containerManaged	ejb20_basic_containerManaged	ejb20_basic_containerManaged.jar	
ejb_basic_statelessSession	ejb_basic_statelessSession	ejb_basic_statelessSession.jar	
ejb_basic_statefulSession	ejb_basic_statefulSession	ejb_basic_statefulSession.jar	
ejb_basic_containerManaged	ejb_basic_containerManaged	ejb_basic_containerManaged.jar	
xml_xslt_content	xml_xslt_content	xml_xslt_content.jar	
jta_ejb_jmsjdbc	jta_ejb_jmsjdbc	jta_ejb_jmsjdbc.jar	

The ejb_xxxxx beans are the basic EJB 1.1 examples that are pre-built and pre-deployed in the *examples* domain. The new EJB 2.0 bean is the fifth item in the list above, the row starting with "ejb20_basic_containerManaged," and may or may not be in your list depending on the somewhat capricious auto-deployment feature.

If the new ejb20_basic_containerManaged bean does not show up on your list, click on the **Configure a new EJB..** link and fill out the form shown in Figure 12 with the data listed below.

Figure 12: Configuring a new bean



Name:	ejb20_basic_containerManaged
URI:	20_basic_containerManaged.jar
Path:	./config/examples/applications
Deployment Order:	1000
☑ Deployed	

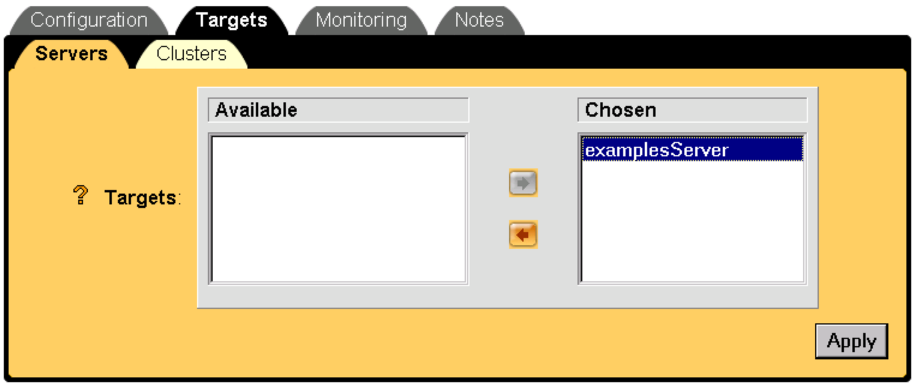
Data for the new EJB:

```
Name:           ejb20_basic_containerManaged
URI:            ejb20_basic_containerManaged.jar
Path (NT):      c:/bea/wlserver6.1/config/examples/applications
Path (Sun):     /opt/bea/wlserver6.1/config/examples/applications
```

Click on **Create** and the new EJB *.jar* file will be read in and deployed. You may need to click on the **EJB** folder in the left navigation pane a few times to force it to show up properly in your console display.

One final step to ensure the new EJB has deployed properly is to check that the component is *targeted* for the examplesServer. Click on the **ejb20_basic_containerManaged** component under the EJB folder in the left pane. You should see the detailed screen for the component. Click on the **Targets** tab and you should see the following screen:

Figure 13: Verifying that the bean is properly "targeted"



If the examplesServer is in the **Available** list instead of the **Chosen** list, the EJB has not been targeted for the server and has not been properly deployed on the server. Pick **examplesServer** in the left list, click on the right-pointing arrow to move it to the **Chosen** side, and click on **Apply** to save the changes. The EJB should immediately deploy, and you should see messages in the log indicating that it has deployed and has registered in JNDI with the name *ejb20-containerManaged-AccountHome*:

```
<Info> <Management> <Configuration changes for domain saved to the
repository.>
<Info> <EJB> <EJB Deploying file: ejb20_basic_containerManaged.jar>
<Info> <JDBC> <Connection for pool "demoPool" created.>
<Info> <EJB> <EJB Deployed EJB with JNDI name ejb20-
containerManaged-AccountHome.>
<Info> <J2EE> <Deployed : ejb20_basic_containerManaged>
```

Testing the EJB20 Examples

Returning to the Command Prompt (or telnet) window you used to build the EJB 2.0 bean, you may now run the simple client application provided with the example by typing:

```
java examples.ejb20.basic.containerManaged.Client
```

Note: The application will run only if you earlier set the `CLASSPATH` properly using the *setExamplesEnv* script, because the *Client.class* file is actually located in the *config/examples/clientclasses* directory structure.

You should see the following output from the client application:

```
Beginning containerManaged.Client...
Starting Part A of the example...
Creating account 10020 with a balance of 3000.0 account type
Savings...
Account 10020 successfully created
Part A: Depositing $2000
Current balance is $5000.0
Attempting to withdraw an amount greater than current balance.
Expecting an exception...
Received expected Processing Error:
examples.ejb20.basic.containerManaged.ProcessingErrorException:
Request to withdraw $5001.0; is more than balance $5000.0 in account
10020
Removing account...
End Part A of the example...

<snip>

Querying for accounts with a null account type
Account ID: 0; account type is null
Account ID: 5; account type is null
Account ID: 10; account type is null
Account ID: 15; account type is null
Removing beans...
End Part B of the example...
End containerManaged.Client...
```

The following error message indicates that the EJB is not properly deployed or has not registered its home interface in the JNDI tree:

```
The client was unable to lookup the EJBHome.
Please make sure that you have deployed the ejb with the JNDI name
ejb20-containerManagedAccountHome on the WebLogic server...
```

Ensure that the EJB component is targeted to the examplesServer. Rebooting the server may help.

An error message similar to the following message indicates that you have not created the required `ejbAccounts` table in the database used by the `demoPool` JDBC pool:

```
Error creating account: EJB Exception:; nested exception is:
java.sql.SQLException: Invalid object name 'ejbAccounts'. Severity
16, State 1, Procedure 'localhost null', Line 5
```

Once you have the ejb20_basic_containerManaged example working, feel free to build and test some of the other EJB 2.0 examples that are provided by WebLogic. In each case, move to the proper */wlserver6.1/samples/ejb20/basic/xxxx* directory and use the `ant all` command to build and deploy the example, performing whatever management-console tasks are required to properly deploy the bean. You may need to create additional tables to support these examples – see the documentation packaged with the example to identify database requirements.

There is an additional EJB 2.0 example ("bands") provided in the normal installation. This example, located in the */examples/ejb20/bands* directory, illustrates the use of CMP 2.0 relationships, sequences, cascade delete, and other EJB 2.0 features. You may build and deploy this example as well, but recognize that we will be examining many of these same features step by step in this workbook by walking through the examples and exercises in the O'Reilly EJB book.

Congratulations! You are now ready to create the JDBC database and *ejbbook* domain we will be using for the workbook examples.

Building the Workbook Database

There are three ways to create the database tables required to support the workbook exercises:

1. Create an empty database in some JDBC-compliant database technology available to you, then create each table when a specific exercise requires you to, using the details spelled out in the workbook text.
2. Create an empty database in a JDBC-compliant database technology for which the workbook download site provides a complete database-creation script. Create the database yourself and run the script to create all of the tables before starting work on the first exercise.
3. Use the evaluation version of the Cloudscape database that ships with WebLogic and download a pre-configured database containing all of the necessary tables and data.

We recommend Option #1 . Building the database one step at a time as it is needed to support the exercises, will help you learn about the specific database requirements for the EJBs and relationships introduced in each exercise.

Use Option #2 if you want to use your own JDBC database rather than Cloudscape, and you wish to skip as much database configuration work as possible.

Option #3 is the fastest approach: it avoids all database-creation and configuration tasks.

- Avoid using an Oracle database for the workbook examples. Oracle-specific changes must be made in two of the tables you will be creating in Exercise 7.1 in order to avoid the reserved word `NUMBER`. These changes complicate your task of mapping bean attributes to database columns in subsequent exercises, requiring you to edit the WebLogic CMP descriptor file in each exercise to reflect the changes. If you choose to use Oracle, details of the required changes are provided in Exercise 7.1, and the downloadable *oracle.sql* script creates tables with legal column names.

Option #1 – Build Empty Database, Add Tables During Exercises

If you are familiar with a specific database technology and have it available for use with these exercises, create a new database with a small amount of space (5-10 MB should suffice). Record the database name, user name, and password, as these will be required in the configuration of the JDBC connection pool in WebLogic.

That's it! You should be ready to proceed to the *ejbbook* domain-configuration step.

If you do not have access to an alternate JDBC-compliant database, the workbook exercises will operate properly using the Cloudscape evaluation database supplied with the WebLogic 6.1 installation.

- Warning: The Cloudscape product will operate for only 30 days from the WebLogic installation date even if you have a valid long-term WebLogic Server license. Cloudscape is a separately-licensed product owned by Informix.

There are at two ways to create and manage Cloudscape databases: the *Cloudview* graphical tool and the *ij* command-line tool. The following sections detail the steps necessary to create an empty Cloudscape database using each of these tools.

Creating Cloudscape Database using Cloudview

Cloudscape ships with a client application called *Cloudview*, which is useful for creating and managing databases.

Cloudview is a Java application that requires two additional archive (*.jar*) files in the classpath:

<WL_HOME>\samples\eval\cloudscape\lib\tools.jar

<WL_HOME>\samples\eval\cloudscape\lib\cloudscape.jar

The `WL_HOME` environment variable depends on your installation and platform, but should be *\bea\wlserver6.1* (NT/Win2K) or */opt/bea/wlserver6.1* (SunOS) if you followed the directions during the product installation step.

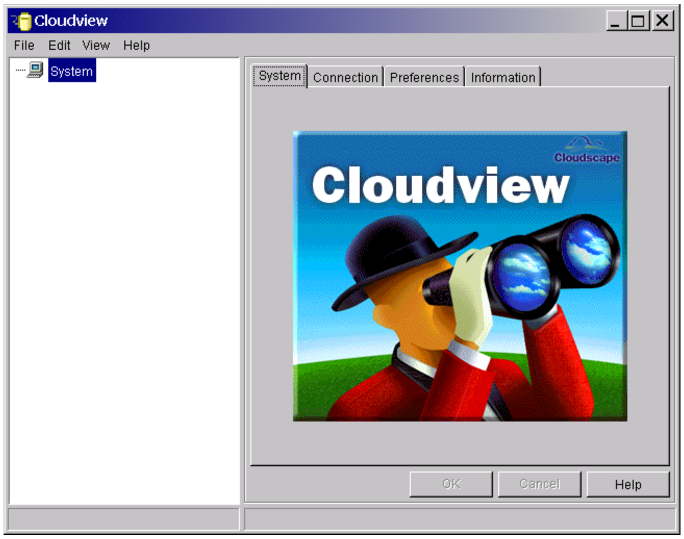
Once these archives are added to your classpath, run Cloudview using:

```
java COM.cloudscape.tools.cview
```

The download site provides a simple script called *cvstart.cmd* (or *cvstart.sh*) in the *dbscripts.jar* archive that you can download and use to start Cloudview. Edit the script if necessary to match your installation.

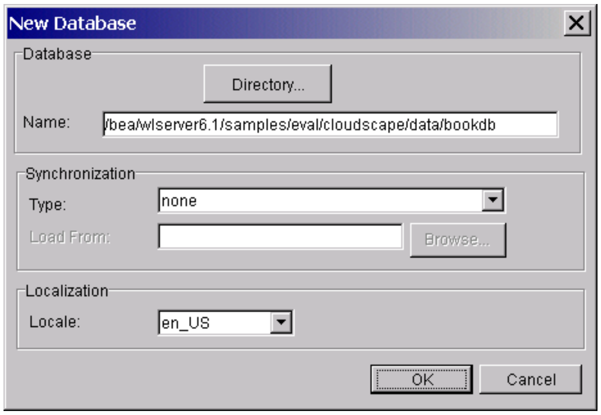
The Cloudview application should start and display the following interface:

Figure 14: Cloudview splash screen



Open the **File** menu and choose **New** and **Database**. Complete the resulting form:.

Figure 15: Creating a new Cloudscape database



Name: /bea/wlserver6.1/samples/eval/cloudscape/data/bookdb

Type: none

Load From:

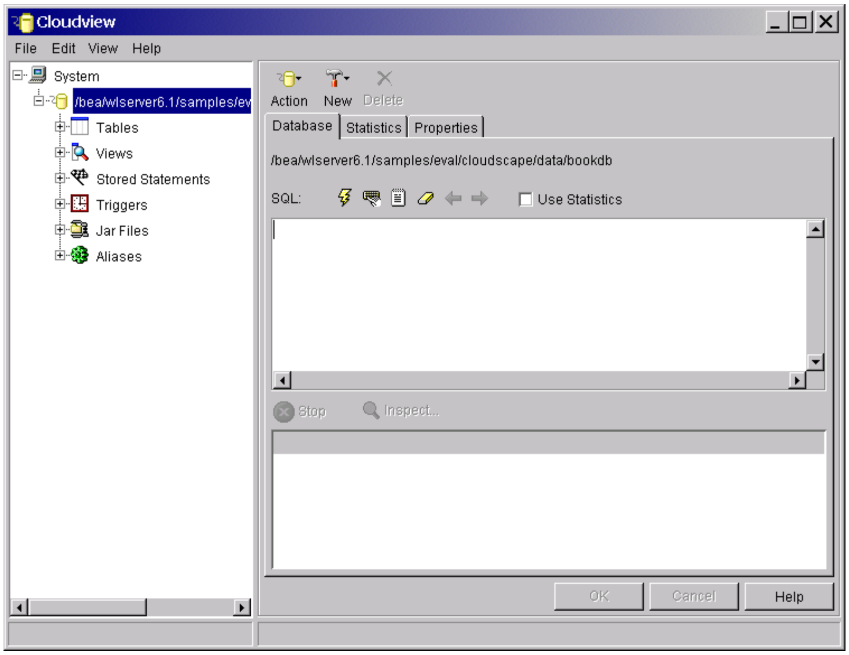
Locale: en_US

The full path to the database will depend on your platform, but should be something like:

/bea/wlserver6.1/samples/eval/cloudscape/data/bookdb

Click on the **OK** button. Cloudview will create the new database and display the main control screen for managing the *bookdb* database:

Figure 16: Main screen for new Cloudscape database



That's all there is to it. Cloudscape has created a new JDBC-compliant database which can be accessed from within WebLogic using the following properties:

```
URL:           jdbc:cloudscape:bookdb
Driver:        COM.cloudscape.core.JDBCDriver
Properties:    user=none
               server=none
Password:      <anything>
```

You will configure the JDBC connection pool in your WebLogic domain to use this database in a future step.

From this point forward, when the exercise text tells you to create a table, start Cloudview and type the correct create-table statement in the **SQL** window. You may also find the *cloudscape.sql* script (available from the workbook download site in the *dbscripts.jar* archive) useful for this purpose. It contains the create-table statements for the entire database, allowing you to copy and paste statements for individual tables to the **SQL** window without keying them in. You may also use the graphical tools within Cloudview to build tables, but this is not recommended.

- Warning: The evaluation license for Cloudscape prohibits multiple processes from opening the database simultaneously. To run the Cloudview program you must first shut down WebLogic, to disconnect it from the database.

If you've chosen to create the required database tables during the exercises (Option #1), you may now close the Cloudview program and proceed to the section "Configuring EJBBook Domain for EJB Exercises."

See the documentation available at *http://www.cloudscape.com/* for detailed instructions and help in using the Cloudview application.

Creating Cloudscape Database using ij Utility

Cloudscape ships with a command-line utility, *ij,* which is useful for creating and managing databases. *ij* is a Java application that requires two additional archive (*.jar*) files in the classpath:

<WL_HOME>\samples\eval\cloudscape\lib\tools.jar

<WL_HOME>\samples\eval\cloudscape\lib\cloudscape.jar

The `WL_HOME` environment variable depends on your installation and platform, but should be *\bea\wlserver6.1* (NT/Win2K) or */opt/bea/wlserver6.1* (SunOS) if you followed the directions during the product installation step.

Once these archives are added to your classpath, run *ij* using:

```
java -Dcloudscape.system.home=<WL_HOME>/samples/eval/cloudscape/data
COM.cloudscape.tools.ij
```

The entire *java* command should be on a single line. The `WL_HOME` variable again depends on your installation.

The download site provides a simple script called *ijstart.cmd* (or *ijstart.sh*) in the *dbscripts.jar* archive that you can download and use to start *ij*. Edit the script to match your installation if necessary.

When *ij* starts you should see a brief message and then a prompt:

```
ij version 3.5 (c) 1997-2000 Informix Software, Inc.
ij>
```

Type `help;` to print a list of commands. Note that all commands must end with a semi-colon.

There is no explicit `create database` command in *ij*; instead, the `connect` command is used with an optional `create` clause. Type the following command at the `ij>` prompt to create the *bookdb* database:

```
ij> connect 'jdbc:cloudscape:bookdb;create=true';
ij>
```

The *bookdb* database is now created and available for use. Use `show connections` to list the current database connections and verify that the *bookdb* connection is present:

```
ij> show connections;
CONNECTION0* -  jdbc:cloudscape:bookdb;create=true
* = current connection
ij>
```

Verify that the new database has been created in the correct location on the disk. There should be a new *bookdb* directory in the *<WL_HOME>/samples/eval/cloudscape/data* directory. If the *bookdb* directory is not present, make sure you included the proper *cloudscape.system.home* definition on the *java* command line for *ij*.

Useful commands in *ij* include:

- `Connect 'URL' – Connects to the specified Cloudscape database`
- `Disconnect` – Disconnects from the current database
- `Run 'filename'` – Runs the commands in the specified file
- `Exit` – Exits the *ij* utility, closing connections

Normal SQL DDL commands like `create table`, `drop table`, `insert`, `update`, and `delete` are recognized by *ij*. Don't forget: all commands must end with a semi-colon.

That's all there is to it. Cloudscape has created a new JDBC-compliant database that can be accessed from within WebLogic using the following properties:

```
URL:            jdbc:cloudscape:bookdb
Driver:         COM.cloudscape.core.JDBCDriver
Properties:     user=none
                server=none
Password:       <anything>
```

You will configure the JDBC connection pool in your WebLogic domain to use this database in a future step.

From this point forward, when the exercise text tells you to create a table, start *ij* and connect to the *bookdb* database using:

```
ij> connect 'jdbc:cloudscape:bookdb';
```

After connecting, simply type the correct SQL create-table statement. For example, the `CABIN` table will be required for Exercise 4.1; you can create it with the following statement:

```
ij> create table CABIN
(
  ID int primary key,
  SHIP_ID int,
  BED_COUNT int,
  NAME char(30),
  DECK_LEVEL int
)
;
0 rows inserted/updated/deleted
ij>
```

You may also find the *cloudscape.sql* script (available from the workbook download site in the *dbscripts.jar* archive) useful for this purpose. It contains the create-table statements for the entire database, allowing you to copy and paste statements for individual tables without keying them in.

> Warning: The evaluation license for Cloudscape prohibits multiple processes from opening the database simultaneously. To run the ij utility and connect to the bookdb database you must first shut down WebLogic, to disconnect it from the database.

If you've chosen to create the required database tables while doing the exercises (Option #1), you may now close the *ij* utility (using the `exit` command) and proceed to the section "Configuring EJBBook Domain for EJB Exercises."

See the documentation available at *http://www.cloudscape.com/* for detailed instructions and help in using the *ij* utility.

Option #2 – Build Empty Database, Create All Tables at Start

If you are familiar with a specific database technology and have it available for use with these exercises, create a new database with a small amount of space (5-10 MB should suffice). Record the database name, user name, and password, as these will be required in the configuration of the JDBC connection pool in WebLogic.

If you do not have access to an alternate JDBC-compliant database, the workbook exercises will operate properly using the Cloudscape evaluation database supplied with the WebLogic 6.1 installation. Follow the instructions in the Option #1 discussion above to create an empty Cloudscape database and return here when that step is complete.

Table-creation scripts are available in the workbook download area of the site for the following database technologies:

- *mssqlserver.sql* – Microsoft SQL Server 7 or higher
- *oracle.sql* – Oracle 8.1.x or higher
- *db2udb.sql* – IBM DB/2 UDB 7.1 or higher
- *cloudscape.sql* – Cloudscape evaluation database supplied with WebLogic 6.1

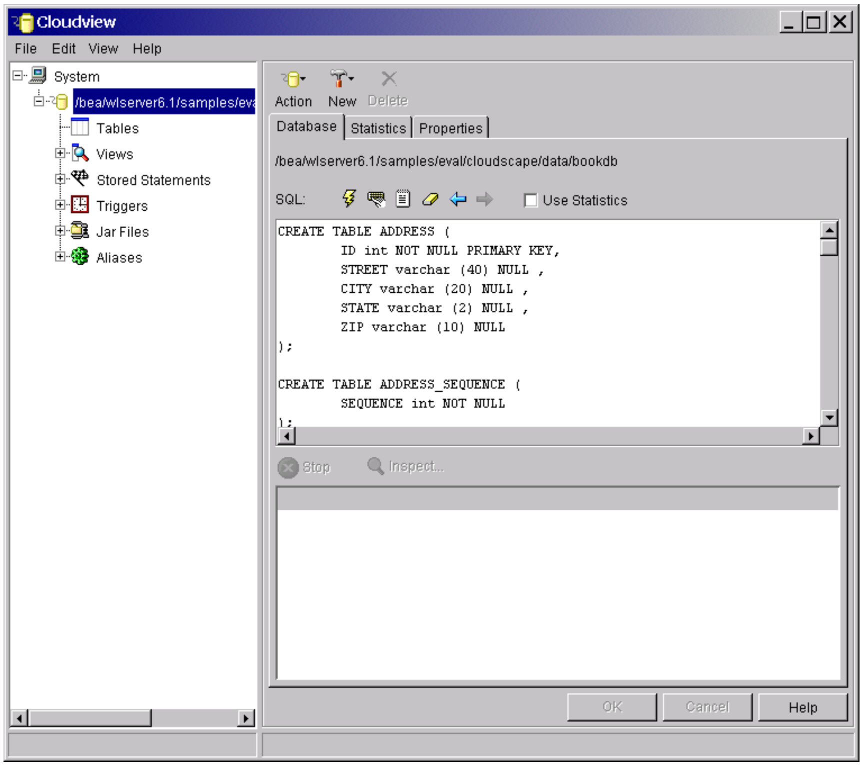
After creating an empty database using your technology of choice, execute the commands in one of these scripts, using the graphical tool or command-line tool supplied with the database technology. Verify that all of the tables were created properly and that the five sequence tables have a valid row before proceeding to the next step.

If you have chosen to use Cloudscape and create the tables yourself, you may use either the Cloudview application or the *ij* command-line utility.

Creating Workbook Tables using Cloudview

Start Cloudview (using *cvstart.cmd* or *.sh*) and use the **File** menu to open the *bookdb* database created in the earlier step. The SQL window on the right side should be empty. Use *Notepad* or some other editing tool to copy the contents of the *cloudscape.sql* script and paste them into this window (Figure 17), or click on the **script icon** and load the *cloudscape.sql* script from disk.

Figure 17: Creating tables using Cloudview SQL window



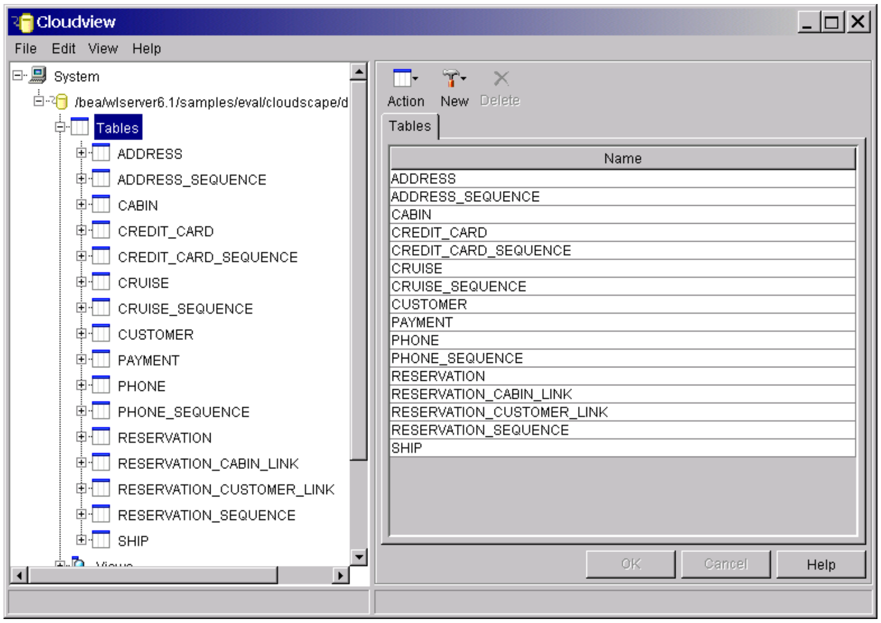
```
CREATE TABLE ADDRESS (
        ID int NOT NULL PRIMARY KEY,
        STREET varchar (40) NULL ,
        CITY varchar (20) NULL ,
        STATE varchar (2) NULL ,
        ZIP varchar (10) NULL
);

CREATE TABLE ADDRESS_SEQUENCE (
        SEQUENCE int NOT NULL
);
```

Click on the **lightning icon** to execute the script. Cloudview will create all tables, and pre-populate each of the sequence tables used for automatic key generation with a valid row of data.

Open the **Tables** folder in the navigation pane and compare the display with Figure 18 to ensure all tables were created successfully.

Figure 18: Verifying all tables were created in database



ADDRESS
ADDRESS_SEQUENCE
CABIN
CREDIT_CARD
CREDIT_CARD_SEQUENCE
CRUISE
CRUISE_SEQUENCE
CUSTOMER
PAYMENT
PHONE
PHONE_SEQUENCE
RESERVATION
RESERVATION_CABIN_LINK
RESERVATION_CUSTOMER_LINK
RESERVATION_SEQUENCE
SHIP

Name
ADDRESS
ADDRESS_SEQUENCE
CABIN
CREDIT_CARD
CREDIT_CARD_SEQUENCE
CRUISE
CRUISE_SEQUENCE
CUSTOMER
PAYMENT
PHONE
PHONE_SEQUENCE
RESERVATION
RESERVATION_CABIN_LINK
RESERVATION_CUSTOMER_LINK
RESERVATION_SEQUENCE
SHIP

The *bookdb* database is now prepared for use. Close Cloudview and proceed to the next section, “Configuring EJBBook Domain for EJB Exercises.”

Creating Workbook Tables using ij Utility

Start the *ij* utility (using *ijstart.cmd* or *.sh*) and connect to the *bookdb* database:

```
C:\work\ejbbook>ijstart
...
C:\work\ejbbook>java
-Dcloudscape.system.home=\bea\wlserver6.1/samples/eval/cloudscape/
data COM.cloudscape.tools.ij
ij version 3.5 (c) 1997-2000 Informix Software, Inc.
ij> connect 'jdbc:cloudscape:bookdb';
ij>
```

Next, run the *cloudscape.sql* script:

```
ij> run 'cloudscape.sql';
```

The script will create all tables, and pre-populate each of the sequence tables used for automatic key generation with a valid row of data.

The *bookdb* database is now prepared for use. Close *ij* and proceed to the next section, “Configuring EJBBook Domain for EJB Exercises.”

Option #3 – Download Complete Cloudscape Database

The workbook download site provides a pre-created *bookdb* database for Cloudscape containing all of the tables and data required for the exercises. There is no need to run Cloudview or *ij*, create an empty database, or run any table-creation scripts. Who could ask for more?

Perform the following steps to install this pre-created database:

1. Download the *bookdb.jar* file from the workbook download site.
2. Copy the file to the */bea/wlserver6.1/samples/eval/cloudscape/data* directory, adjusting this path to reflect your installation details.
3. Extract the contents of the file to the *data* directory using `jar xvf bookdb.jar`, creating a *bookdb* subdirectory and a series of additional Cloudscape files and subdirectories under the *bookdb* directory.

That's all there is to it. The *bookdb* Cloudscape database is now prepared for use and is available in WebLogic using the JDBC properities:

```
URL:            jdbc:cloudscape:bookdb
Driver:         COM.cloudscape.core.JDBCDriver
Properties:     user=none
                server=none
Password:       <anything>
```

Configuring EJBBook Domain for EJB Exercises

While the workbook examples could conceivably run in the *examples* domain, or in the empty *mydomain* domain created by the installation process, by creating a new domain you will learn the process and better understand the structure and configuration of a WebLogic domain.

Creating an Empty EJBBook Domain

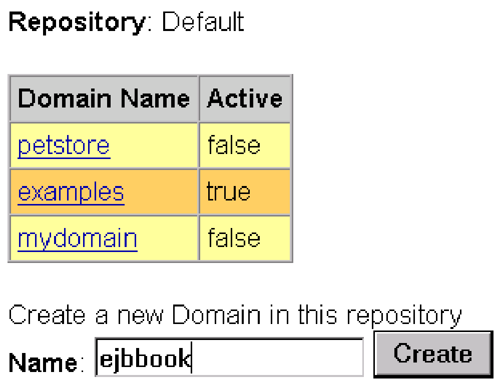
There are two options for creating a new domain:

- Simply copy and rename an entire existing domain directory structure to create the new domain, editing all of the files in the domain root directory to reflect the new domain name.
- Use the management console to create the new domain, copy a few key files from an empty domain like *mydomain* in to the new domain, and create the required services and domain configuration items by hand in the console.

Option 1 is appropriate for more advanced users who will be able to troubleshoot all of the name-related problems. We'll be using Option 2 since it will provide a good opportunity to walk step by step through the creation of the many components in a working domain.

To use the WebLogic console to create the new domain, boot a working domain (like *examples*) and open the console. Go to the main home page if not already there. The right side of the screen should look like Figure 7. Click on the **Domain Configurations** link and you should see a page like Figure 19.

Figure 19: Creating a new domain



Repository: Default

Domain Name	Active
petstore	false
examples	true
mydomain	false

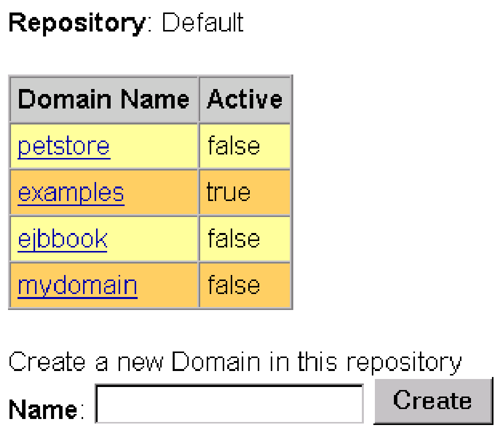
Create a new Domain in this repository

Name: ejbbook

Enter the name of the new domain (*ejbbook*) in the text field and click on **Create**.

WebLogic will create the new domain directory under the *wlserver6.1/config* directory and place a skeleton *config.xml* in that new domain root directory. You should be back at the main home page for the console. Click on **Domain Configurations** again and the resulting list of domain names should now include your new *ejbbook* domain, as in Figure 20:

Figure 20: Verifying the ejbbook domain was created



Repository: Default

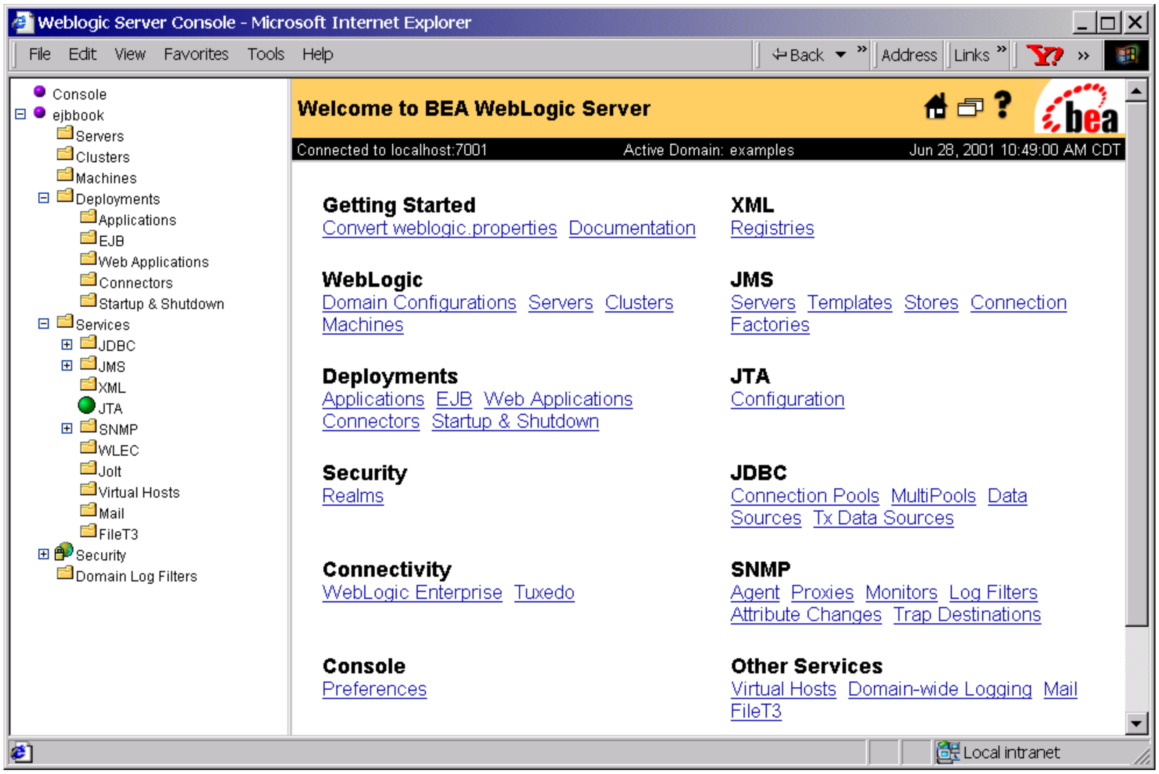
Domain Name	Active
petstore	false
examples	true
ejbbook	false
mydomain	false

Create a new Domain in this repository

Name:

Click on the **ejbbook** link and the console will display the main home page with the new *ejbbook* domain hierarchy displayed in the navigation pane on the left:

Figure 21: Main home page for new ejbbook domain


Weblogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address
Links
Console
ejbbook
Servers
Clusters
Machines
Deployments
Applications
EJB
Web Applications
Connectors
Startup & Shutdown
Services
JDBC
JMS
XML
JTA
SNMP
WLEC
Jolt
Virtual Hosts
Mail
FileT3
Security
Domain Log Filters
Welcome to BEA WebLogic Server
bea
Connected to localhost:7001
Active Domain: examples
Jun 28, 2001 10:49:00 AM CDT
Getting Started
Convert weblogic.properties Documentation
XML
Registries
WebLogic
Domain Configurations Servers Clusters Machines
JMS
Servers Templates Stores Connection Factories
Deployments
Applications EJB Web Applications Connectors Startup & Shutdown
JTA
Configuration
Security
Realms
JDBC
Connection Pools MultiPools Data Sources Tx Data Sources
Connectivity
WebLogic Enterprise Tuxedo
SNMP
Agent Proxies Monitors Log Filters Attribute Changes Trap Destinations
Console
Preferences
Other Services
Virtual Hosts Domain-wide Logging Mail FileT3
Local intranet

Click on a few of the folders on the left side and you will notice that there are no servers, applications, EJBs, web applications, JDBC pools, or other services defined in the domain. Also note that when you are editing a domain different from the “active” or running domain, you cannot modify security information such as users, groups, and ACLs.

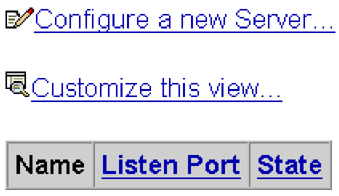
Configuring EJBBook Domain

You can accomplish the first few configuration steps using the management console. Note that you are still running the *examples* domain and are essentially using that domain to edit the *config.xml* file in the new *ejbbook* domain. You will be unable to boot the new domain until the basic configuration steps are complete and you’ve copied some scripts to the new domain root directory.

Step 1 – Create and Configure a Server in the ejbbook domain

Click on the **Servers** folder in the navigation pane on the left. The page on the right should display an empty list of servers:

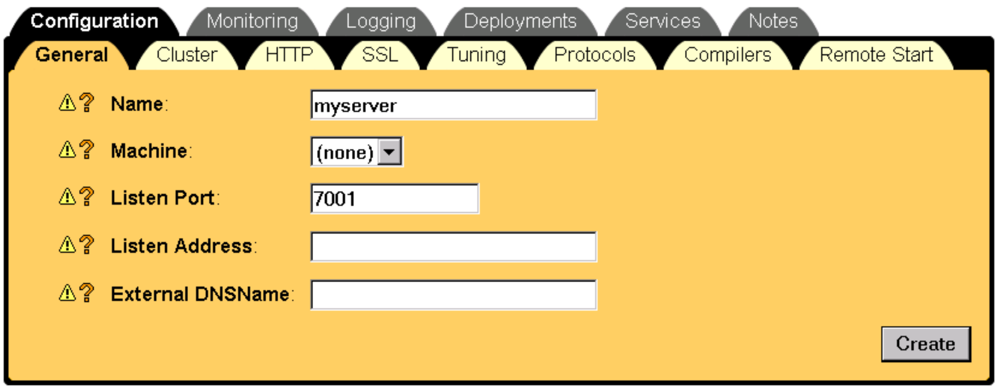
Figure 22: Starting to create a new server



Name	Listen Port	State

Click on the **Configure a new Server...** link and fill out the form as shown below:

Figure 23: Configuring the new server



Name:	myserver
Machine:	(none)
Listen Port:	7001
Listen Address:	
External DNSName:	

Note that *myserver* is all lower case. Click on **Create** to continue.

Now click on the **Logging** tab and change the Severity Threshold Level to *Info* and apply changes.

This is about as far as you can safely go without booting the *ejbbook* domain, so shut down the examples server (you can do this through the management console, or by simply stopping the process in the command/telnet window with control-C).

Step 2 – Configure ejbbook domain root directory

Booting the new *ejbbook* domain requires the following files in the */config/ejbbook* directory:

- *startWebLogic.cmd* or *startWebLogic.sh*

- *fileRealm.properties* and *SerializedSystemIni.dat*
- *ca.pem*, *democert.pem*, *demokey.pem*, and 1024-bit versions of these files if present

There should also be an *applications* directory and a *logs* directory.

Win2K/NT

In a command-prompt window, perform the following steps to copy and configure these required files and directories:

1. Change to *\bea\wlserver6.1\config\ejbbook* directory.
2. Copy **.pem* files from the *mydomain* domain root directory.
   ```
   copy ..\mydomain\*.pem .
   ```
3. Copy security/realm files from *mydomain*. Overwrite any existing copy in *ejbbook*.
   ```
   copy /Y ..\mydomain\fileRealm.properties .
   copy /Y ..\mydomain\SerializedSystemIni.dat .
   ```
4. Copy the *startWebLogic.cmd* script from *mydomain*.
   ```
   copy ..\mydomain\startWebLogic.cmd .
   ```
5. Edit *startWebLogic.cmd* and make the following changes:
   - Replace all occurrences of the word *mydomain* with *ejbbook* – be sure to fix the *java* command-line occurrence (`-Dweblogic.Domain=ejbbook)`
   - Modify the line that sets `STARTMODE=true` to be `STARTMODE=false` in order to run WebLogic in "development" mode and enable automatic deployment, a feature discussed in a later section.
6. Create *applications* and *logs* directories and copy the *applications* directory structure from *mydomain*, including all subdirectories and files (*xcopy* is a good tool for this, or use *Explorer* to copy the folder).
   ```
   mkdir logs
   mkdir applications
   xcopy /E ..\mydomain\applications applications
   ```

SunOS

In a telnet window perform the following steps to copy and configure these required files and directories:

```
cd /opt/bea/wlserver6.1/config/ejbbook
cp -p ../mydomain/*.pem .
cp -p ../mydomain/fileRealm.properties .
cp -p ../mydomain/SerializedSystemIni.dat .
cp -p ../mydomain/startWebLogic.sh .
vi startWebLogic.sh
```

Change *mydomain* to *ejbbook* everywhere it appears

Change `STARTMODE=true` to `STARTMODE=false` to enable auto deploy

```
mkdir logs
mkdir applications
cp -r -p ../mydomain/applications/* applications
```

Win2K/NT or SunOS

On either platform, the *startWebLogic* script may need a slight modification to support the JDBC-compliant database you've chosen to use. The version copied from the *mydomain* domain will have a line similar to the following:

```
set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar
```

If you are using the Cloudscape evaluation database provided with WebLogic, add the *cloudscape.jar* file to the CLASSPATH by appending it to this line:

```
set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar;.\samples\e
val\cloudscape\lib\cloudscape.jar
```

Other database technologies will require different Java libraries in the CLASSPATH for the server. Consult your database documentation for the name and location of the required files and append them to the CLASSPATH in a similar manner.

For example, IBM DB/2 UDB database requires the following lines be added in *startWebLogic*:

```
set CLASSPATH=%CLASSPATH%;\sqllib\java\db2java.zip
set PATH=%PATH%;\sqllib\bin
```

Cloudscape also requires an additional variable definition on the *java* command used to run the WebLogic Server. Edit the *startWebLogic* script and add the following definition to the main *java* command near the bottom of the file:

```
-Dcloudscape.system.home=./samples/eval/cloudscape/data
```

For reference, here is a listing of the *startWebLogic.cmd* file for a typical NT/Win2K installation with the changes required for Cloudscape highlighted in the listing:

```
@echo off
...
:runWebLogic
echo on
set PATH=.\bin;%PATH%


set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar;.\samples\e
val\cloudscape\lib\cloudscape.jar

echo off
...
echo on
```

```
"%JAVA_HOME%\bin\java" -hotspot -ms64m -mx64m -classpath %CLASSPATH%
-Dweblogic.Domain=ejbbook
-Dcloudscape.system.home=./samples/eval/cloudscape/data
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Dweblogic.Name=myserver "-Dbea.home=C:\bea"
"-Djava.security.policy==C:\bea\wlserver6.1/lib/weblogic.policy"
-Dweblogic.management.password=weblogic weblogic.Server
goto finish

:finish
cd config\ejbbook
ENDLOCAL
```

➢ Note that, in the file, the *java* command should be all on one line.

That should be it. Cross your fingers and execute the *startWebLogic* script to attempt to boot the new *ejbbook* domain. The final line in the log should be:

```
... <Notice> <WebLogicServer> <Started WebLogic Admin Server
"myserver" for domain "ejbbook" running in Development Mode>
```

Verify that the domain is correct (*ejbbook*) and that the server is running in development mode. If something is wrong, stop the server and edit the *startWebLogic* script to fix the problem.

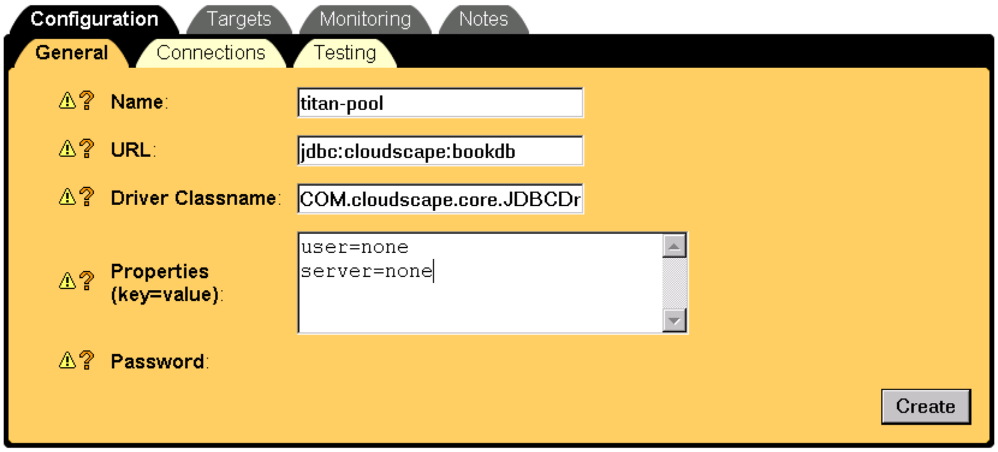
Step 3 – Create a JDBC pool and datasource in ejbbook domain

Open a new management console browser window. You should see the main home page for the *ejbbook* domain.

Click on the **JDBC** folder on the left to open it, then click on the **Connection Pools** subfolder under JDBC to view the current connection pools. The page on the right side should show an empty list of pools.

Click on the **Configure a new JDBC Connection Pool…** link and fill out the form with the appropriate JDBC connection-pool information for your database, as in Figure 24:

Figure 24: Configuring a connection pool for ejbbook



Name: titan-pool
URL: jdbc:cloudscape:bookdb
Driver Classname: COM.cloudscape.core.JDBCDr
Properties (key=value):
```
user=none
server=none
```
Password:

The pool information (URL, Driver Classname, Properties, etc.) should be the same as the information used when you configured the *examples* domain to use your selected JDBC database. If you created a new *bookdb* Cloudscape database for use in the exercises, the pool information would be:

```
URL:           jdbc:cloudscape:bookdb
Driver:        COM.cloudscape.core.JDBCDriver
Properties:    user=none
               server=none
Password:      <anything>
```

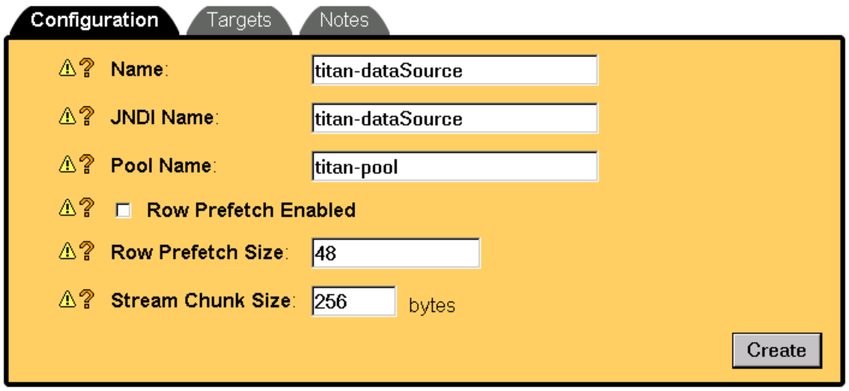
The pool name should be *titan-pool* to be consistent with the downloadable example code and subsequent discussions in this workbook, although the pool name itself is not actually used in the deployment descriptors for the EJBs. Click on **Create** to continue.

Configure the initial and maximum number of connections to be higher than the defaults of one initial and one maximum set automatically by WebLogic. Click on the **Connections** tab and set the **Initial Capacity** to 2 and the **Maximum Capacity** to 10, then click on **Apply** to save the change. Allowing additional connections will be important during Exercise 13.2.

Next, target the JDBC pool to the server we created in the first step. Click on the **Targets** tab and move *myserver* from the **Available** list to the **Chosen** list and apply the change.

The final step is the creation of a *data source* associated with the JDBC pool you just created. Open the **JDBC** folder in the navigation pane and click on the **Data Sources** subfolder. The right page should display an empty list of JDBC Data Sources. Click on the **Configure a new JDBC Data Source…** link and fill out the form as shown in Figure 25.

Figure 25: Associating a data source with the connection pool



Name:	titan-dataSource
JNDI Name:	titan-dataSource
Pool Name:	titan-pool
☐ Row Prefetch Enabled	
Row Prefetch Size:	48
Stream Chunk Size:	256 bytes

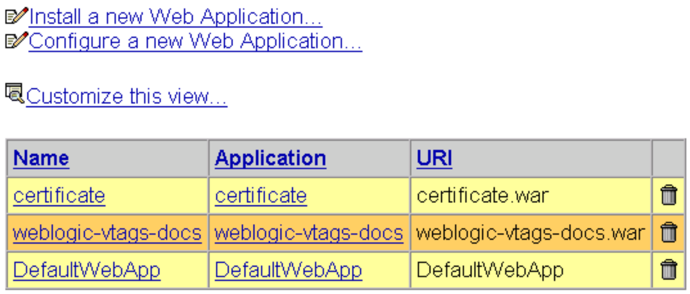
Enter the data source name exactly as shown (*titan-dataSource*) to be consistent with the downloadable example code and descriptors for the exercises. Click on **Create** to create the data source.

Finally, click on the **Targets** tab and move *myserver* from **Available** to **Chosen** to target the *titan-dataSource* data source to the *myserver* server.

Congratulations, you now have a valid, running *ejbbook* domain!

This might be a good time to configure the default web application for the server and perform a simple HTTP test. When you copied the applications directory from mydomain, this included a *DefaultWebApp* directory structure containing a simple web application (*WEB-INF* directory, *web.xml* file, etc). Because the *DefaultWebApp* directory is located in the magic *applications* directory, its presence was detected and the web application was automatically deployed during the boot process. Click on the **Web Applications** folder in the navigation pane on the left side of the console. You should see a list of applications similar to the following:

Figure 26: Finding the default web application


Install a new Web Application...
Configure a new Web Application...
Customize this view...

Name	Application	URI	
certificate	certificate	certificate.war	🗑
weblogic-vtags-docs	weblogic-vtags-docs	weblogic-vtags-docs.war	🗑
DefaultWebApp	DefaultWebApp	DefaultWebApp	🗑

You should see the *DefaultWebApp* application somewhere on the list. Click on the link under the **Name** column and examine some of the details for the application. Look in the **Targets** tab and make sure the *DefaultWebApp* web application is targeted to *myserver*, fixing it to be so if necessary.

Finally, verify that the *DefaultWebApp* web application is actually the default web application for *myserver*. Open the server configuration page (click on the **myserver** item in the **Servers** folder in the navigation pane) and open the **Configuration/HTTP** page in the nested notebook on the right. Ensure that *DefaultWebApp* appears in the first droplist as the selected default web application for *myserver*, fixing it and applying the change to make it so if not. This change does not take effect until you reboot the domain in the current version of WebLogic, so exit the console and reboot the server at this time.

Once you have associated the default application with your server, you can test the HTTP and web-application functionality of the server . Open a separate browser window and attempt to access the URL:

> *http://servername:7001/index.html*

Replace "servername" with *localhost* or the address of your server as appropriate. You should see a simple WebLogic index page. You can find the file for this page at:

> */config/ejbbook/applications/DefaultWebApp/index.html*

Feel free to edit this file or create other HTML files in this directory to convince yourself that your *ejbbook* domain is able to serve web pages from this default web application.

Note that you do not specify the full context path */DefaultWebApp/index.html* in the URL because the server was configured to use this web application as the default if no other context path is specified. Contrast this with the *console* web application which requires that you specify *http://server:port/console/...* when you use the console application because it is not the default web application for the server.

Configuring TitanApp Application in EJBBook Domain

There are a number of ways to deploy web-application components and EJB components in J2EE server products such as WebLogic. A detailed discussion of .war files, .jar files, and .ear files is beyond the scope of this workbook. The online documentation describes the many options and issues; see these two locations:

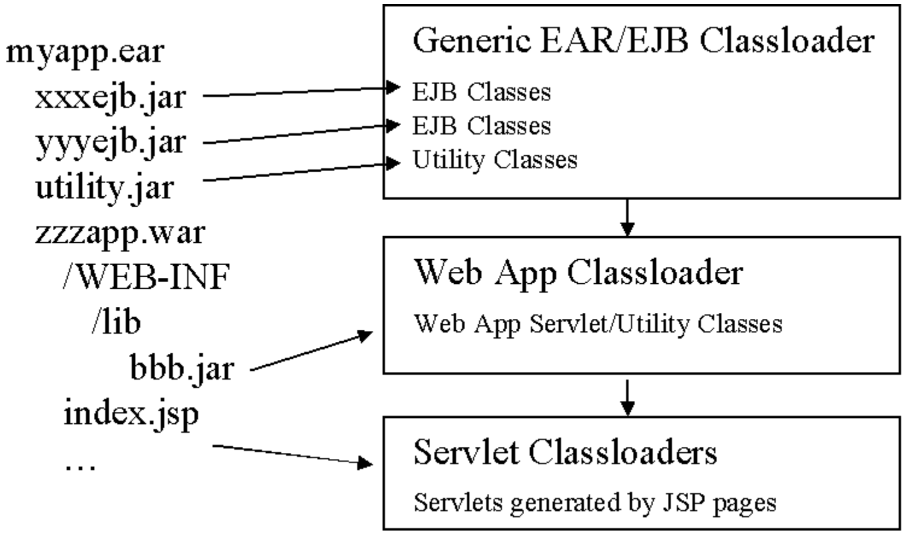
> *http://edocs.bea.com/wls/docs61/adminguide/appman.html*
>
> *http://e-docs.bea.com/wls/docs61/programming/packaging.html*).

The basic rule of thumb is that any component placed in the magic *applications* directory is loaded by a separate classloader (actually a heirarchy of classloaders, see below) in WebLogic, allowing for individual re-deployment of that component. This is true for .war files containing web applications, .jar files containing one or more EJB components, and .ear files containing combinations of .jar and .war files. Exploded versions of these archive-file types are also treated in the same manner, meaning that if you create the proper directory structure and descriptor files required for an archive, the resulting directory structure will be treated (more or less) as a single archive file.

The downside to having many individual .jar files in the applications directory (the way the *examples* domain was configured during installation) is that no component loaded by one classloader can see or use classes loaded by a different classloader at the same "level" in the hierarchy of loaders. For this reason, it is very common to create a single .ear archive file containing all web-application components, EJB components, and supporting classes and deploy this monolithic .ear file as a single application in WebLogic.

WebLogic 6.1 uses a heirarchy of classloaders for an enterprise application archive (.ear) file as illustrated in Figure 27. All EJB components and utility classes are loaded by the top-level classloader and have mutual visibility, and all web-application components and servlets are loaded by child classloaders of the top-level classloader and therefore have visibility to all top-level classes. This is important because it allows web-application components in .ear files to use all classes located in top-level .jar archive files, including EJB remote interfaces, local interfaces (references to the beans themselves), custom exception classes, utility classes, etc. There is no need to place common classes such as these in the system classpath or in"client" jar files in the web application */WEB-INF/lib* directory as was commonly done in WebLogic 5.1.

Figure 27: Class loader heirarchy used for .ear files



There are many pros and cons involved in choosing a deployment technique. For this workbook, we will be using the single monolithic .ear file approach to give good mutual visibility between components in the application; but we will deploy the .ear file as an exploded directory structure under the *applications* directory to avoid the final archive step and to make the individual pieces of an enterprise application archive more visible.

All of the example EJB components and JSP pages created in the following sections of this workbook will be part of a new exploded enterprise (.ear) application called *titanapp*. Enterprise applications have the following basic form:

```
rootdir\
       EJB .jar files
       Web-App .war files
       META-INF\
               application.xml      enterprise app descriptor file
```

Each of the archive files in the enterprise application (.ear) file can likewise be a true archive file or an exploded file. In our case, we will include a simple web application called *webapp* in our *titanapp* enterprise application, using the exploded .war format, so our final exploded directory structure will be:

```
titanapp\
        EJB .jar files
        webapp\
              .html files
              .jsp files
              WEB-INF\
                     web.xml          web-app descriptor file
                     weblogic.xml     weblogic-specific descriptor
        META-INF\
                application.xml      enterprise app descriptor
                REDEPLOY             empty file used by WebLogic
                                     to redeploy application
```

Before we can begin building and deploying the example EJBs, we must create this set of directories and files under the *ejbbook/applications* directory. The download site for this workbook includes a prebuilt version of this empty *titanapp* enterprise archive directory structure (*titanapp_empty.jar*). You may download and extract the directory structure and files to your *config/ejbbook/applications* directory to create the exploded *titanapp* application or build the application manually.

If you choose to build the structure and descriptor files manually, use the following as minimal models for the respective files:

web.xml

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <display-name>EJBBook Examples Web Application</display-name>
</web-app>
```

weblogic.xml

```
<?xml version="1.0" ?>
<!DOCTYPE weblogic-web-app PUBLIC
"-//BEA Systems, Inc.//DTD Web Application 6.0//EN"
"http://www.bea.com/servers/wls600/dtd/weblogic-web-jar.dtd">
<weblogic-web-app>
     <jsp-descriptor>
       <jsp-param>
         <param-name>pageCheckSeconds</param-name>
         <param-value>1</param-value>
       </jsp-param>
       <jsp-param>
         <param-name>verbose</param-name>
         <param-value>true</param-value>
       </jsp-param>
     </jsp-descriptor>
</weblogic-web-app>
```

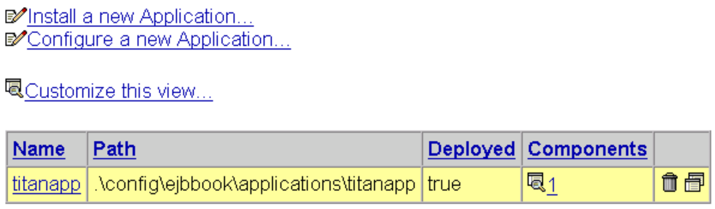
application.xml

```
<?xml version="1.0" ?>
<!DOCTYPE application PUBLIC
'-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN'
'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>
<application>
   <display-name>titanapp</display-name>
   <description>Examples from OReilly EJB Book</description>
   <module>
         <web>
               <web-uri>webapp</web-uri>
               <context-root>webapp</context-root>
         </web>
   </module>
</application>
```

Important note: Stop the ejbbook domain while you are building these directory structures until everything seems to be correct, then boot the domain. It should see the new exploded enterprise application and automatically deploy it in the domain.

Open the management console and click on the **Applications** folder in the navigation pane on the left. You should see the *titanapp* application in the list on the right:

Figure 28: Finding the titanapp application


Install a new Application...
Configure a new Application...

Customize this view...

Name	Path	Deployed	Components	
titanapp	.\config\ejbbook\applications\titanapp	true	1	

Click on the **titanapp** link in the **Name** column and drill down to the *webapp* component within the application and verify that it is properly targeted to the *myserver* server.

Place a simple JSP page such as the following in the *webapp* directory in the exploded structure and attempt to view this JSP page using a URL like:

http://servername:7001/webapp/simple.jsp

simple.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Simple JSP Page</TITLE></HEAD>
<BODY>
<H2>Counting Fun</H2>
<% for (int jj=1; jj<=10; jj++) { %>
   <%= jj %><br>
<% } %>
</BODY>
</HTML>
```

Congratulations! You have now successfully created and configured an enterprise application in the *ejbbook* domain, including a web application able to serve up JSP pages.

You are ready to begin building, deploying, and testing the example EJB components from the O'Reilly EJB book! Let's go!

Exercise Code Setup and Configuration

The example code is organized as a series of directories, one for each "exercise" called out in the O'Reilly EJB book. These directories must be located beneath a common directory containing a handful of common configuration files. The discussion in this workbook will assume that the working directory is organized as shown below:

```
\work\ejbbook\
             common.properties     used by build.xml scripts
             setEnv.cmd or .sh     script to set envir vars
             ex04_1\               Exercise 4-1 files
                   build.xml
                   setEnv.cmd or .sh    calls common version
                   <descriptors>
                   com\
                      titan\
                           pkg1\
                               EJB .java files
                           pkg2\
                               EJB .java files
                           clients\
                                  Client .java files

             ex04_2\               Exercise 4-2 files
             ...
```

The individual *ex##_#* directories will be known in this workbook as the "exercise work root" directories. You should always perform builds for a given exercise from the exercise work root directory for that exercise.

Some EJB components are built/deployed/tested in one exercise and then used in a subsequent exercise (`CabinEJB` is an example). Other beans are created in one exercise and then modified heavily in subsequent exercises. Keeping each exercise in a separate directory structure is a way to insulate previous work from changes required for subsequent exercises even when the same beans are involved in both exercises.

The download site for this workbook provides copies of the common files required in the */work/ejbbook* directory in archive files called *work_ejbbook_win.jar* and *work_ejbook_sol.jar*. Download one of these files, create a working directory (*/work/ejbbook*) to use as the root for all of the exercises, and extract the contents of the archive file into that working directory. The *common.properties* file in the archive is shown below and may require modification if your installation does not match the assumed directory structure:

common.properties – NT/Win2K

```
JAVA_HOME=/bea/jdk131
WEBLOGIC_HOME=/bea/wlserver6.1
DOMAIN=ejbbook
APPLICATION=titanapp
```

common.properties – SunOS

```
JAVA_HOME=/opt/bea/jdk131
WEBLOGIC_HOME=/opt/bea/wlserver6.1
DOMAIN=ejbbook
APPLICATION=titanapp
```

The download site for this workbook also contains individual *.jar* archives for each exercise called out in the O'Reilly EJB book. When you extract one of these exercise archives, be sure to extract it in the */work/ejbbook* directory so that the resulting *ex##_#* exercise work root directory is a direct subdirectory of the */work/ejbbook* directory, allowing the *build.xml* files in each exercise to find the *common.properties* file in the ".." directory relative to themselves.

Each downloaded exercise archive will typically include the following files:

- *build.xml* – Ant build script, normal targets are *clean*, *compile*, *ejbjar*, *dist,* and *redeploy*.
- *setEnv.cmd/.sh* – Scripts to set environment variables before building the exercise.
- *ejb-jar.xml*, *weblogic-jar.xml*, other xml descriptor files.
- Source code for EJB beans in *com/titan/xxxx* package directory.
- Source code for Client java programs in *com/titan/clients* directory.
- Client JSP files in */jsp* directory.

Finally, there is a single archive file (*everything.jar*) available on the download site that contains all exercise archive files and all of the other scripts and archives described in the previous sections if this is more convenient than downloading the files you need one at a time.

Enough preparation – why not build your first example?

Exercises for Chapter 4

Exercise 4.1: A Simple Entity Bean

The Cabin EJB demonstrates basic CMP 2.0 capability for a simple entity bean mapped to a single table. The following sections outline the steps necessary to download, build, deploy, and test the Cabin EJB.

Download and Build the Example Programs

Perform the following steps:

1. Download the *ex04_1.jar* file from the download site for the workbook and place it in your main work root directory (*/work/ejbbook* or equivalent).
2. Open a Command Prompt or telnet window and change to this directory.
3. Extract the files from the archive using `jar xvf`. If *jar* is not in your path, modify your user profile or system environment variables to add the *<BEAHOME>/jdk131/bin* directory in the BEA installation to your default path. The correct *jar* command is:

   ```
   jar xvf ex04_1.jar
   ```
4. Change directories down to the *ex04_1* directory created by the extraction process.
5. Execute the *setEnv.cmd* script or *setEnv.sh* script to set environment variables properly before attempting to build the example. On SunOS, be sure to use `. ./setEnv.sh` to add the environment variables in the script to your shell environment.
6. Perform the build by typing `ant dist` to run the *ant* utility and execute everything in the *build.xml* file required to compile and deploy the EJB files.

If you are not familiar with the *ant* application, visit the Ant web site in the Jakarta project at *http://jakarta.apache.org/ant/index.html* to learn about this utility. The *build.xml* file provided with the workbook exercises contains the following tasks:

- *clean* – Removes all .class and .jar files from the working directory (and subdirectories)
- *compile* – Compiles all EJB and client Java files
- *ejbjar* – Performs the ejbc process to create a packaged EJB application *.jar* file (*titanejb.jar*), containing all of the EJB components in this exercise
- *dist* – Copies the *titanejb.jar* file and any JSP files to their proper locations in the *titanapp* enterprise application within the *ejbbook* domain directory structure
- *redeploy* – Forces a running copy of WebLogic to redeploy the *titanapp* application (discussed in detail in Exercise 4-2)

There are dependencies expressed in the script using the `depends` attributes in the task tags. When the *dist* task is run, for example, it first performs the *compile* and *ejbjar* tasks if these are required based on file time stamps, etc. The workbook will typically instruct you to execute a specific task like *dist* knowing that *ant* will perform other tasks like *compile* automatically if required. Feel free to perform builds one step at a time if it helps you understand the process.

To see details of the operations performed by *ant*, use the `-verbose` option on the command line.

Create the Required Database Objects

Exercise 4-1 requires a single table (`CABIN`) in the database used by the JDBC pool we configured, *titan-pool*. The schema should be:

```
create table CABIN
(
  ID int primary key,
  SHIP_ID int,
  BED_COUNT int,
  NAME char(30),
  DECK_LEVEL int
)
```

Examine the WebLogic-Specific Files/Components

Take a moment to examine the descriptors in the *ex04_1* directory to understand the custom descriptors required by WebLogic for the Cabin EJB. Every EJB deployed in WebLogic requires a descriptor file called *weblogic-ejb-jar.xml* which defines WebLogic-specific container parameters and provides other run-time information required by the server. Think of this file as an extension to the standard *ejb-jar.xml* file required by all EJBs. See the online documentation at *http://e-docs.bea.com/wls/docs61/ejb/reference.html* for a detailed description of all elements and valid values.

In the case of the CabinEJB, this file is fairly straightforward:

weblogic-ejb-jar.xml

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC
'-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB//EN'
'http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd'>
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>CabinEJB</ejb-name>
    <entity-descriptor>
      <entity-cache>
        <max-beans-in-cache>100</max-beans-in-cache>
      </entity-cache>
      <persistence>
        <persistence-type>
          <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
          <type-version>6.0</type-version>
          <type-storage>
            META-INF/weblogic-cmp-rdbms-jar.xml
          </type-storage>
        </persistence-type>
        <persistence-use>
          <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
          <type-version>6.0</type-version>
        </persistence-use>
      </persistence>
    </entity-descriptor>
    <jndi-name>CabinHomeRemote</jndi-name>
  </weblogic-enterprise-bean>
  <!-- Map the normal weblogic users to the everyone role -->
  <security-role-assignment>
    <role-name>everyone</role-name>
    <principal-name>guest</principal-name>
    <principal-name>system</principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>
```

Because this exercise is the first one, we'll take the time to walk through this descriptor and discuss the important elements.

```
<ejb-name>CabinEJB</ejb-name>
```

The `<ejb-name>` element provides a tie back to the `<ejb-name>` element in *ejb-jar.xml*.

```
      <entity-cache>
        <max-beans-in-cache>100</max-beans-in-cache>
      </entity-cache>
```

This set of elements defines the maximum number of beans of this class (CabinBean) allowed in memory at the same time. WebLogic will begin passivating beans if this limit is reached during execution, swapping less-recently-used beans to disk to make room for new instances.

```
        <persistence>
          <persistence-type>
            …
          </persistence-type>
          <persistence-use>
            …
          </persistence-use>
        </persistence>
```

The <persistence> elements indicate the CMP persistence type, version, and associated *weblogic-cmp* descriptor file to be used at runtime for bean persistence services. The *ejbc* process also reads this element to find the CMP descriptor file to use in generating the CMP java class.

```
      <jndi-name>CabinHomeRemote</jndi-name>
```

The <jndi-name> element specifies the JNDI name the container will use when registering this bean's Home interface in the JNDI tree. This name will be used by clients (and other beans) to perform JNDI lookup operations.

```
    <security-role-assignment>
      <role-name>everyone</role-name>
      <principal-name>guest</principal-name>
      <principal-name>system</principal-name>
    </security-role-assignment>
```

The final elements in the *weblogic-jar.xml* file map the security role everyone defined in the *ejb-jar.xml* file to specific users within the WebLogic default realm. By mapping guest to the required role, you ensure that no additional authentication or credentials will be required when invoking methods on the bean.

The other WebLogic-specific descriptor file is *weblogic-cmp-rdbms-jar.xml*. This file is required only for CMP entity beans:

weblogic-cmp-rdbms-jar.xml

```
<!DOCTYPE weblogic-rdbms-jar PUBLIC
 '-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB RDBMS
Persistence//EN'
 'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms20-
persistence-600.dtd'>
```

```
<weblogic-rdbms-jar>
  <weblogic-rdbms-bean>
    <ejb-name>CabinEJB</ejb-name>
    <data-source-name>titan-dataSource</data-source-name>
    <table-name>CABIN</table-name>
    <field-map>
      <cmp-field>id</cmp-field>
      <dbms-column>ID</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>shipId</cmp-field>
      <dbms-column>SHIP_ID</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>bedCount</cmp-field>
      <dbms-column>BED_COUNT</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>name</cmp-field>
      <dbms-column>NAME</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>deckLevel</cmp-field>
      <dbms-column>DECK_LEVEL</dbms-column>
    </field-map>
  </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

The Cabin EJB is a very simple bean with a small number of columns, so the CMP descriptor file is very simple:

```
    <ejb-name>CabinEJB</ejb-name>
```

The `<ejb-name>` element again provides a tie back to the previous descriptor files.

```
    <data-source-name>titan-dataSource</data-source-name>
    <table-name>CABIN</table-name>
```

The `<data-source-name>` and `<table-name>` elements indicate the Data Source to be used for persistence services (recall that a Data Source is a J2EE-defined abstraction on top of a specific JDBC connection pool) as well as the specific table in the schema containing this bean's persistent data. Note that there can be only one `<table-name>` element, so mapping a bean across multiple tables requires a view in the database, or other custom technique.

```
    <field-map>
      <cmp-field>id</cmp-field>
      <dbms-column>ID</dbms-column>
    </field-map>
```

Each attribute in the bean is mapped to a specific database column using a `<field-map>` element.

The *ejbc* process uses the descriptor information in these two WebLogic-specific files along with the standard *ejb-jar.xml* file to generate all of the persistence services and concrete implementations of the abstract methods defined in the bean class.

Deploy the EJB Components to WebLogic

After using the *ant ejbjar* task, place the *titanejb.jar* file in the root directory of the exploded enterprise archive directory structure. Assuming that you are using the suggested domain/application names, the pathname will be:

./config/ejbbook/applications/titanapp/titanejb.jar

The provided *ant dist* task places the completed EJB jar file in the correct location.

Because you are using an exploded .ear type of structure, WebLogic will not automatically deploy the EJB when the file appears in the directory, even after a reboot of the server. This minor disadvantage is offset by the advantage inherent in WebLogic's policy of using a single class loader for everything in an .ear file, whether exploded or not.

The console enables you to deploy EJBs through a set of configuration screens if the EJB .jar file is located in the *applications* directory, but you cannot use it to deploy an EJB within an exploded .ear file. For this reason, and to be consistent with the structure and requirements of a valid .ear file, you should add the new EJB in the application descriptor file so it will be deployed automatically when the server boots and loads the exploded .ear file.

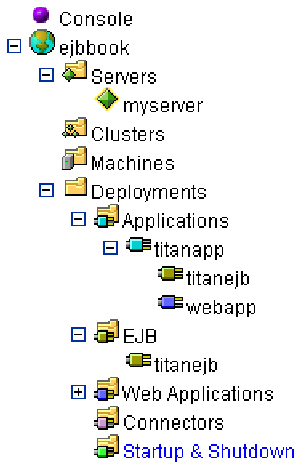
Steps to add an EJB to an exploded .ear file:

1. Edit *titanapp/META-INF/application.xml* and add an `<ejb>` module element to the application. The next time WebLogic is booted it will pick up the new EJB and deploy it within the application.

```
<application>
   <display-name>titanapp</display-name>
   <description>Examples from OReilly EJB Book</description>
   <module><web>
         <web-uri>webapp</web-uri>
         <context-root>webapp</context-root>
   </web></module>
   <module><ejb>titanejb.jar</ejb></module>
</application>
```

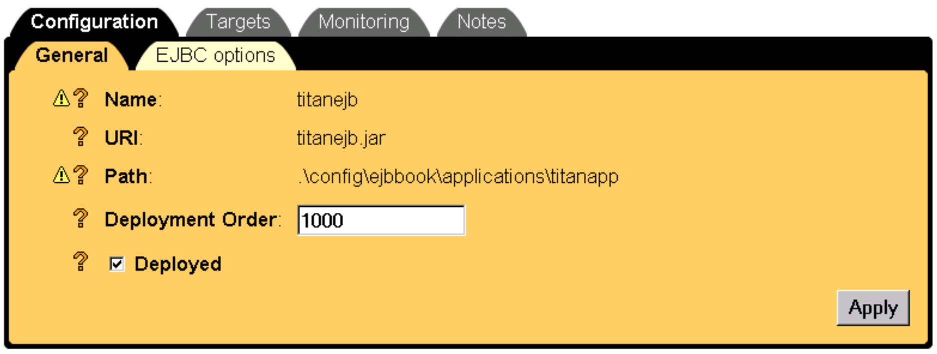
2. Reboot WebLogic and open the console. In the navigation tree on the left side of the console you should now see the *titanejb* deployed as an EJB component within the *titanapp* application:

Figure 29: Verifying the deployment of titanejb component



3. Click on the *titanejb* component within the *titanapp* application. The right pane should show the name, URI, and path of *titanejb*, and the Deployed checkbox should be set.

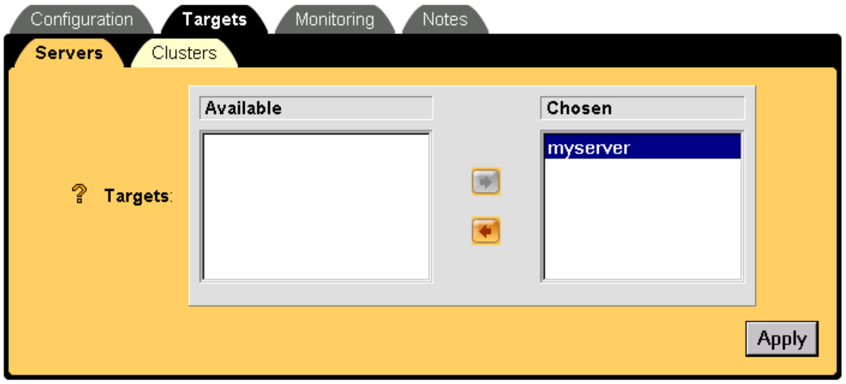
Figure 30: Verifying configuration of titanejb component



Name: titanejb
URI: titanejb.jar
Path: .\config\ejbbook\applications\titanapp
Deployment Order: 1000
☑ Deployed

4. Click on the **Targets** tab to bring up two lists of servers:

Figure 31: Verifying that titanejb is properly "targeted"



Targets:

Available	Chosen
	myserver

5. When an EJB component is deployed for the first time by reading the *application.xml* file in the .ear (exploded or not), WebLogic does not know on which servers this component should be available. Move the *myserver* target from **Available** to **Chosen** to indicate that this component should be deployed on the server *myserver*, then press **Apply**.

The bean is now deployed and ready for testing.

WebLogic saves the configuration of this EJB component, including target information, in the *config.xml* file. Subsequent reboots of the server will deploy the EJB component properly, whether the `<module><ejb>` entry continues to appear in the *application.xml* file or not.

The GUI-averse can avoid using the console when deploying a new EJB component within an exploded .ear file:

1. Add the EJB component to the *META-INF/application.xml* file as described above.
2. Ensure the *ejbbook* domain is not running.
3. Modify the *config.xml* file for the ejbbook domain to add a new `<EJBComponent>` element for the *titanejb* component to the *titanapp* `<Application>` element:

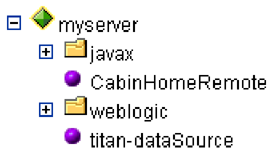
```
<Application Deployed="true" Name="titanapp"
             Path=".\config\ejbbook\applications\titanapp">
    <EJBComponent Name="titanejb" Targets="myserver"
                  URI="titanejb.jar"/>
    <WebAppComponent Name="webapp" Targets="myserver" URI="webapp"/>
</Application>
```

4. Restart the WebLogic server. The bean should now be deployed and ready for testing.

Verify that the bean is properly deployed and has registered its Home interface in the JNDI tree by examining the contents of the JNDI tree. In WebLogic 6.1 the only way to view the JNDI tree is to right-click on the server icon in the navigation tree on the left side, and select **View JNDI**

Tree from the popup menu. You should see a new browser window displaying the JNDI tree in the left navigation pane, including entries for the `titan-dataSource` and `CabinHomeRemote`:

Figure 32: Viewing the JNDI tree for myserver



Browse through the tree and examine the various entries to get a better sense of how the JNDI service works in a WebLogic environment. If you do not see the `CabinHomeRemote` and data source entries in the tree, examine the log file during a reboot for any error messages that might indicate the cause of the problem.

Examine and Run the Client Applications

Two client applications provided in the workbook download follow the examples in the O'Reilly EJB book:

- *Client_41.java* – Creates a single Cabin bean and populates its attributes, then looks it up again by primary key.
- *Client_42.java* – Creates 99 additional beans with a variety of data useful in subsequent client programs.

Examine the Client_41 source code to identify the WebLogic-specific code and review the basics of JNDI lookups and bean creation.

Client_41.java

```
package com.titan.clients;

import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

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

```
public class Client_41 {
    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref = jndiContext.lookup("CabinHomeRemote");
            CabinHomeRemote home = (CabinHomeRemote)
        PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
            CabinRemote cabin_1 = home.create(new Integer(1));
            cabin_1.setName("Master Suite");
            cabin_1.setDeckLevel(1);
            cabin_1.setShipId(1);
            cabin_1.setBedCount(3);

            Integer pk = new Integer(1);
            CabinRemote cabin_2 = home.findByPrimaryKey(pk);
            System.out.println(cabin_2.getName());
            System.out.println(cabin_2.getDeckLevel());
            System.out.println(cabin_2.getShipId());
            System.out.println(cabin_2.getBedCount());

        } catch (java.rmi.RemoteException re)
                  {re.printStackTrace();}
          catch (javax.naming.NamingException ne)
                  {ne.printStackTrace();}
          catch (javax.ejb.CreateException ce)
                  {ce.printStackTrace();}
          catch (javax.ejb.FinderException fe)
                  {fe.printStackTrace();}
    }

    public static Context getInitialContext()
        throws javax.naming.NamingException {

        Properties p = new Properties();
         p.put(Context.INITIAL_CONTEXT_FACTORY,
                   "weblogic.jndi.WLInitialContextFactory");
         p.put(Context.PROVIDER_URL, "t3://localhost:7001");
        return new javax.naming.InitialContext(p);
    }
}
```

The first line in the `main` method calls the `getInitialContext` method to acquire a reference to the JNDI context. The `getInitialContext` method specifies the proper values for `INITIAL_CONTEXT_FACTORY` and `PROVIDER_URL` for attaching to the WebLogic JNDI context.

Note that the PROVIDER_URL assumes that the client is executing on the same machine as the WebLogic server instance (i.e., the *localhost* in the URL).

The next 7-8 lines employ the JNDI tree to look up the CabinHomeRemote interface, create an empty entity bean with the primary key of 1, and set the persistent attributes for the bean to some valid values.

The next six lines find this bean by specifying the primary key, then display the persistent attributes of the bean using the newly-acquired reference to it. The displayed values should be the same as the values set via the original reference.

Run the client application by invoking *java* and specifying the fully-qualified class name of the client. Don't forget to run the *setEnv.cmd* or *.sh* script before running the client if you have not already done so.

```
C:\work\ejbbook\ex04_1>setenv
...
C:\work\ejbbook\ex04_1>java com.titan.clients.Client_41
```

The output of the client application should be:

```
Master Suite
1
1
3
```

Note that Client_41 adds a row to the database representing the bean and does not delete it at the conclusion of the application. You can examine the database and see the row in the table for this bean to convince yourself it truly did get saved to the database. Unfortunately, if you attempt to run Client_41 again it will attempt to insert the same bean in the database, and you will get an error:

```
java.rmi.RemoteException: EJB Exception:; nested exception is:
java.sql.SQLException: Violation of PRIMARY KEY constraint
'PK_CABIN'.
Cannot insert duplicate key in object 'CABIN'.
```

If you do not get an error like this, look at your table definition again – it is likely you forgot to set the ID column as a unique primary key in the CABIN table and you now have multiple rows in the database with the same ID. If you find yourself in this situation, delete everything in the table, fix the ID column to be a unique primary key, and run Client_41 once again to create the row. Then, run the client one *more* time, to verify that the database reports some sort of duplicate-key exception.

Examine *Client_42.java* yourself to understand the use of repeated calls to create on the Home interface to make many beans. The output from running this example should look something like:

```
PK = 1, Ship = 1, Deck = 1, BedCount = 3, Name = Master Suite
PK = 2, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 100
PK = 3, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 101
PK = 4, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 102
PK = 5, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 103
PK = 6, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 104
PK = 7, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 105
...
PK = 97, Ship = 3, Deck = 4, BedCount = 2, Name = Suite 406
PK = 98, Ship = 3, Deck = 4, BedCount = 3, Name = Suite 407
PK = 99, Ship = 3, Deck = 4, BedCount = 2, Name = Suite 408
PK = 100, Ship = 3, Deck = 4, BedCount = 3, Name = Suite 409
```

After running Client_42, check that you have rows in the `CABIN` table for `ID` values 1-100, with no holes or duplicates. Like Client_41, this example creates rows in the table and does not delete them when finished, so Client_42 can be run only once without causing duplicate-key exceptions.

Examine and Run the Client JSP Pages

The *ant dist* task copied a number of JavaServer Pages (JSP) files from the */ex04_1/jsp* directory to the *webapp* directory contained in the *titanapp* application. This workbook includes JSP-based versions of many of the client applications, and in some cases provides JSP examples in lieu of Java client applications (Chapter 7&8 exercises). If a JSP page is provided for a client application, it will have the same root name as the Java client (e.g., *Client_41.java* has a matching *Client_41.jsp* file).

Recall that any JSP page located in the *webapp* directory inside the *titanapp* application is accessed through the WebLogic server using a URL like:

http://servername:7001/webapp/Client_41.jsp

The *servername* portion of the URL will be *localhost* if you are running WebLogic on your own workstation.

If you attempt to run this JSP after successfully executing the Java versions of Client_41 and Client_42, be prepared for a Primary Key Constraint error from the database, because the rows you are trying to create already exist (you did remember to set the `ID` column in the `CABIN` table as a unique primary key column, right?). You'll need to delete all of the data in the `CABIN` table before running the JSP versions of these clients.

Provided with the client JSP pages in this exercise is a small utility JSP page called *ViewDB.jsp*, which provides a handy (and cross-platform) mechanism for viewing and deleting the data in your database. Try invoking this JSP now by opening a new browser and accessing this page with a URL like this one:

http://servername:7001/webapp/ViewDB.jsp

You should see a page with content similar to this (assuming you've successfully run the *Client_41* and *Client_42 Java* programs already):

```
View and Delete Data in Database

Delete All Rows in All Workbook Tables

CABIN
ID=1 SHIP_ID=1 BED_COUNT=3 NAME=Master Suite DECK_LEVEL=1 Delete Row
ID=2 SHIP_ID=1 BED_COUNT=2 NAME=Suite 100 DECK_LEVEL=1 Delete Row
...
ID=98 SHIP_ID=3 BED_COUNT=3 NAME=Suite 407 DECK_LEVEL=4 Delete Row
ID=99 SHIP_ID=3 BED_COUNT=2 NAME=Suite 408 DECK_LEVEL=4 Delete Row
ID=100 SHIP_ID=3 BED_COUNT=3 NAME=Suite 409 DECK_LEVEL=4 Delete Row

Delete all rows in CABIN table

SHIP
Error reading table, may not be present yet.

CUSTOMER
Error reading table, may not be present yet.

ADDRESS
Error reading table, may not be present yet.

-- more tables listed and not present yet --
```

The *ViewDB.jsp* page essentially dumps the contents of all tables created in the workbook exercises, to give you an easy way to see all related database rows in a single display. It also provides links to delete individual rows from a table, delete all rows in a table, or delete all rows in all workbook tables, to make it much easier to clean up the database after a bad run or when you wish to re-run a client program which creates data. Note that if you have referential integrity turned on between related tables, you'll have to perform the delete operations in the proper order (children first, then parents), or the database won't allow the deletes. The **Delete All Rows in All Workbook Tables** link should perform deletes in the right order based on the relationships defined in these exercises.

In this case you want to run the *Client_41.jsp* and *Client_42.jsp* pages, so click on the link to **Delete all rows in CABIN table**. The table should now appear (and be!) empty. Now you can execute the Client_41 and _42 JSP pages to re-create the `CABIN` rows and use the ViewDB JSP to verify the rows are present again before proceeding to Exercise 4-2.

ViewDB.jsp will come in handy many times over the course of the workbook. Modify it to your own purposes if you wish, or ignore it completely and use some other database viewing and manipulation program, as you see fit.

Exercise 4.2: A Simple Session Bean

The TravelAgent EJB demonstrates the use of a stateless session bean to encapsulate process-related logic, including the use of Cabin entity beans.

Download and Build the Example Programs

Perform the following steps:

1. Download *ex04_2.jar* from the download site for the workbook and place it in your main work root directory (*/work/ejbbook* or equivalent).
2. Open a Command Prompt or telnet window and change to this directory.
3. Extract the files from the archive using `jar xvf`. See the details in Exercise 4-1 if you need more information.
4. Change directories down to the *ex04_2* directory created by the extraction process.
5. Execute the *setEnv.cmd* script or *setEnv.sh* script to set environment variables properly before attempting to build the example.
6. Perform the build by typing `ant dist` to run the *ant* utility and execute everything in the *build.xml* required to compile and deploy the EJB files.

Create the Required Database Objects

Because the TravelAgent EJB is a stateless session bean, no additional database configuration is required. The `CABIN` table must still be available from the previous exercise, and should contain the 100 rows created by the successful execution of the Client_41 and Client_42 programs.

Examine the WebLogic-Specific Files/Components

The *weblogic-ejb-jar.xml* descriptor for a stateless EJB is normally very short. In this case it has an additional element mapping the Cabin EJB `<ejb-ref>` tag from the standard *ejb-jar.xml* file to the JNDI name of the Cabin bean:

weblogic-ejb-jar.xml

```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>CabinEJB</ejb-name>
      …
  </weblogic-enterprise-bean>
  <weblogic-enterprise-bean>
    <ejb-name>TravelAgentEJB</ejb-name>
    <stateless-session-descriptor>
      <pool>
        <max-beans-in-free-pool>100</max-beans-in-free-pool>
      </pool>
    </stateless-session-descriptor>
    <reference-descriptor>
       <ejb-reference-description>
         <!-- Matches entry in ejb-jar.xml file -->
         <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
         <jndi-name>CabinHomeRemote</jndi-name>
       </ejb-reference-description>
    </reference-descriptor>
    <jndi-name>TravelAgentHome</jndi-name>
  </weblogic-enterprise-bean>
  <!-- Map the normal weblogic users to the everyone role -->
  <security-role-assignment>
    <role-name>everyone</role-name>
    <principal-name>guest</principal-name>
    <principal-name>system</principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>
```

Look at some of the new elements introduced in this exercise:

```
    <stateless-session-descriptor>
      <pool>
        <max-beans-in-free-pool>100</max-beans-in-free-pool>
      </pool>
    </stateless-session-descriptor>
```

The first new element is the `<max-beans-in-free-pool>` element which controls the maximum number of bean instances available in the free pool before the container begins synchronizing access to these stateless session beans. This mechanism can be a powerful way to throttle or limit the number of simultaneous requests for a resource-intensive service.

```
      <reference-descriptor>
         <ejb-reference-description>
           <!-- Matches entry in ejb-jar.xml file -->
           <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
           <jndi-name>CabinHomeRemote</jndi-name>
         </ejb-reference-description>
      </reference-descriptor>
```

The next new section of the *weblogic-ejb-jar.xml* descriptor file is related to some elements in the *ejb-jar.xml* file. Recall that *ejb-jar.xml* had a section like this:

```
       <ejb-ref>
         <!-- requires an entry in weblogic-ejb-jar.xml also -->
         <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
         <ejb-ref-type>Entity</ejb-ref-type>
         <home>com.titan.cabin.CabinHomeRemote</home>
         <remote>com.titan.cabin.CabinRemote</remote>
       </ejb-ref>
```

The `<ejb-ref-name>` element in the *ejb-jar.xml* file provides the portable JNDI name available within the TravelAgent EJB to look up the Cabin EJB Home interface. The corresponding `<ejb-reference-description>` element in the *weblogic-ejb-jar.xml* file maps this portable name back to the specific JNDI name used to register the Cabin Home interface (defined in the Cabin EJB section of the *weblogic-ejb-jar.xml* descriptor file).

Finally, note that the TravelAgent EJB will be registering its Home interface using the JNDI name `TravelAgentHomeRemote`. This will be the name client applications use during lookup operations.

Deploy the EJB Components to WebLogic

After using *ant ejbjar* to build the new *titanejb.jar* file, containing both the Cabin EJB and TravelAgent EJB, place it in the root directory of the exploded enterprise archive directory structure. Assuming that you are using the suggested domain/application names, the pathname will be:

./config/ejbbook/applications/titanapp/titanejb.jar.

The *ant dist* task places the completed EJB jar file in the correct location.

Because we've simply replaced the old version of *titanejb.jar* with a new one containing both beans, there is no additional deployment or targeting effort necessary to configure the new EJB in WebLogic. If the server was not running during the *ant dist* task, simply reboot WebLogic and it should deploy both beans.

WebLogic provides the capability to "hot deploy" and "hot redeploy" application components in a running server. If the server was running when the *ant dist* task placed the updated copy of *titanejb.jar* in the proper location, you can now perform a hot redeployment of the enterprise application to test this functionality.

Redeployment of an .ear application archive is normally performed automatically when the time stamp on the .ear file itself changes. WebLogic polls for this change according to a schedule defined in the domain configuration. Ensure that this feature is enabled in the *ejbbook* domain by opening a console, clicking on the *ejbbook* node in the navigation pane, opening the **Configuration-Applications** tab, and checking that the **Auto Deployment Enabled** checkbox is selected for the domain.

Because we're using an exploded .ear deployment mechanism there is no single .ear file for WebLogic to monitor for time-stamp changes. Rather than constantly scanning the directory structure for any files with time-stamp changes, WebLogic monitors a single "magic" file called *REDEPLOY* located in the *META-INF* directory of the exploded .ear structure as a surrogate for monitoring the entire exploded structure. In other words, if the time stamp on *REDEPLOY* changes, WebLogic treats the change as a request to re-deploy the entire enterprise application.

The *build.xml* file includes a *redeploy* task to make this process straightforward. While the server is running, execute the *ant redeploy* task to touch the *REDEPLOY* file and force a re-deployment of the entire *titanapp* enterprise application. If you are monitoring the output from the WebLogic server instance, you should see many messages related to the undeploy/redeploy process. Assuming everything redeploys smoothly, you should be able to avoid rebooting the server for this and subsequent exercises using the *redeploy* task when a modified application is ready for deployment. If the redeploy task fails to work, it may be caused by WebLogic's new "production mode" feature discussed during the server setup portion of the workbook. Redeploy will work only in development mode, so ensure that the *startWebLogic.cmd* or *.sh* file for the *ejbbook* domain sets the `STARTMODE` variable to false.

Whether you performed a hot re-deployment of the *titanapp* application or simply re-booted the server to deploy the new EJB components, verify that the TravelAgent EJB is properly deployed: Examine the JNDI tree for the server (right-click on the server in the navigation pane and choose **View JNDI tree**), and check that both `CabinHomeRemote` and `TravelAgentHomeRemote` are registered in the tree.

Examine and Run the Client Applications

There is one client application for this exercise that uses the TravelAgent EJB to list cabins which meet certain criteria. The important code is shown below.

Client_43.java

```
...
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("TravelAgentHomeRemote");
TravelAgentHomeRemote home = (TravelAgentHomeRemote)
PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

TravelAgentRemote travelAgent = home.create();

// Get a list of all cabins on ship 1 with a bed count of 3.
String list [] = travelAgent.listCabins(SHIP_ID,BED_COUNT);

for(int i = 0; i < list.length; i++){
    System.out.println(list[i]);
}
...
```

Note that the Home interface's `create` method is used when a client wishes to acquire a reference (a remote interface) to a session bean. The client uses the resulting remote reference to call business methods. Each business method invocation is performed by some instance of the bean in the pool without regard for previous invocations. In this example the business method is `listCabins`, which accepts two parameters and returns an array of `String` objects.

Let's examine the code in the TravelAgent EJB `listCabins` method for any WebLogic-specific code or other interesting snippets.

TravelAgentBean.java

```
public String [] listCabins(int shipID, int bedCount) {
    try {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
              "weblogic.jndi.WLInitialContextFactory");
        p.put(Context.PROVIDER_URL, "t3://localhost:7001");
        javax.naming.Context jndiContext = new InitialContext(p);
        Object obj =
            jndiContext.lookup("java:comp/env/ejb/CabinHomeRemote");

        CabinHomeRemote home = (CabinHomeRemote)
          PortableRemoteObject.narrow(obj,CabinHomeRemote.class);
        ...
```

The first few lines of `listCabins` should look familiar – it is acquiring a JNDI context object using the WebLogic-specific properties we've normally seen used in a client application. In a way, the TravelAgent EJB is a client of the Cabin EJB just like any other client, so this makes some sense.

Note that because both objects are contained in a single copy of WebLogic, the setting of these properties could be eliminated, and the empty constructor for `InitialContext` used with the same effective result.

The JNDI lookup appears different than before. The code is acquiring a reference to the Home interface for the Cabin EJB, but rather than look it up by the name "CabinHomeRemote" the code is using the JNDI Environment Naming Context (ENC) name specified in the `<ejb-ref-name>` element in the TravelAgent *ejb-jar.xml* descriptor. As discussed in the EJB book, the root for these JNDI ENC lookups is always *java:comp/env* so the final JNDI lookup becomes *java:comp/env/ejb/CabinHomeRemote*, as shown in the code.

The rest of the `listCabins` method is a fairly straightforward (if awkward) mechanism for looking at every Cabin EJB, and placing information in the returned array of strings if the current bean matches the requested criteria. Note that in a real application this type of "query" would be performed by a "finder" method, something we will be discussing and creating in later exercises.

Run the client application by invoking *java* and specifying the fully-qualified class name of the client. Don't forget to run the *setEnv.cmd* or *.sh* script before running the client if you have not already done so.

```
C:\work\ejbbook\ex04_2>setenv
...
C:\work\ejbbook\ex04_2>java com.titan.clients.Client_43
```

The output of the client application should be:

```
1,Master Suite,1
3,Suite 101,1
5,Suite 103,1
7,Suite 105,1
9,Suite 107,1
12,Suite 201,2
14,Suite 203,2
16,Suite 205,2
18,Suite 207,2
20,Suite 209,2
22,Suite 301,3
24,Suite 303,3
26,Suite 305,3
28,Suite 307,3
30,Suite 309,3
```

Examine and Run the Client JSP Pages

The *Client_43.jsp* page is the same as the *Client_43.java* client program. Try this page to verify that it provides the same information as the Java client. It was copied to the *titanapp/webapp* directory during the *ant dist* task, so it should be available using the typical URL:

http://servername:7001/webapp/Client_43.jsp

Exercises for Chapter 5

Exercise 5.1: The Remote Component Interfaces

Exercise 5.1 explores a number of the features of the Home interface for an EJB, including the `remove` method and the metadata available on the EJB's definition and interfaces.

Download and Build the Example Programs

Download and extract the example directory *ex05_1* and examine the contents. This exercise is a "pure client" exercise – it uses the EJB components built in the previous exercises but does not actually define them or modify them in any way.

Use the *ant dist* task to build the client programs.

Create the Required Database Objects

This exercise requires no additional database objects.

Examine the WebLogic-Specific Files/Components

This exercise introduces no WebLogic-specific files or components.

Deploy the EJB Components to WebLogic

This exercise introduces no new EJB components. The Cabin EJB and TravelAgent EJB must be properly deployed and running in the WebLogic server. (If you have successfully completed Exercise 4-2, they will be.)

Examine and Run the Client Applications

There are two client applications and one "utility" client application provided in the workbook download. These follow the examples in the O'Reilly EJB book:

- *Client_51.java* – Demonstrates use of `remove` on the Cabin EJB Home interface.
- *Client_52.java* – Demonstrates use of metadata methods.
- *Client_51_undo.java* – Re-creates the single Cabin bean deleted in Client_51 to allow re-running of Client_51 or the equivalent JSP page.

Because this exercise does not actually create or include all of the EJB code, the classpath must include the *titanejb-client.jar* file when running the client applications. The script *setEnv.cmd* (or the *.sh* version) sets the classpath properly, and the *ant compile* task in the build script ensures that the *titanejb-client.jar* is copied from the previous exercise to be available in the classpath.

Examine the Client Java code if you want to, but it is fairly straightforward and follows the example in the EJB book closely.

The output from Client_51 should look as described in the EJB book, and the output from Client_52 should be something like:

```
com.titan.cabin.CabinHomeRemote
com.titan.cabin.CabinRemote
java.lang.Integer
false
Master Suite
```

Note that if you attempt to run Client_51 again without running the Client_51_undo application, the `remove` call will fail and you will receive an exception report like the following:

```
java.rmi.RemoteException: EJB Exception:; nested exception is:
javax.ejb.NoSuchEntityException: Bean with primary key: '30' not
found.
```

Examine and Run the Client JSP Pages

The three client applications are duplicated as JSP pages for your use and examination. Once deployed to the */webapp* directory by the *ant dist* task, they should be available using the normal URL syntax for workbook JSP pages.

Note that there is no equivalent to *setEnv* required when JSP pages in the *webapp* web application attempt to access EJB components. This is a byproduct of our decision to use an enterprise application (i.e., an exploded *.ear* file) containing all EJB components as well as the web application itself, an arrangement which causes all classes in the EJB components to be visible to the *webapp* components.

Exercise 5.2: The EJBObject, Handle, and Primary Key

Exercise 5.2 explores some esoteric capabilities of EJB components, including `EJBObject` methods and `Handle`s, and demonstrates the importance of the primary key for a bean.

Download and Build the Example Programs

Download and extract the example directory *ex05_2* and examine the contents. This exercise is also a "pure client" exercise – it uses the EJB components built in the previous exercises but does not actually define them or modify them in any way.

Use the *ant dist* task to build the client programs.

Create the Required Database Objects

This exercise requires no additional database objects.

Examine the WebLogic-Specific Files/Components

There are no WebLogic-specific files or components introduced in this exercise.

Deploy the EJB Components to WebLogic

This exercise introduces no new EJB components. The Cabin EJB and TravelAgent EJB must be properly deployed and running in the WebLogic server.

Examine and Run the Client Applications

Three client applications provided in the workbook download follow the examples in the O'Reilly EJB book:

- *Client_53.java* – Demonstrates using EJBObject to retrieve the home interface using a bean reference.
- *Client_54.java* – Demonstrates serialization of primary keys and equivalence of remote references to the same bean.
- *Client_55.java* – Demonstrates use of handles.

This exercise is similar to Exercise 5-1 in that you must run *setEnv.cmd* (or *.sh*) before running the client applications to place the *titanejb-client.jar* file in the classpath properly.

Let's examine the interesting portions of these client programs individually.

Client_53.java

This client application demonstrates acquiring a reference to the Home interface for a bean using a remote interface to the same bean. The following code in the `main` method acquires a remote reference to the EJB and passes it to the other method, `getTheEJBHome`:

```
...
// Get a remote reference to the bean (EJB object).
TravelAgentRemote agent = home.create();
// Pass the remote reference to some method.
getTheEJBHome(agent);
...
```

The `getTheEJBHome` method then uses the remote reference to re-acquire a reference to the Home interface for the bean and display some metadata:

```
Object ref = agent.getEJBHome();
TravelAgentHomeRemote home = (TravelAgentHomeRemote)
   PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

// Do something useful with the home interface
EJBMetaData meta = home.getEJBMetaData();
System.out.println(meta.getHomeInterfaceClass().getName());
System.out.println(meta.getRemoteInterfaceClass().getName());
System.out.println(meta.isSession());
```

Client_54.java

This client application explores the importance of the primary key in identifying specific entity beans and demonstrates the equivalence of two different references to the same entity bean. The `testReferences` method creates a bean and acquires two references to the same bean:

```
System.out.println
  ("Creating Cabin 101 and retrieving additional reference by pk");
CabinRemote cabin_1 = home.create(new Integer(101));
Integer pk = (Integer)cabin_1.getPrimaryKey();
CabinRemote cabin_2 = home.findByPrimaryKey(pk);
```

Next the application performs a series of tests to prove the two references actually refer to the same entity bean:

```
System.out.println("Testing reference equivalence");
// We now have two remote references to the same bean -- Prove it!
cabin_1.setName("Keel Korner");
if (cabin_2.getName().equals("Keel Korner")) {
   System.out.println("Names match!");
}

// Test the isIdentical() function
if (cabin_1.isIdentical(cabin_2)) {
   System.out.println("cabin_1.isIdentical(cabin_2) returns true -
This is correct");
} else {
   System.out.println("cabin_1.isIdentical(cabin_2) returns false -
This is wrong!");
}
```

The second method of the Client_54 example program, `testSerialization`, uses output streams to write the primary key for the bean to a file, reads it back in again, and uses it to locate the desired entity bean once again. Because the primary key object is serializable and contains everything required to identify the unique bean it represents, the new reference does in fact refer to the same entity bean as the original reference.

Client_55.java

This client application explores the `Handle` and `HomeHandle` objects and demonstrates the process of serializing and deserializing these objects and re-acquiring references to the related `EJBObject` and `Home` interfaces. The code is well documented and fairly self-explanatory, so there will be no detailed examination in this text.

Examine and Run the Client JSP Pages

The three client applications are duplicated as JSP pages for your examination. Note that WebLogic uses the */wlserver6.1* directory as the location for the temporary files when the JSP pages write the serialized handles/keys to disk. This is an interesting by-product of the directory change (`cd ..\..`) that occurs in the *startWebLogic* script provided by WebLogic.

Exercise 5.3:
The Local Component Interfaces

This exercise demonstrates the use of local interfaces for the Cabin entity bean from within the TravelAgent stateless session bean and shows the changes required in the descriptor files to configure local interfaces and register multiple interfaces in the JNDI tree.

Download and Build the Example Programs

Download and extract the example directory *ex05_3* and examine the contents. This exercise is an extension of Exercise 4.2 with new "local" versions of the Cabin home and bean interfaces, so there are two new source files in the *com/titan/cabin* directory:

- *CabinLocal.java* – contains the definition of the local interface for the bean
- *CabinHomeLocal.java* – contains the local home interface for the bean

These files are identical to the versions described in the EJB book.

Use the *ant dist* task to build and deploy the EJB components.

Create the Required Database Objects

This exercise requires no additional database objects.

The `CABIN` table should contain all of the rows created during Exercise 4.1.

Examine the Standard EJB Descriptor File

The standard descriptor file *ejb-jar.xml* is nearly identical to the version from Exercise 4.2, with the minor changes outlined in the EJB book to configure the new local interfaces:

```
<enterprise-beans>
   <entity>
      <ejb-name>CabinEJB</ejb-name>
      <home>com.titan.cabin.CabinHomeRemote</home>
      <remote>com.titan.cabin.CabinRemote</remote>
      <local-home>com.titan.cabin.CabinHomeLocal</local-home>
      <local>com.titan.cabin.CabinLocal</local>
      <ejb-class>com.titan.cabin.CabinBean</ejb-class>
      ...
   </entity>
```

These elements inform the *ejbc* process and the container that the Cabin EJB will have both a local and remote set of interfaces available.

The next change is in the TravelAgent EJB descriptor, where the use of the local interfaces in the bean code requires a change to the reference descriptor:

```
 <session>
   <ejb-name>TravelAgentEJB</ejb-name>
   ...
   <ejb-local-ref>
       <!-- requires an entry in weblogic-ejb-jar.xml also -->
       <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
       <ejb-ref-type>Entity</ejb-ref-type>
       <local-home>com.titan.cabin.CabinHomeLocal</local-home>
       <local>com.titan.cabin.CabinLocal</local>
   </ejb-local-ref>
   ...
</session>
```

Application code within the TravelAgent bean will now use a JNDI ENC lookup with *ejb/CabinHomeLocal* to access the Cabin home interface.

Examine the WebLogic-Specific Files/Components

The *weblogic-ejb-jar.xml* file is very similar to the version from Exercise 4.2, with a few changes required to support the new local Cabin interfaces.

First, the Cabin EJB section in *weblogic-ejb-jar.xml* contains two elements defining JNDI names for the bean:

```
<weblogic-enterprise-bean>

  <ejb-name>CabinEJB</ejb-name>

  <entity-descriptor>
    ...
  </entity-descriptor>

  <jndi-name>CabinHomeRemote</jndi-name>
  <local-jndi-name>CabinHomeLocal</local-jndi-name>

</weblogic-enterprise-bean>
```

By specifying both JNDI names we've told WebLogic to register the Cabin home interface twice in the JNDI tree. The names cannot be identical, so we've followed the convention in the EJB book and appended `Remote` and `Local` to the appropriate JNDI names.

Next, the `<ejb-local-ref>` elements in the *ejb-jar.xml* file must have corresponding mapping elements in the *weblogic-ejb-jar.xml* file to map the `ejb/CabinHomeRemote` JNDI ENC lookup syntax to the actual JNDI name in the tree:

```
<weblogic-enterprise-bean>
  <ejb-name>TravelAgentEJB</ejb-name>
  ...
  <reference-descriptor>
      <ejb-local-reference-description>
          <!-- Matches entry in ejb-jar.xml file -->
          <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
          <jndi-name>CabinHomeLocal</jndi-name>
      </ejb-local-reference-description>
  </reference-descriptor>
  ...
</weblogic-enterprise-bean>
```

There is no change in the contents of the *weblogic-cmp-rdbms-jar.xml* file from the version in Exercise 4.2. Changing the interface definitions for a bean to include a local interface has no effect on the underlying persistence mechanism or CMP fields.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *titanapp* directory in the *ejbbook* domain. Use the *redeploy* task to touch the *REDEPLOY* file and force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Use the console to verify that `TravelAgentHomeRemote`, `CabinHomeRemote`, and `CabinHomeLocal` are registered in the JNDI tree.

Examine and Run the Client Applications

One client application is provided in the workbook download, *Client_58.java*.

The Client_58 example program is identical to the Client_43 application from Exercise 4.2, calling the `listCabins` method on the TravelAgent EJB to list the cabins having a specific deck level and bed count. The Cabin beans created in Exercise 4.1 must still exist in the database for this example to operate properly.

Set the environment variables and run Client_58 in the normal manner:

```
C:\work\ejbbook\ex05_3>setenv
...
C:\work\ejbbook\ex05_3>java com.titan.clients.Client_58
1,Master Suite,1
3,Suite 101,1
5,Suite 103,1
7,Suite 105,1
9,Suite 107,1
12,Suite 201,2
14,Suite 203,2
16,Suite 205,2
18,Suite 207,2
20,Suite 209,2
22,Suite 301,3
24,Suite 303,3
26,Suite 305,3
28,Suite 307,3
30,Suite 309,3
```

Examine and Run the Client JSP Pages

The *Client_58.jsp* page was copied to the proper *webapp* directory in the *titanapp* application during the *ant dist* task, so it should be available using the normal URL:

http://servername:7001/webapp/Client_58.jsp

Exercises for Chapter 6

Exercise 6.1: Basic Persistence in CMP 2.0

This exercise covers the creation of a new CMP entity bean, the Customer EJB. You will be starting over with a new *ejb-jar.xml* file and building up a large set of interrelated entity beans throughout this chapter and next.

Download and Build the Example Programs

Download and extract the example directory *ex06_1* and examine the contents. We're starting over in the creation of our EJB archive, so the Cabin EJB and TravelAgent EJB are absent.

Use the *ant dist* task to build and deploy the Customer EJB component and the related client program.

Create the Required Database Objects

This exercise requires a new table, `CUSTOMER`:

```
CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 HAS_GOOD_CREDIT BIT[1]
)
```

Make sure the `ID` column is a unique primary key to avoid duplicate records with the same identifier.

Examine the Standard EJB Descriptor File

The *ejb-jar.xml* file contains descriptor information for a single bean, the Customer EJB:

```
<entity>
    <ejb-name>CustomerEJB</ejb-name>
    <home>com.titan.customer.CustomerHomeRemote</home>
    <remote>com.titan.customer.CustomerRemote</remote>
    <ejb-class>com.titan.customer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
```

```
        <abstract-schema-name>Customer</abstract-schema-name>
        <cmp-field><field-name>id</field-name></cmp-field>
        <cmp-field><field-name>lastName</field-name></cmp-field>
        <cmp-field><field-name>firstName</field-name></cmp-field>
        <cmp-field><field-name>hasGoodCredit</field-name></cmp-field>
        <primkey-field>id</primkey-field>
        <security-identity><use-caller-identity/></security-identity>
    </entity>
```

Very similar to the first *ejb-jar.xml* file developed in Exercise 4-1, this version simply defines the `cmp` fields present in the Customer bean along with standard descriptor elements. The remaining section of the file defines the security and transactional attributes of the bean:

```
        <assembly-descriptor>
            <security-role>
                <role-name>Employees</role-name>
            </security-role>
            <method-permission>
                <role-name>Employees</role-name>
                <method>
                    <ejb-name>CustomerEJB</ejb-name>
                    <method-name>*</method-name>
                </method>
            </method-permission>
            <container-transaction>
                <method>
                    <ejb-name>CustomerEJB</ejb-name>
                    <method-name>*</method-name>
                </method>
                <trans-attribute>Required</trans-attribute>
            </container-transaction>
        </assembly-descriptor>
```

Note that the security role name has changed from `everyone` to `Employees` for this exercise and the remaining exercises in the workbook.

Examine the WebLogic-Specific Files/Components

The WebLogic-specific files for this exercise are the same two files always present for CMP 2.0 entity beans: *weblogic-ejb-jar.xml* and *weblogic-cmp-rdbms-jar.xml*.

♦ *weblogic-ejb-jar.xml*
  This file serves to extend the information in the standard *ejb-jar.xml* file, as required by the WebLogic container. Like the version in Exercise 4-1, it specifies the run-time caching limits, the name of the CMP-related descriptor file, and the name to use in the JNDI registration, and it maps users in the WebLogic realm to the role name used in the *ejb-jar.xml* file.

- *weblogic-cmp-rdbms-jar.xml*
  Like the file from Exercise 4-1, this version provides specific CMP mapping information needed to build the CMP persistence functions that read and write data from the database. The JDBC data source is specified, along with the specific table name in the database, and each attribute in the entity bean is mapped to a column in the database through a single `<field-map>` element:

```
...
<field-map>
  <cmp-field>lastName</cmp-field>
  <dbms-column>LAST_NAME</dbms-column>
</field-map>
...
```

Refer to the detailed discussion of this file in Exercise 4-1 for more details.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *titanapp* directory in the *ejbbook* domain. Use the *redeploy* task to touch the *REDEPLOY* file and force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Use the console to verify that `CustomerHomeRemote` is registered in the JNDI tree.

Examine and Run the Client Applications

A single client application (Client_61) is provided for this exercise. It is modeled after the example in the EJB book and requires careful use of command-line arguments for proper operation. Let's examine some of the code in the client program:

Client_61.java

First, we need some preliminary code to ensure the number of command-line arguments is a multiple of three for the application to work properly (the reason will become clear momentarily):

```
if (args.length<3 || args.length%3!=0) {
   System.out.println("Usage: java com.titan.clients.Client_61 <pk1>
<fname1> <lname1> ...");
   System.exit(-1);
}
```

Next there is some code to break up the command-line arguments and create a Customer bean for each set of three arguments (primary key, first name, and last name):

```
// obtain CustomerHome
Context jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeRemote");
CustomerHomeRemote home = (CustomerHomeRemote)
    PortableRemoteObject.narrow(obj, CustomerHomeRemote.class);

// create Customers
for(int i = 0; i < args.length; i++) {
    Integer primaryKey = new Integer(args[i]);
    String firstName = args[++i];
    String lastName = args[++i];
    CustomerRemote customer = home.create(primaryKey);
    customer.setFirstName(firstName);
    customer.setLastName(lastName);
    customer.setHasGoodCredit(true);
}
```

Please don't code this way at home. At the end of this loop there will be a set of persistent Customer beans in the database. The next section looks them up via primary key (again looping through the command-line arguments), reports their names, and deletes them again:

```
// find and remove Customers
for(int i = 0; i < args.length; i+=3) {
   Integer primaryKey = new Integer(args[i]);
   CustomerRemote customer =
                    home.findByPrimaryKey(primaryKey);
   String lastName = customer.getLastName( );
   String firstName = customer.getFirstName( );
   System.out.print(primaryKey+" = ");
   System.out.println(firstName+" "+lastName);

   // remove Customer
   customer.remove();
}
```

Run the client by supplying a set of the `Customer`'s `pk`, `firstname`, and `lastname` values on the command line as shown here:

```
java com.titan.clients.Client_61 777 Greg Nyberg 888 Bob Smith
```

The resulting output should be:

```
777 = Greg Nyberg
888 = Bob Smith
```

Because the beans are removed within the second loop, there will be no data in the database at the conclusion of the application. Feel free to modify the client application to eliminate the `remove` call in the loop and examine the `CUSTOMER` table at the conclusion of a test run.

Note that the `CUSTOMER` table definition limits the size of the first and last names to 20 characters each. If you attempt to create a customer with either name longer than 20 characters, the program will fail at the corresponding `set` method within the first loop processing the command-line arguments. The customer will be half saved, with a row created in the database missing one or both name values. In addition, because the second loop never gets executed, any other customers created in the loop using earlier sets of command-line arguments will be left in the database. Experiment with this client program to get some appreciation for the importance of doing all work on a bean in a transaction to avoid causing the container to save each create and set change independently, as this client does.

Please be sure the table is empty before proceeding to the next exercise.

Examine and Run the Client JSP Pages

There is no JSP page provided for the Client_61 application. Although it would not be difficult to build a simple HTML form to accept `pk`, `firstname`, and `lastname` values and perform the same activity in a JSP page, this is left as an exercise for the ambitious reader.

Exercise 6.2: Dependent Value Classes in CMP 2.0

This exercise demonstrates the use of a dependent value class to combine multiple cmp fields in a single serializable object which can be passed in and out of entity-bean business methods.

Download and Build the Example Programs

Download and extract the example directory *ex06_2* and examine the contents.

The new Name class is a serializable class encapsulating last-name and first-name values, providing a constructor for setting the values and accessors for retrieving the values:

```
public class Name implements java.io.Serializable {

    private String lastName;
    private String firstName;

    public Name(String lname, String fname){
        lastName = lname;
        firstName = fname;
    }
    public String getLastName() {
        return lastName;
    }
    public String getFirstName() {
        return firstName;
    }
}
```

The Name class is then used as a parameter for a setName method and as the return type for a getName method in the CustomerRemote interface and the CustomerBean class:

```
public interface CustomerRemote extends javax.ejb.EJBObject {

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

    public boolean getHasGoodCredit() throws RemoteException;
    public void setHasGoodCredit(boolean flag)
    throws RemoteException;
}
```

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    ...
    public Name getName() {
        Name name = new Name(getLastName(),getFirstName());
        return name;
    }
    public void setName(Name name) {
        setLastName(name.getLastName());
        setFirstName(name.getFirstName());
    }
    ...
}
```

As shown in this listing, the getName and setName methods provide the translation between the Name object and the entity bean's cmp fields.

Comparing the CustomerRemote interface in this exercise with the one in the previous exercise, you will note that the get and set methods for the lastName and firstName cmp fields are no longer available in the remote interface. The CustomerBean class still defines the abstract get and set functions for these cmp fields, but they will no longer be directly accessible from outside the bean itself.

Because this might be a little confusing, the following tables compare the CustomerRemote and CustomerBean definitions for these two exercises:

CustomerRemote.java

Exercise 6-1	Exercise 6-2
getLastName	
setLastName	getName
getFirstName	setName
setFirstName	
getHasGoodCredit	getHasGoodCredit
setHasGoodCredit	setHasGoodCredit

CustomerBean.java

Exercise 6-1	Exercise 6-2
	getName
	setName
getLastName	getLastName
setLastName	setLastName
getFirstName	getFirstName
setFirstName	setFirstName
getHasGoodCredit	getHasGoodCredit
setHasGoodCredit	setHasGoodCredit

Use the *ant dist* task to build and deploy the components and the related client program.

Create the Required Database Objects

This exercise requires no additional database objects. To avoid potential primary-key conflicts, be sure the `CUSTOMER` table is empty before running the client program.

Examine the Standard EJB Descriptor File

The *ejb-jar.xml* file is unchanged from the previous exercise. The addition of a dependent value object and new business methods has no impact on the standard descriptor file unless the new methods need special security or transaction descriptor elements

Examine the WebLogic-Specific Files/Components

The WebLogic-specific descriptor files have not changed since the previous exercise.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *titanapp* directory in the *ejbbook* domain. Use the *redeploy* task to touch the *REDEPLOY* file and force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Examine and Run the Client Applications

The client application for this exercise demonstrates the use of the `Name` dependent value class:

Client_62.java

```
...
// create example customer
System.out.println("Creating customer for use in example...");
Integer primaryKey = new Integer(1);
Name name = new Name("Monson", "Richard");
CustomerRemote customer = home.create(primaryKey);
customer.setName(name);

// find Customer by key
System.out.println("Getting name of customer using getName()..");
customer = home.findByPrimaryKey(primaryKey);
name = customer.getName();
System.out.print(primaryKey+" = ");
System.out.println(name.getFirstName( )+" "+name.getLastName( ));
```

```
// change customer's name
System.out.println("Setting name of customer using setName()..");
name = new Name("Monson-Haefel", "Richard");
customer.setName(name);
System.out.println("Getting name of customer using getName()..");
name = customer.getName();
System.out.print(primaryKey+" = ");
System.out.println(name.getFirstName( )+" "+name.getLastName( ));

// remove Customer to clean up
System.out.println("Removing customer...");
customer.remove();
...
```

This straightforward example highlights some implications of using dependent value classes:

- Only `Name` objects appear in the interface. There are no separate access methods for first name and last name.
- Because `Name` objects are immutable, a new one must be created whenever a change in either `cmp` field is desired.
- The `setName` method is an all-or-nothing mechanism; it always replaces both `cmp` field values.

By packaging multiple gets or sets in a single method, a dependent value object effectively reduces the number of individual remote bean invocations an external client makes. In this example the `Name` class reduces the number of calls only slightly. You begin to see the benefit when you imagine applying the same approach to a bean with many attributes. Without it, the overhead of the many get and set calls becomes prohibitive very quickly in real-world examples.

Caution: The client's `import` statement must include the `Name` class specifically. If it says simply `import com.titan.customer.*`, the customer `Name` class will conflict with the `Name` class in the *javax.naming* package.

Examine and Run the Client JSP Pages

The downloadable *Client_62.jsp* example program implements the same type of Customer EJB interaction as the Java client, with one wrinkle: The page includes a form displaying the last name and first name of Customer #1, if it exists, and allows you to modify the bean through the JSP page. Feel free to examine and modify this page to explore the various options available when a dependent value object like `Name is used` to make the round trip from entity bean to HTML form and back to entity bean.

One optional activity that you may wish to try is a demonstration of the concurrency hole that exists in entity beans used in web applications such as this one. Here's what happens: Two different browser windows can display the same `Customer` at the same time. The second window to submit will always overwrite updates made by the first window to submit, because the second window's `setName` method unconditionally overwrites both fields in the bean with its own values. As a result, the CMP-generated persistence code for the bean believes both fields should be updated in the database.

A simple test will show you the problem:

1. Open two separate browser windows in your desktop, both displaying *Client_62.jsp*.
2. In Window #1, change the **first** name to `FromWindow1` and leave the last name unchanged. Hit **Submit** and everything looks fine. Hit **Refresh** in Window #1 and convince yourself that the first change is saved in the database.
3. (Don't hit **Refresh** in Window #2 before performing this step or you will miss the point of the test.) In Window #2, change the **last** name to `FromWindow2`, leave the first name unchanged, and hit **Submit**. Notice that, even though it hasn't changed, the first name shown in Window #2 overrides the change made in Window #1.
4. Now hit **Refresh** on Window #1 and verify that the change you made there has vanished without a trace.

This effect occurs regardless of application-server locking schemes or database isolation levels, because it is not related to transactions or bean-instance sharing, but is simply a by-product of the all-or-nothing set methods and the "stale" data inherent in an HTML form.

In a client-server application or simple JDBC-based web application this concurrency hole is typically plugged through the use of "predicated updates" or time-stamp checking during the write operation, to ensure that no other application or process has updated the database row since the data was read and displayed to the user. If this condition is sensed, the second window receives an error message. Typically, it then refreshes the form with updated data from the bean (i.e., from the database) and allows the user to retry the submit.

Predicated updates for CMP beans are not yet available in WebLogic server, but unofficial comments indicate they are in the plans for future releases.

When you are done with the page or client application, make sure the `CUSTOMER` table is empty; delete the row using the *ViewDB.jsp* or other convenient mechanism.

Exercise 6.3: A Simple Relationship in CMP 2.0

This exercise demonstrates the use of a dependent object to pass data from an entity bean with a local interface back to a remote client. You will introduce a new EJB, `Address`, in the application you are building. The exercise also introduces the concept of a relationship `cmp` field in preparation for a detailed examination of relationships in the next chapter of the EJB book.

Download and Build the Example Programs

Download and extract the example directory *ex06_3* and examine the contents.

The Address EJB introduced in this exercise is a fairly simple bean with four string attributes and a primary key field. As described in the EJB book, Address is configured to have local rather than remote interfaces. Three source files are still required to define the bean code, but the interface names have `Local` rather than `Remote` where appropriate:

CustomerBean.java	*AddressBean.java*	Bean class
CustomerHomeRemote.java	*AddressHomeLocal.java*	Home interface
CustomerRemote.java	*AddressLocal.java*	Local and remote interfaces

This particular naming convention is not dictated by the specification, but is a reasonable way to differentiate between the two types of interfaces. Before the advent of local interfaces, the `Remote` portion of the name was typically absent. Note that a single EJB can have both sets of interfaces.

The *AddressBean.java* file defines a single `create` method for the bean:

AddressBean.java

```
public abstract class AddressBean implements javax.ejb.EntityBean {

    public Integer ejbCreateAddress
                         (String street, String city,
                          String state,  String zip )
    {
        setStreet(street);
        setCity(city);
        setState(state);
        setZip(zip);
        return null;
    }
    ...
}
```

Notice that the bean's `create` method does not include its primary key in the list of parameters. Contrast this with the `create` method in the `Customer` bean:

```
    public Integer ejbCreate(Integer id){
          this.setId(id);
          return null;
    }
```

The Address EJB will demonstrate a new feature of the WebLogic 6.1 server platform, automatic key generation using a sequence table. This feature will be discussed when you examine the WebLogic-specific files and components.

The *AddressLocal.java* interface defines the `get` and `set` methods available:

AddressLocal.java

```
public interface AddressLocal extends javax.ejb.EJBLocalObject {
    public String getStreet();
    public void setStreet(String street);
    public String getCity();
    public void setCity(String city);
    public String getState();
    public void setState(String state);
    public String getZip();
    public void setZip(String zip);
}
```

Note that `throws RemoteException` is absent from all of the method declarations. This is a local interface; invoking these methods on the bean does not involve RMI communication, so they cannot throw `RemoteException`s.

Finally, the *AddressHomeLocal.java* file defines the `Home` interface for the bean:

AddressHomeLocal.java

```
public interface AddressHomeLocal extends javax.ejb.EJBLocalHome
{
    public AddressLocal createAddress(String street, String city,
                                  String state,  String zip )
                      throws javax.ejb.CreateException;

    public AddressLocal findByPrimaryKey(Integer primaryKey)
    throws javax.ejb.FinderException;
}
```

Note that the `AddressHomeLocal` interface returns local references as expected, and that these methods will not throw any `RemoteException`s because the communication with the home interface does not involve RMI calls.

The next class to examine is the dependent-object class, `AddressDO`. This class is very similar to the `Name` class from the previous exercise, having a constructor to set private data members and `get` methods to retrieve them. Note that `AddressDO` implements `Serializable`, so it can be passed to remote clients via RMI communication.

The Customer EJB includes a unidirectional one-to-one relationship called `homeAddress` referring to an Address EJB. Chapter 7 in the EJB book covers all of the different relationship types and will explain these terms in detail. For now, just examine the changes made to the Customer EJB.

First, `CustomerBean` is modified to include the new relationship, `homeAddress`, as a set of abstract `get` and `set` methods, much as any new attribute would be added:

```
    public abstract AddressLocal getHomeAddress();
    public abstract void setHomeAddress(AddressLocal address);
```

If the intention was to expose this relationship to outside callers, these same methods would appear in the `CustomerRemote` interface file. For this exercise the need is not to expose the actual relationship `get` and `set` methods, but instead to provide methods on the remote interface that accept and return copies of the `AddressDO` value object:

CustomerRemote.java

```
public interface CustomerRemote extends javax.ejb.EJBObject {

    public void setAddress(String street, String city,
                           String state, String zip)
    throws RemoteException, CreateException, NamingException;

    public void setAddress(AddressDO address)
    throws RemoteException, CreateException, NamingException;
    public AddressDO getAddress() throws RemoteException;

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

    public boolean getHasGoodCredit() throws RemoteException;
    public void setHasGoodCredit(boolean flag)
    throws RemoteException;

}
```

The new interface methods are in bold in the listing above. Note that the actual EJB interface, `AddressLocal`, does not appear in these methods – they accept or return simple Java types or `AddressDO` objects only.

The code implementing these new methods will not be created by the CMP generation tool, so it is up to you to define these methods in the `CustomerBean` class:

CustomerBean.java

```
public AddressDO getAddress() {

    AddressLocal addrLocal = this.getHomeAddress();

    String street = addrLocal.getStreet();
    String city = addrLocal.getCity();
    String state = addrLocal.getState();
    String zip = addrLocal.getZip();
    AddressDO addrValue = new AddressDO(street,city,state,zip);

    return addrValue;
}
```

The `getAddress` method uses the CMP-created `getHomeAddress` relationship method to obtain a local reference to the Address EJB linked to this customer. (How it knows which address to fetch is something we will cover in detail later). Accessor methods are called on this reference to obtain the values of the fields, and an `AddressDO` object is constructed for return to the caller.

```
public void setAddress(AddressDO addrValue)
    throws CreateException, NamingException {

    String street = addrValue.getStreet();
    String city = addrValue.getCity();
    String state = addrValue.getState();
    String zip = addrValue.getZip();

    setAddress(street,city,state,zip);
}
```

The `setAddress` method that accepts an `AddressDO` object obtains from it the four strings it needs, and simply calls the overloaded version of the method presented below:

```
public void setAddress(String street, String city, String state,
String zip)
    throws CreateException, NamingException {

    AddressLocal addr = this.getHomeAddress( );

    if (addr == null) {
        // Customer doesn't have an addr yet. Create a new one.
        InitialContext cntx = new InitialContext( );
        AddressHomeLocal addrHome =
              (AddressHomeLocal)cntx.lookup("AddressHomeLocal");

        addr = addrHome.createAddress(street, city, state, zip);
        this.setHomeAddress(addr);
    } else {
       // Customer already has an address. Change its fields
       addr.setStreet(street);
       addr.setCity(city);
       addr.setState(state);
       addr.setZip(zip);
    }
}
```

This method is a little more complicated than you might expect because it must handle the case where a Customer is told to set its Address to be equal to a set of values and it does not yet have a related Address EJB. The basic result of this method, whichever branch is taken in the code, is that the Customer will be associated with an Address EJB through the `homeAddress` relationship field that has the specified four strings as field values.

Use the *ant dist* task to build and deploy the components and the related client programs.

Create the Required Database Objects

The Address EJB requires the following table in the database:

```
CREATE TABLE ADDRESS
(
 ID INT PRIMARY KEY,
 STREET CHAR(40),
 CITY CHAR(20),
 STATE CHAR(2),
 ZIP CHAR(10)
)
```

The relationship between the Customer EJB and the Address EJB must be reflected in the database in some manner to make it persistent. There are at least three different ways:

1. Adding a CUSTOMER_ID column to the ADDRESS table (classic approach of parent key in child table).
2. Adding an ADDRESS_ID column to the CUSTOMER table (child key in parent table).
3. Introduction of a LINK table containing pairs of CUSTOMER_ID and ADDRESS_ID values (required for many-to-many relationships, may be used for all relationships).

Chapter 7 will demonstrate the first and third options. Option 2 was chosen for this exercise to demonstrate the CMP engine's ability to make a relationship persistent in this manner.

Alter the CUSTOMER table to match the new schema:

```
CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 HAS_GOOD_CREDIT BIT(1),
 ADDRESS_ID INT
)
```

Finally, the Address EJB will be making use of an automatic key-generation facility available in WebLogic CMP. The required elements in *weblogic-cmp-rdbms-jar.xml* file will be described momentarily. The automatic key generation requires a table having the following schema:

```
CREATE TABLE ADDRESS_SEQUENCE
(
 SEQUENCE INT
}
```

Create this table and populate it with a single row having a value of 100 in the SEQUENCE column. As Address beans are created by the container, the value in the table will increase to reflect the next available primary key value for the bean. Access to this table uses the SERIALIZABLE isolation level within WebLogic. Caching of values is also available.

Before proceeding, double-check that you have made the required changes to the database:

- New ADDRESS table
- ADDRESS_ID added to CUSTOMER table
- New ADDRESS_SEQUENCE table containing a single row

All tables other than ADDRESS_SEQUENCE should be empty before proceeding.

Examine the Standard EJB Descriptor File

You will examine the relationship elements in the *ejb-jar.xml* file in detail in the next exercise. At this point, examine the non-relationship portions of the descriptor:

```
<entity>
  <ejb-name>CustomerEJB</ejb-name>
  <home>com.titan.customer.CustomerHomeRemote</home>
  <remote>com.titan.customer.CustomerRemote</remote>
  <ejb-class>com.titan.customer.CustomerBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Customer</abstract-schema-name>
  <cmp-field><field-name>id</field-name></cmp-field>
  <cmp-field><field-name>lastName</field-name></cmp-field>
  <cmp-field><field-name>firstName</field-name></cmp-field>
  <cmp-field><field-name>hasGoodCredit</field-name></cmp-field>
  <primkey-field>id</primkey-field>
  <security-identity><use-caller-identity/></security-identity>
</entity>
```

Note that the `<entity>` elements for the Customer EJB did not change with the introduction of the new relationship, despite the addition of the new `getHomeAddress` and `setHomeAddress` methods in the bean. Also note that we chose to not expose the `ADDRESS_ID` as a `cmp` field in the bean, although we could have. For this reason there is no `<cmp-field>` element related to the `ADDRESS_ID` relationship (and no `getAddressId` or `setAddressId` methods in the bean or the remote interface).

The Address EJB `<entity>` elements are fairly straightforward, defining the attributes of the bean:

```
<entity>
  <ejb-name>AddressEJB</ejb-name>
  <local-home>com.titan.address.AddressHomeLocal</local-home>
  <local>com.titan.address.AddressLocal</local>
  <ejb-class>com.titan.address.AddressBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
```

```
    <abstract-schema-name>Address</abstract-schema-name>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>street</field-name></cmp-field>
    <cmp-field><field-name>city</field-name></cmp-field>
    <cmp-field><field-name>state</field-name></cmp-field>
    <cmp-field><field-name>zip</field-name></cmp-field>
    <primkey-field>id</primkey-field>
    <security-identity><use-caller-identity/></security-identity>
</entity>
```

Skip the relationship sections of the file. The final change is the addition of Address EJB elements to the transaction and security sections of the `<assembly-descriptor>`:

```
<assembly-descriptor>
    <security-role>
        <role-name>Employees</role-name>
    </security-role>
    <method-permission>
        <role-name>Employees</role-name>
        <method>
            <ejb-name>CustomerEJB</ejb-name>
            <method-name>*</method-name>
        </method>
        <method>
            <ejb-name>AddressEJB</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <container-transaction>
        <method>
            <ejb-name>CustomerEJB</ejb-name>
            <method-name>*</method-name>
        </method>
        <method>
            <ejb-name>AddressEJB</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

Examine the WebLogic-Specific Files/Components

The *weblogic-ejb-jar.xml* file includes information for both the CustomerEJB and the AddressEJB in this exercise, but introduces no new elements or sections as a result of the relationship between the beans. The only difference from previous exercises is the change in the JNDI name element tag for the `Address` home interface:

```
      <local-jndi-name>AddressHomeLocal</local-jndi-name>
```

Because the `Home` interface for the `Address` is local, the tag is `<local-jndi-name>` rather than `<jndi-name>`.

The *weblogic-cmp-rdbms-jar.xml* descriptor file contains a number of new sections and elements in this exercise. A detailed examination of the relationship elements will wait until the next exercise, but there are some other changes to observe and examine.

The file contains a section mapping the `Address` `<cmp-field>` attributes from the *ejb-jar.xml* file to the database columns in the `ADDRESS` table, in addition to a new section related to the automatic key generation used for primary key values in this bean:

```
<weblogic-rdbms-bean>
    <ejb-name>AddressEJB</ejb-name>
    <data-source-name>titan-dataSource</data-source-name>
    <table-name>ADDRESS</table-name>
    <field-map>
      <cmp-field>id</cmp-field>
      <dbms-column>ID</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>street</cmp-field>
      <dbms-column>STREET</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>city</cmp-field>
      <dbms-column>CITY</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>state</cmp-field>
      <dbms-column>STATE</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>zip</cmp-field>
      <dbms-column>ZIP</dbms-column>
    </field-map>
    <!-- Automatically generate the value of ID in the database on
inserts using sequence table -->
    <automatic-key-generation>
        <generator-type>NAMED_SEQUENCE_TABLE</generator-type>
        <generator-name>ADDRESS_SEQUENCE</generator-name>
        <key-cache-size>10</key-cache-size>
    </automatic-key-generation>
</weblogic-rdbms-bean>
```

The <automatic-key-generation> section within the <weblogic-rdbms-bean> element enables this feature and provides configuration information to the CMP code-generation tool. In this example we are using the NAMED_SEQUENCE_TABLE approach and supplying the name of the table in the database (ADDRESS_SEQUENCE), containing a single column (SEQUENCE) and a single row with an integer value. The <key-cache-size> element defines the number of keys to cache in memory without requiring a database hit to update the sequence.

This key-generation scheme is only one of the options available, but it will work with any JDBC-compliant database because it uses a table to contain the current sequence value. Other key-generation techniques are available in WebLogic 6.1 for specific database platforms.

If the target database is Oracle, for example, the primary key can be generated using an Oracle Sequence by specifying the name of the sequence:

```
<automatic-key-generation>
    <generator-type>ORACLE</generator-type>
    <generator-name>ADDRESS_SEQ</generator-name>
    <key-cache-size>10</key-cache-size>
</automatic-key-generation>
```

According to the on-line documentation, the sequence should have the same increment in the database as the <key-cache-size> increment in the descriptor file.

If the target database is MS SQL Server, the primary key column in the table can use the IDENTITY data type, which automatically sets the value when a row is inserted in the table. To inform the WebLogic CMP engine that this table will be using this approach, use the following elements to configure the key-generation facility:

```
<automatic-key-generation>
    <generator-type>SQL_SERVER</generator-type>
</automatic-key-generation>
```

When a bean using this key-generation approach is created, SQL Server will assign the primary key automatically during the database insert. WebLogic will obtain the value of the key and use it as the primary key for subsequent operations.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *titanapp* directory in the *ejbbook* domain. Use the *redeploy* task to force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Examine and Run the Client Applications

A single client Java application for this exercise demonstrates the creation of a Customer and the use of the public getAddress/setAddress interface on the Customer to manipulate the underlying Address EJB.

Client_63.java

```
// obtain CustomerHome
Context jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeRemote");
CustomerHomeRemote home = (CustomerHomeRemote)
javax.rmi.PortableRemoteObject.narrow(obj,CustomerHomeRemote.class);

System.out.println("Creating Customer 1..");
// create a Customer
Integer primaryKey = new Integer(1);
CustomerRemote customer = home.create(primaryKey);
```

The first few lines of the client program obtain a reference to the home interface for the Customer EJB and create a single bean with a primary key value of 1. At this point in the program a row has been created in the `CUSTOMER` table in the database with `NULL` values for first name and last name as well as a `NULL` in the `ADDRESS_ID` column. These intermediate database states are difficult to see during the execution of the client because they exist for only a short period of time, but the individual transactions created by each method call to the bean cause the states to be committed to the database after each method invocation. The JSP page version of the client includes dumps of the `CUSTOMER` and `ADDRESS` table at each intermediate point in the program.

```
// create an address data object
System.out.println("Creating AddressDO data object..");
AddressDO address = new AddressDO("1010 Colorado",
                                  "Austin", "TX", "78701");

// set address in Customer bean
System.out.println("Setting Address in Customer 1...");
customer.setAddress(address);
```

This section creates an `AddressDO` data object using its constructor, and calls the `setAddress` method on the `Customer` remote interface. Turn back to the code examined earlier in this exercise to review the logic within the `CustomerBean setAddress` method to create or update the Address EJB related to the `Customer`.

After this invocation there will be a new row in the `ADDRESS` table in the database, and the primary key of that row will be in the `ADDRESS_ID` column in the `CUSTOMER` table.

```
System.out.println("Acquiring Address do from Customer 1...");
address = customer.getAddress();

System.out.println("Customer 1 Address data: ");
System.out.println(address.getStreet( ));
System.out.println(address.getCity( )+","+
                        address.getState()+" "+
                        address.getZip());
```

The client code now retrieves the customer's address by calling the getAddress method on the Customer remote interface to obtain the AddressDO data object. Don't be confused by the fact that the client program uses the same variable address again. It is obtaining a new AddressDO object created by the getAddress method on the CustomerBean class, serialized, sent via RMI to the client, and deserialized for use in the client program.

```
// create a new address
System.out.println("Creating new AddressDO data object..");
address = new AddressDO("1600 Pennsylvania Avenue NW",
                                  "DC", "WA", "20500");

// change customer's address
System.out.println("Setting new Address in Customer 1...");
customer.setAddress(address);
```

This section creates a new AddressDO data object on the client and again calls the setAddress method on the Customer remote interface. Because the Customer is already linked to an Address EJB, the setAddress code in CustomerBean will modify the fields in that Address bean rather than creating a new one. After the setAddress invocation, the database row for the associated Address EJB will reflect the new values, but there should be no other changes to the database.

```
address = customer.getAddress();
System.out.println("Customer 1 Address data: ");
System.out.println(address.getStreet( ));
System.out.println(address.getCity( )+","+
                        address.getState()+" "+
                        address.getZip());

// remove Customer to clean up
System.out.println("Removing Customer 1...");
customer.remove();
```

The final section retrieves another copy of the AddressDO data object, reports its values, and removes the Customer bean. There is a problem with this code, however! It does not delete the Address row in the database when the parent Customer is removed. You will learn how to configure CMP beans for cascade deletion in Exercise 7.3, but until then you will need to clean up these rows manually after you run this client program.

Examine and Run the Client JSP Pages

The *Client_63.jsp* page follows exactly the steps outlined above, with the addition of data dumps for the CUSTOMER and ADDRESS tables after each significant step, to make visible the database changes produced by each step.

The CUSTOMER and ADDRESS tables must be empty before proceeding to the next exercise.

Exercises for Chapter 7

Exercise 7.1: Entity Relationships in CMP 2.0: Part 1

This exercise begins the construction of a complex set of interrelated entity beans defined throughout Chapter 7 of the EJB book. We will explore relationship-modeling features of CMP 2.0 in this exercise and the two that follow it.

Exercise 7.1 starts by defining a few simple relationships centered on the Customer EJB.

Download and Build the Example Programs

Download and extract the example directory *ex07_1* and examine the contents.

To simplify the example programs and allow more straightforward get and set functions for related entity beans, the Customer EJB has been converted. It now has local rather than remote interfaces. The following changes were required:

- `CustomerHomeRemote` became `CustomerHomeLocal`
- `CustomerRemote` became `CustomerLocal`
- `RemoteException` was removed from the `throws` clauses of interface methods
- Descriptor elements in *ejb-jar.xml* were changed to the "local" versions:

```
<ejb-name>CustomerEJB</ejb-name>
<home>com.titan.customer.CustomerHomeRemote</home>
<remote>com.titan.customer.CustomerRemote</remote>
<ejb-class>com.titan.customer.CustomerBean</ejb-class>
```

...became...

```
<ejb-name>CustomerEJB</ejb-name>
<local-home>com.titan.customer.CustomerHomeLocal</local-home>
<local>com.titan.customer.CustomerLocal</local>
<ejb-class>com.titan.customer.CustomerBean</ejb-class>
```

- The JNDI name element in weblogic-ejb-jar.xml was changed to the local version:

```
<jndi-name>CustomerHomeRemote</jndi-name>
```

...became...

```
<local-jndi-name>CustomerHomeLocal</local-jndi-name>
```

This change to "local" interfaces allows methods on the `CustomerLocal` interface to include local references as parameters or return types, something `CustomerRemote` methods cannot do. `CustomerLocal` methods can therefore accept and return references to other local entity beans such as Address, CreditCard, and Phone, which you will be creating in this exercise. It also removes the need for specialized data object classes (`AddressDO`) and overloaded set methods that accept primitive parameters instead of references to other beans. This streamlining simplifies the entity bean significantly and allows you to concentrate primarily on the relationship-modeling features in CMP 2.0 instead of spending your time creating more data objects and specialized get and set methods for them in the Customer bean.

The one downside to making the Customer EJB local is that client Java programs will no longer be able to acquire a remote reference to the bean and make RMI calls to it from outside the WebLogic server JVM. Designers often use session beans with remote interfaces as a "façade" for entity beans with local interfaces. Rather than introduce this complexity, we'll take advantage of the fact that servlets and JSP pages within the same .ear file can use local entity beans without restriction. All of the exercises in Chapter 7 will therefore use JSP pages rather than remote Java client applications.

The downloadable code includes a revised version of the original Cabin EJB from Exercise 4.1. It has been modified to use local rather than remote interfaces, and relationships with other beans are not present.

The Customer EJB in this exercise includes the following relationship definitions:

- `homeAddress` (a reference to an Address bean)
- `creditCard` (a reference to a CreditCard bean)
- `phoneNumbers` (a Collection of Phone beans)

The methods in the `CustomerLocal` interface and `CustomerBean` class which were added to support these relationships will be described when you examine the standard EJB descriptor file.

Use the *ant dist* task to build and deploy the components and the related client programs.

Create the Required Database Objects

You need two new tables to support the CreditCard and Phone EJBs introduced in this exercise:

```
CREATE TABLE CREDIT_CARD
(
 ID INT PRIMARY KEY,
 EXP_DATE DATE,
 NUMBER CHAR(20),
 NAME CHAR(40),
 ORGANIZATION CHAR(20),
 CUSTOMER_ID INT
}
```

```
CREATE TABLE PHONE
(
 ID INT PRIMARY KEY,
 NUMBER CHAR(20),
 TYPE INT,
 CUSTOMER_ID INT
}
```

Note that both of these tables contain `CUSTOMER_ID`, a foreign key related to the `ID` column in the `CUSTOMER` table.

➢ Important note for Oracle users only: These create scripts will fail in Oracle because they use the reserved word `NUMBER` as a column name. You must modify the `CREDIT_CARD` and

PHONE tables slightly to rename the NUMBER columns to CARD_NUMBER and PHONE_NUMBER, respectively. You must also edit weblogic-cmp-rdbms-jar.xml to reflect these column-name changes in the related <field-map> elements for the CreditCard and Phone EJBs -- in this exercise and all subsequent exercises.

You will be using automatic key generation for both of the new EJBs. Unless you are using the Oracle-specific or SQL Server-specific approach to key generation, this exercise also requires two new sequence tables:

```
CREATE TABLE CREDIT_CARD_SEQUENCE
(
 SEQUENCE INT
}

CREATE TABLE PHONE_SEQUENCE
(
 SEQUENCE INT
}
```

Create a single row in each of these tables containing an integer value in the SEQUENCE column.

All non-sequence tables in the database should be empty at the start of the exercise.

Examine the Standard EJB Descriptor File

The standard *ejb-jar.xml* descriptor file contains important information about the relationships between beans in the application. As described in the EJB book, each relationship is defined in a single <ejb-relation> element with two <ejb-relationship-role> elements, one for each "direction" in the relationship:

```
<ejb-relation>
    <ejb-relation-name>Customer-HomeAddress</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Customer-has-a-Address
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <relationship-role-source>
            <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>homeAddress</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Address-belongs-to-Customer
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <relationship-role-source>
            <ejb-name>AddressEJB</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
```

This relationship is an example of a uni-directional one-to-one relationship. It has a multiplicity of "one" on both ends of the relationship and a `<cmr-field-name>` defined on only one end of the relationship. The specific role names are not important – except to the extent they help document the role each EJB plays in the relationship. The names are also used to link these elements with corresponding elements in the WebLogic-specific descriptor file discussed in a moment.

The presence of a `<cmr-field>` in the Customer EJB side of the relationship requires the existence of a properly-named set of get and set functions in the `CustomerBean` and optionally in the `CustomerLocal` interface. For the Customer-HomeAddress relationship, you must define these in the `CustomerBean`:

```
    public abstract AddressLocal getHomeAddress();
    public abstract void setHomeAddress(AddressLocal address);
```

Corresponding method definitions should also be defined in the CustomerLocal interface:

```
    public AddressLocal getHomeAddress();
    public void setHomeAddress(AddressLocal address);
```

The Customer-CreditCard relationship is also defined in the *ejb-jar.xml* file as a one-to-one relationship. Note that in this relationship both sides have a `<cmr-field>` element defined, indicating that this is a bi-directional relationship. Both entity beans must therefore have get and set functions defined for this relationship, but otherwise it is very similar to the Customer-HomeAddress relationship.

The final relationship defined in the *ejb-jar.xml* is the Customer-Phones relationship:

```
<ejb-relation>
    <ejb-relation-name>Customer-Phones</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Customer-has-many-Phone-numbers
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <relationship-role-source>
            <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>phoneNumbers</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Phone-belongs-to-Customer
        </ejb-relationship-role-name>
        <multiplicity>many</multiplicity>
        <relationship-role-source>
            <ejb-name>PhoneEJB</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
```

The multiplicity elements brand this association as a one-to-many relationship, with one Customer bean potentially related to many Phone beans. In addition, the lack of a `<cmr-field>` element on the Phone EJB side of the relationship indicates that it is uni-directional. The Customer EJB requires a matching set of get and set functions for the `phoneNumbers` cmr field:

```
public abstract Collection getPhoneNumbers( );
public abstract void setPhoneNumbers(Collection phones);
```

Do not confuse the location and presence of cmr-related get and set functions and fields in entity beans with the physical database keys and foreign keys required to implement these relationships. They are not the same. For example, even though this relationship will eventually be implemented using a `CUSTOMER_ID` foreign key in the `PHONE` table, the Phone EJB is not required to expose this relationship via a `customer` cmr field. The container-specific descriptor file will provide the required mapping of this relationship to the correct foreign keys and primary keys.

A number of convenience methods were also created in the CustomerBean to help manipulate the contents of the phoneNumbers relationship:

```
public void addPhoneNumber(String number, byte type)
         throws NamingException, CreateException {

    InitialContext jndiEnc = new InitialContext( );
    PhoneHomeLocal phoneHome =
        (PhoneHomeLocal)(jndiEnc.lookup("PhoneHomeLocal"));

    PhoneLocal phone = phoneHome.create(number,type);

    Collection phoneNumbers = this.getPhoneNumbers( );
    phoneNumbers.add(phone);

}
```

The addPhoneNumber method accepts the desired phone number and type as parameters, creates a Phone EJB using these values, retrieves the current collection of Phone beans related to this Customer, and adds the new Phone bean to the collection.

Note that the result of the jndiEnc.lookup() call is simply cast to the desired class, which is valid for local interfaces.

```
public void updatePhoneNumber(String number,byte typeToUpdate) {

    Collection phoneNumbers = this.getPhoneNumbers( );
    Iterator iterator = phoneNumbers.iterator();
    while(iterator.hasNext()){
        PhoneLocal phone = (PhoneLocal)iterator.next();
        if (phone.getType() == typeToUpdate) {
            phone.setNumber(number);
            break;
        }
    }
}
```

The updatePhoneNumber method accepts the desired new phone number and the specific type of the phone number to change. This method retrieves the collection of Phone beans related to the Customer, searches the collection to find a bean with the desired type, and applies the new number to the matching bean.

```
public void removePhoneNumber(byte typeToRemove) {

    Collection phoneNumbers = this.getPhoneNumbers( );
    Iterator iterator = phoneNumbers.iterator();
    while(iterator.hasNext()){
       PhoneLocal phone = (PhoneLocal)iterator.next();
       if (phone.getType() == typeToRemove) {
            phoneNumbers.remove(phone);
            break;
       }
    }

}
```

The `removePhoneNumber` method searches the collection of related Phone beans for the specific type and removes the Phone bean from the collection. This operation unlinks the Phone bean from the Customer, but does not delete the row from the Phone table.

For the convenience of client applications, another method iterates through the collection and returns a collection of `String` objects representing the data in the related Phone beans.

Examine the WebLogic-Specific Files/Components

The database-mapping information for the relationships defined in the *ejb-jar.xml* file must be defined in the WebLogic-specific CMP descriptor file. Three relationships are defined in the *ejb--jar.xml* file, so there are also three relationship sections in the *weblogic-cmp-rdbms-jar.xml* file.

First, examine the mapping information for the Customer-HomeAddress relationship:

```
<weblogic-rdbms-relation>
    <relation-name>Customer-HomeAddress</relation-name>
    <weblogic-relationship-role>
        <relationship-role-name>Customer-has-a-Address
        </relationship-role-name>
        <column-map>
            <foreign-key-column>ADDRESS_ID</foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

Notice in this section:

- The `<relation-name>` element in this file matches the `<ejb-relation-name>` element from the *ejb-jar.xml* file.
- The `<relationship-role-name>` element in this file matches the `<ejb-relationship-role-name>` element from the *ejb-jar.xml* file.

- This descriptor file includes only one side of the relationship, not both, because this relationship was implemented with an ADDRESS_ID foreign key in the CUSTOMER table.

The second section provides mapping information for the Customer-CreditCard relationship:

```
<weblogic-rdbms-relation>
    <relation-name>Customer-CreditCard</relation-name>
    <weblogic-relationship-role>
        <relationship-role-name>CreditCard-belongs-to-Customer
        </relationship-role-name>
        <column-map>
            <foreign-key-column>CUSTOMER_ID</foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

As in the preceding section, the descriptor needs to specify only one side of the relationship to provide mapping information, the CreditCard side, because the CREDIT_CARD table contains the linking CUSTOMER_ID foreign key. Placing the parent key (Customer ID) in the child table (CREDIT_CARD) is the normal technique for one-to-one and one-to-many relationships.

Finally, the third section provides mapping information for the Customer-Phones relationship:

```
<weblogic-rdbms-relation>
    <relation-name>Customer-Phones</relation-name>
    <weblogic-relationship-role>
        <relationship-role-name>Phone-belongs-to-Customer
        </relationship-role-name>
        <column-map>
            <foreign-key-column>CUSTOMER_ID</foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

This section defines mapping information for the Phone side of the relationship alone because the relationship is implemented in the database using a CUSTOMER_ID foreign key in the PHONE table. Note how closely this section resembles the Customer-CreditCard section you examined earlier. Both relationships use parent keys in the child table for linking the beans, and the differences in multiplicity do not affect the contents of the WebLogic CMP descriptor file.

The *weblogic-ejb-jar.xml* descriptor file contains the typical elements for each EJB, defining the JNDI home name, bean cache information, and other WebLogic-specific information unrelated to the CMP process.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *titanapp* directory in the *ejbbook* domain. Use the *redeploy* task to force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Use the console to verify that `CustomerHomeLocal`, `AddressHomeLocal`, `CreditCardHomeLocal`, and `PhoneHomeLocal` are registered in the JNDI tree.

Examine and Run the Client JSP Pages

Three client JSP pages provided in the download demonstrate the Customer relationships created in this exercise. The *ant dist* task copied them to the *webapp* directory in the *titanapp* exploded *.ear* file, making them available using the standard URLs:

http://servername:7001/webapp/Client_71.jsp

Client_71.jsp

The Client_71 JSP page demonstrates the simple one-to-one relationship between the Customer bean and the CreditCard bean. First, it looks up the necessary home interfaces and creates a single Customer bean:

```
Context jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeLocal");
CustomerHomeLocal customerhome = (CustomerHomeLocal)
   PortableRemoteObject.narrow(obj, CustomerHomeLocal.class);

obj = jndiContext.lookup("CreditCardHomeLocal");
CreditCardHomeLocal cardhome = (CreditCardHomeLocal)
   PortableRemoteObject.narrow(obj, CreditCardHomeLocal.class);

out.print("<H2>Creating Customer 71</H2>");

Integer primaryKey = new Integer(71);
CustomerLocal customer = customerhome.create(primaryKey);
customer.setName( new Name("Smith","John") );
```

Next, it creates a single CreditCard bean:

```
Calendar now = Calendar.getInstance();
CreditCardLocal card = cardhome.create(now.getTime(),
                 "370000000000001", "John Smith", "O'Reilly");
```

Note that the `create` method for the CreditCard EJB does not require a primary key as a parameter. The primary key will be generated automatically by the WebLogic CMP engine using the `CREDIT_CARD_SEQUENCE` table.

Next, the JSP page calls the `setCreditCard` method on the Customer bean to link the two objects:

```
customer.setCreditCard(card);
```

Because this is a bi-directional one-to-one relationship, it could just as easily use the `setCustomer` method on the CreditCard bean to link the two objects, and achieve the same effect on the beans and in the database.

The next section tests the bi-directional nature of the relationship by traversing the relationship both ways and reporting information on the related object:

```
String cardname = customer.getCreditCard().getNameOnCard();
out.print("customer.getCreditCard().getNameOnCard()="
                                       +cardname+"<br>");

Name name = card.getCustomer().getName();
String custfullname = name.getFirstName()+" "+name.getLastName();
out.print("card.getCustomer().getName()="+custfullname+"<br>");
```

The references returned by traversing relationships such as this are identical to the references returned by looking up the bean using a finder in the local home interface. In other words, the following expression would evaluate as `true`:

```
card.getCustomer().isIdentical(
                   custhome.findByPrimaryKey(new Integer(71))
```

In a one-to-one relationship such as this, what happens to one side of the relationship always affects the other side. The last section of the page demonstrates this interdependence by unlinking the beans using the CreditCard side of the relationship and verifying that the Customer's `getCreditCard` method also reflects the loss of the relationship between the beans:

```
card.setCustomer(null);

CreditCardLocal newcardref = customer.getCreditCard();
if (newcardref == null) {
  out.print("Card is properly unlinked from customer bean<br>");
} else {
  out.print("Whoops, customer still thinks it has a card!<br>");
}
```

Notice that this code did not remove the CreditCard bean, so it still exists in the database but is no longer linked to a Customer.

Experiment with this JSP page and test additional scenarios to better understand one-to-one relationships of this type. You should edit the JSP page in the */ex07_1/jsp* directory and use the *ant dist* target to move the modified version to the *webapp* directory in the *ejbbook* domain. There is no need to reboot the server – the next time you hit the page the server will re-compile the JSP and present the new contents.

The next client JSP page expects Customer 71 to exist, so make sure it exists in the database before moving on.

Client_72.jsp

The *Client_72.jsp* example page illustrates many of the same concepts Client_71 does, but focuses on the Customer-HomeAddress relationship. This is a uni-directional relationship, so the Customer EJB has the cmr-field methods `getHomeAddress` and `setHomeAddress`, but the Address EJB does not have corresponding `setCustomer` and `getCustomer` methods as the CreditCard EJB did in the previous example.

The page creates an Address bean and attaches it to the Customer if it does not already have one, then proceeds to create a new Address and perform the `setHomeAddress` method again with the new address. It is important to recognize that this second call to `setHomeAddress` effectively orphans the first Address EJB; the Customer will now be linked to the second Address object only.

There is a big difference between

- getting the Address reference from the Customer and manipulating its cmp fields directly, and
- creating a new Address bean and calling the `setHomeAddress` method on the Customer.

For example, this code modifies the data within the Address bean already linked to the Customer, essentially changing only one column (`STREET`) in the `ADDRESS` table to reflect the new value:

```
AddressLocal addr = customer.getHomeAddress();
addr.setStreet("700 Main Street");
```

...whereas this code, taken from *Client_72.jsp*, creates a completely new Address EJB with the data passed to the `createAddress` constructor, inserts a new row in the `ADDRESS` table, and modifies the `ADDRESS_ID` column in the `CUSTOMER` table representing this Customer bean:

```
AddressLocal addr = addresshome.createAddress(
                "700 Main Street","St. Paul","MN","55302");
customer.setHomeAddress(addr);
```

➢ Bottom line: Don't call relationship set methods to modify the values in the related bean. Get a reference to the related bean and use its set methods to modify its cmp-field values directly.

The next example page does not require anything beyond the existence of the Customer 71 record in the database, so feel free to experiment with *Client_72.jsp* to learn more about the two techniques described above.

Client_73.jsp

The final client page in this exercise demonstrates the proper use of one-to-many relationships such as the Customer-Phones relationship. Being a uni-directional one-to-many relationship, the

Customer EJB has a cmr field called `phoneNumbers` with access methods `getPhoneNumbers` and `setPhoneNumbers` created by the CMP code-generation process.

Because the Customer EJB includes helper methods (`addPhoneNumber`, `updatePhoneNumber`, etc.), the client JSP page may use these methods rather than manipulating the phone collection directly. In other words, this simple code in the JSP page…

```
customer.addPhoneNumber("800-333-3333",(byte)2);
```

…eliminates the need for all of this code in the client program:

```
UserTransaction tran = (UserTransaction)
              jndiContext.lookup("java:comp/UserTransaction");
try {
    tran.begin();
    Collection phones = customers.getPhoneNumbers();
    PhoneLocal newphone = phonehome.create(
                             ("800-333-3333",(byte)2);
    phones.add(newphone);
    tran.commit();
}
catch (Exception e) {
    e.printStackTrace();
    tran.rollback();
}
```

Obviously, helper methods substantially reduce client code. Note that essentially the same code appears in the `addPhoneNumber` helper method itself, but that the helper method does not have any transaction-related code:

```
public void addPhoneNumber(String number, byte type)
         throws NamingException, CreateException {

    InitialContext jndiEnc = new InitialContext( );
    PhoneHomeLocal phoneHome =
          (PhoneHomeLocal)(jndiEnc.lookup("PhoneHomeLocal"));

    PhoneLocal phone = phoneHome.create(number,type);

    Collection phoneNumbers = this.getPhoneNumbers( );
    phoneNumbers.add(phone);

}
```

The EJB container will not allow manipulation of a relationship collection (including iteration through the collection) outside the context of a transaction. This restriction explains why the client-based version required explicit transaction creation and commitment, with all collection manipulation done within the transaction.

Why doesn't the helper method (addPhoneNumber) also require an explicit transaction? Because it uses declarative transactions, as specified by these elements in the ejb-jar.xml file:

```
<container-transaction>
    <method>
        <ejb-name>CustomerEJB</ejb-name>
        <method-name>*</method-name>
    </method>
    ...
    <trans-attribute>Required</trans-attribute>
</container-transaction>
```

We've declared that all methods in the CustomerEJB will require a transaction, so when the addPhoneNumber method is entered, the container automatically creates a transaction (unless one is already associated with the relevant thread) and commits it on exit from the method if the operation was successful (i.e., no exception was thrown).

Chapter 14 in the EJB book discusses transactions in more detail and explores the nuances of explicit versus declarative transactions. For now, the example code will take advantage of declarative transactions and helper methods to simplify greatly the code in our client JSP page.

The *Client_73* page performs the following operations:

1. Locates the Customer 71 EJB and reports the current contents of the phoneNumbers collection using the getPhoneList helper function
2. Adds a new Phone EJB to the collection using the addPhoneNumber helper method
3. Reports on the contents of the phone collection again
4. Adds an additional phone of a different "type" to the collection
5. Reports on the contents of the phone collection
6. Uses the updatePhoneNumber helper method to update the type=1 phone number in the collection to have a different "number" value
7. Reports on the contents of the phone collection
8. Uses the removePhoneNumber helper method to remove a phone number, by type, from the collection of related phones for this Customer
9. Reports on the contents of the phone collection
10. Displays the contents of the CUSTOMER and PHONE tables

Note that the removePhoneNumber method removes the link between the Customer and the Phone bean, but does not actually delete the bean from the database. We've orphaned yet another EJB, just like the orphaned Address beans from the previous client program.

We invite the ambitious reader to create an additional helper method on the Customer EJB that removes the bean from the collection (unlinking it) and also removes the bean itself (deletes the Phone from the database). The code would look something like this:

```
public void removePhoneNumberWithDelete(byte typeToRemove)
   throws javax.ejb.RemoveException {

   Collection phoneNumbers = this.getPhoneNumbers( );
   Iterator iterator = phoneNumbers.iterator();

   while(iterator.hasNext()){
      PhoneLocal phone = (PhoneLocal)iterator.next();
      if (phone.getType() == typeToRemove) {
              phoneNumbers.remove(phone);
              phone.remove();
              break;
      }

   }
}
```

The bold lines are the only difference from the existing `removePhoneNumber` helper method. Don't forget to modify the `CustomerLocal` interface to include this method declaration as well.

Modify the *Client_73.jsp* file to use this new helper method instead of `removePhoneNumber`, and verify that the program successfully deletes the phone number from the database as well as removing it from the `phoneNumbers` collection.

The non-sequence workbook tables should be empty before proceeding to next exercise.

Exercise 7.2: Entity Relationships in CMP 2.0: Part 2

This exercise demonstrates the remaining four entity-bean relationship types:

- Many-to-One Unidirectional (Cruise-Ship)
- One-to-Many Bidirectional (Cruise-Reservation)
- Many-to-Many Bidirectional (Customer-Reservation)
- Many-to-Many Unidirectional (Cabin-Reservation)

Download and Build the Example Programs

All entity beans participating in relationships must be in the same .jar file.

This exercises re-introduces the Cabin EJB, with a new relationship to the Reservation EJB. The Cabin EJB has also been converted to be a local bean in these exercises.

Create the Required Database Objects

The following tables must be present in the database for the example programs to operate:

```
CREATE TABLE SHIP
(
 ID INT PRIMARY KEY,
 NAME CHAR(30),
 TONNAGE DECIMAL (8,2)
)
```

```
CREATE TABLE CRUISE
(
 ID INT PRIMARY KEY,
 NAME CHAR(30),
 SHIP_ID INT
)
```

```
CREATE TABLE RESERVATION
(
 ID INT PRIMARY KEY,
 CRUISE_ID INT,
 AMOUNT_PAID DECIMAL (8,2),
 DATE_RESERVED DATE
)
```

```
CREATE TABLE
RESERVATION_CUSTOMER_LINK
(
 RESERVATION_ID INT,
 CUSTOMER_ID INT,
)
```

```
CREATE TABLE
RESERVATION_CABIN_LINK
(
 RESERVATION_ID INT,
 CABIN_ID INT,
)
```

In addition, the Cruise and Reservation beans will be using automatic sequencing for primary keys, so the following sequence tables should also be created and populated with a single row:

```
CREATE TABLE CRUISE_SEQUENCE
{
 SEQUENCE INT
}
```

```
CREATE TABLE RESERVATION_SEQUENCE
{
 SEQUENCE INT
}
```

Examine the Standard EJB Descriptor File

The standard *ejb-jar.xml* file defines the relationships between the beans for this exercise. Each relationship is defined in a single `<ejb-relation>` element with two `<ejb-relationship-role>` elements, one for each "direction" of the relationship.

Many-to-One Unidirectional (Cruise-Ship)

```
<ejb-relation>
    <ejb-relation-name>Cruise-Ship</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Cruise-has-a-Ship
        </ejb-relationship-role-name>
        <multiplicity>many</multiplicity>
        <relationship-role-source>
            <ejb-name>CruiseEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>ship</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Ship-has-many-Cruises
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <relationship-role-source>
            <ejb-name>ShipEJB</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
```

The Cruise-Ship relationship is a Many-to-One Unidirectional relationship because the multiplicity on the Cruise side of the relationship is “many” and only one side has a `<cmr-field>` element. Therefore, the Cruise bean must have methods defined for the `ship` attribute:

```
public abstract ShipLocal getShip();
public abstract void setShip(ShipLocal ship);
```

Corresponding method definitions are also placed in the `CruiseLocal` interface:

```
public ShipLocal getShip();
public void setShip(ShipLocal ship);
```

The Ship bean has no methods to allow access to the list of cruises related to a given ship because this relationship is unidirectional.

One-to-Many Bidirectional (Cruise-Reservation)

```
<ejb-relation>
    <ejb-relation-name>Cruise-Reservation</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Cruise-has-many-Reservations
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <relationship-role-source>
            <ejb-name>CruiseEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>reservations</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Reservation-has-a-Cruise
        </ejb-relationship-role-name>
        <multiplicity>many</multiplicity>
        <relationship-role-source>
            <ejb-name>ReservationEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>cruise</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

The Cruise-Reservation relationship defines <cmr-field> elements for both sides of the relationship, making it bidirectional. Because the multiplicity is one-to-many, the <cmr-field> element on the Cruise side is actually a collection of related Reservation objects.

The presence of <cmr-field> elements on both sides requires the existence of related get and set methods on both the Cruise and Reservation beans. The methods on the Cruise bean are:

```
public abstract void setReservations(Collection res);
public abstract Collection getReservations( );
```

The method definitions on the CruiseLocal interface are:

```
public void setReservations(Collection res);
public Collection getReservations( );
```

The methods on the Reservation bean are:

```
public abstract CruiseLocal getCruise();
public abstract void setCruise(CruiseLocal cruise);
```

And the related ReservationLocal interface method definitions are:

```
public CruiseLocal getCruise();
public void setCruise(CruiseLocal cruise);
```

Many-to-Many Bidirectional (Customer-Reservation)

```
<ejb-relation>
    <ejb-relation-name>Customer-Reservation</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Customer-has-many-Reservations
        </ejb-relationship-role-name>
        <multiplicity>many</multiplicity>
        <relationship-role-source>
            <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>reservations</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
```

```
      <ejb-relationship-role>
          <ejb-relationship-role-name>
              Reservation-has-many-Customers
          </ejb-relationship-role-name>
          <multiplicity>many</multiplicity>
          <relationship-role-source>
              <ejb-name>ReservationEJB</ejb-name>
          </relationship-role-source>
          <cmr-field>
              <cmr-field-name>customers</cmr-field-name>
              <cmr-field-type>java.util.Set</cmr-field-type>
          </cmr-field>
      </ejb-relationship-role>
  </ejb-relation>
```

Both sides of the relationship are defined with "many" multiplicity and have `<cmr-field>` elements, so this is a many-to-many bidirectional relationship. As described in the EJB book, the use of a `java.util.Set` field type, rather than a `Collection` field type for the customer's cmr field, indicates to the container that we do not want duplicate Customer beans in our collection.

The presence of `<cmr-field>` elements on both sides requires the existence of related get and set methods on both the Customer and Reservation beans. The methods on the Customer bean are:

```
public abstract Collection getReservations();
public abstract void setReservations(Collection reservations);
```

The method definitions on the `CustomerLocal` interface are:

```
public Collection getReservations();
public void setReservations(Collection reservations);
```

The methods on the Reservation bean are:

```
public abstract Set getCustomers( );
public abstract void setCustomers(Set customers);
```

And the related `ReservationLocal` interface method definitions are:

```
public Set getCustomers( );
public void setCustomers(Set customers);
```

Many-to-Many Unidirectional (Cabin-Reservation)

```
<ejb-relation>
    <ejb-relation-name>Cabin-Reservation</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Cabin-has-many-Reservations
        </ejb-relationship-role-name>
        <multiplicity>many</multiplicity>
        <relationship-role-source>
            <ejb-name>CabinEJB</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Reservation-has-many-Cabins
        </ejb-relationship-role-name>
        <multiplicity>many</multiplicity>
        <relationship-role-source>
            <ejb-name>ReservationEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>cabins</cmr-field-name>
            <cmr-field-type>java.util.Set</cmr-field-type>
    </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

Both sides of the relationship are defined with "many" multiplicity, but only the Reservation side of the relationship has a `<cmr-field>` element, making this a many-to-many unidirectional relationship. The Reservation bean therefore requires these get and set methods:

```
public abstract Set getCabins( );
public abstract void setCabins(Set cabins);
```

...And the related method definitions in the `ReservationLocal` interface are:

```
public Set getCabins( );
public void setCabins(Set customers);
```

The same warning applies in this exercise as in the last exercise: Do not confuse the location and presence of cmr-related get and set functions and `<cmr-field>` elements in bean definitions with the physical database schema required to implement these relationships. The WebLogic-specific descriptor file defines the physical mapping, while this standard *ejb-jar.xml* file defines only the logical relationships between the beans.

Examine the WebLogic-Specific Files/Components

The database-mapping information for the relationships defined in the *ejb-jar.xml* file must be defined in the WebLogic-specific CMP descriptor file. Four new relationships are defined in the *ejb-jar.xml* file, so there are also four new relationship sections in the *weblogic-cmp-rdbms-jar.xml* file.

Many-to-One Unidirectional (Cruise-Ship)

The Cruise-Ship relationship is implemented by a SHIP_ID foreign key in each row in the CRUISE table:

```
<weblogic-rdbms-relation>
    <relation-name>Cruise-Ship</relation-name>
    <weblogic-relationship-role>
     <relationship-role-name>Cruise-has-a-Ship
        </relationship-role-name>
     <column-map>
         <foreign-key-column>SHIP_ID</foreign-key-column>
         <key-column>ID</key-column>
     </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

One-to-Many Bidirectional (Cruise-Reservation)

The Cruise-Reservation relationship is implemented by a CRUISE_ID foreign key in each row in the RESERVATION table:

```
<weblogic-rdbms-relation>
    <relation-name>Cruise-Reservation</relation-name>
    <weblogic-relationship-role>
        <relationship-role-name>Reservation-has-a-Cruise
        </relationship-role-name>
        <column-map>
            <foreign-key-column>CRUISE_ID</foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

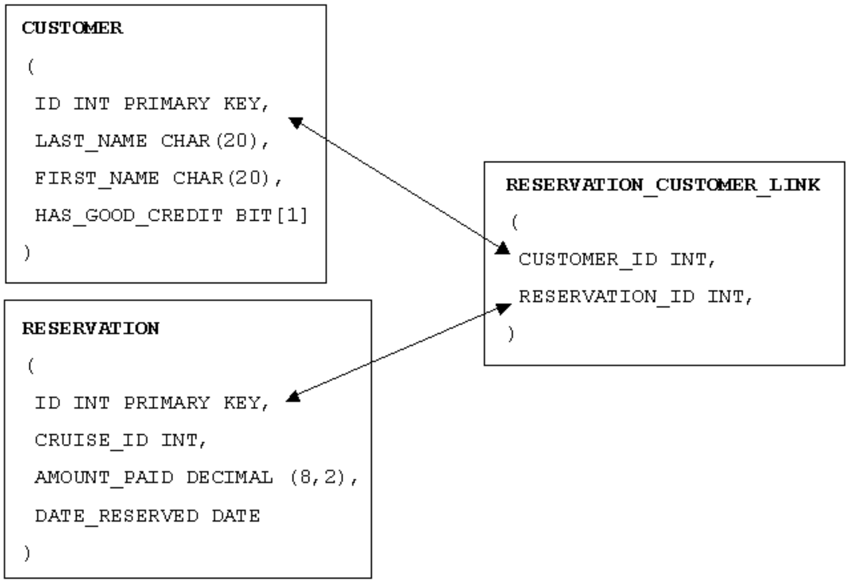
Many-to-Many Bidirectional (Customer-Reservation)

Many-to-Many relationships require a separate “link” table containing the key for each side of the relationship. The column names and name of the link table are defined in this section of the *weblogic-cmp-rdbms-jar.xml* descriptor:

```
<weblogic-rdbms-relation>
    <relation-name>Customer-Reservation</relation-name>
    <table-name>RESERVATION_CUSTOMER_LINK</table-name>
    <weblogic-relationship-role>
        <relationship-role-name>Customer-has-many-Reservations
        </relationship-role-name>
        <column-map>
            <foreign-key-column>CUSTOMER_ID</foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
    <weblogic-relationship-role>
        <relationship-role-name>Reservation-has-many-Customers
        </relationship-role-name>
        <column-map>
            <foreign-key-column>RESERVATION_ID
            </foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

The following diagram depicts how the `RESERVATION_CUSTOMER_LINK` table connects a specific `CUSTOMER` row and `RESERVATION` row in the database, thereby linking the beans these rows represent:

Figure 33:Link table used to implement Customer-Reservation relationship

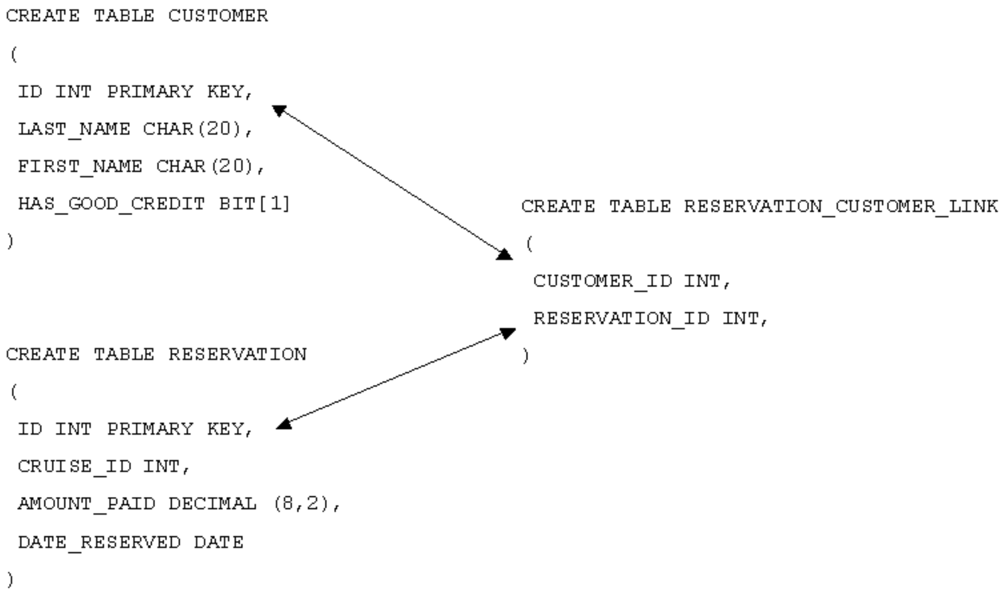


```
CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 HAS_GOOD_CREDIT BIT[1]
)

RESERVATION
(
 ID INT PRIMARY KEY,
 CRUISE_ID INT,
 AMOUNT_PAID DECIMAL (8,2),
 DATE_RESERVED DATE
)

RESERVATION_CUSTOMER_LINK
(
 CUSTOMER_ID INT,
 RESERVATION_ID INT,
)
```

Note that there is no primary key in the link table, nor is there any code in either bean to query or



```
CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 HAS_GOOD_CREDIT BIT[1]
)

CREATE TABLE RESERVATION_CUSTOMER_LINK
(
 CUSTOMER_ID INT,
 RESERVATION_ID INT,
)

CREATE TABLE RESERVATION
(
 ID INT PRIMARY KEY,
 CRUISE_ID INT,
 AMOUNT_PAID DECIMAL (8,2),
 DATE_RESERVED DATE
)
```

manipulate rows in this table explicitly. The CMP-related code implements the get and set methods defined on each side of the relationship and performs all database updates automatically whenever relationship change occurs.

Many-to-Many Unidirectional (Cabin-Reservation)

The Cabin-Reservation relationship is also a many-to-many relationship, requiring a separate "link" table containing references to both sides of the relationship. The fact that the relationship is unidirectional does not affect the physical mapping of the relationship:

```
<weblogic-rdbms-relation>
    <relation-name>Cabin-Reservation</relation-name>
    <table-name>RESERVATION_CABIN_LINK</table-name>
    <weblogic-relationship-role>
        <relationship-role-name>Cabin-has-many-Reservations
        </relationship-role-name>
        <column-map>
            <foreign-key-column>CABIN_ID</foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
    <weblogic-relationship-role>
        <relationship-role-name>Reservation-has-many-Cabins
        </relationship-role-name>
        <column-map>
            <foreign-key-column>RESERVATION_ID
            </foreign-key-column>
            <key-column>ID</key-column>
        </column-map>
    </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

Finally, the *weblogic-cmp-rdbms-jar.xml* file also defines the basic mapping of the new entity beans to their respective tables, and defines the automatic-sequence parameters for the Cruise and Reservation beans.

It is instructive at this point to step back and consider the total amount of descriptor information required to define the eight entity beans in this exercise and the various relationships between them. The total size of the *ejb-jar.xml* file at this point is approximately 440 lines and the *weblogic-cmp-rdbms-jar.xml* file is over 350 lines. While these counts include a fair amount of white space and boilerplate, it is not excessive, so they probably represent a reasonable size-per-bean expectation for EJB-based systems using CMP persistence.

For example, a system with 100 beans (not uncommon for a real-life system) might be expected to have an *ejb-jar.xml* file of 5,000-6,000 lines and a WebLogic-specific *weblogic-cmp-rdbms-jar.xml* file of perhaps 4,000-5,000 lines, depending on the number of relationships. Contrast this with the amount of custom Java code required to implement all of the persistence logic for the beans and their relationships, and the advantage of CMP 2.0 becomes much more evident.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task to force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Use the console to verify that all of the new beans are properly deployed by checking for their home interfaces in the JNDI tree.

Examine and Run the Client JSP Pages

This exercise includes a large number of client JSP pages which demonstrate the relationships between the entity beans. The *ant dist* task copied them to the *webapp* directory in the *titanapp* exploded .ear file, making them available using the standard URLs:

http://servername:7001/webapp/Client_75.jsp

The following list summarizes the purpose of the example JSP pages; more detail follows:

- *Client_75.jsp* – Example demonstrating the Cruise/Ship relationship, including the sharing of a reference (see EJB Book Figure 7-12)
- *Client_76a.jsp* – Example demonstrating the Cruise/Reservation relationship, including the use of `set` to modify the reservations associated with a Cruise (see Figure 7-14)
- *Client_76b.jsp* – Example demonstrating the Cruise/Reservation relationship, including the use of `addAll` to modify the Reservations associated with a Cruise (Figure 7-15)
- *Client_77a.jsp* – Example demonstrating the Customer/Reservation relationship, including the use of `addAll` to modify the Customers for a Reservation (Figure 7-17)

- *Client_77b.jsp* – Example demonstrating the Customer/Reservation relationship, including the use of `setCustomers` to modify the Customers for a Reservation (Figure 7-18)
- *Client_77c.jsp* – Example demonstrating the Cabin/Reservation relationship, including the removal of Cabins from the Reservation using an iterator (Figure 7-20)
- *Client_77d.jsp* – Example demonstrating the Cabin/Reservation relationship, including the use of `setCabins` to modify the Cabins for a Reservation

Client_75.jsp

This example demonstrates the Cruise/Ship relationship, including the sharing of a reference (see EJB Book Figure 7-12).

First, the example creates two Ship beans and an array of six Cruise beans. The first three Cruise beans are linked to Ship A and the last three are linked with Ship B. This section of code demonstrates some properties of relationship cmr fields:

```
ShipLocal newship = cruises[4].getShip();
cruises[1].setShip(newship);
```

The fourth Cruise bean is asked for a reference to its related Ship bean (which should be Ship B) and this reference is used in a call to the `setShip` method on the first Cruise bean. This operation links the first Cruise bean to the Ship B bean rather than the Ship A bean, as the output will show.

Note that you cannot run this example twice without generating a duplicate-key problem on the `SHIP` table because the program attempts to create the same Ship beans during each run. To run the example again, first delete all rows in the `SHIP` and `CRUISE` tables.

Delete all rows in the `SHIP` and `CRUISE` tables before proceeding to the next example.

Client_76a.jsp

This example demonstrates the Cruise/Reservation relationship, including the use of set methods to modify the reservations associated with a Cruise (see Figure 7-14).

First, the example creates two Cruise beans (Cruise A and B) and six Reservation beans. Note that the `create` method for a Reservation bean requires a Cruise bean as an input parameter, thereby providing a mechanism to link a Reservation to a Cruise from the start. The first three Reservation beans are linked to Cruise A, the last three to Cruise B.

The next section of code demonstrates the effect of calling `setReservations` on a Cruise bean:

```
Collection a_reservations = cruiseA.getReservations();
cruiseB.setReservations( a_reservations );
```

Cruise A's `getReservations` method provides a collection of references to the reservations associated with Cruise A. This collection is then passed to Cruise B's `setReservations` method.

The net effect, as shown in Figure 7-14 in the EJB book, is that the Reservation beans originally linked to Cruise B are orphaned (they are no longer linked to any Cruise).

This example may be run more than once without errors, because the Cruise and Reservation beans are assigned new unique primary keys automatically when created. To avoid confusion when examining the tables, however, be sure the `CRUISE` and `RESERVATION` tables are empty before proceeding to the next example.

Client_76b.jsp

This example demonstrates the Cruise/Reservation relationship, including the use of `addAll` to modify the Reservations associated with a Cruise (Figure 7-15).

This example creates the same set of two Cruise and six Reservation beans, with the same relationships between them. The next section of code demonstrates the effect of using `addAll` instead of a `set` method:

```
UserTransaction tran =
(UserTransaction)jndiContext.lookup("java:comp/UserTransaction");
try {
    tran.begin();
    Collection a_reservations = cruiseA.getReservations();
    Collection b_reservations = cruiseB.getReservations();
    b_reservations.addAll(a_reservations);
    tran.commit();
}
catch (Exception e) {
    e.printStackTrace();
    tran.rollback();
}
```

Manipulating the contents of `Collection` objects representing the bean relationships must be done in the context of a UserTransaction, either created explicitly (as we've done here) or within a transaction created by the container automatically (using declarative transactions). We covered this requirement in the previous exercise.

Within the transaction block, the three lines that actually perform the manipulation are:

```
    Collection a_reservations = cruiseA.getReservations();
    Collection b_reservations = cruiseB.getReservations();
    b_reservations.addAll(a_reservations);
```

This code obtains collections containing the Reservation beans associated with each Cruise bean. It is important to recognize that the objects returned by relationship get methods such as `getReservations` may look like simple Java `Collection` objects, but they are actually container-generated objects representing the relationships. These objects implement the `Collection` interface, and any manipulation of the objects using methods available in the `Collection` interface causes corresponding changes to the underlying relationships and database tables.

In this example, we use the addAll method in the Collection interface to add the contents of the a_reservations collection to the current contents of the b_reservations collection. Because the multiplicity on the Reservation side of the relationship is "one," there can be only one Cruise associated with each Reservation bean. Therefore, the addAll invocation has the effect of linking all of the Reservation beans only to Cruise B, leaving Cruise A with none (see Figure 7-15 in the EJB book).

Automatic key creation makes it possible to run this example more than once without error. The CRUISE and RESERVATION tables, however, should be empty before proceeding to the next example, to avoid confusion when examining the tables.

Client_77a.jsp

This example demonstrates the Customer/Reservation relationship including the use of addAll to modify the Customers for a Reservation (Figure 7-17).

Six Customer beans and two Reservation beans are created, with each Reservation bean related to three Customer beans. In a manner very similar to *Client_76b.jsp*, we manipulate the relationship between the beans using collection (Set) objects and the addAll method:

```
Set customers_a = reservations[1].getCustomers();
Set customers_b = reservations[2].getCustomers();
customers_a.addAll(customers_b);
```

As in the previous client program, addAll adds the contents of one collection to the other, essentially linking all six Customer beans to the first Reservation bean. The difference in this example, however, is that both sides of the relationship have a multiplicity of "many," so the act of linking the Customer beans associated with the second Reservation to the first Reservation does not affect their relationships with the second Reservation bean. They are simply linked to both Reservation beans after this operation, as shown in Figure 7-17 in the EJB book.

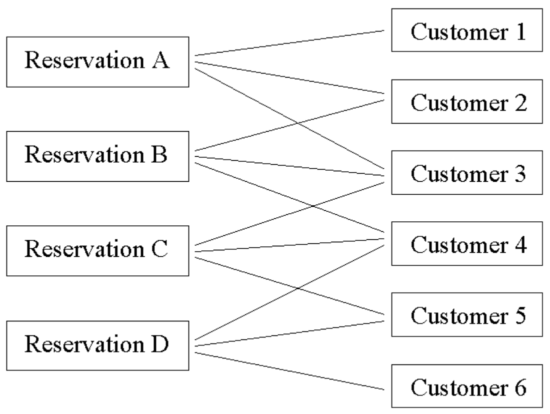
Because this example creates Customer objects with specific primary keys (1-6) you may not run it more than once unless you clear out the tables first. In addition, the CUSTOMER, RESERVATION, and RESERVATION_CUSTOMER_LINK tables should be empty before proceeding to the next example.

Client_77b.jsp

This example demonstrates the Customer/Reservation relationship, including the use of setCustomers to modify the Customers for a Reservation (Figure 7-18).

Six Customer beans are again created along with four Reservation beans. The relationships between the Customer beans and the Reservation beans is more complex in this example, with each Reservation linked to a different set of three Customer beans:

Figure 34: Initial relationships in Client_77b example



The next section manipulates these relationships using the `setCustomers` method on one of the Reservation beans:

```
try {
   tran.begin();
   Set customers_a = reservations[1].getCustomers();
   reservations[4].setCustomers(customers_a);
   tran.commit();
}
catch (Exception e) {
   e.printStackTrace();
   tran.rollback();
}
```

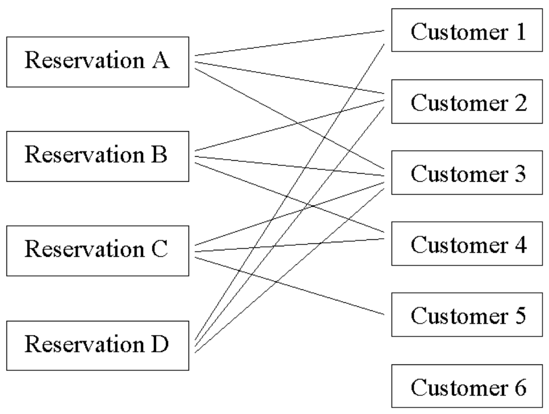
We explained the need for a transaction in the previous examples. The important lines are inside the transaction block:

```
   Set customers_a = reservations[1].getCustomers();
   reservations[4].setCustomers(customers_a);
```

A collection (`Set`) of Customer beans is retrieved from the first Reservation bean. This collection will contain references to Customers 1-3 in this example. The `setCustomers` method is called on the fourth Reservation bean to change its list of related Customer beans to be equal to the passed-in collection.

Because the multiplicity on both sides of the relationship is "many," this operation does not remove any other links between Customers 1-3 and their related Reservation beans, but simply changes the links associated with the fourth Reservation bean to be as they are shown in the following figure:

Figure 35: Relationships after setCustomers operation on Reservation D



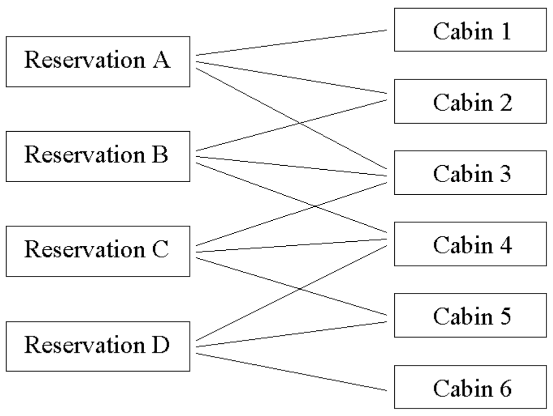
Because this example creates Customer objects with specific primary keys (1-6) you may not run it more than once unless you clear out the tables first. In addition, the `CUSTOMER`, `RESERVATION`, and `RESERVATION_CUSTOMER_LINK` tables should be empty before proceeding to the next example.

Client_77c.jsp

This example demonstrates the Cabin/Reservation relationship, including the removal of Cabins from the Reservation using an iterator (Figure 7-20).

A set of four Reservation beans and another of six Cabin beans are created and linked to each other as shown in Figure 36 below:

Figure 36: Initial relationships in Client_77c example



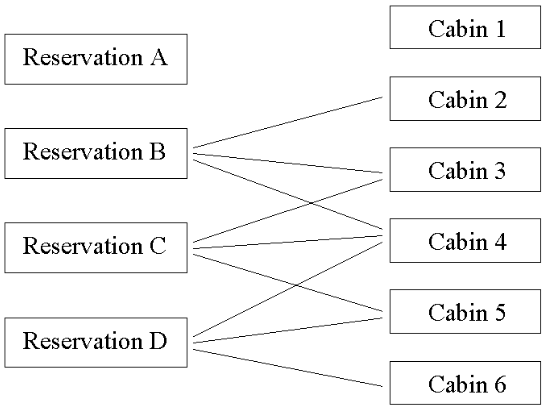
The application starts a transaction, acquires the collection of Cabin beans associated with the first Reservation, and uses it to retrieve an `Iterator` object:

```
try {
    tran.begin();
    Set cabins_a = reservations[1].getCabins();
    Iterator iterator = cabins_a.iterator();
    while (iterator.hasNext()) {
        CabinLocal cc = (CabinLocal)iterator.next();
        System.out.println("...");
        out.print("...");
        iterator.remove();
    }
    tran.commit();
}
catch (Exception e) {
    e.printStackTrace();
    tran.rollback();
}
```

The loop iterates through the elements of the collection and removes them from it, effectively removing these cabins from the relationship with the first Reservation bean. The Cabin beans themselves are not removed, of course; just the links with the Reservation.

After this loop is complete, the Reservation/Cabin relationships will look like this:

Figure 37: Relationships after removing cabins from Reservation A's collection



Because this example creates Cabin objects with specific primary keys (1-6) you may not run it more than once unless you clear out the tables first. In addition, the `CABIN`, `RESERVATION`, and `RESERVATION_CABIN_LINK` tables should be empty before proceeding to the next example.

Client_77d.jsp

This example demonstrates the Cabin/Reservation relationship, including the use of `setCabins` to modify the Cabins for a Reservation.

The code in the example walks step-by-step through the creation of Reservations, Cabins, and their relationships before using the `setCabins` method to modify the relationship. We hope this example will cement your understanding of relationships in CMP 2.0, and how you can create and manipulate them using cmr fields and method calls.

First, three Cabin beans are created using primary key values 1-3:

```
cabins[1] = cabinhome.create(new Integer(1));
cabins[1].setShip(shipA);
cabins[1].setDeckLevel(1);
cabins[1].setName("Minnesota Suite");
cabins[1].setBedCount(2);
cabins[2] = cabinhome.create(new Integer(2));
...
cabins[3] = cabinhome.create(new Integer(3));
...
```

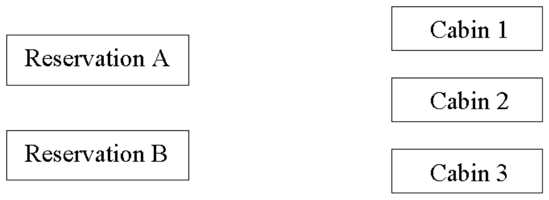
Next, two Reservation beans are created (via automatic key generation):

```
reservations[1] = reservationhome.create(cruiseA, null);
reservations[1].setDate(...);
reservations[1].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[2] = reservationhome.create(cruiseA, null);
...
```

The following figure illustrates the status of the Reservation and Cabin beans at this point. The beans exist but are not yet related to each other:

Figure 38: Reservation and Cabin beans after bean creation

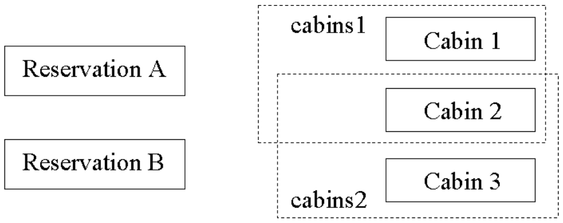


The next section builds two simple `HashSet` objects containing Cabins 1 and 2, and Cabins 2 and 3, respectively:

```
Set cabins1 = new HashSet(2);
cabins1.add(cabins[1]);
cabins1.add(cabins[2]);
Set cabins2 = new HashSet(2);
cabins2.add(cabins[2]);
cabins2.add(cabins[3]);
```

These `Set` objects contain references to the Cabin beans as shown below:

Figure 39: Set objects contain Cabin beans

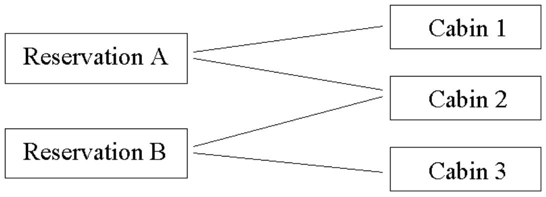


The next two lines of code use these `Set` objects in calls to the Reservation beans:

```
reservations[1].setCabins(cabins1);
reservations[2].setCabins(cabins2);
```

The resulting Reservation/Cabin relationships look like this:

Figure 40: Relationships after setCabins calls on both Reservation beans

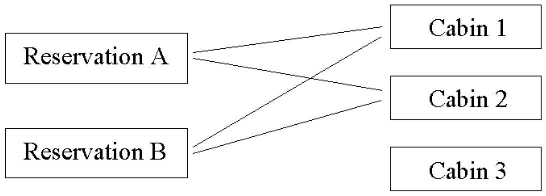


The final section of code uses the `setCabins` method to replace previous links between the second Reservation bean and its Cabins with links to the specified set of Cabin beans, much as you've seen before.

```
try {
    tran.begin();
    Set cabins_a = reservations[1].getCabins();
    reservations[2].setCabins(cabins_a);
    tran.commit();
}
catch (Exception e) {
    e.printStackTrace();
    tran.rollback();
}
```

The resulting relationships look like this:

Figure 41: Relationships after modifying Reservation B's cabins



Because this example creates Cabin objects with specific primary keys (1-3) you may not run it more than once unless you clear out the tables first. In addition, the CABIN, RESERVATION, and RESERVATION_CABIN_LINK tables should be empty before proceeding to the next exercise.

Optional Additional Tasks

Change the Ship and Cabin beans to use automatic sequence generation. This variation entails:

1. Making changes to the *weblogic-cmp-rdbms-jar.xml* file
2. Creating SHIP_SEQUENCE and CABIN_SEQUENCE tables with single rows in each
3. Modifying the create and postcreate methods in bean and local interfaces to remove the primary key from the parameter list and method code

Test your changes by modifying one of the client JSP pages to create some beans without specifying the primary key.

Exercise 7.3: Cascade Deletes in CMP 2.0

This is a very short exercise to demonstrate the use of automatic cascade-delete handling by the container, using the Customer bean and some related "children" beans.

Download and Build the Example Programs

Download and extract the *ex07_3* directory in the familiar way.

There are no differences between this set of code and descriptors and the files in the *ex07_2* example other than the additional `<cascade-delete/>` tags in the *ejb-jar.xml* file and the new example JSP page for this exercise.

Use *ant dist* to build the example code and place the *titanejb.jar* file in the correct directory.

Create the Required Database Objects

This exercise requires no additional database objects. All tables should be empty at the start of the exercise to avoid duplicate-key errors.

Examine the Standard EJB Descriptor File

The special `<cascade-delete/>` tag appears in the relationship section of the standard descriptor file (*ejb-jar.xml*) within the "role" section for the bean that should be deleted when the parent is deleted. For example, the Address bean associated with the Customer bean should be deleted when the Customer is removed, so the pertinent section of the *ejb-jar.xml* file now has the `<cascade-delete/>` tag:

```
<ejb-relation>
    <ejb-relation-name>Customer-HomeAddress</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Customer-has-a-Address
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <relationship-role-source>
            <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>homeAddress</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            Address-belongs-to-Customer
        </ejb-relationship-role-name>
        <multiplicity>one</multiplicity>
        <cascade-delete/>
        <relationship-role-source>
            <ejb-name>AddressEJB</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
```

Note that the `<cascade-delete/>` tag is placed on the "dependent object" side of the relationship, in this case on the Address EJB side of the Customer/HomeAddress relationship.

Scan through *ejb-jar.xml* and find the other places this tag has been added to relationships.

Examine the WebLogic-Specific Files/Components

This exercise requires no changes to the WebLogic-specific descriptor files.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task to force a re-deployment of the entire *titanapp* enterprise application, or simply reboot the server to deploy the new *titanejb.jar* file.

Use the console to verify that all of the example beans are properly deployed by checking for their home interfaces in the JNDI tree.

Examine and Run the Client JSP Pages

There is a single client JSP page available in this exercise, *Client_78.jsp*. It demonstrates the creation of a Customer EJB having a related CreditCard, Address, and list of Phone beans. It then shows the container's ability to perform cascading deletions. A simple call removes the Customer, and the container's CMP functionality automatically removes its children.

The example page is straightforward and will not be discussed in detail in this workbook.

The example should not leave any new rows in the database, but do ensure that the workbook tables are empty before proceeding to the next exercise.

Exercises for Chapter 8

Exercise 8.1: Simple EJB QL Statements

This exercise explores some of the basic EJB QL functionality available in CMP 2.0, including finder methods, ejbSelect methods, and the use of the `IN` operation in queries. More complex queries and operations will be examined in the next exercise.

Download and Build the Example Programs

Download and extract the *ex08_1* directory as you have other example directories.

This example extends the same EJB components you worked with in the previous exercise, adding new finder methods to home interfaces and ejbSelect methods to beans.

Use *ant dist* to build the example code and place the *titanejb.jar* file in the correct directory.

Create the Required Database Objects

This exercises requires no additional database objects. To avoid duplicate-key errors, make sure all tables are empty at the start of the exercise.

Examine the Standard EJB Descriptor File

EJB QL is a standard mechanism for declaring the queries (both finder methods and ejbSelect methods) available in the beans and their home interfaces. Because it is a standard mechanism, all of the descriptor information required for these queries is normally found in the standard *ejb-jar.xml* file.

The first minor change this exercise requires is a change in the `<abstract-schema-name>` element for all of the entity beans. This is the name that will appear in the query definitions to represent a given entity bean. It need not match the EJB name or the physical database table name. For previous exercises it looked like this:

```
<abstract-schema-name>CustomerEJB</abstract-schema-name>
```

To make the query definitions look a little more like SQL, the element for each bean now looks something like:

```
<abstract-schema-name>Customer</abstract-schema-name>
```

There are a number of new `<query>` sections in the *ejb-jar.xml* file. We will examine them in the order in which they appear in the file.

The first set of `<query>` elements defines three different finder methods in the Customer home interface for locating Customer beans by name, by credit rating, or by city:

```
<query>
    <query-method>
        <method-name>findByName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.lastName = ?1 AND c.firstName = ?2
    </ejb-ql>
</query>
<query>
    <query-method>
        <method-name>findByGoodCredit</method-name>
        <method-params></method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.hasGoodCredit = TRUE
    </ejb-ql>
</query>
<query>
    <query-method>
        <method-name>findByCity</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.homeAddress.city=?1 AND c.homeAddress.state=?2
    </ejb-ql>
</query>
```

Some important things to note about these finder queries:

- They all return zero or more Customer beans matching the criteria specified. Finder methods must return the beans managed by that home interface. They may not return other beans or other data types like `String`, `Integer`, etc.
- The `findByName` and `findByCity` methods expect parameters and use the placeholders `?1` and `?2` to refer to these parameters within the `<ejb-ql>` element.

- The WHERE clause for the findByCity method demonstrates traversing a single-valued relationship (i.e., “something-to-one” not “something-to-many”) to obtain an attribute value in a related bean. The method uses the cmr field name homeAddress to traverse the relation, not the physical database foreign key, ADDRESS_ID.

Each finder query must have a corresponding method definition in the home interface:

CustomerHomeLocal.java

```
public CustomerLocal findByName(String lastName, String firstName)
    throws FinderException;

public Collection findByGoodCredit()
    throws FinderException;

public Collection findByCity(String city, String state)
    throws FinderException;
```

Note that the method names and parameter types must exactly match the <method-name> and <method-param> elements in the query definition, but the parameter names are unimportant.

Finders can return either a single bean reference or a Collection of bean references. If the query returns multiple beans but the method definition in the home interface specifies a single bean as the return type (as findByName's definition does above), only the first matching bean will be returned.

> Warning: The specific bean returned in this case is impossible to determine with confidence. It is likely to be either 1) the first bean encountered in the table while scanning a database index on the specified columns, or 2) the first bean inserted in the table if no corresponding index exists. In addition, the fact that multiple matching beans exist is unknown to the caller because there is no exception thrown or return value present indicating this fact. Unless the criteria are known to represent a unique bean, the use of a single bean reference as a return type for a finder method should be avoided in favor of a collection.

The next query section in *ejb-jar.xml* defines some ejbSelect methods in the Address bean:

```
<query>
    <query-method>
        <method-name>ejbSelectZipCodes</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT a.zip FROM Address AS a
        WHERE a.state = ?1
    </ejb-ql>
</query>
```

```
<query>
    <query-method>
        <method-name>ejbSelectAll</method-name>
        <method-params></method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(a) FROM Address AS a
    </ejb-ql>
</query>
<query>
    <query-method>
        <method-name>ejbSelectCustomer</method-name>
        <method-params>
            <method-param>com.titan.customer.AddressLocal
            </method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer AS c
        WHERE c.homeAddress = ?1
    </ejb-ql>
</query>
```

These ejbSelect queries demonstrate some additional functionality:

- The `ejbSelectZipCodes` query returns a simple String datatype rather than a bean reference.
- The `ejbSelectAll` query returns Address beans, something we could just as easily have accomplished with a finder method in the Address home interface.
- The `ejbSelectCustomer` query returns a Customer bean, rather than an Address bean, something we could not do in a finder method in the Address home interface (although it could be done in the Customer home interface).
- The `WHERE` clause in the final query also uses the relationship-traversal syntax ("c.homeAddress") and demonstrates checking for equality with a specific bean (the desired Address is passed in as a query method parameter).

Each ejbSelect query must have a corresponding method in the Address bean class:

AddressBean.java

```
public abstract Set ejbSelectZipCodes(String state)
    throws FinderException;

public abstract Collection ejbSelectAll()
    throws FinderException;

public abstract EntityBean ejbSelectCustomer(AddressLocal addr)
    throws FinderException;
```

The method names and parameter types must match the definitions in the `<query>` elements perfectly, but the parameter names are not important. The choice of return type is very important for methods which return multiple objects (bean references, `String`s, etc). As the EJB book points out, the use of `Set` implies duplicate values should be removed, while `Collection` allows for duplicates in the returned collection.

The return type for the `ejbSelectCustomer` method should be `CustomerLocal`, but a bug in the current version of WebLogic requires the use of `javax.ejb.EntityBean` as a return type to avoid CMP compilation errors. It will still return a reference to a Customer bean, but the returned type will be EntityBean and the caller must cast it to `CustomerLocal` before use.

- Note also: As this workbook went to press, a pending WebLogic bug (#CR051294) prevented this ejbSelect method from operating properly.

A significant downside to using ejbSelect methods is their definition as `private` methods in the CMP-generated bean code, making them available only within the bean class itself.

To facilitate testing the methods and building simple client example programs, the example code uses a public `Home` method in the Address home interface with a corresponding public ejbHome method in the bean which calls the desired internal ejbSelect method:

AddressHomeLocal.java

```
public Collection selectZipCodes(String state)
    throws FinderException;
```

AddressBean.java

```
public Collection ejbHomeSelectZipCodes(String state)
    throws FinderException {
        return this.ejbSelectZipCodes(state);
}
```

Home methods are described briefly in Chapter 5 of the EJB book and are covered in detail in Chapter 11. Adding these methods makes it easy to test the **`private`** `ejbSelectZipCodes` query method by invoking the **`public`** `selectZipCodes` method in the Address home interface.

The next query section in *ejb-jar.xml* defines a simple finder method of the Cruise bean:

```
<query>
    <query-method>
        <method-name>findByName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
       SELECT OBJECT(c) FROM Cruise c WHERE c.name = ?1
    </ejb-ql>
</query>
```

This very straightforward method accepts a `String` parameter and returns zero or more Cruise beans. The method definition in the Cruise home interface is:

CruiseHomeLocal.java

```
public CruiseLocal findByName(String name)
    throws FinderException;
```

The use of `CruiseLocal` as the return type rather than `Collection` implies that there should only be one Cruise with a given name – which may or may not be true. If more than one Cruise has a given name, the finder method will return the first one it finds.

The final query section in *ejb-jar.xml* defines a finder and an ejbSelect method for the Cabin bean:

```
<query>
    <query-method>
        <method-name>findAllOnDeckLevel</method-name>
        <method-params>
            <method-param>java.lang.Integer</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Cabin as c WHERE c.deckLevel = ?1
    </ejb-ql>
</query>
```

```
<query>
    <query-method>
        <method-name>ejbSelectAllForCustomer</method-name>
        <method-params>
            <method-param>com.titan.customer.CustomerLocal
            </method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(cab) FROM Customer AS cust,
          IN(cust.reservations) AS res,
          IN(res.cabins) AS cab
        WHERE cust = ?1
    </ejb-ql>
</query>
```

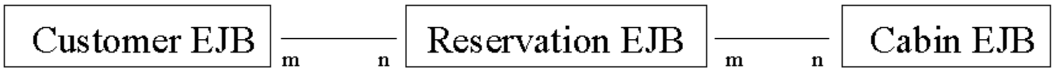
The finder, `findAllOnDeckLevel`, is very straightforward. It accepts an integer parameter and returns zero or more Cabin beans that match the criterion. There is a corresponding method in the Cabin home interface:

CabinHomeLocal.java

```
public abstract Collection findAllOnDeckLevel(Integer level)
    throws FinderException;
```

The second query in the Cabin bean, `ejbSelectAllForCustomer`, is more complex. It is designed to return a collection of all Cabin beans related to a given Customer. Recall that the relationships among Customer, Reservation, and Cabin are "many-sided":

Figure 42: Many-to-many relationships in ejbSelectAllForCustomer query



The query uses the `IN` operation discussed in the EJB book to traverse the relationships from the Customer bean down through the "reservations" cmr field to a list of Reservation beans, then through the "cabins" cmr field on each Reservation bean to a list of Cabin beans:

```
SELECT OBJECT(cab) FROM Customer AS cust,
  IN(cust.reservations) AS res,
  IN(res.cabins) AS cab
WHERE cust = ?1
```

An alternate non-standard syntax available in WebLogic makes this process of traversing cmr fields a little clearer:

```
SELECT OBJECT(cab) FROM Customer AS cust,
  cab in cust.reservations.cabins
WHERE cust = ?1
```

The alternate syntax replaces the multiple `IN` operations with a single chain of cmr fields. This syntax is available only in WebLogic, so don't use it if you wish your EJB-based application to be vendor- and platform-independent.

A public Home method called `selectAllForCustomer` is available in the Cabin home interface that calls this private `ejbSelectAllForCustomer` select method when invoked:

CabinHomeLocal.java

```
public Set selectAllForCustomer(CustomerLocal cust)
    throws FinderException;
```

CabinBean.java

```
public abstract Set ejbSelectAllForCustomer(CustomerLocal cust)
    throws FinderException;

public Set ejbHomeSelectAllForCustomer(CustomerLocal cust)
    throws FinderException {
        return this.ejbSelectAllForCustomer(cust);
}
```

Note that we are using `Set` as the return type for these methods to indicate to the container that the resulting list of Cabins should not contain duplicates.

Examine the WebLogic-Specific Files/Components

This exercise requires no changes to the WebLogic-specific descriptor files.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server as needed to deploy the new *titanejb.jar* file.

Use the console to verify that all the example beans are properly deployed by checking for their home interfaces in the JNDI tree.

Examine and Run the Client JSP Pages

This exercise includes a large number of client JSP pages that demonstrate use of the query methods you've just seen. The *ant dist* task should have copied the pages to the *webapp* directory in the *titanapp* exploded *.ear* file, making them available using the standard URLs; e.g.:

http://servername:7001/webapp/Client_81.jsp

The example JSP pages are first summarized, then described in more detail:

- *Client_81.jsp* demonstrates the `findByName` finder method in the Customer home interface
- *Client_82.jsp* demonstrates the `findByCity` finder method in the Customer home interface
- *Client_83.jsp* demonstrates the `ejbSelectZipCodes` select method in the Address bean
- *Client_84a.jsp* sets up the database for the subsequent examples.
- *Client_84b.jsp* demonstrates the `findAllOnDeckLevel` finder method in the Cabin home interface
- *Client_84c.jsp* demonstrates the `ejbSelectAllForCustomer` select method in the Cabin bean

Client_81.jsp

This example creates a small set of Customer beans and demonstrates the findByName finder method in the Customer home interface.

The first section of code creates Customer beans with key values from 80-99 and populates them with some appropriate attributes along with Address and Phone information.

The next section performs the call to the findByName finder method to acquire a reference to a single Customer bean with the desired name:

```
CustomerLocal cust85 = customerhome.findByName("Smith85","John");
```

The client displays information about the retrieved Customer to verify that the finder method properly returned a Customer having the desired name.

The Customer beans created in this exercise are used in Client_82 and Client_83.jsp, so do not delete them unless you wish to modify Client_81 and run it again.

Client_82.jsp

This example demonstrates the findByCity finder method in the Customer home interface. The Customer beans created in Client_81.jsp must still exist in the database, along with their Address information.

The code first invokes the Customer's findByCity method to retrieve a collection of matching Customer bean references:

```
Collection mplscusts = custhome.findByCity("Minneapolis","MN");
```

The client then iterates through the collection and displays information about the Customer beans returned to let you verify that they indeed match the criteria. Note that this search is case-sensitive, so the names must match exactly. In the next exercise (8.2) we will use the LIKE comparison operator to perform searches with partial matches, but unfortunately EJB QL lacks any simple way to perform case-insensitive searches.

Client_83.jsp

This example demonstrates the `ejbSelectZipCodes` select method in the Address bean. The Customer beans created in Client_81.jsp must still exist in the database, along with their Address information.

Normally, ejbSelect methods defined on EJBs are not accessible to outside callers such as our client JSP page. We created the public Home method `selectZipCodes` and matching `ejbHomeSelectZipCodes` method on the bean for just this purpose.

The code in the client page simply calls the public `selectZipCodes` Home method and iterates through the returned Collection:

```
Collection mnzips = addresshome.selectZipCodes("MN");
Iterator iterator = mnzips.iterator();
while (iterator.hasNext()) {
    String zip = (String)(iterator.next());
    out.print(zip+"<br>");
}
```

As an optional exercise, try modifying the ejbSelect method in the Address bean to return a `Collection` instead of a `Set`:

AddressBean.java

```
public abstract Collection ejbSelectZipCodes(String state)
    throws FinderException;
```

This simple change will have a big impact on the code generated by the CMP process and the resulting output of the method call. Use *ant dist* to rebuild the EJBs, and redeploy or reboot as necessary to effect the change.

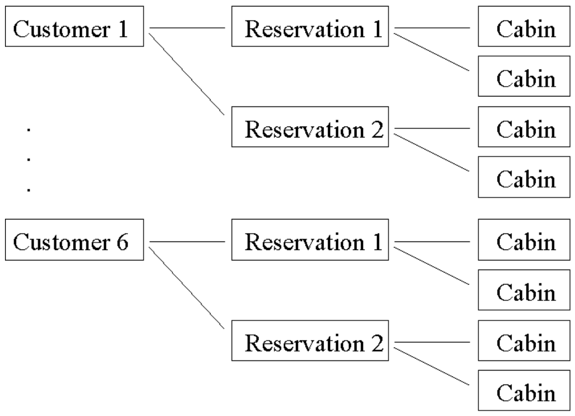
When you re-run this client page, instead of seeing each matching zip code only once, you should see a longer list with the same zip codes appearing multiple times.

The next three example programs use a different set of sample data, so empty all of the workbook tables before proceeding to Client_84a.jsp.

Client_84a.jsp

This example sets up the database for the subsequent two examples by creating a set of six Customer beans, each with two related Reservation beans, each having two Cabins:

Figure 43: Beans and relationships created by Client_84a.jsp



Client_84b.jsp

This example demonstrates the `findAllOnDeckLevel` finder method in the Cabin home interface. It is a fairly straightforward program with a single call to the finder method to retrieve a collection of Cabin bean references:

```
Collection cabins = cabinhome.findAllOnDeckLevel(new Integer(3));

Iterator iter = cabins.iterator();
while (iter.hasNext()) {
    CabinLocal cabin = (CabinLocal)(iter.next());
    out.print(cabin.getName()+ ... );
}
```

❖ You may be wondering why we did not have to enclose the iteration through this collection in a transaction, as in the examples in Exercise 7.2. In that exercise, the `Collection` references we received by calling cmr field get functions actually represented relationships between beans, and anything we did to the collection was applied to the relationships. The container insists that activity such as this be done in the context of a transaction. The reason we don't need a transaction in this case is that the *collection* returned by the finder method is truly a simple Java collection of references to Cabin beans (stubs) created by the finder for our use. It does not represent a relationship between beans as it did in the Chapter 7 exercises, so we are free to manipulate the collection, iterate through it, remove items from it, etc. with no possible impact on the underlying entity beans. Therefore, no transaction spanning our activity is required.

Client_84c.jsp

This example demonstrates the `ejbSelectAllForCustomer` select method in the Cabin bean, and shows the effects of the IN function in the <ejb-ql> query definition. Just as *Client_83* did, *Client_84c* calls a public Home method that delegates the work to the matching ejbHome method on the bean which in turn calls the private `ejbSelectAllForCustomer` method to perform a query based on the <ejb-ql> elements in the *ejb-jar.xml* file:

```
SELECT OBJECT(cab) FROM Customer AS cust,
  IN(cust.reservations) AS res,
  IN(res.cabins) AS cab
WHERE cust = ?1
```

The net result is a collection of Cabin bean references for all cabins that are related to the desired Customer bean through Reservation bean relationships.

`ejbSelectAllForCustomer` returns a collection of Cabin beans, begging the question of why it could not simply be a finder method in the Cabin home interface. That implementation would eliminate the need for a Home method, would be easier to use, and would make more sense. Unfortunately, as this workbook went to press, the current version (6.1) of WebLogic did not seem able to create queries of this complexity as finder methods.

Make sure all workbook tables are empty before proceeding to the next exercise.

Exercise 8.2: Complex EJB QL Statements

This exercise explores the more complex operations and comparison operators available in the EJB query langage. The final example program also demonstrates use of a WebLogic-specific extension to the EJB QL language, `ORDERBY`.

Download and Build the Example Programs

Download and extract the *ex08_2* directory as in preceding exercises.

This example extends the same EJB components as the previous exercise, adding new finder methods to the Customer and Ship home interfaces only.

Use *ant dist* to build the example code and place the *titanejb.jar* file in the correct directory.

Create the Required Database Objects

This exercises requires no additional database objects. To avoid duplicate-key errors, make sure all tables are empty at the start of the exercise.

Examine the Standard EJB Descriptor File

Most of the changes to the standard EJB descriptor file are located in the CustomerEJB section of the *ejb-jar.xml* file. New finder methods have been created to demonstrate more complex operators available in EJB QL.

Start by examining the new finder methods in the ShipEJB definition:

```
<query>
    <query-method>
        <method-name>findByTonnage</method-name>
        <method-params>
            <method-param>java.lang.Double</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(s) FROM Ship s
        WHERE s.tonnage = ?1
    </ejb-ql>
</query>
```

```
<query>
    <query-method>
        <method-name>findByTonnage</method-name>
        <method-params>
            <method-param>java.lang.Double</method-param>
            <method-param>java.lang.Double</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(s) FROM Ship s
        WHERE s.tonnage BETWEEN ?1 AND ?2
    </ejb-ql>
</query>
```

Items to note:

- The two finders have the same name but different sets of parameters. You may overload finder methods just as you do other methods in Java. The appropriate version will be called, depending on the parameters supplied in the method call.
- The two-parameter version of the method uses the `BETWEEN` operator in the `WHERE` clause to select only those ships whose tonnage falls between the values passed .

The `ShipHomeLocal` interface must declare both of these finder methods:

```
public Collection findByTonnage(Double tonnage)
    throws javax.ejb.FinderException;

public Collection findByTonnage(Double tonnage1, Double tonnage2)
    throws javax.ejb.FinderException;
```

The query method to examine is a redefinition of one seen earlier. We redefined the `findByName` method in the CustomerEJB to use a `LIKE` operator in the `WHERE` clause, then renamed the old method, which used equality comparisons, to `findByExactName`:

```
<query>
    <query-method>
        <method-name>findByExactName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.lastName = ?1 AND c.firstName = ?2
    </ejb-ql>
</query>
```

```
<query>
    <query-method>
        <method-name>findByName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.lastName LIKE ?1 AND c.firstName LIKE ?2
    </ejb-ql>
</query>
```

Recognize that we could not use overloading in this case because the two methods have the same number and types of parameters.

Also note that, because the `LIKE` operations compare bean attribute values to parameters passed to `findByName`, it will be important to include the proper wildcard characters in the parameters, if the wildcard-matching capability of `LIKE` is to work as expected.

For example, these two finder calls will act exactly the same:

```
Collection custs1 = custhome.findByExactName("S","Jo");
Collection custs2 = custhome.findByName("S","Jo");
```

Because the parameters passed to `findByName` do not include wildcards, the `LIKE` operators will in effect compare for equality. One proper way to call `findByName` would be:

```
Collection custs2 = custhome.findByName("S%","Jo%");
```

This call will return all Customers having first names beginning with "Jo" and last names beginning with the letter "S", matching people like "Joan Smith" and "Joseph Star."

The next query section defines an additional finder that uses both name and state to demonstrate again how the `AND` operator works in a `WHERE` clause:

```
<query>
    <query-method>
        <method-name>findByNameAndState</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
```

```
        <ejb-ql>
            SELECT OBJECT(c) FROM Customer c
            WHERE c.lastName LIKE ?1 AND c.firstName LIKE ?2
              AND c.homeAddress.state = ?3
        </ejb-ql>
    </query>
```

Skip two query definitions left over from the previous exercise and you will find that the next section added for this exercise demonstrates use of the `IN` operator in the `WHERE` clause:

```
    <query>
        <query-method>
            <method-name>findInHotStates</method-name>
            <method-params></method-params>
        </query-method>
        <ejb-ql>
            SELECT OBJECT(c) FROM Customer c
            WHERE c.homeAddress.state IN ('FL','TX','AZ','CA')
        </ejb-ql>
    </query>
```

This query will return only those customers having a home address in one of the specified states.

One unfortunate limitation of the `IN` operator is inherited from SQL itself: There is no easy way to pass a list of values for use in the `IN` operator.

The next query demonstrates the use of `IS EMPTY` to identify beans having an "empty" relationship:

```
    <query>
        <query-method>
            <method-name>findWithNoReservations</method-name>
            <method-params></method-params>
        </query-method>
        <ejb-ql>
            SELECT OBJECT(c) FROM Customer c
            WHERE c.reservations IS EMPTY
        </ejb-ql>
    </query>
```

This query will return all Customer beans that have no related Reservation beans.

The next query demonstrates the use of `MEMBER OF` to identify beans that are members of a relationship with some other bean:

```
<query>
    <query-method>
        <method-name>findOnCruise</method-name>
        <method-params>
            <method-param>com.titan.cruise.CruiseLocal
            </method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(cust) FROM Customer cust, Cruise cr
        WHERE cr = ?1
          AND cust MEMBER OF cr.reservations.customers
    </ejb-ql>
</query>
```

This query has a single `CruiseLocal` parameter and uses the `MEMBER OF` operator to limit the Customer beans returned to only those customers that are related to reservations for the Cruise bean specified. This ability to traverse multiple "many" relationships and create a single set of Customer beans is a powerful tool for creating queries, and is a much cleaner solution than the three-way join a pure SQL query would have required.

The final new query in the descriptor file demonstrates the use of a WebLogic-specific extension to the EJB QL language, the `ORDERBY` clause:

```
<query>
    <query-method>
        <method-name>findByState</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.homeAddress.state = ?1
        ORDERBY c.lastName,c.firstName
    </ejb-ql>
</query>
```

The `findByState` method selects a list of all Customers with a home address in the state passed to it as a parameter. The `ORDERBY` clause in the query definition directs the container to return the collection of customers sorted by last name and first name. Whether it uses an `ORDER BY` clause in the actual SQL statement or simply sorts the Collection after retrieving it is up to the CMP code-generation algorithm. WebLogic 6.1 uses an `ORDER BY` clause in the underlying SQL statement.

All of the Customer finder methods must be declared in the CustomerHomeLocal class:

```
public CustomerLocal findByExactName(String lastName,
    String firstName)
    throws FinderException;

public Collection findByName(String lastName, String firstName)
    throws FinderException;

public Collection findByNameAndState(String lastName,
    String firstName, String state)
    throws FinderException;

public Collection findByGoodCredit()
    throws FinderException;

public Collection findByCity(String city, String state)
    throws FinderException;

public Collection findInHotStates()
    throws FinderException;

public Collection findWithNoReservations()
    throws FinderException;

public Collection findOnCruise(CruiseLocal cruise)
    throws FinderException;

public Collection findByState(String state)
    throws FinderException;
```

Examine the WebLogic-Specific Files/Components

This exercise requires no changes to the WebLogic-specific descriptor files.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server as needed to deploy the new *titanejb.jar* file.

Use the console to verify that all of the example beans are properly deployed by checking for their home interfaces in the JNDI tree.

Examine and Run the Client JSP Pages

This exercise includes a large number of client JSP pages that demonstrate the finder methods added to the beans in this exercise. The *ant dist* task should have copied the pages to the *webapp* directory in the *titanapp* exploded *.ear* file, making them available using the standard URLs:

http://servername:7001/webapp/Client_85.jsp

The following list summarizes the purpose of the example JSP pages, with more detail in the sections following the list:

- *Client_85.jsp* demonstrates the `findByTonnage` methods of the Ship home interface
- *Client_86a.jsp* performs Customer setup for subsequent examples
- *Client_86b.jsp* demonstrates the two methods in the Customer home interface that find Customers by name
- *Client_87.jsp* demonstrates the `findInHotStates` method of the Customer home interface
- *Client_88.jsp* demonstrates the `findWithNoReservation` and `findOnCruise` methods of the Customer home interface
- *Client_89.jsp* demonstrates the `findByState` method of the Customer home interface

Client_85.jsp

This example demonstrates the `findByTonnage` methods of the Ship home interface.

The client first creates a variety of Ship beans with different tonnage values, then calls the one-parameter and two-parameter versions of `findByTonnage` to retrieve collections of Ship beans matching the criteria:

```
Collection ships100k =
      shiphome.findByTonnage(new Double(100000.0));

Iterator iterator = ships100k.iterator();
while (iterator.hasNext()) {
    ShipLocal ship = (ShipLocal)(iterator.next());
    out.print("pk="+ship.getId()+ ... );
}

Collection ships50110k = shiphome.findByTonnage(
                  new Double(50000.0), new Double(110000.0));

iterator = ships50110k.iterator();
while (iterator.hasNext()) {
    ShipLocal ship = (ShipLocal)(iterator.next());
    out.print("pk="+ship.getId()+ ... );
}
```

The BETWEEN operator in EJB QL is inclusive, so output for the call specifying lower and upper limits of 50,000 and 110,000 tons should include Ship beans with exactly these displacement values as well as values between them.

The rows created in the SHIP table by this example will be used in subsequent examples.

Client_86a.jsp

This example performs Customer setup for subsequent examples. It creates a fairly large set of Customer beans with associated Address, Phone, Cruise, and Reservation beans sufficient for the remaining examples in this exercise.

If you need to run this example again because of an error or because you wish to create a different set of data for experimentation, you must first empty all workbook tables except the SHIP table to avoid duplicate-key problems.

Client_86b.jsp

This example demonstrates the two methods in the Customer home interface that find Customers by name.

The client uses findByExactName and findByName to produce lists of Customer beans matching various criteria. First it calls the exact-match method, supplying a name that should exist in the database:

```
CustomerLocal customer =
    customerhome.findByExactName("Star","Joe");
```

This will return a single Customer bean with the desired name if one exists – the first matching bean if more than one meets the criteria.

Next, the client calls the wildcard-matching method, the one that uses a LIKE comparison operator. We've purposefully neglected to include the wildcard character ('%') in the first call to findByName to verify that it acts like the exact-match method and returns no matching Customer beans:

```
Collection customers = customerhome.findByName("S","Jo");

Iterator iterator = customers.iterator();
while (iterator.hasNext()) {
    CustomerLocal customer = (CustomerLocal)(iterator.next());
    AddressLocal addr = customer.getHomeAddress();
    out.print( ... );
}
```

The second call uses the wildcard character and should return some matching Customer beans:

```
customers = customerhome.findByName("S%","Jo%");

iterator = customers.iterator();
while (iterator.hasNext()) {
    CustomerLocal customer = (CustomerLocal)(iterator.next());
    AddressLocal addr = customer.getHomeAddress();
    out.print( ... );
}
```

The final section of the code demonstrates the `findByNameAndState` method, passing a partial name (including wildcards) and the desired state:

```
customers = customerhome.findByNameAndState("S%","Jo%","MN");
```

Feel free to edit this JSP page to experiment with these methods – but don't forget to perform the *ant dist* task to push any modified JSP pages to the *webapp* directory within the enterprise application before trying to access them through your browser.

Client_87.jsp

This example demonstrates the `findInHotStates` method of the Customer home interface. It is straightforward and will not be examined in this text.

Client_88.jsp

This example demonstrates the `findWithNoReservation` and `findOnCruise` methods of the Customer home interface.

The first call to the `findWithNoReservation` method returns a list of Customer beans having no Reservation beans in their `reservations` cmr field:

```
Collection poorcustomers = customerhome.findWithNoReservations();

Iterator iterator = poorcustomers.iterator();
while (iterator.hasNext()) {
    CustomerLocal customer = (CustomerLocal)(iterator.next());
    AddressLocal addr = customer.getHomeAddress();
    out.print( ... );
}
```

The next section of the code demonstrates the use of the `findOnCruise` method. First, the client retrieves a desired Cruise bean using its own `findByName` method:

```
CruiseLocal cruiseA = cruisehome.findByName("Alaska Cruise");
```

This Cruise bean is then passed to the `findOnCruise` method to retrieve a list of Customer beans who have reservations on the desired cruise:

```
Collection alaskacustomers = customerhome.findOnCruise(cruiseA);
iterator = alaskacustomers.iterator();
while (iterator.hasNext()) {
    ...
}
```

Recall that this finder method employed the powerful `MEMBER OF` operator to examine all of the reservations for the given cruise to find the customers with reservations on that cruise:

```
<ejb-ql>
    SELECT OBJECT(cust) FROM Customer cust, Cruise cr
    WHERE cr = ?1 AND cust MEMBER OF cr.reservations.customers
</ejb-ql>
```

We supplied the “Alaska Cruise” Cruise bean, and the CMP engine performed all the necessary database logic to build the correct list of customers.

If you want to experiment with this example page, you might try passing the other Cruise bean (“Bohemian Cruise”), or a null value, or a new Cruise bean you create that has no reservations.

Client_89.jsp

This example demonstrates the `findByState` method of the Customer home interface.

The only noteworthy aspect here is the use of the WebLogic-specific `ORDERBY` extension to provide the Customer beans in sorted order. Run the example and verify that the Customer beans are sorted by last name then first name.

Feel free to play with the `findByState` query definition in *ejb-jar.xml*, changing the sort order or removing the extension completely to see the effects. Don’t forget to perform the *ant dist* task to rebuild the *titanejb.jar* file if you make changes to the descriptor, and to redeploy or reboot as needed to activate the new version of the bean.

Note that it is also possible to avoid WebLogic-specific extensions in the standard *ejb-jar.xml* file and still achieve the same sorting behavior. In *ejb-jar.xml* you can define `findByState` to have the standard query without the `ORDERBY` clause:

```
<query>
    <query-method>
        <method-name>findByState</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.homeAddress.state = ?1
    </ejb-ql>
</query>
```

In the *weblogic-cmp-rdbms-jar.xml* file you can then redefine the finder method to use the `<weblogic-query>` element and the proper tags:

```
<weblogic-query>
    <query-method>
        <method-name>findByState</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <weblogic-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.homeAddress.state = ?1
        ORDERBY c.lastName
    </weblogic-ql>
</weblogic-query>
```

💣 If you want to include the WebLogic-specific extensions, the finder name and parameters must match the version in the *ejb-jar.xml* file exactly. The *ejb-jar.xml* file will then be portable to a different EJB container – except that it will not sort the beans as it should.

This redefinition of the finder method in the *weblogic-cmp-rdbms-jar.xml* file is also the location for WebLogic-specific tuning elements such as the `<include-updates>` element. This element controls whether or not bean changes occurring within the current transaction are reflected in the result set from the query. By default, WebLogic ignores such changes , thereby improving performance by removing the need to write all unsaved changes to the database before performing the query. See the online documentation for more information.

Empty all of the non-sequence workbook tables before proceeding to the next exercise.

Exercise for Chapter 10

Exercise 10.1: A BMP Entity Bean

In this exercise we will build and examine a simple EJB that uses bean-managed persistence (BMP) to synchronize the contents of the bean with the database.

Download and Build the Example Programs

Download and extract the *ex10_1* directory in the usual manner.

This example starts from scratch, retaining none of the Container-Managed Persistence (CMP) beans built in the previous examples.

Compare the *ShipBean.java* files for this exercise and for Exercise 7.2 and note some differences:

- The BMP version is much longer (270+ lines vs. 37 lines). This difference is not surprising because we are coding all of the persistence logic ourselves.
- Both versions have get and set methods for attributes, but the BMP version actually implements these functions and uses attributes defined in the class. The CMP version simply declares the methods `abstract` and leaves it to the CMP-generated subclass to implement the methods and store the values in attributes we never see – or ever need to see.
- Both versions define the same standard methods (`ejbLoad`, `ejbStore`, `ejbRemove`, `ejbCreate`, and `ejbPostCreate`). These functions are trivial placeholders in the CMP version but contain a great deal of Java code in the BMP version.
- The BMP version also implements the get and set methods for the `EntityContext`, while the CMP version simply defines empty placeholders.
- The BMP version must implement the two finder methods declared in the Ship home interface (`ejbFindByPrimaryKey` and `ejbFindByCapacity`), both of which the CMP code-generation process creates automatically in the CMP version.

Recognize that this bean is a trivial example, containing only three attributes, two finder methods, and no relationships. It is easy to see how the sheer amount of coding required to build a BMP entity bean drives most designers to use CMP whenever they can.

Use *ant dist* to build the example code and place the *titanejb.jar* file in the correct directory.

Create the Required Database Objects

The Ship bean in this exercise includes a new integer attribute called `capacity`, which was not present in the previous exercises.

If you already have a SHIP table, alter the schema to add a new column called CAPACITY with an integer data type. If you skipped the CMP exercises and are starting with no SHIP table, create one with the following schema:

```
CREATE TABLE SHIP
(
  ID INT PRIMARY KEY,
  NAME CHAR(30),
  TONNAGE DECIMAL (8,2),
  CAPACITY INT
)
```

To avoid duplicate-key problems, be sure the SHIP table is empty before running the example programs.

Examine the Standard EJB Descriptor File

Many portions of the standard descriptor file *ejb-jar.xml* are the same as in the version used in the CMP exercises. The first section defines the Ship bean:

```
<enterprise-beans>
   <entity>
      <description>This bean represents a cruise ship.
      </description>
      <ejb-name>ShipEJB</ejb-name>
      <home>com.titan.ship.ShipHomeRemote</home>
      <remote>com.titan.ship.ShipRemote</remote>
      <ejb-class>com.titan.ship.ShipBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <security-identity><use-caller-identity/>
      </security-identity>
      <resource-ref>
           <description>DataSource for the Titan database
           </description>
           <res-ref-name>jdbc/titanDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

Some features to note:

- Setting the <persistence-type> element to Bean informs the *ejbc* process that we will be using bean-managed persistence.

- There is no `<abstract-schema-name>` element.
- There are no `<cmp-field>` elements to define the attributes of the bean.
- The `<resource-ref>` section defines the information necessary for the bean to obtain a reference to a JDBC data source to perform SQL operations.

Note that we could eliminate the `<resource-ref>` section and have the bean code obtain a JDBC connection directly from the WebLogic JDBC pool, but doing so would have two very bad side effects:

- The bean code would not be portable.
- The container would not be aware of requests for a JDBC connection within the transaction, so the container would not coordinate the work properly and would not ensure that all SQL operations in the transaction are performed with a single JDBC connection.

The final section of *ejb-jar.xml* contains the standard elements defining transactional behavior and security parameters:

```
<assembly-descriptor>
   <security-role>
      <description>
         This role represents ...
      </description>
     <role-name>everyone</role-name>
   </security-role>
   <method-permission>
     <role-name>everyone</role-name>
     <method>
         <ejb-name>ShipEJB</ejb-name>
         <method-name>*</method-name>
     </method>
   </method-permission>
   <container-transaction>
     <method>
        <ejb-name>ShipEJB</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
 </assembly-descriptor>
```

Some points to note:

- We are back to using `everyone` as the role name rather than `Employees`.

- All methods on the Ship bean require a transaction, so unless one is already active the container will start one whenever a bean method is called.
- All database work done through the JDBC connection obtained using the `jdbc/titanDB` resource will be part of this container-created transaction, so the bean code need not include explicit begin, commit, or rollback logic.

Examine the WebLogic-Specific Files/Components

The only WebLogic-specific file in this exercise is the *weblogic-ejb-jar.xml* file. There is no CMP-related descriptor file in this exercise because we are using bean-managed persistence.

We need one new section in *weblogic-ejb-jar.xml*, to map the `jdbc/titanDB` resource defined *ejb-jar.xml* to the actual JDBC data source available in the container. The code in the ShipBean class that obtains a JDBC connection looks like this:

```
private Connection getConnection() throws SQLException {
    try {
      Context jndiCntx = new InitialContext();
      DataSource ds =
       (DataSource)jndiCntx.lookup("java:comp/env/jdbc/titanDB");
      return ds.getConnection();
    }
...
```

To map this connection to the *titan-dataSource* JDBC data source we created before we started working on the exercises, the `<weblogic-enterprise-bean>` section for the Ship bean must include the following:

```
<reference-descriptor>
    <resource-description>
        <res-ref-name>jdbc/titanDB</res-ref-name>
        <jndi-name>titan-dataSource</jndi-name>
    </resource-description>
</reference-descriptor>
```

The rest of the *weblogic-ejb-jar.xml* file is straightforward and has been discussed in previous exercises. Exercise 4.1 includes a complete walkthrough of this file in case you are starting with this exercise or need a review. Recognize that *weblogic-ejb-jar.xml* in the current exercise does not include the `<persistence>` elements required to configure container-managed persistence in Exercise 4.1 because we are using bean-managed persistence.

Deploy the EJB Components to WebLogic

Use the *ant dist* task to copy the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server as needed to deploy the new *titanejb.jar* file.

Use the console to verify that the Ship bean is properly deployed by checking for the `ShipHomeRemote` home interface in the JNDI tree.

Examine and Run the Client Applications

The single client example program in this exercise, *Client_101.java*, is patterned after the first CMP example, *Client_41.java*. We will examine the program in detail in case you are starting with this exercise.

The first section in Client_101.java creates a Ship bean by acquiring a reference to the home interface and invoking the two-parameter version of its `create` method:

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("ShipHomeRemote");
ShipHomeRemote home = (ShipHomeRemote)
    PortableRemoteObject.narrow(ref,ShipHomeRemote.class);

System.out.println("Creating Ship 101..");
ShipRemote ship1 =
    home.create(new Integer(101),"Edmund Fitzgerald");
```

Calling the `create` method of the home interface causes the container to invoke the four-parameter version of the `ejbCreate` method in the `ShipBean` class, which will write a row to the `SHIP` table in the database.

The next two lines call methods to set the bean's capacity and tonnage attributes to reasonable values:

```
ship1.setTonnage(50000.0);
ship1.setCapacity(300);
```

Recognize that because this code is not operating within an explicit transaction, each of these calls will cause an entire life cycle of the bean to occur in the container, including the following callbacks to our *ShipBean.java* code:

- `ejbLoad` to load Ship attributes from the database
- `setCapacity` or `setTonnage` method to modify the attribute
- `ejbStore` to store the Ship attributes in the database

When you run the client application, each method in *ShipBean.java* will write a message to the WebLogic server log, to let you see these callback methods being invoked by the container.

The next section uses the home interface's `findByPrimaryKey` method to acquire a reference to the Ship bean, which the container forwards to the `ejbFindByPrimaryKey` method in our *ShipBean.java* code:

```
Integer pk = new Integer(101);
ShipRemote ship2 = home.findByPrimaryKey(pk);
```

This new Ship reference is used to get the values of the bean attributes:

```
System.out.println(ship2.getName());
System.out.println(ship2.getTonnage());
System.out.println(ship2.getCapacity());
```

Again, we are not in a transaction, so each get method will cause a full life cycle of load and store callbacks. Examine the WebLogic server log after a run to confirm this behavior is actually occurring.

You might be surprised to see `ejbStore` calls after the "get" method calls in the log. Why does the container bother to call `ejbStore` when all we did was return the value of one bean attribute? It must do so because the container cannot see the attributes of the bean, it cannot tell if anything is different, and it is unwilling to trust that a method that starts with the word "get" has no side effects. When beans manage their own persistence, the container will call `ejbStore` at the end of *every* transaction.

We could improve the bean's performance in at least three ways:

- Create a method on the bean that returns multiple attributes, similar to the `Name` dependent-value class in Exercise 6-2. The container will still invoke `ejbStore` after each call to the get method, but at least returning multiple fields in a value object would reduce the number of calls.
- Add a Boolean "dirty" flag to the bean, have set methods set it to true, and have `ejbStore` check it to see whether a database operation is actually required. The container will still call `ejbStore` each time get is invoked, but we will avoid an expensive database hit until a set method is called. (Container-managed persistence uses essentially the same approach to solve the "unnecessary ejbStore problem" for us. CMP controls access to the bean's attributes through its get and set methods and keeps track of which set methods were called.)
- Place operations that work with the entity bean in a session bean that starts a transaction (either explicit or declarative) around all the get and set operations performed on the entity bean. The container will invoke the `ejbStore` method only at the end of the transaction, saving many database hits. Note that this tactic also reduces the number of `ejbLoad` calls.

Finally, we delete the bean using the `remove` method of the reference, which the container forwards to our `ejbRemove` method in the ShipBean class:

```
ship2.remove();
```

Run the client in the normal manner – first remembering to execute the *setEnv* command or shell script to set environment variables properly.

The client should output text similar to this:

```
C:\work\ejbbook\ex10_1>java com.titan.clients.Client_101
Creating Ship 101..
Finding Ship 101 again..
Edmund Fitzgerald
50000.0
300
Removing Ship 101..
```

The database will be empty after the client program is finished because the last step in the program removes the only bean.

Examine and Run the Client JSP Pages

The *ant dist* task should have copied the example JSP page to the *webapp* directory in the *titanapp* exploded .ear file, making it available at the standard URL:

http://servername:7001/webapp/Client_101.jsp

The *Client_101.jsp* page is identical to the *Client_101.java* program in operation, but dumps the database contents between steps so you can track their impact.

Make sure the `SHIP` table is empty before proceeding to the next exercise.

As an optional task in this exercise, implement the “dirty” flag solution from the list in the preceding section to reduce database hits. You will need to:

- Add a private `Boolean` attribute named `dirty` to the ShipBean class
- Modify all of the set methods to set `dirty` to `true` (only if old and new values are different, perhaps)

Modify ejbStore to save the bean attributes to the database only if `dirty` is `true`, and to reset the flag to `false` after updating the data.

Exercises for Chapter 12

Exercise 12.1: A Stateless Session Bean

In this exercise we will build and test a new stateless session EJB that writes payment information to the database during the booking process.

Download and Build the Example Programs

Download and extract the *ex12_1* directory in the normal manner.

This exercise reuses the `CustomerEJB` component from Exercise 6.3, along with the `Name` object, `AddressEJB` component, `HomeAddress` relationship, and `AddressDO` object from that exercise.

The new stateless session bean `ProcessPaymentEJB` has a public interface with three payment methods:

ProcessPaymentRemote.java

```
public interface ProcessPaymentRemote extends javax.ejb.EJBObject {

    public boolean byCheck(CustomerRemote customer,
                           CheckDO check, double amount)
      throws RemoteException, PaymentException;

    public boolean byCash(CustomerRemote customer, double amount)
      throws RemoteException, PaymentException;

    public boolean byCredit(CustomerRemote customer,
                            CreditCardDO card, double amount)
      throws RemoteException, PaymentException;
}
```

All three of these public methods eventually delegate the request to a single private method `process`, which performs the database operation. Because there is no Payment EJB with CMP-generated persistence logic, our `process` method must construct SQL statements and use JDBC calls to perform the database insert:

```
...
con = getConnection();
ps = con.prepareStatement
 ("INSERT INTO payment (customer_id, amount, type, check_bar_code,
check_number, credit_number, credit_exp_date)"+
     " VALUES (?,?,?,?,?,?,?)");
```

```
ps.setInt(1,customerID.intValue());
ps.setDouble(2,amount);
ps.setString(3,type);
ps.setString(4,checkBarCode);
ps.setInt(5,checkNumber);
ps.setLong(6,creditNumber);
ps.setDate(7,creditExpDate);
int retVal = ps.executeUpdate();
...
```

The use of stateless session beans to encapsulate and perform simple JDBC operations directly, without employing entity beans, is a very common practice in high-volume applications.

This example illustrates one other aspect of EJB communication: The public methods `byCheck`, `byCash`, and `byCredit` accept a `CustomerRemote` interface as an input parameter. Before calling any of them, the client must first acquire such a reference (using the `CustomerHomeRemote` interface and the appropriate "find" method). A remote client can pass a remote Customer reference to a method such as byCheck because the `CustomerRemote` object is a `Serializable` object that the underlying RMI communication process can pass by value.

Use *ant dist* to build the example code and place the *titanejb.jar* file in the correct directory.

Create the Required Database Objects

The ProcessPaymentEJB inserts new records in the PAYMENT table. Create this table in your database:

```
CREATE TABLE PAYMENT
(
    CUSTOMER_ID     INT,
    AMOUNT          DECIMAL(8,2),
    TYPE            CHAR(10),
    CHECK_BAR_CODE  CHAR(50),
    CHECK_NUMBER    INTEGER,
    CREDIT_NUMBER   CHAR(20),
    CREDIT_EXP_DATE DATE
)
```

Make sure the last four columns allow nulls, because the table will include rows for some payments in which these columns do not make sense; e.g., a payment by check has no `CREDIT_NUMBER`.

Note that there is no primary key in this table, nor will we be building an entity EJB to represent a Payment object.

Ensure that all non-sequence workbook tables are empty before running the example programs.

Examine the Standard EJB Descriptor File

You are probably becoming very familiar with the general structure and contents of the standard *ejb-jar.xml* descriptor file. The version in this exercise combines elements describing the ProcessPaymentEJB stateless session bean and the Customer EJB and related beans.

The first section contains elements describing the ProcessPayment EJB:

```
<session>
    <description>
        A service that handles monetary payments.
    </description>
    <ejb-name>ProcessPaymentEJB</ejb-name>
    <home>com.titan.processpayment.ProcessPaymentHomeRemote
    </home>
    <remote>com.titan.processpayment.ProcessPaymentRemote
    </remote>
    <ejb-class>com.titan.processpayment.ProcessPaymentBean
    </ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <env-entry>
         <env-entry-name>minCheckNumber</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>2000</env-entry-value>
    </env-entry>
    <resource-ref>
        <description>DataSource for the Titan database
        </description>
        <res-ref-name>jdbc/titanDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</session>
```

The `<session-type>` element declares the bean to be `Stateless`. The `<env-entry>` element defines the variable `minCheckNumber` and gives it the integer value “2000.” The ProcessPayment EJB can now obtain this threshold value using code like this:

```
private int getMinCheckNumber() {
    try {
        InitialContext jndiCntx = new InitialContext( );
        Integer value = (Integer)
            jndiCntx.lookup("java:comp/env/minCheckNumber");
        return value.intValue();
    } catch(NamingException ne){throw new EJBException(ne);}
}
```

The `<resource-ref>` element in this section makes the *titanDB* data source available to the ProcessPayment EJB. As you saw in Exercise 10.1, the bean gains access to the data source with syntax such as:

```
private Connection getConnection() throws SQLException {
    try {
        InitialContext jndiCntx = new InitialContext();
        DataSource ds = (DataSource)
            jndiCntx.lookup("java:comp/env/jdbc/titanDB");
        return ds.getConnection();
    } catch(NamingException ne){throw new EJBException(ne);}
}
```

The next few sections of the *ejb-jar.xml* descriptor file define the Customer and Address EJBs, their individual cmp fields, and the Customer-HomeAddress relationship. These sections are identical to their counterparts in Exercise 6.3.

Finally, the `<assembly-descriptor>` section defines the access rights and transactional properties of all methods of all beans. The role `everyone` may access all methods, and a transaction is required for all methods.

Examine the WebLogic-Specific Files/Components

The two WebLogic-specific descriptor files contain the normal entries for defining pool/cache parameters, security mapping, and the CMP-specific descriptor location. In addition, the `<reference-descriptor>` section for the bean in *weblogic-ejb-jar.xml* must include a `<resource-description>` element that matches the `<resource-ref>` element in *ejb-jar.xml*:

ejb-jar.xml

```
<resource-ref>
   <description>DataSource for the Titan database</description>
   <res-ref-name>jdbc/titanDB</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

weblogic-ejb-jar.xml

```
<reference-descriptor>
  <resource-description>
   <res-ref-name>jdbc/titanDB</res-ref-name>
   <jndi-name>titan-dataSource</jndi-name>
  </resource-description>
</reference-descriptor>
```

This mapping allows the code in the bean to use the "*java:comp/env/jdbc/titanDB*" syntax to refer to the data source we've registered as *titan-dataSource* in our WebLogic JNDI tree.

The *weblogic-cmp-rdbms-jar.xml* descriptor file contains the customary entries for field and relationship mapping for the Customer and Address EJB. This file is basically unchanged from Exercise 6.3.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server as needed to deploy the new *titanejb.jar* file.

Use the console to verify that all the example beans are properly deployed by checking for their home interfaces in the JNDI tree.

Examine and Run the Client Applications

The download provides two simple example programs, *Client_121.java* and *Client_122.java*.

Client_121.java

This program creates a single Customer bean for use in the next program. It is very similar to the the Client_63.jsp example page from Exercise 6.3. Run it from the command line after setting environment variables properly with *setEnv.cmd* or *setEnv.sh* as appropriate:

```
C:\work\ejbbook\ex12_1>setEnv
...
C:\work\ejbbook\ex12_1>java com.titan.clients.Client_121
Creating Customer 1..
Creating AddressDO data object..
Setting Address in Customer 1...
Acquiring Address data object from Customer 1...
Customer 1 Address data:
1010 Colorado
Austin,TX 78701

C:\work\ejbbook\ex12_1>
```

Client_122.java

Client_122 uses the new ProcessPayment EJB to insert rows in the `PAYMENT` table. We'll walk through this code in some detail to reinforce your understanding of the proper use of a stateless session bean.

First we obtain a reference to the JNDI context and look up the ProcessPayment and Customer home interfaces:

```
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("ProcessPaymentHome");
ProcessPaymentHomeRemote procpayhome = (ProcessPaymentHomeRemote)
 PortableRemoteObject.narrow(ref,ProcessPaymentHomeRemote.class);

ref = jndiContext.lookup("CustomerHome");
CustomerHomeRemote custhome = (CustomerHomeRemote)
 PortableRemoteObject.narrow(ref,CustomerHomeRemote.class);
```

Next we create a stateless session bean for our use and obtain a remote reference:

```
ProcessPaymentRemote procpay = procpayhome.create();
```

Because the public methods of the session bean require the Customer to be passed in the form of a `CustomerRemote` reference, we obtain a reference of that type from a finder method of CustomerHomeRemote – in this case, a reference to the Customer (ID=1) created by the previous example program:

```
CustomerRemote cust = custhome.findByPrimaryKey(new Integer(1));
```

Now we are ready to begin making calls to the ProcessPayment session bean. The first call will be to the `byCash` method which requires no additional information apart from the Customer reference and the amount of the payment:

```
procpay.byCash(cust,1000.0);
```

This call will insert a row in the `PAYMENT` table that has the proper values for a cash transaction.

The next call, to the `byCheck` method, requires a `CheckDO` parameter. We create a `CheckDO` object and pass it to the method:

```
CheckDO check = new CheckDO("010010101101010100011", 3001);
procpay.byCheck(cust,check,2000.0);
```

Note that the check number (3001) easily exceeds the minimum check number (2000) imposed in the `byCheck` method via the `minCheckNumber` entry in the descriptor file.

The next call, to the `byCredit` method, requires a `CreditCardDO` parameter. We create a dummy `CreditCardDO` object and pass it:

```
Calendar expdate = Calendar.getInstance();
expdate.set(2003,1,28); // month=1 is February
CreditCardDO credit = new CreditCardDO(
    "370000000000002",expdate.getTime(),"AMERICAN_EXPRESS");
procpay.byCredit(cust,credit,3000.0);
```

Like the other calls, this one inserts the appropriate row in the `PAYMENT` table and commits before returning to the client program.

Finally, we invoke `byCheck` again, this time using a check number that will not pass the method's minimum-value test:

```
CheckDO check2 = new CheckDO("111000100111010110101", 1001);
try {
    procpay.byCheck(cust,check2,9000.0);
}
catch (PaymentException pe) {
    System.out.println(
            "Caught PaymentException: "+pe.getMessage());
}
```

The `byCheck` method should raise the `PaymentException`, and the `catch` block here should catch and report it.

Note that each call to a method of the session bean creates and commits a separate container-managed transaction because we are using declarative transactions defined in the *ejb-jar.xml* file rather than creating a long-running explicit transaction spanning multiple calls. Therefore, database changes produced by completed calls will commit, regardless of errors in subsequent calls to the session bean.

The final step is to call the `remove` method to invalidate the reference and return the session bean to the "does not exist" state. This has very little effect in the case of a stateless session bean, but is a good practice nevertheless:

```
procpay.remove();
```

Examine the database after running *Client_122.java*. You should see three new rows in the `PAYMENT` table representing the successful payments made by this client program.

Examine and Run the Client JSP Pages

Two client JSP pages, *Client_121.jsp* and *Client_122.jsp*, duplicate the functions of the Java client applications described above.

The *ant dist* task should have copied the example JSP pages to the *webapp* directory in the *titanapp* exploded .ear file, making them available using the standard URLs:

http://servername:7001/webapp/Client_121.jsp

Make sure the `CUSTOMER` and `ADDRESS` tables are empty before proceeding to the next exercise.

Exercise 12.2: A Stateful Session Bean

If there is a "mother of all exercises" in this workbook, this is it. We'll be combining most of the EJB components created in preceding exercises into a single application that takes advantage of a new stateful session EJB, the TravelAgent bean.

- ❖ Note well: The code in this workbook will not match the examples in the EJB book as exactly as in all other exercises. We're going to take advantage of the EJB 2.0 capabilities of WebLogic and retain all of the relationships and beans created in Chapters 6 and 7 rather than simplifying some of the beans as shown in the EJB book.

Download and Build the Example Programs

Download and extract the *ex12_2* directory in the normal manner.

This exercise contains all of the EJB components from Exercise 7.3 with the following additions and changes:

- ♦ Augments the Customer EJB's local interfaces with remote home and bean interfaces.
- ♦ Has Customer use an `AddressDO` object to expose the Address EJB relationship in the remote interface.
- ♦ Includes the ProcessPayment stateless session EJB from Exercise 12.1.
- ♦ Adds a new TravelAgent stateful session EJB and a TicketDO value object.
- ♦ Adds to the Reservation EJB a new create method that takes multiple parameters.

Before we examine the descriptor files, let's walk through some important methods in the new TravelAgent EJB to point out items of interest.

As described in the EJB book, the TravelAgent EJB is a stateful session bean; i.e., one that maintains values of bean attributes between one method invocation and the next. Client applications call methods of the bean to set the desired Customer, Cruise, and Cabin values before invoking the `bookPassage` method to perform all of the activities related to booking.

The typical calling pattern for the TravelAgent EJB is therefore:

1. The caller creates a TravelAgent EJB using the create method that includes a `CustomerRemote` reference; the bean saves this reference in its `customer` attribute.
2. The caller invokes `setCruiseID` to attach a particular Cruise to this booking; the bean looks up the local reference to this Cruise EJB and saves the reference in the `cruise` attribute.
3. The caller invokes `setCabinID` to attach a particular Cabin to this booking; the bean looks up the local reference to this Cabin EJB and saves the reference in the `cabin` attribute.

4. The caller invokes the `bookPassage` method to perform the booking and return ticket information.

Examining the setCruiseID method will help you understand the process better:

```
public void setCruiseID(Integer cruiseID)
   throws javax.ejb.FinderException {

   try {
         CruiseHomeLocal home = (CruiseHomeLocal)
         jndiContext.lookup("java:comp/env/ejb/CruiseHome");
         cruise = home.findByPrimaryKey(cruiseID);
   } catch (NamingException ne) {
         throw new EJBException(ne);
   }
}
```

The method uses the `Integer` parameter `cruiseID` and the local home interface's `findByPrimaryKey` method to acquire a local reference to the proper Cruise bean.

Note that the method performs the JNDI lookup using the "*java:comp/env/ejb/CruiseHome*" syntax rather than simply specifying a JNDI name. This approach works only if *ejb-jar.xml* and *weblogic-ejb-jar.xml* include entries that make this resource available to the bean, and map it to the proper name in the JNDI tree.

Our `bookPassage` method is identical to the version in the EJB book:

```
public TicketDO bookPassage(CreditCardDO card,

                            double price)
    throws IncompleteConversationalState, RemoteException {

    if (customer == null || cruise == null || cabin == null)
    {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal reshome = (ReservationHomeLocal)
         jndiContext.lookup("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation = reshome.create(
                   customer, cruise, cabin, price, new Date());
        Object ref = jndiContext.lookup(
                  "java:comp/env/ejb/ProcessPaymentHome");

        ProcessPaymentHomeRemote ppHome =
            (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow(
                ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);
        process.remove();

        TicketDO ticket =
            new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch (Exception e) {
        throw new EJBException(e);
}
```

We create a Reservation bean using the current values of the `customer`, `cruise`, and `cabin` references, which must be non-null if we are to avoid an `IncompleteConversationalState` exception.

Creating a Reservation EJB inserts rows in the `RESERVATION`, `RESERVATION_CUSTOMER_LINK`, and `RESERVATION_CABIN_LINK` tables as needed to make the Reservation bean attributes and relationships persistent. These rows will not be committed to the database until the containing transaction commits.

Next, the `bookPassage` code invokes the `byCredit` method of the `ProcessPayment` stateless session bean to write the appropriate row to the `PAYMENT` table. Finally, `bookPassage` constructs a `TicketDO` value object containing textual information confirming the reservation and returns it to the caller.

Note that only after all steps in the booking process are complete and the method is ready to return to the caller does the Container commit the transaction it created automatically when bookPassage was invoked. All changes to the database are made permanent simultaneously.

The next method in the TravelAgent bean is `listAvailableCabins`, a public method clients call to obtain a list of the cabins available for a specific cruise.

In the workbook example programs, the database schema for the Reservation and Cabin EJBs includes a many-to-many link table `RESERVATION_CABIN_LINK`. Because a link table is used, the query in TravelAgent's `listAvailableCabins` method requires a slightly more complex `WHERE` clause than the one in the EJB book:

```
try {
    Integer cruiseID = (Integer)cruise.getPrimaryKey();
    Integer shipID = (Integer)cruise.getShip().getPrimaryKey();

    con = getConnection();
    ps = con.prepareStatement(
        "select ID, NAME, DECK_LEVEL from CABIN "+
        "where SHIP_ID = ? and BED_COUNT = ? and ID NOT IN "+
        "(SELECT RCL.CABIN_ID FROM RESERVATION_CABIN_LINK AS RCL,
          RESERVATION AS R "+
        " WHERE RCL.RESERVATION_ID = R.ID AND R.CRUISE_ID = ?)");
    ps.setInt(1,shipID.intValue());
    ps.setInt(2,bedCount);
    ps.setInt(3,cruiseID.intValue());
    result = ps.executeQuery();
    ...
```

The goal remains the same: Select only those Cabins which do not already have a related Reservation for the specified Cruise. Our SQL statement must join the `RESERVATION` and `RESERVATION_CABIN_LINK` tables to perform this test.

It would be instructive to build this same query as a “finder” or “ejbSelect” query in the Cabin EJB, using EJB QL. It is likely that the `MEMBER OF` operation would make traversing the relationships from Cruise to Reservation to Cabin a fairly simple operation. We leave this to the industrious reader as an optional exercise.

The final public method in *TravelAgentBean.java* is a utility method called `buildSampleData`. This method allows a remote client to build a set of Customer, Cabin, Cruise, and Ship objects for use in this exercise. It builds a set of interrelated beans using create and set methods as appropriate, and returns a collection of strings verifying the activity to the caller.

Use *ant dist* to build the example code and place the *titanejb.jar file* in the correct directory.

Create the Required Database Objects

This exercises requires no additional database tables.

The tables should not be modified as shown in the EJB Book for use by EJB 1.1-compliant containers because we’ve already created and tested in previous exercises all of the entity beans and relationships we need for this exercise.

As a reference, here are the correct schemas for all of the database tables, including the sequence tables we’re using for primary-key generation, expressed as Cloudscape DDL commands:

```
CREATE TABLE ADDRESS (
   ID int NOT NULL PRIMARY KEY,
   STREET varchar (40) NULL ,
   CITY varchar (20) NULL ,
   STATE varchar (2) NULL ,
   ZIP varchar (10) NULL
);

CREATE TABLE ADDRESS_SEQUENCE (
   SEQUENCE int NOT NULL
);

CREATE TABLE CABIN (
   ID int NOT NULL PRIMARY KEY,
   SHIP_ID int NULL ,
   BED_COUNT int NULL ,
   NAME varchar (30) NULL ,
   DECK_LEVEL int NULL
);

CREATE TABLE CREDIT_CARD (
   ID int NOT NULL PRIMARY KEY,
   EXP_DATE date NULL ,
   NUMBER varchar (20) NULL ,
   NAME varchar (40) NULL ,
   ORGANIZATION varchar (20) NULL ,
   CUSTOMER_ID int NULL
);

CREATE TABLE CREDIT_CARD_SEQUENCE (
   SEQUENCE int NOT NULL
);

CREATE TABLE CRUISE (
   ID int NOT NULL PRIMARY KEY,
   NAME varchar (30) NULL ,
   SHIP_ID int NULL
);

CREATE TABLE CRUISE_SEQUENCE (
   SEQUENCE int NULL
);
```

```
CREATE TABLE CUSTOMER (
   ID int NOT NULL PRIMARY KEY,
   LAST_NAME varchar (20) NULL ,
   FIRST_NAME varchar (20) NULL ,
   ADDRESS_ID int NULL ,
   HAS_GOOD_CREDIT boolean NULL
);

CREATE TABLE PAYMENT (
   customer_id int NOT NULL ,
   amount decimal(8, 2) NOT NULL ,
   type char (10) NOT NULL ,
   check_bar_code char (50) NULL ,
   check_number int NULL ,
   credit_number char (20) NULL ,
   credit_exp_date date NULL
);

CREATE TABLE PHONE (
   ID int NOT NULL PRIMARY KEY,
   NUMBER varchar (20) NULL ,
   TYPE int NULL ,
   CUSTOMER_ID int NULL
);

CREATE TABLE PHONE_SEQUENCE (
   SEQUENCE int NOT NULL
);

CREATE TABLE RESERVATION (
   ID int NOT NULL PRIMARY KEY,
   CRUISE_ID int NULL ,
   AMOUNT_PAID float NULL ,
   DATE_RESERVED date NULL
);

CREATE TABLE RESERVATION_CABIN_LINK (
   RESERVATION_ID int NOT NULL ,
   CABIN_ID int NOT NULL
);

CREATE TABLE RESERVATION_CUSTOMER_LINK (
   RESERVATION_ID int NOT NULL ,
   CUSTOMER_ID int NOT NULL
);
```

```
CREATE TABLE RESERVATION_SEQUENCE (
   SEQUENCE int NULL
);

CREATE TABLE SHIP (
   ID int NOT NULL PRIMARY KEY,
   NAME varchar (30) NULL ,
   TONNAGE float NULL ,
   CAPACITY int NULL
);
```

The following tables should have the ID column set as a unique primary key:

```
ADDRESS
CABIN
CREDIT_CARD
CRUISE
CUSTOMER
PHONE
RESERVATION
SHIP
```

The SQL scripts in the root *ejbbook* working directory contain statements that create these tables and primary-key constraints.

Ensure that all of the non-sequence workbook tables are empty before running the first example program.

Examine the Standard EJB Descriptor File

The standard descriptor file, *ejb-jar.xml*, has gotten very long in this exercise (550+ lines). Not surprising, considering our application has grown to include ten beans and eight relationships. We now need to include a number of `<ejb-ref>` sections to provide portable access to our beans from other beans.

The ProcessPaymentEJB section is the same as it was in Exercise 12.1, and all entity-bean sections of the *ejb-jar.xml* file are essentially unchanged from the version of the file used in Exercise 7.3. The few differences will be highlighted as we walk through the file.

The new TravelAgent stateful session bean is represented in a `<session>` section:

```
<session>
  <ejb-name>TravelAgentEJB</ejb-name>
  <home>com.titan.travelagent.TravelAgentHomeRemote</home>
  <remote>com.titan.travelagent.TravelAgentRemote</remote>
  <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>

  ... many ejb-ref and ejb-local-ref elements ...

  <resource-ref>
      <res-ref-name>jdbc/titanDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
  </resource-ref>
</session>
```

The elements shown above should be familiar by now. The `<session-type>` element declares this bean to be a `Stateful` session bean, and the `<resource-ref>` element makes the *jdbc/titanDB* data source available to it.

The sections left out of the listing above are a series of `<ejb-ref>` and `<ejb-local-ref>` elements that allow the TravelAgent EJB to look up the following home interfaces, using the `ejb/`*`Something`*`Home` syntax:

- ProcessPaymentEJB (remote home interface)
- CustomerEJB (remote)
- CabinEJB (local)
- ShipEJB (local)
- CruiseEJB (local)
- ReservationEJB (local)

The next section of the *ejb-jar.xml* file defines the CustomerEJB:

```
<entity>
  <ejb-name>CustomerEJB</ejb-name>
  <home>com.titan.customer.CustomerHomeRemote</home>
  <remote>com.titan.customer.CustomerRemote</remote>
  <local-home>com.titan.customer.CustomerHomeLocal</local-home>
  <local>com.titan.customer.CustomerLocal</local>
  <ejb-class>com.titan.customer.CustomerBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
```

```
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Customer</abstract-schema-name>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>lastName</field-name></cmp-field>
    <cmp-field><field-name>firstName</field-name></cmp-field>
    <cmp-field><field-name>hasGoodCredit</field-name></cmp-field>
    <primkey-field>id</primkey-field>
    <security-identity><use-caller-identity/></security-identity>
  </entity>
```

Did you catch the difference we've just introduced? In previous exercises the Customer EJB had either a remote or a local interface, but here we've provided both interfaces for the same bean class.

The two home-interface definitions are identical except for the basic differences in return types, exceptions, etc. we'd expect to see between a home local and a home remote interface:

CustomerHomeLocal.java

```
package com.titan.customer;

import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CustomerHomeLocal extends javax.ejb.EJBLocalHome {
    public CustomerLocal create(Integer id)
        throws CreateException;

    public CustomerLocal findByPrimaryKey(Integer id)
        throws FinderException;
}
```

CustomerHomeRemote.java

```
package com.titan.customer;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CustomerHomeRemote extends javax.ejb.EJBHome
{
    public CustomerRemote create(Integer id)
        throws CreateException, RemoteException;

    public CustomerRemote findByPrimaryKey(Integer id)
        throws FinderException, RemoteException;
}
```

Now compare the bean's local and remote interfaces. Are they identical like the home-interface definitions? Not by a long shot. Remember that it is illegal to expose a relationship or attribute based on a local interface in the remote interface for a bean. Therefore, Customer remote interface methods must:

1. Use value objects such as `Name` and `AddressDO`, *or*
2. Be helper methods like `addPhoneNumber`, *or*
3. Be get or set methods for primitive attributes of the Customer bean itself; e.g., `hasGoodCredit`.

CustomerRemote.java

```
package com.titan.customer;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.naming.NamingException;
import java.util.Vector;

public interface CustomerRemote extends javax.ejb.EJBObject {

    public void setAddress(String street, String city,
                           String state, String zip)
        throws RemoteException, CreateException, NamingException;
    public void setAddress(AddressDO address)
        throws RemoteException, CreateException, NamingException;

    public AddressDO getAddress() throws RemoteException;
    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;
    public void addPhoneNumber(String number, byte type)
        throws NamingException, CreateException, RemoteException;
    public void removePhoneNumber(byte typeToRemove)
        throws RemoteException;
    public void updatePhoneNumber(String number,
                                  byte typeToUpdate)
        throws RemoteException;
    public Vector getPhoneList() throws RemoteException;
    public boolean getHasGoodCredit() throws RemoteException;
    public void setHasGoodCredit(boolean flag)
        throws RemoteException;
}
```

The Customer local interface, on the other hand, can expose everything as a public method, including cmr relationships such as `reservations` and `phoneNumbers`:

CustomerLocal.java

```
package com.titan.customer;

import javax.ejb.CreateException;
import javax.naming.NamingException;
import java.util.Date;
import java.util.Vector;
import java.util.Collection;

public interface CustomerLocal extends javax.ejb.EJBLocalObject {

    public Name getName();
    public void setName(Name name);

    public boolean getHasGoodCredit();
    public void setHasGoodCredit(boolean flag);

    public void addPhoneNumber(String number, byte type)
        throws NamingException, CreateException;
    public void removePhoneNumber(byte typeToRemove);
    public void updatePhoneNumber(String number,
                                  byte typeToUpdate);
    public Vector getPhoneList();

    public AddressLocal getHomeAddress();
    public void setHomeAddress(AddressLocal address);

    public CreditCardLocal getCreditCard();
    public void setCreditCard(CreditCardLocal card);

    public Collection getPhoneNumbers();
    public void setPhoneNumbers(Collection phones);

    public Collection getReservations();
    public void setReservations(Collection reservations);
}
```

In a sense, the local interface is a superset of the methods available in the remote interface, adding additional methods for direct control of relationship fields.

Each component (client program, JSP page, other EJB, etc.) that needs to use the Customer EJB will obtain either the local or remote home interface, then use this to obtain either a local or remote reference to the desired bean.

❖ Important: Don't confuse multiple interfaces with multiple bean instances! Both references refer to the same business object and underlying database row, and calls to both interfaces will result in execution of the same *CustomerBean.java* code. Your choice of interface will, however, affect performance and remote-access capability, as discussed in Chapter 5 of the EJB book.

Referring back to the TravelAgentEJB section of the *ejb-jar.xml* file, you can see that we've decided to make the `CustomerHomeRemote` interface available to the bean for lookup in JNDI by including an `<ejb-ref>` element rather than an `<ejb-local-ref>` element. Somewhere in the TravelAgentEJB we must look up the home interface for the Customer EJB and we must require the remote interface rather than the local interface for some reason. You'll find the remote reference used in the `buildSampleData` method where we require a `CustomerRemote` reference in order to call the create method for Reservation beans.

Moving down *ejb-jar.xml* to the ReservationEJB section, you'll find an `<ejb-local-ref>` section in the definition for this entity bean:

```
<entity>
  <ejb-name>ReservationEJB</ejb-name>
  <local-home>com.titan.reservation.ReservationHomeLocal
  </local-home>
  <local>com.titan.reservation.ReservationLocal</local>
  <ejb-class>com.titan.reservation.ReservationBean</ejb-class>
  ...
  <primkey-field>id</primkey-field>
  <ejb-local-ref>
       <ejb-ref-name>ejb/CustomerHomeLocal</ejb-ref-name>
       <ejb-ref-type>Entity</ejb-ref-type>
       <local-home>
          com.titan.customer.CustomerHomeLocal
       </local-home>
       <local>com.titan.customer.CustomerLocal</local>
  </ejb-local-ref>
  <security-identity><use-caller-identity/></security-identity>
</entity>
```

Why would an entity bean like Reservation need to look up the local home interface for the Customer EJB? Recall that we've modified the Reservation bean to include a `create` method that accepts a `CustomerRemote` reference as a parameter. The Reservation EJB has a many-to-many relationship with Customer which should be initialized using the passed-in Customer remote reference, but CMP relationships always use local references, not remote references. To initialize the relationship, the Reservation bean's `create` method must convert the remote Customer reference to a local Customer reference.

To perform this conversion and obtain a local reference, the Reservation's `ejbPostCreate` method uses the primary key obtained from the passed-in Customer remote reference to look up the local reference for the same Customer. It then places that local reference in the relationship field `customers`, thereby linking this Reservation bean to the desired Customer bean:

ReservationBean.java

```
public void ejbPostCreate(CustomerRemote customer,
                          CruiseLocal cruise,
                          CabinLocal cabin, double price)
    throws javax.ejb.CreateException
{
    System.out.println("ReservationBean::ejbPostCreate");
    setCruise(cruise);
    // Our bean has many cabins, use the cmr set method here..
    Set cabins = new HashSet();
    cabins.add(cabin);
    this.setCabins(cabins);
    try {
        Integer primKey = (Integer)customer.getPrimaryKey();
        javax.naming.Context jndiContext = new InitialContext();
        CustomerHomeLocal home =
           (CustomerHomeLocal)jndiContext.lookup(
                "java:comp/env/ejb/CustomerHomeLocal");
        CustomerLocal custL = home.findByPrimaryKey(primKey);
        // Our bean has many customers, use the cmr set here..
        Set customers = new HashSet();
        customers.add(custL);
        this.setCustomers(customers);
    } catch (RemoteException re) {
        throw new CreateException(
            "Invalid Customer - Bad Remote Reference");
    } catch (FinderException fe) {
        throw new CreateException(
            "Invalid Customer - Unable to Find Local Reference");
    } catch (NamingException ne) {
        throw new CreateException(
"Invalid Customer - Unable to find CustomerHomeLocal Reference");
    }
}
```

The remaining sections of the *ejb-jar.xml* descriptor file contain the relationship and assembly information we've covered in previous exercises.

Examine the WebLogic-Specific Files/Components

The two WebLogic-specific descriptor files are also longer than the ones you've encountered before, weighing in at 320+ lines for *weblogic-ejb-jar.xml* and 360+ lines for the CMP descriptor file. For the most part, they just contain more of what you've encountered in previous exercises. We'll direct your attention to a few details, starting in the *weblogic-ejb-jar.xml* file.

The first thing to note in the *weblogic-ejb-jar.xml* file is that it includes a section that corresponds to each `<ejb-ref>` and `<ejb-local-ref>` section in the *ejb-jar.xml* file. The TravelAgentEJB section of *ejb-jar.xml* declared all the bean home interfaces that should be available to the TravelAgent bean:

ejb-jar.xml

```
<ejb-ref>
   <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>
     com.titan.processpayment.ProcessPaymentHomeRemote
   </home>
   <remote>
     com.titan.processpayment.ProcessPaymentRemote
   </remote>
</ejb-ref>
<ejb-ref>
   <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>
     com.titan.customer.CustomerHomeRemote
   </home>
   <remote>com.titan.customer.CustomerRemote</remote>
</ejb-ref>
<ejb-local-ref>
   <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local-home>
     com.titan.cabin.CabinHomeLocal
   </local-home>
   <local>com.titan.cabin.CabinLocal</local>
</ejb-local-ref>

...

<ejb-local-ref>
   <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local-home>
     com.titan.reservation.ReservationHomeLocal
   </local-home>
   <local>com.titan.reservation.ReservationLocal</local>
</ejb-local-ref>
```

weblogic-ejb-jar.xml contains a corresponding section, in the `<weblogic-enterprise-bean>` element for the TravelAgentEJB:

weblogic-ejb-jar.xml

```
<weblogic-enterprise-bean>

    <ejb-name>TravelAgentEJB</ejb-name>
    <stateful-session-descriptor>
        <stateful-session-cache>
            <max-beans-in-cache>100</max-beans-in-cache>
            <idle-timeout-seconds>300</idle-timeout-seconds>
        </stateful-session-cache>
    </stateful-session-descriptor>

    <reference-descriptor>
        <resource-description>
            <res-ref-name>jdbc/titanDB</res-ref-name>
             <jndi-name>titan-dataSource</jndi-name>
        </resource-description>
        <ejb-reference-description>
            <!-- Matches entry in ejb-jar.xml file -->
            <ejb-ref-name>ejb/ProcessPaymentHomeRemote
                </ejb-ref-name>
            <jndi-name>ProcessPaymentHomeRemote</jndi-name>
        </ejb-reference-description>
        <ejb-reference-description>
            <!-- Matches entry in ejb-jar.xml file -->
            <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
            <jndi-name>CustomerHomeRemote</jndi-name>
        </ejb-reference-description>
        <ejb-local-reference-description>
            <!-- Matches entry in ejb-jar.xml file -->
            <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
            <jndi-name>CabinHomeLocal</jndi-name>
        </ejb-local-reference-description>
        ...
        <ejb-local-reference-description>
            <!-- Matches entry in ejb-jar.xml file -->
            <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
            <jndi-name>ReservationHomeLocal</jndi-name>
        </ejb-local-reference-description>
    </reference-descriptor>

    <jndi-name>TravelAgentHomeRemote</jndi-name>
</weblogic-enterprise-bean>
```

Note that `<ejb-ref>` elements in *ejb-jar.xml* have corresponding `<ejb-reference-description>` elements here, while `<ejb-local-ref>` elements in *ejb-jar.xml* have matching `<ejb-local-reference-description>` elements here. These elements map the JNDI ENC lookup names back to the actual JNDI names WebLogic uses for these components.

As shown in the listing above, stateful session beans have additional descriptor elements you've not encountered before:

```
    <stateful-session-descriptor>
        <stateful-session-cache>
            <max-beans-in-cache>100</max-beans-in-cache>
            <idle-timeout-seconds>300</idle-timeout-seconds>
        </stateful-session-cache>
    </stateful-session-descriptor>
```

The `<max-beans-in-cache>` element defines the maximum number of instances available for immediate use in the instance pool. If necessary, the container will passivate inactive beans to allow for new requests. Passivated beans that remain inactive for the length of time specified in the `<idle-timeout-seconds>` element are subject to removal from disk storage. See the on-line documentation for more information on caching strategies used for stateful session beans.

The next interesting feature is in the CustomerEJB section. At the bottom you will see two separate JNDI-name elements:

```
<weblogic-enterprise-bean>

    <ejb-name>CustomerEJB</ejb-name>
    ...
    <jndi-name>CustomerHomeRemote</jndi-name>
    <local-jndi-name>CustomerHomeLocal</local-jndi-name>

</weblogic-enterprise-bean>
```

This descriptor is telling WebLogic to register both a remote and a local home interface in the JNDI tree, using the slightly different names specified in the respective elements. Beans will create reference elements in *ejb-jar.xml* and *weblogic-ejb-jar.xml* mapping JNDI ENC names to either the remote or local interface for the Customer EJB depending on their needs. For example, the previous section for TravelAgentEJB mapped the remote reference thus…

```
        <ejb-reference-description>
            <!-- Matches entry in ejb-jar.xml file -->
            <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
            <jndi-name>CustomerHomeRemote</jndi-name>
        </ejb-reference-description>
```

…while the ReservationEJB, which needs a local interface, mapped the JNDI ENC name to the local JNDI name thus:

```
        <ejb-local-reference-description>
            <!-- Matches entry in ejb-jar.xml file -->
            <ejb-ref-name>ejb/CustomerHomeLocal</ejb-ref-name>
            <jndi-name>CustomerHomeLocal</jndi-name>
        </ejb-local-reference-description>
```

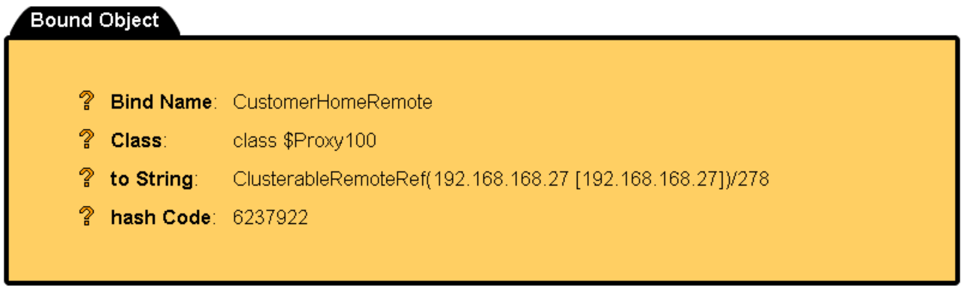
The rest of the *weblogic-ejb-jar.xml* file and everything in the *weblogic-cmp-rdbms-jar.xml* file are identical to the previous versions of these files from Exercise 7.3 and 12.1.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server to deploy the new *titanejb.jar* file.

If you bring up the WebLogic console for the domain, you should see both versions of the Customer home interface registered in the JNDI tree. To open the JNDI tree view for a server, click on **servers** in the navigation tree on the left to open the list of servers, right-click on the **myserver** node, and select **View JNDI tree**. If you click on the **CustomerHomeRemote** name, the right pane should display something like this:

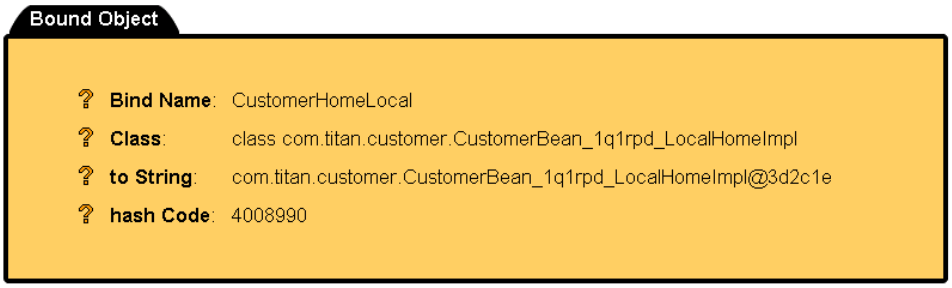
Figure 44:CustomerHomeRemote binding in JNDI tree



Bind Name: CustomerHomeRemote
Class: class $Proxy100
to String: ClusterableRemoteRef(192.168.168.27 [192.168.168.27])/278
hash Code: 6237922

If you click on **CustomerHomeLocal** in the list, you should see something like this:

Figure 45:CustomerHomeLocal binding in JNDI tree



Bind Name: CustomerHomeLocal
Class: class com.titan.customer.CustomerBean_1q1rpd_LocalHomeImpl
to String: com.titan.customer.CustomerBean_1q1rpd_LocalHomeImpl@3d2c1e
hash Code: 4008990

The important thing to note is that the object registered with the name `CustomerHomeLocal` is actually the WebLogic-generated `Customer_LocalHomeImpl` object itself, while the other registration is a "proxy" or "stub" object with information about the network address of the home implementation, etc. Obviously the local reference will be much faster and is more suitable for, say, caching in a session bean.

Examine and Run the Client Applications

Three client programs available in the download exercise the new TravelAgent EJB and the methods it makes available:

- *Client_125.java* uses the `buildSampleData` method to create Customer, Cabin, Ship, and Cruise objects for use in this exercise.
- *Client_126.java* uses the `bookPassage` method to reserve for a Customer a specified Cabin on a particular Cruise.
- *Client_127.java* lists available cabins for a specific Cruise having a desired number of beds using the `listAvailableCabins` method.

Client_125.java

This example program uses the buildSampleData method to create Customer, Cabin, Ship, and Cruise objects for use in this exercise. Ensure the non-sequence workbook tables in the database are empty before running this program.

When you run the *Client_125.java* program, it will display some information about the beans created by the `buildSampleData` method:

```
C:\work\ejbbook\ex12_2>java com.titan.clients.Client_125
Calling TravelAgentBean to create sample data..
Created customers with IDs 1 and 2..
Created ships with IDs 101 and 102..
Created cabins on Ship A with IDs 100-109
Created cabins on Ship B with IDs 200-209
Created cruises on ShipA with IDs 180, 181, 182
Created cruises on ShipB with IDs 183, 184, 185
Made reservation for Customer 1 on Cruise 180 for Cabin 103
Made reservation for Customer 1 on Cruise 185 for Cabin 208
Made reservation for Customer 2 on Cruise 181 for Cabin 105
Made reservation for Customer 2 on Cruise 185 for Cabin 202
```

Please note the primary key values the program uses for Cabin beans and Cruise beans associated with each of the two ships (the bold lines in the listing above). The program cannot control the Cruise IDs because they are generated automatically by the container-managed persistence code, using the `CRUISE_SEQUENCE` table in the database. When you execute the *Client_126* example program you will need to specify a Cruise ID and a Cabin ID that are related to a particular Ship. For example, Cruise 180 and Cabin 101 are both associated with Ship A.

Client_126.java

The *Client_126.java* example program uses the TravelAgent bean to demonstrate the steps required to book a cabin on a cruise. The program reads command-line arguments to determine the desired Customer, Cruise, Cabin, and booking price. The first three arguments must be the actual primary keys of the desired beans.

Examine the code in the *Client_126.java* file, paying particular attention to the steps required to book a cruise using the TravelAgent EJB. First we obtain references to the remote home interfaces of the Customer and TravelAgent beans:

```
Context jndiContext = getInitialContext();
Object obj = jndiContext.lookup("TravelAgentHomeRemote");
TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
  PortableRemoteObject.narrow(obj,TravelAgentHomeRemote.class);

obj = jndiContext.lookup("CustomerHomeRemote");
CustomerHomeRemote custhome = (CustomerHomeRemote)
  PortableRemoteObject.narrow(obj,CustomerHomeRemote.class);
```

Next, we use the first command-line argument to acquire a remote reference to the desired Customer bean:

```
Integer customerID = new Integer(args[0]);
CustomerRemote cust = custhome.findByPrimaryKey(customerID);
```

If this lookup fails, a `FinderException` will be thrown and the entire method will abort.

Next we create an instance of the TravelAgent stateful session bean and supply it with our Customer remote reference, passing it as a parameter to the `create` method:

```
TravelAgentRemote tagent = tahome.create(cust);
```

The TravelAgent EJB is a stateful session bean, not a stateless session bean, so the container directs the next two calls to the same instance of the bean:

```
tagent.setCruiseID(cruiseID);
tagent.setCabinID(cabinID);
```

These lines continue to modify the session bean's state, preparing it for the final `bookPassage` call to follow. The next few lines create a `CreditCardDO` object for use in booking the reservation (no, it's not my actual AMEX card so don't even try it):

```
Calendar expdate = Calendar.getInstance();
expdate.set(2003,1,5);
CreditCardDO card = new CreditCardDO(
    "370000000000002",expdate.getTime(),"AMERICAN EXPRESS");
```

Clearly, in a real application all these hard-coded items would be retrieved from the Customer bean, an HTML form, or through some other method, but the code would not change materially from the simple steps shown here.

Finally, the moment the Customer has been waiting for: the actual call to make the booking:

```
TicketDO ticket = tagent.bookPassage(card,price);
```

Review the code in the `bookPassage` method in *TravelAgentBean.java* if you need to refresh your memory about what it does. The return type is a value object containing a description of the booking, which we display proudly:

```
System.out.println("Result of bookPassage:");
System.out.println(ticket.description);
```

Don't forget to call `remove` on the reference to the stateful session bean:

```
tagent.remove();
```

This call invalidates our remote reference, ensuring that we don't use it again, and informs the container that we are done with the bean and it is free to use that "slot" in the instance pool for a different user's bean.

If you repeatedly forget to `remove` the reference, eventually the instance pool will fill up and the container will be forced to passivate beans that haven't been accessed recently, to make room for new requests. Finally, when the timeout period (controlled by the `<idle-timeout-secs>` element) for the bean is reached, the stateful bean is removed entirely by the container. The client will receive an exception if he or she attempts to use that bean instance again.

Here is a typical command line and the resulting output from the client program:

```
...\ex12_2>java com.titan.clients.Client_126 1 180 101 2000.0
Finding reference to Customer 1
Starting TravelAgent Session...
Setting Cruise and Cabin information in TravelAgent..
Booking the passage on the Cruise!
Ending TravelAgent Session...
Result of bookPassage:
Bob Smith has been booked for the Alaska Cruise cruise on ship
Nordic Prince.
 Your accommodations include Suite 101 a 1 bed cabin on deck level
1.
 Total charge = 2000.0
```

Note that the command line above specified a Cruise ID of 180 and a Cabin ID of 101. If you supply different command-line arguments, remember to check their values against the output of *Client_125* to ensure that you are giving *Client_126* valid Cruise and Cabin IDs. This program has been simplified to demonstrate core logic more clearly, and presents a variety of opportunities for error:

- There is no mechanism to select the reservation date or credit card information.
- Nothing prevents booking passage on a cruise on Ship A and a cabin on Ship B.
- Nothing prevents booking the same Cabin multiple times for the same cruise.

Experiment with the example program if you care to:

- Try booking a cruise using an expired credit card and verify that the exception raised in the ProcessPayment call made toward the end of `bookPassage` prevents the program from committing any of the booking process to the database.
- Find a way to avoid booking the same Cabin twice for the same Cruise. You might try creating a new method in the TravelAgent bean that checks for a conflict by joining `RESERVATION` and `RESERVATION_CABIN_LINK` in the manner you saw in the `listAvailableCabins` method. Have the method throw an exception if the desired Cabin is already booked for the specified Cruise, and have `bookPassage` handle the exception appropriately.

Client_127.java

This example demonstrates the `listAvailableCabins` method of the TravelAgent EJB. It accepts two command-line parameters, CruiseID and bed count, and displays a list of the Cabins having that bed count that are available for that Cruise:

```
C:\work\ejbbook\ex12_2>java com.titan.clients.Client_127 180 1
Starting TravelAgent Session...
Setting Cruise information in TravelAgent..
Ending TravelAgent Session...
Result of listAvailableCabins:
100,Suite 100,1
102,Suite 102,1
104,Suite 104,1
105,Suite 105,1
106,Suite 106,1
107,Suite 107,1
108,Suite 108,1
109,Suite 109,1
```

The code itself is very straightforward: We simply obtain a reference to a TravelAgent bean by calling `create` with a null Customer reference (we aren't planning to use it for a Customer, so this hurts nothing), set the Cruise reference in the bean, and invoke the `listAvailableCabins` method:

```
Integer cruiseID = new Integer(args[0]);
int bedCount = new Integer(args[1]).intValue();

Context jndiContext = getInitialContext();
Object obj = jndiContext.lookup("TravelAgentHome");
TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
  PortableRemoteObject.narrow(obj, TravelAgentHomeRemote.class);

TravelAgentRemote tagent = tahome.create(null);

tagent.setCruiseID(cruiseID);

String[] results = tagent.listAvailableCabins(bedCount);
```

If this seems like a strange way to find the cabins available, that's because it probably is. More common would be a stateless session bean with a method expecting the same two parameters, or perhaps a "finder" or "ejbSelect" method in the Cabin bean that accepts the required parameters and returns a collection of local references to Cabin beans. The primary purpose of these examples is to explore possible techniques for developing beans and queries rather than to recommend the best design pattern for a given problem.

Exercise 13.1 requires the sample data created in this exercise. Do not empty the database tables before proceeding to the next exercise.

Examine and Run the Client JSP Pages

Three client JSP pages, *Client_125.jsp, Client_126.jsp*, and *Client_127.jsp*, duplicate the functions of the Java client applications described above.

The *ant dist* task should have copied the example JSP pages to the *webapp* directory in the *titanapp* exploded .ear file, making them available using the standard URLs:

```
http://servername:7001/webapp/Client_125.jsp
```

Note that a JSP page running in the same JVM as the EJB components can access the local interface for beans directly, but these pages do not make use of this ability.

- ➢ If the flexibility to host the web application and EJB components on different servers is important to you, use remote interfaces in JSP pages and servlets.

As noted above, Exercise 13.1 requires the sample data created in this exercise. Do not empty the database tables before proceeding to the next exercise.

Exercises for Chapter 13

Exercise 13.1: JMS as a Resource

In this exercise we will modify the *TravelAgent* EJB to publish a simple text message on a JMS topic when a reservation is made.

Download and Build the Example Programs

Download and extract the *ex13_1* directory in the normal manner.

The `bookPassage` method in the *TravelAgent* EJB has been modified to publish a message in a JMS Topic when the reservation is complete:

TravelAgentBean.java

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
    ...
    String ticketDescription = ticket.toString();

    TopicConnectionFactory factory = (TopicConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/TopicFactory");

    Topic topic = (Topic)
        jndiContext.lookup("java:comp/env/jms/TicketTopic");

    TopicConnection connect = factory.createTopicConnection();
    TopicSession session = connect.createTopicSession(false,0);
    TopicPublisher publisher = session.createPublisher(topic);

    TextMessage textMsg = session.createTextMessage();
    textMsg.setText(ticketDescription);

    System.out.println(
        "Sending text message to jms/TicketTopic..");
    publisher.publish(textMsg);

    connect.close();

    return ticket;
}
```

Note the use of *jms/TopicFactory* and *jms/TicketTopic* in the JNDI lookup calls. Similar to previous lookups for *jdbc/titanDB* and other EJBs, these lookups require `<resource-ref>` tags in *ejb-jar.xml* as well as mapping tags in *weblogic-ejb-jar.xml*.

Also, note the parameters in the `createTopicSession` method call. Despite the specification which states that these parameters should be ignored for any JMS resource obtained using the JNDI ENC, WebLogic will not correctly publish the message unless the first parameter is `false`.

Use *ant dist* to build the example code and place the *titanejb.jar file* in the correct directory.

Configure the Required JMS Components

This exercise requires a number of new JMS components in the WebLogic *ejbbook* domain. We will walk you through the steps needed to create and configure the components, summarized as follows:

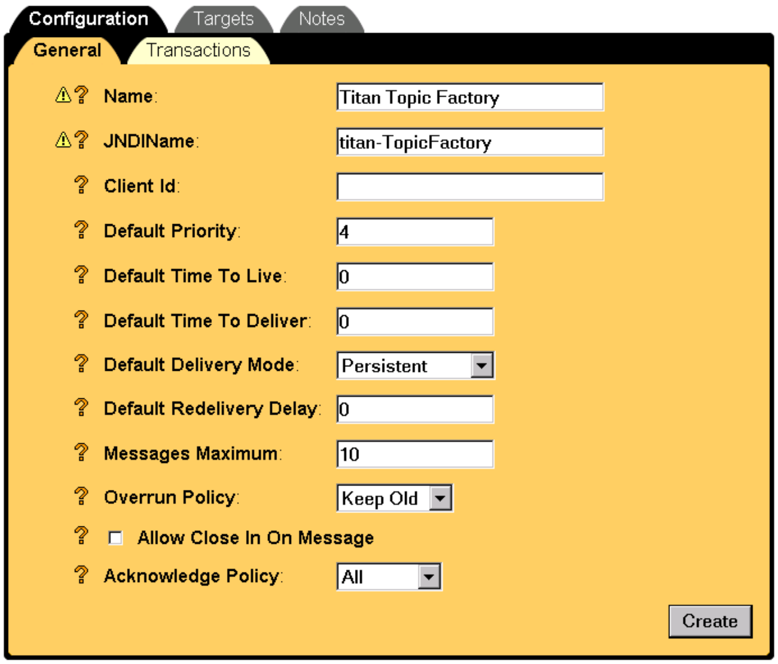
1. Create the *jmsstore* directory in the *ejbbook* domain root directory, for use by a JMSFileStore you will create.
2. Create a JMS Connection Factory called *Titan Topic Factory* with the JNDI name *titan-TopicFactory*.
3. Create a JMSFileStore called *TitanJMSStore*, which will use the *jmsstore* directory.
4. Create a JMSServer called *TitanJMSServer*, which uses the *TitanJMSStore*.
5. Configure a JMSTopic in this server called *Titan Ticket Topic* with the JNDI name *titan-TicketTopic*.

First, use Explorer or a command prompt to navigate to the *ejbbook* root directory for your domain. Create a new subdirectory of it called *jmsstore* to contain the persistent information for a JMS Store that you will be creating presently.

Next, boot the *ejbbook* domain and open the WebLogic console application. Open the **JMS** folder in the navigation pane on the left side of the console. Click on the **Connection Factories** folder and you should see an empty list of Connection Factories.

Click on the **Configure a new JMS Connection Factory…** link and fill out the resulting form:

Figure 46: Creating the Titan Topic Factory

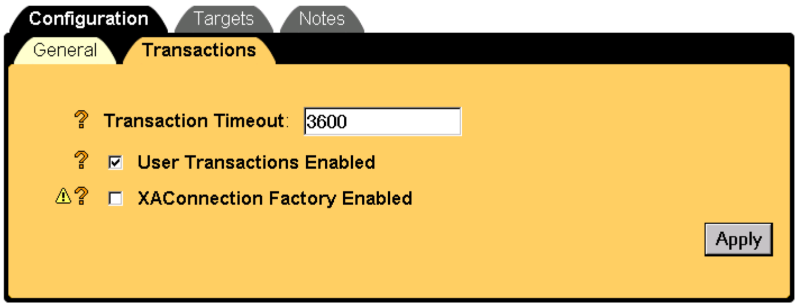


Name:	Titan Topic Factory
JNDIName:	titan-TopicFactory
Client Id:	
Default Priority:	4
Default Time To Live:	0
Default Time To Deliver:	0
Default Delivery Mode:	Persistent
Default Redelivery Delay:	0
Messages Maximum:	10
Overrun Policy:	Keep Old
☐ Allow Close In On Message	
Acknowledge Policy:	All

Be precise about the JNDI name *titan-TopicFactory* because the resource reference *jms/TopicFactory* is mapped to this JNDI name in the *ejb-jar.xml* file.

Click on **Create** to create the new factory. Next, select the **Transactions** tab and enable **User Transactions** for this factory:

Figure 47: Enabling User Transactions for Titan Topic Factory



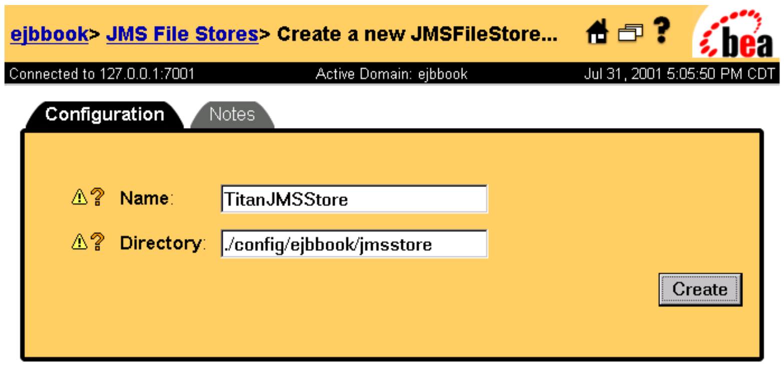
Transaction Timeout: 3600

☑ User Transactions Enabled

☐ XAConnection Factory Enabled

This option ensures that JMS activity using a connection obtained through this factory will be part of any container-managed or explicit transaction which might be active. Apply this change, then click on the **Targets** tab and move the **myserver** entry from the **Available** list to the **Chosen** list to make this factory available on the server. Apply this change. In the navigation pane's **JMS** area, click on the **Stores** folder. You should see an empty list of JMS Stores for the domain. We will be using a file-based JMS Store, so click on the link to **Configure a new JMSFile Store...** and fill out the resulting form:

Figure 48: Creating the Titan JMS Store


ejbbook> JMS File Stores> Create a new JMSFileStore...
Connected to 127.0.0.1:7001
Active Domain: ejbbook
Jul 31, 2001 5:05:50 PM CDT
Configuration
Notes
Create

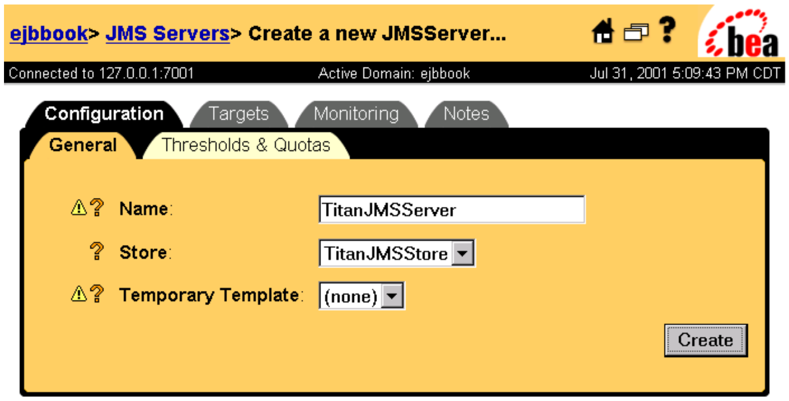
Name: TitanJMSStore

Directory: ./config/ejbbook/jmsstore

Click on **Create** to save this information. Verify you've created this new directory under the *ejbbook* directory before going on to create and target the JMS Server.

You can now create the JMS Server for the *ejbbook* domain. In the navigation pane's **JMS** area, click on the **Servers** folder. You should see an empty list of JMS Servers for this domain. Click on the link to **Configure a new JMS Server...** and fill out the resulting form:

Figure 49: Creating the Titan JMS Server


ejbbook> JMS Servers> Create a new JMSServer...
Connected to 127.0.0.1:7001
Active Domain: ejbbook
Jul 31, 2001 5:09:43 PM CDT
Configuration
Targets
Monitoring
Notes
General
Thresholds & Quotas
Create

Name: TitanJMSServer

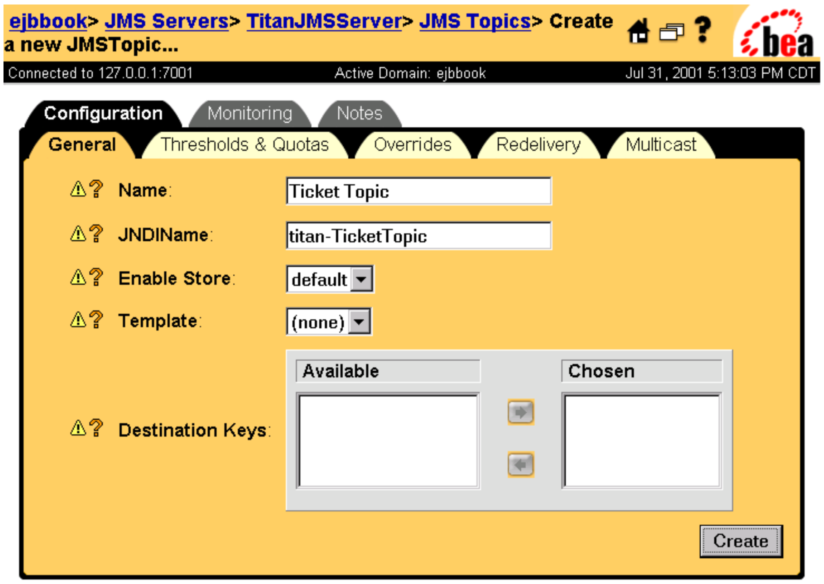
Store: TitanJMSStore

Temporary Template: (none)

Click on **Create** to continue. Click on the **Targets** tab and make sure this new JMSServer is targeted to the **myserver** server. Apply this change and you are ready to proceed to the next step.

Open the new **TitanJMSServer** folder in the navigation pane and click on the **Destinations** folder. An empty list of Topics and Queues should appear on the right side of the screen. You need to create the Topic for this exercise, so click on the link to **Configure a new JMSTopic...** and fill out the resulting form:

Figure 50: Creating the Ticket Topic



ejbbook> JMS Servers> TitanJMSServer> JMS Topics> Create a new JMSTopic...

Connected to 127.0.0.1:7001 Active Domain: ejbbook Jul 31, 2001 5:13:03 PM CDT

Configuration | Monitoring | Notes

General | Thresholds & Quotas | Overrides | Redelivery | Multicast

Name:	Ticket Topic
JNDIName:	titan-TicketTopic
Enable Store:	default
Template:	(none)
Destination Keys:	Available / Chosen

Create

Click on **Create** and the new Topic is ready for use. Note that the JNDI name for the Topic, *titan-TicketTopic,* will appear in the WebLogic-specific descriptor file as a mapping element.

Review all the JMS-related components you've created in the console. Verify that each of these is properly configured in your domain before attempting to deploy the new *titanejb.jar* file or run the example programs.

- JMS Connection Factory
  - Name: *Titan Topic Factory*
  - JNDI Name: *titan-TopicFactory*
  - Targeted to *myserver* server

- JMS Store
    - Name: *TitanJMSStore*
    - Type: *JMSFileStore*
    - Directory: *./config/ejbbook/jmsstore*
    - This directory exists in *ejbbook* root directory
- JMS Server
    - Name: *TitanJMSServer*
    - Store: *TitanJMSStore*
- JMS Destination
    - Name: *Ticket Topic*
    - Type: *JMSTopic*
    - JNDI Name: *titan-TicketTopic*

Examine the Standard EJB Descriptor File

The *ejb-jar.xml* file for this exercise is nearly identical to the file from Exercise 12.2; we've added only some additional `<resource-ref>` tags in the descriptor information for the *TravelAgent* EJB:

```
<session>
  <ejb-name>TravelAgentEJB</ejb-name>
  <home>com.titan.travelagent.TravelAgentHomeRemote</home>
  <remote>com.titan.travelagent.TravelAgentRemote</remote>
  <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
  ...
  <resource-ref>
   <res-ref-name>jms/TopicFactory</res-ref-name>
   <res-type>javax.jms.TopicConnectionFactory</res-type>
   <res-auth>Container</res-auth>
  </resource-ref>
  <resource-env-ref>
   <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
   <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
  </resource-env-ref>
</session>
```

Examine the WebLogic-Specific Files/Components

The WebLogic-specific descriptor files are also nearly identical to their counterparts in Exercise 12.2; we've added only some additional tags to map the generic resources specified in the *ejb-jar.xml* file to specific JNDI names in the domain:

weblogic-ejb-jar.xml

```
<weblogic-enterprise-bean>
    <ejb-name>TravelAgentEJB</ejb-name>
    ...
    <reference-descriptor>
        ...
        <resource-description>
            <res-ref-name>jms/TopicFactory</res-ref-name>
            <jndi-name>titan-TopicFactory</jndi-name>
        </resource-description>
        <resource-env-description>
            <res-env-ref-name>jms/TicketTopic</res-env-ref-name>
            <jndi-name>titan-TicketTopic</jndi-name>
        </resource-env-description>
        ...
    </reference-descriptor>
    <jndi-name>TravelAgentHome</jndi-name>
</weblogic-enterprise-bean>
```

This file needs no additional JMS-related descriptor elements to configure the JMS components. All of the configuration elements for the JMS components are stored in the domain configuration file, *config.xml*. Feel free to browse through this file and locate the elements related to the JMS Store, JMS Server, JMS Connection Factory, and JMS Topic created earlier through the Administration Console.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server as needed to deploy the new *titanejb.jar* file.

Examine the WebLogic server output during startup to ensure that no JMS-related error messages are displayed. If errors appear, read them carefully and make appropriate corrections to the components you just created.

Verify proper deployment of the new JMS components by examining the JNDI tree for the server in the normal manner and confirming that the new registrations *titan-TopicFactory* and *titan-TicketTopic* are present. If these components are not registered properly in the JNDI tree, check that the ConnectionFactory and JMSServer components are targeted to the *myserver* server.

Examine and Run the Client Applications

The client applications in this exercise require the sample data created in Exercise 12.2. If you deleted the sample data from the database, return to the Exercise 12.2 work root directory (*/work/ejbbook/ex12_2*) and run the *Client_125* program again to re-create the sample data.

This exercise contains a copy of the *Client_126.java* example program from the previous exercise, along with a working version of the example program in the EJB book, *JmsClient_1.java*.

The *Client_126.java* program is provided to enable you to create reservations for a specific Customer, Cruise, Cabin, and price using the same syntax as in Exercise 12.2:

```
java com.titan.clients.Client_126 1 180 101 2000.0
```

As in the preceding exercise, it is important that the Customer ID and Cruise ID represent valid data in the database created during the execution of *Client_125*. Run the *Client_126* example to verify that it continues to operate properly with the revised version of the *TravelAgent* EJB and produces the same output as before.

The *JmsClient_1.java* program you downloaded is patterned after the example in the EJB book. It creates the necessary JMS objects to become a subscriber to the *titan-TicketTopic* JMS Topic and waits indefinitely for messages to arrive:

```
public JmsClient_1() throws Exception {

    Context jndiContext = getInitialContext();

    TopicConnectionFactory factory = (TopicConnectionFactory)
        jndiContext.lookup("titan-TopicFactory");

    Topic topic = (Topic)jndiContext.lookup("titan-TicketTopic");

    TopicConnection connect = factory.createTopicConnection();

    TopicSession session =
      connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

    TopicSubscriber subscriber = session.createSubscriber(topic);

    subscriber.setMessageListener(this);

    System.out.println(
        "Listening for messages on titan-TicketTopic...");

    connect.start();
}
```

As shown in the listing, the download version of this program uses hard-coded factory and topic names rather than reading them from the command-line arguments, simply to reduce the potential for typographical errors when running the example.

When a message arrives, the `onMessage` method will be invoked:

```
public void onMessage(Message message) {
    try {
        TextMessage textMsg = (TextMessage)message;
        String text = textMsg.getText();
        System.out.println("\n RESERVATION RECEIVED:\n"+text);
    } catch(JMSException jmsE) {
        jmsE.printStackTrace();
    }
}
```

The program will extract and display the text from the JMS message.

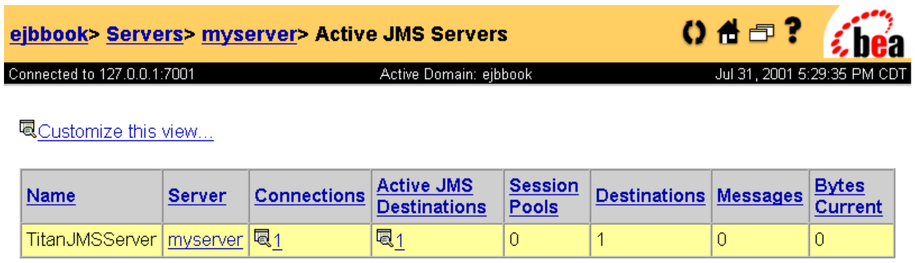
Open a new command prompt or telnet window, navigate to the *ex13_1* directory, set the environment variables properly, and run the *JmsClient_1* application using the standard command-line syntax:

```
C:\work\ejbbook\ex13_1>setenv
...
C:\work\ejbbook\ex13_1>java com.titan.clients.JmsClient_1
Listening for messages on titan-TicketTopic...
```

The client should report that it is listening for messages on *titan-TicketTopic* and then simply wait.

While the *JmsClient_1* application is running, open the WebLogic Administration Console and navigate to the **Services/JMS** page for *myserver*. Click on the link to **Monitor all Active JMS Servers...** to see a few statistics about our JMSServer:

Figure 51: Monitoring the Titan JMS Server


ejbbook> Servers> myserver> Active JMS Servers

Connected to 127.0.0.1:7001 Active Domain: ejbbook Jul 31, 2001 5:29:35 PM CDT

Customize this view...

Name	Server	Connections	Active JMS Destinations	Session Pools	Destinations	Messages	Bytes Current
TitanJMSServer	myserver	1	1	0	1	0	0

Note that WebLogic reports a single connection, our *JmsClient_1* program, and a single active destination, the Titan Ticket Topic. You may view additional information for these objects by clicking on the hourglass icon or number hyperlink in the table.

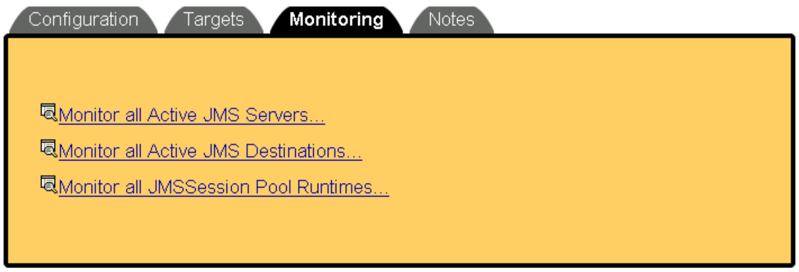
Leaving *JmsClient_1* running in its own window, run the *Client_126* example program from a different window to create a reservation. Shortly afterward you should see *JmsClient_1* display a confirmation message:

```
C:\work\ejbbook\ex13_1>java com.titan.clients.JmsClient_1
Listening for messages on titan-TicketTopic...

 RESERVATION RECEIVED:
Bob Smith has been booked for the Alaska Cruise cruise on ship
Nordic Prince.
 Your accommodations include Suite 100 a 1 bed cabin on deck level
1.
 Total charge = 999.0
```

The WebLogic console also allows monitoring of JMS Destinations in the domain. There are a variety of ways to navigate to the monitoring screens, but one easy technique is to navigate to the **TitanJMSServer** folder and click on the **Monitoring** tab:

Figure 52: Monitoring JMS Server Activity



Monitor all Active JMS Servers...

Monitor all Active JMS Destinations...

Monitor all JMSSession Pool Runtimes...

Click on the **Monitor all Active JMS Destinations...** link to see a list of the JMS destinations defined in this JMS Server and statistics regarding active message consumers, number of messages in queue, messages delivered, etc. Activate the automatic refresh feature for this window and watch these values change as you make additional reservations and start and stop *JmsClient_1* applications.

Optional uses of *Client_126* and this simple subscriber example program bring to light some characteristics of JMS Topics. Perform the activities if you are interested in learning more about JMS Topic behavior.

A JMS Topic does not save messages and does not care if there are any active subscribers when a message is published. To test this feature:

1. Stop the copy of *JmsClient_1* currently running so there is no active destination listening on the *titan-TicketTopic* JMS Topic.
2. In the Administration Console, verify that the subscriber/consumer count is zero (use the refresh icon if necessary).
3. Run *Client_126* to create a reservation and publish a message on the Topic. The *TravelAgent* EJB will happily publish a message on the Topic even though it has no active listeners.
4. Start *JmsClient_1* again. The program will subscribe to the Topic – but too late to pick up the message sent in step 3.

A JMS Topic can have multiple subscribers, all of which will receive the message published to the Topic. To see this behavior:

1. Start multiple copies of *JmsClient_1* in separate command-prompt or telnet windows.
2. Verify that the Administration Console reports multiple active consumers on this Topic.
3. Run the *Client_126* example program to create a reservation and publish a message on the Topic.
4. Check the active *JmsClient_1* programs to see that all display the same reservation confirmation.

Examine and Run the Client JSP Pages

There are no JSP-based examples in this exercise. The *Client_126.jsp* should still be available in the web application if you find it more convenient for creating reservations.

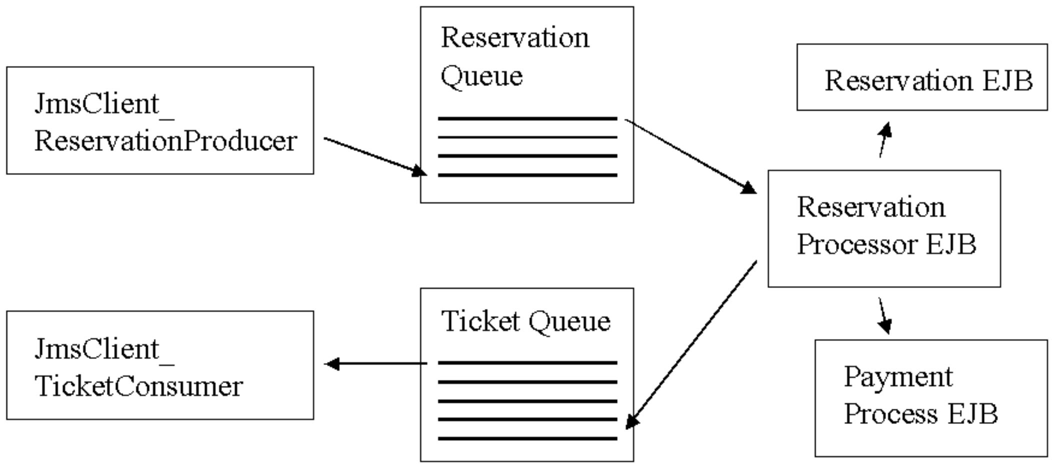
Exercise 13.2: The Message-Driven Bean

The final exercise in this workbook demonstrates three additional features of JMS and EJB 2.0:

- Configuration and use of a JMS Queue
- Message-Driven Beans listening on a JMS Queue
- JMS Reply-To Queues

The number of JMS Queues, EJB components, and client programs required to implement this exercise might seem a little overwhelming at first. The basic approach is summarized in the following figure:

Figure 53: Basic approach for message-driven bean example


JmsClient_ ReservationProducer
Reservation Queue
Reservation EJB
Reservation Processor EJB
JmsClient_ TicketConsumer
Ticket Queue
Payment Process EJB

1. The *JmsClient_ReservationProducer* program places a reservation request in the Reservation JMS Queue.
2. The Reservation Processor message-driven bean processes the request using the EJBs created in previous exercises and places a `TicketDO` message in the Ticket JMS Queue.
3. The *JmsClient_TicketConsumer* client program listens for this message and displays confirmation information.

This exercise will also give you an opportunity to test the performance of your EJB environment and examine the behavior of WebLogic Server under load.

Download and Build the Example Programs

Download and extract the *ex13_2* directory in the usual manner.

A single new EJB component is added in this exercise, the *ReservationProcessor* Message-Driven Bean. This EJB is designed to perform essentially the same booking activity as the *TravelAgent* EJB from Exercise 12.2, with the following changes:

- Booking activity occurs when a message containing all of the required data is placed in a Reservation Queue.
- The resulting `TicketDO` object is placed in an `ObjectMessage` on a reply queue.

As described in the EJB book, the *ReservationProcessor* bean implements two interfaces: `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener`. It contains two important methods required to implement these interfaces:

ReservationProcessorBean.java

```
public void setMessageDrivenContext(MessageDrivenContext mdc){
    ejbContext = mdc;
    try {
       jndiContext = new InitialContext();
    } catch(NamingException ne) {
       throw new EJBException(ne);
    }
}

public void onMessage(Message message) {
    try {
        MapMessage reservationMsg = (MapMessage)message;

        Integer customerPk = (Integer)
                reservationMsg.getObject("CustomerID");
        ...
}
```

The container will call the `onMessage` method when a JMS Client application places a message in the queue associated with this message-driven bean. The bean extracts fields from the message, acquires references to the required *Customer*, *Cruise*, and *Cabin* EJBs, creates a temporary `CreditCardDO` object, and performs the same business logic as the `bookPassage` method in the *TravelAgent* EJB:

```
ReservationHomeLocal resHome = (ReservationHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/ReservationHome");

ReservationLocal reservation =
    resHome.create(customer, cruise, cabin, price, new Date());

Object ref =
    jndiContext.lookup("java:comp/env/ejb/ProcessPaymentHome");

ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

ProcessPaymentRemote process = ppHome.create();

process.byCredit(customer, card, price);

TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
```

Rather than return the resulting `TicketDO` object, however, `onMessage` calls a helper method to deliver the confirmation information to an interested recipient:

```
deliverTicket(reservationMsg, ticket);
```

The `deliverTicket` method performs all of the steps required to open a connection to a JMS Queue and place the `TicketDO` in an `ObjectMessage` on the queue:

```
public void deliverTicket(MapMessage reservationMsg,
                          TicketDO ticket)
throws NamingException, JMSException {

    // create a ticket and send it to the proper destination
    Queue queue = (Queue)reservationMsg.getJMSReplyTo();
    QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/QueueFactory");
    QueueConnection connect = factory.createQueueConnection();
    QueueSession session = connect.createQueueSession(false,0);
    QueueSender sender = session.createSender(queue);
    ObjectMessage message = session.createObjectMessage();
    message.setObject(ticket);

    sender.send(message);

    connect.close();
}
```

This method obtains the *ConnectionFactory* using the typical JNDI ENC lookup with the resource *jms/QueueFactory*, but where is the JMS Queue name in this code? Rather than hard-coding the reply Queue name in the `deliverTicket` method, the *JmsClient_ReservationProducer* program specifies the Queue to use for `TicketDO` delivery as a "Reply-To" Queue within the incoming reservation message itself:

JmsClient_ReservationProducer.java

```
Queue ticketQueue =
    (Queue)jndiContext.lookup("titan-TicketQueue");
...
MapMessage message = session.createMapMessage();
message.setJMSReplyTo(ticketQueue);
```

This powerful feature of JMS allows each message to specify the proper reply queue, rather than leaving this decision in the hands of the *Reservation Processor* EJB.

The details of the two client programs will be covered in a later section in this exercise.

Use *ant dist* to build the example code and place the *titanejb.jar* file in the correct directory.

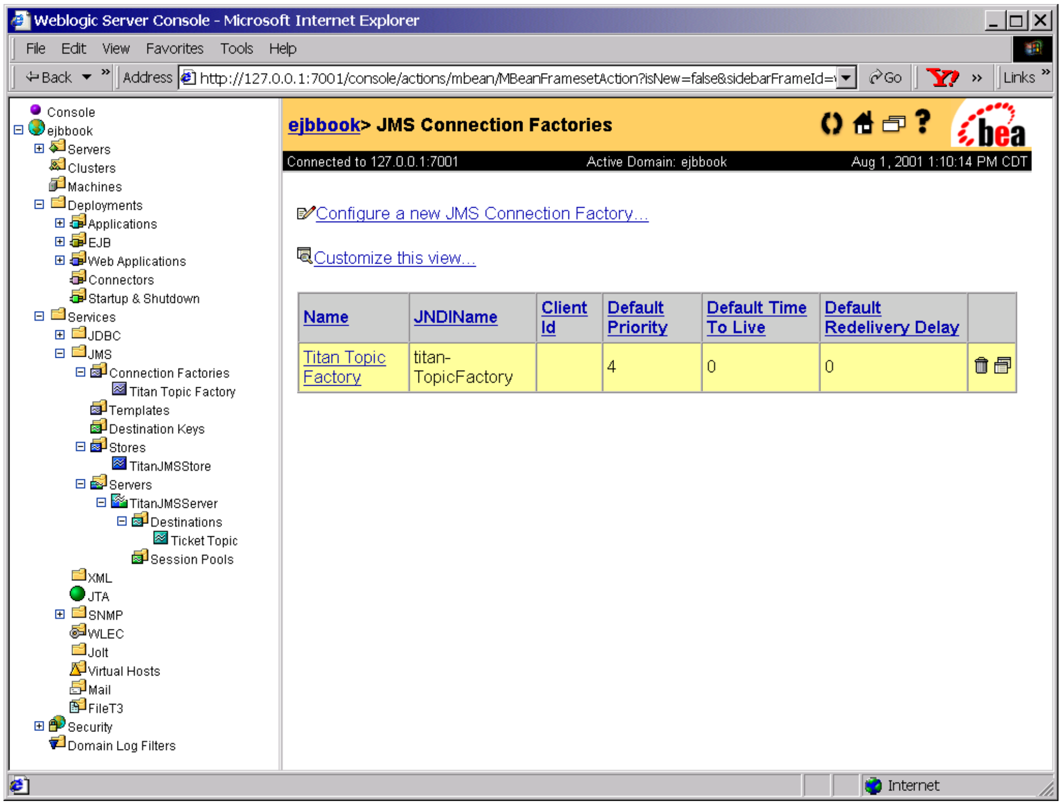
Configure the Required JMS Components

This exercise uses two new JMS Queue components and a new JMS ConnectionFactory. We will walk you through the steps needed to create and configure the components in WebLogic, summarized as follows:

1. Create a JMS Connection Factory called *Titan Queue Factory* with the JNDI name *titan-TopicFactory*.
2. Configure a JMSQueue in the TitanJMSServer server called *Reservation Queue* with the JNDI name *titan-ReservationQueue*.
3. Configure a JMSQueue in the TitanJMSServer server called *Ticket Queue* with the JNDI name *titan-TicketQueue*.

First, boot the *ejbbook* domain and open the WebLogic console application. In the navigation pane on the left side of the console, open the **JMS** folder and click on the **Connection Factories** folder and you should see the Connection Factory you created in the last exercise:

Figure 54: Preparing to createl a new Connection Factory



ejbbook> JMS Connection Factories

Connected to 127.0.0.1:7001 Active Domain: ejbbook Aug 1, 2001 1:10:14 PM CDT

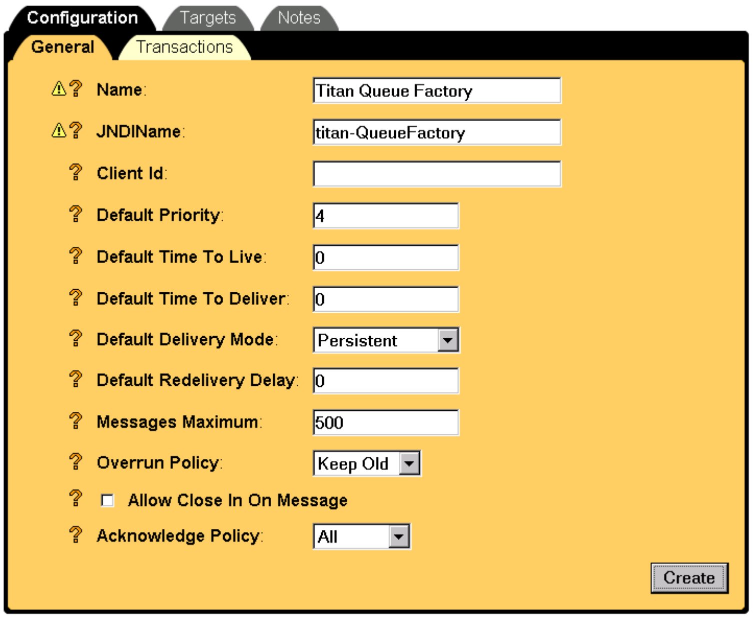
Configure a new JMS Connection Factory...

Customize this view...

Name	JNDIName	Client Id	Default Priority	Default Time To Live	Default Redelivery Delay	
Titan Topic Factory	titan-TopicFactory		4	0	0	

Click on the link to **Configure a new JMS Connection Factory...** and fill out the resulting form:

Figure 55: Creating the Titan Queue Factory

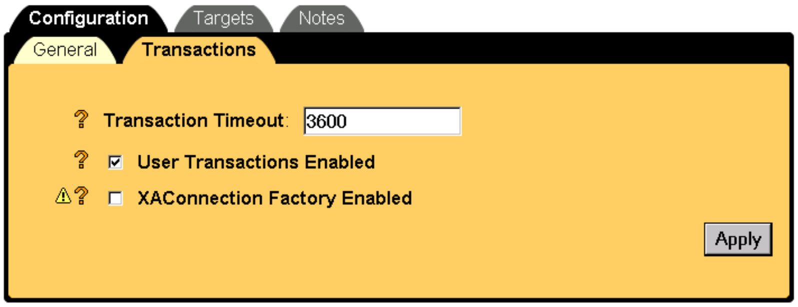


Name:	Titan Queue Factory
JNDIName:	titan-QueueFactory
Client Id:	
Default Priority:	4
Default Time To Live:	0
Default Time To Deliver:	0
Default Delivery Mode:	Persistent
Default Redelivery Delay:	0
Messages Maximum:	500
Overrun Policy:	Keep Old
☐ Allow Close In On Message	
Acknowledge Policy:	All

We recommend setting the **Messages Maximum** field higher than the default value of 10, to allow the server to accept a larger number of JMS messages while others in the queue are awaiting delivery to a listener. Click on **Create** to create this new JMS Connection Factory.

Next, select the **Transactions** tab and, as before, enable **User Transactions** for this factory:

Figure 56: Enabling User Transactions for Titan Queue Factory

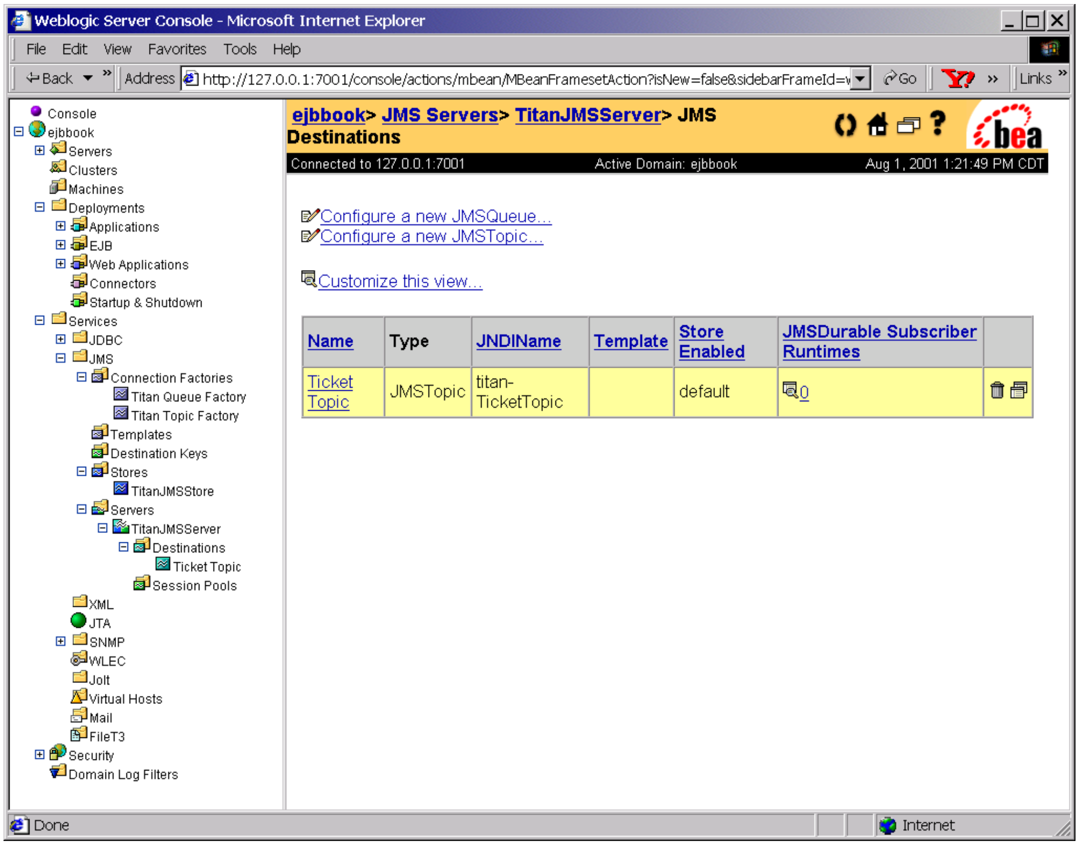


Transaction Timeout:	3600
☑ User Transactions Enabled	
☐ XAConnection Factory Enabled	

This option ensures that JMS activity using a connection obtained through this factory will be part of any container-managed or explicit transaction which might be active. Use the **Targets** tab to assign this factory to the server *myserver* before continuing to the next step.

Next, in the navigation pane open the **JMS** folder and navigate to the JMS Server you created in the last exercise (*TitanJMSServer*). Open this JMS Server and click on the **Destinations** folder in the navigation pane. You should see the JMS Topic you built in the last exercise:

Figure 57: Preparing to create new Queues



ejbbook> JMS Servers> TitanJMSServer> JMS Destinations

Connected to 127.0.0.1:7001 Active Domain: ejbbook Aug 1, 2001 1:21:49 PM CDT

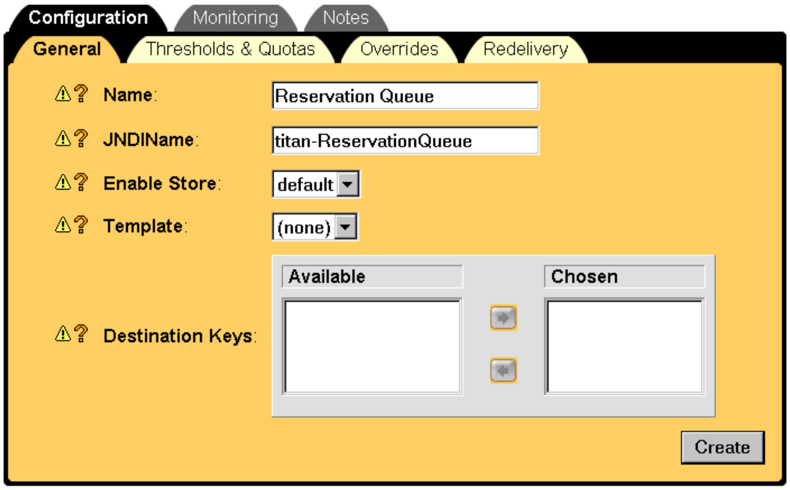
Configure a new JMSQueue...
Configure a new JMSTopic...

Customize this view...

Name	Type	JNDIName	Template	Store Enabled	JMSDurable Subscriber Runtimes	
Ticket Topic	JMSTopic	titan-TicketTopic		default	0	

You need to create another JMS Queue, so click on the **Configure a new JMSQueue…** link and fill out the form:

Figure 58: Creating the Reservation Queue

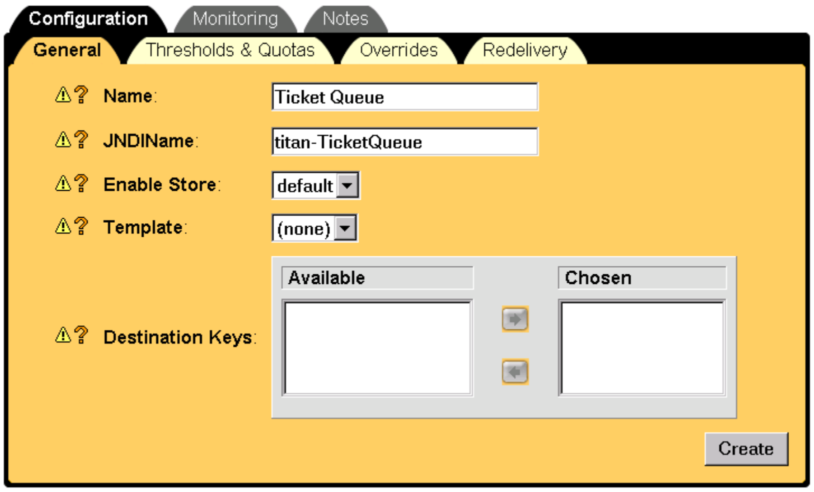


Name:	Reservation Queue
JNDIName:	titan-ReservationQueue
Enable Store:	default
Template:	(none)
Destination Keys:	

Click on **Create** and the new Queue is ready for use. Note that its JNDI name, *titan-ReservationQueue*, will appear in the WebLogic-specific descriptor file as a mapping element, so be sure spelling and capitalization are as in the figure.

The *ReservationProcessor* EJB responds to the booking request by placing a message on a new Ticket Queue, rather than publishing it on the Ticket Topic we used in the previous exercise. Click on the **Destinations** folder one more time, choose the **Configure a new JMSQueue…** link, and configure the Ticket JMS Queue as shown here:

Figure 59: Creating the Ticket Queue



Name: Ticket Queue
JNDIName: titan-TicketQueue
Enable Store: default
Template: (none)
Destination Keys:

Review the list of JMS-related components you've created in the Console. Verify that each of these is properly configured in your domain before attempting to deploy the new *titanejb.jar* file or run the example programs. This list includes the components created in the last exercise as well as the new ones:

- JMS Connection Factory
  - Name: *Titan Topic Factory*
  - JNDI Name: *titan-TopicFactory*
  - Targeted to *myserver* server
- JMS Connection Factory
  - Name: *Titan Queue Factory*
  - JNDI Name: *titan-QueueFactory*
  - Targeted to *myserver* server
- JMS Store
  - Name: *TitanJMSStore*
  - Type: *JMSFileStore*
  - Directory: *./config/ejbbook/jmsstore*
  - This directory exists in *ejbbook* root directory

- JMS Server
    - Name: *TitanJMSServer*
    - Store: *TitanJMSStore*
- JMS Destination
    - Name: *Ticket Topic*
    - Type: *JMSTopic*
    - JNDI Name: *titan-TicketTopic*
- JMS Destination
    - Name: *Reservation Queue*
    - Type: *JMSQueue*
    - JNDI Name: *titan-ReservationQueue*
- JMS Destination
    - Name: *Ticket Queue*
    - Type: *JMSQueue*
    - JNDI Name: *titan-TicketQueue*

Examine the Standard EJB Descriptor File

A single new section in the *ejb-jar.xml* descriptor file specifies deployment information for the new *ReservationProcessor* EJB:

```
<message-driven>
    <ejb-name>ReservationProcessorEJB</ejb-name>
    <ejb-class>
      com.titan.reservationprocessor.ReservationProcessorBean
    </ejb-class>
    <transaction-type>Container</transaction-type>
    <message-selector>MessageFormat = 'Version 3.4'
    </message-selector>
    <acknowledge-mode>auto-acknowledge</acknowledge-mode>
    <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
    </message-driven-destination>
    ...
</message-driven>
```

The missing `<ejb-ref>` and `<resource-ref>` elements will be discussed after some observations about these initial elements:

- The bean has no home or remote interface. Message-driven beans cannot be invoked through any direct mechanism and have no need for create or find methods.
- The `<message-selector>` element instructs the container to pass to this bean only those messages that have a property called `MessageFormat`, with a value equal to `Version 3.4`.
- This bean listens on a JMS Queue, but the queue name itself is not defined in this descriptor file.

Next in the descriptor are `<ejb-ref>` tags providing access to the entity and stateless-session beans required to process a reservation:

```
<ejb-ref>
    <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>
        com.titan.processpayment.ProcessPaymentHomeRemote
    </home>
    <remote>
        com.titan.processpayment.ProcessPaymentRemote
    </remote>
</ejb-ref>
<ejb-ref>
    <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>
        com.titan.customer.CustomerHomeRemote
    </home>
    <remote>com.titan.customer.CustomerRemote</remote>
</ejb-ref>
<ejb-local-ref>
    <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>
        com.titan.cruise.CruiseHomeLocal
    </local-home>
    <local>com.titan.cruise.CruiseLocal</local>
</ejb-local-ref>
```

```
    <ejb-local-ref>
        <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>
            com.titan.cabin.CabinHomeLocal
        </local-home>
        <local>com.titan.cabin.CabinLocal</local>
    </ejb-local-ref>
    <ejb-local-ref>
        <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>
            com.titan.reservation.ReservationHomeLocal
        </local-home>
        <local>com.titan.reservation.ReservationLocal</local>
    </ejb-local-ref>
```

As in the previous exercises, these elements must have corresponding elements in the *weblogic-ejb-jar.xml* descriptor file to provide the mapping to actual JNDI names.

The next element defines the default security context the bean should use when making calls to other EJBs in the environment:

```
<security-identity>
   <run-as><role-name>everyone</role-name></run-as>
</security-identity>
```

Finally, there is a `<resource-ref>` tag to allow the bean to access the JMS Queue Connection Factory using the JNDI ENC syntax:

```
<resource-ref>
   <res-ref-name>jms/QueueFactory</res-ref-name>
   <res-type>javax.jms.QueueConnectionFactory</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

This element will require a mapping element in the *weblogic-ejb-jar.xml* file to define the actual factory name in the JNDI tree.

The rest of the *ejb-jar.xml* file is the same as in the previous exercise.

Examine the WebLogic-Specific Files/Components

The *weblogic-ejb-jar.xml* file contains a new section for the new *ReservationProcessor* EJB, but is otherwise unchanged from the previous exercise. The new section defines deployment-specific mapping information for the JNDI ENC lookup values and WebLogic-specific elements defining pool behavior and security information:

```
<weblogic-enterprise-bean>
  <ejb-name>ReservationProcessorEJB</ejb-name>
  <message-driven-descriptor>
    <pool>
      <max-beans-in-free-pool>50</max-beans-in-free-pool>
      <initial-beans-in-free-pool>5</initial-beans-in-free-pool>
    </pool>
    <destination-jndi-name>titan-ReservationQueue
    </destination-jndi-name>
  </message-driven-descriptor>

  <reference-descriptor>
    <resource-description>
      <res-ref-name>jms/QueueFactory</res-ref-name>
      <jndi-name>titan-QueueFactory</jndi-name>
    </resource-description>
    <ejb-reference-description>
      <!-- Matches entry in ejb-jar.xml file -->
      <ejb-ref-name>ejb/ProcessPaymentHome</ejb-ref-name>
      <jndi-name>ProcessPaymentHome</jndi-name>
    </ejb-reference-description>
    ...
  </reference-descriptor>

  <run-as-identity-principal>guest</run-as-identity-principal>

  <jndi-name>ReservationProcessor</jndi-name>

</weblogic-enterprise-bean>
```

Interesting items in this section include:

- The `<max-beans-in-free-pool>` element, which limits the number of simultaneous message-driven bean instances to 50, thereby limiting our concurrent message-handling capability to this same value.
- The `<destination-jndi-name>` element, which defines the actual JMS Queue on which this message-driven bean should be listening.
- The `<run-as-identity-principal>` element, which WebLogic requires because the role name we chose in the `<run-as>` element in the *ejb-jar.xml* descriptor file actually maps to multiple principals, and the bean must perform its activity as a single principal.
- The `<jndi-name>` element for the bean, which defines the name for this bean in the JNDI tree – although there is no reason for our code to look the bean up and use it via this name.

This file needs no additional JMS-related descriptor elements to configure the JMS components. All of the configuration elements for the JMS components are stored in the domain configuration file, *config.xml*, in the domain root directory. Feel free to browse through this file and locate the elements related to the JMS Store, JMS Server, JMS Connection Factories, JMS Topic, and JMS Queues created earlier through the Administration Console.

Deploy the EJB Components to WebLogic

The *ant dist* task copies the *titanejb.jar* file to the proper location in the *ejbbook* domain. Use the *redeploy* task or reboot the server as needed to deploy the new *titanejb.jar* file.

Examine the WebLogic server output during startup to ensure that no JMS-related error messages are displayed. If errors appear, read them carefully and make appropriate corrections to the components you created in this exercise or the last.

Verify proper deployment of the new JMS components by examining the JNDI tree for the server and confirming that the new registrations *titan-QueueFactory*, *titan-TicketQueue*, and *titan-ReservationQueue* are present. If these components are not registered properly in the JNDI tree, check that the ConnectionFactory and JMSServer components are targeted to the *myserver* server.

Examine and Run the Client Applications

This exercise includes the two client applications described in the EJB book:

- *JmsClient_ReservationProducer* creates messages in the Reservation Queue
- *JmsClient_TicketConsumer* listens on the Ticket Queue for ticket confirmations

Both programs are simple JMS client applications, using straight JNDI calls to look up JMS `ConnectionFactory` objects and `Queue` objects, and standard JMS techniques to manipulate messages.

The client applications in this exercise require the sample data created in Exercise 12.2. If you deleted the sample data from the database, return to the Exercise 12.2 work root directory (*/work/ejbbook/ex12_2*) and run the *Client_125* program again to re-create the sample data.

JmsClient_ReservationProducer.java

This application accepts two command-line arguments:

```
java JmsClient_ReservationProducer <cruiseID> <numrez>
```

The `<cruiseID>` argument is required because the program has no way to know which keys were used during the execution of *Client_125* to create the sample data used in this exercise. The `<numrez>` argument specifies the number of reservations to create, allowing this program to be used as a crude load generator.

The program uses JNDI to look up the required JMS components and prepares the Connection, Session, and Sender objects needed to place messages on the Reservation Queue:

```
Context jndiContext = getInitialContext();

QueueConnectionFactory factory =
    (QueueConnectionFactory)
    jndiContext.lookup("titan-QueueFactory");

Queue reservationQueue =
    (Queue)jndiContext.lookup("titan-ReservationQueue");
Queue ticketQueue =
    (Queue)jndiContext.lookup("titan-TicketQueue");
QueueConnection connect = factory.createQueueConnection();
QueueSession session =
    connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

QueueSender sender = session.createSender(reservationQueue);
```

Note that the program also uses JNDI to look up the Ticket Queue and obtains a reference to this queue as well. The reason will become obvious when you examine the next section of code.

Next, the program loops the number of times specified by the second command-line parameter (which we captured in the `count` variable), each time creating a `MapMessage` object containing the information the *ReservationProcessor* EJB needs to perform a booking:

```
for (int i = 0; i < count; i++) {

    MapMessage message = session.createMapMessage();
    message.setJMSReplyTo(ticketQueue);
    message.setStringProperty("MessageFormat", "Version 3.4");
    message.setInt("CruiseID", cruiseID.intValue());
    message.setInt("CustomerID",i%2+1);  // Customer 1 or 2 only
    message.setInt("CabinID",i%10+100);  // cabins 100-109 only
    message.setDouble("Price", (double)1000+i);

    // the card expires in about 30 days
    Date expDate =
        new Date(System.currentTimeMillis()+43200000);

    message.setLong("CreditCardNum", 923830283029L);
    message.setLong("CreditCardExpDate", expDate.getTime());
    message.setString(
        "CreditCardType", CreditCardDO.MASTER_CARD);

    System.out.println("Sending reservation message #"+i);
    sender.send(message);
}
```

The program puts a variety of information in each `MapMessage` object:

- The Ticket Queue reference is placed in the `JMSReplyTo` attribute. It will be used by the bean's `deliverTicket` method to obtain the correct reply destination for `TicketDO` confirmations.
- The property `MessageFormat` is set to the proper value defined in the *ejb-jar.xml* file, to ensure that our message-driven bean will receive this message.
- Parameters required by the *ReservationProcessor* EJB are placed in the message, including customer, cruise, cabin, price, and credit-card information.

Note that we cannot pre-create a `CreditCardDO` object and place it in the `MapMessage` object because `MapMessage` accepts only simple types such as `Integer`, `String`, etc. Even the expiration date must be sent as a `Long` rather than as a `java.util.Date` object.

The program then places the `MapMessage` on the Reservation Queue for delivery by the JMS services to the *ReservationProcessor* EJB.

Before running the *JmsClient_ReservationProducer* application, examine the other client application in this exercise.

JmsClient_TicketConsumer.java

This program is a straightforward JMS listener, designed to accept messages from the Ticket Queue and display the contents of the `TicketDO` object attached to the message. The steps required to listen on a JMS Queue are very similar to the steps required to listen on a JMS Topic, as *JmsClient_1* did in the last exercise. In this program we connect to the desired Queue and wait indefinitely for messages:

```
public JmsClient_TicketConsumer() throws Exception {

    Context jndiContext = getInitialContext();

    QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup("titan-QueueFactory");
    Queue ticketQueue = (Queue)
        jndiContext.lookup("titan-TicketQueue");
    QueueConnection connect = factory.createQueueConnection();
    QueueSession session =
      connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
    QueueReceiver receiver = session.createReceiver(ticketQueue);
    receiver.setMessageListener(this);

    System.out.println(
        "Listening for messages on titan-TicketQueue...");
    connect.start();
}
```

Note the hard-coded ConnectionFactory and Queue lookup names. When a message is delivered to this listener, the `onMessage` method will be invoked:

```
public void onMessage(Message message) {

    try {
        ObjectMessage objMsg = (ObjectMessage)message;
        TicketDO ticket = (TicketDO)objMsg.getObject();
        System.out.println("********************************");
        System.out.println(ticket);
        System.out.println("********************************");
    }
    catch (JMSException jmsE) {
        jmsE.printStackTrace();
    }
}
```

This message extracts the `TicketDO` object from the message and displays the confirmation information.

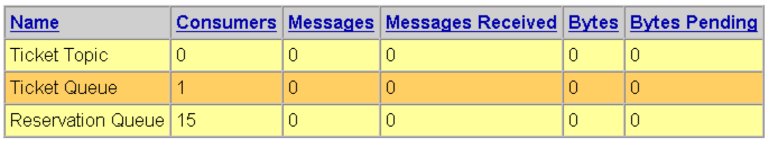
Open a separate command prompt or telnet window, navigate to the *ex13_2* directory, set the environment variables properly, and run the *JmsClient_TicketConsumer* application using the standard command-line syntax:

```
C:\work\ejbbook\ex13_2>setenv
...
...\ex13_2>java com.titan.clients.JmsClient_TicketConsumer
Listening for messages on titan-TicketQueue...
```

The client should report that it is listening for messages on the Ticket Queue and then simply wait. If you receive errors, ensure that the *ejbbook* domain is running and that *titan-QueueFactory* and *titan-TicketQueue* are properly registered in the JNDI tree.

While *JmsClient_TicketConsumer* is running, open the WebLogic Administration Console, navigate to the **TitanJMSServer** folder and click on the **Monitoring** tab. Click on the link to **Monitor all Active JMS Destinations...** to view a list of the active Topic and Queue destinations in the server:

Figure 60: Monitoring destinations in Titan JMS Server



Name	Consumers	Messages	Messages Received	Bytes	Bytes Pending
Ticket Topic	0	0	0	0	0
Ticket Queue	1	0	0	0	0
Reservation Queue	15	0	0	0	0

Note the single active consumer registered on the TicketQueue and the multiple consumers registered on the Reservation Queue. This page will be interesting to watch while the

JmsClient_ReservationProducer application is running, especially when many reservation requests are made in rapid sucession. Use the refresh icon on the right side of the console to activate automatic refresh for this display.

You may wonder where the 15 consumers for the Reservation Queue are coming from. We set the `<initial-beans-in-free-pool>` element for the message-driven bean to only five, so why are there 15 listeners on this queue? WebLogic registers interest in the queue on behalf of every execute thread configured in the default execute queue in your domain. By default, WebLogic uses 15 execute threads, hence 15 listeners on the Reservation Queue.

➢ You can change the number of execute threads on the default execute queue using the Administration Console. Right-click on the server and choose **View Execute Queues**, pick the **default** queue from the list presented on the right, and configure the **Execute Thread Count** to the desired value. You need to reboot for this change to take effect. If you do so now, remember to restart *JmsClient_TicketConsumer* before proceeding.

It is finally time to create some reservations using the new client program. Run the program using the standard syntax, supplying a valid Cruise ID and requesting five iterations:

```
...>java com.titan.clients.JmsClient_ReservationProducer 180 5
Sending reservation message #0
Sending reservation message #1
Sending reservation message #2
Sending reservation message #3
Sending reservation message #4
```

The output in this window is not very interesting, but you should see a great deal of output in the WebLogic server log, as well as some confirmation messages appearing in the window running the *JmsClient_TicketConsumer* client application.

Examine the output in the WebLogic server log and compare the timing of the various messages from the *ReservationProcessor* EJB, the `ejbCreate` methods on the *Reservation* EJB, and the `process` method on the *TravelAgent* EJB. There should be evidence of simultaneous processing of multiple reservations. We configured our instance pool for the *ReservationProcessor* EJB to have multiple beans in the pool, so the container will fire up as many beans as it needs to process the incoming messages, up to the limit set in the `<max-beans-in-free-pool>` element.

Now examine the output from the *JmsClient_TicketConsumer* program. You should see reservation confirmation messages for the five reservations created by the producer program. If you see the output, everything you built and configured in this exercise worked properly, and you should pat yourself on the back for a job well done!

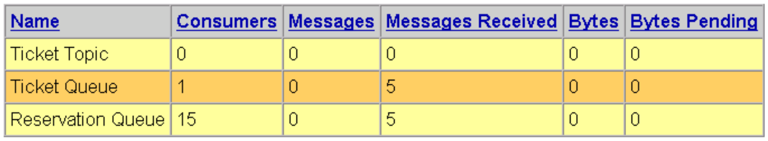
It is very possible that the reservation confirmation messages are not displayed in order from Suite 100 to Suite 104, suggesting that they were not processed in the order in which they were created and placed on the Reservation Queue. According to the specification, the container must deliver the messages to a message-driven bean instance in the order they are placed in the queue. How can the reservation confirmations be out of order?

By configuring multiple Reservation Processor beans in the instance pool you have allowed for asynchronous processing of these requests, essentially giving up the right to pure first-in-first-out processing of incoming reservation requests. In other words, the different bean instances are racing to complete their work booking individual reservations, and there is no guarantee that the beans will finish their work in the order they were started by incoming messages.

As an optional experiment, modify the *weblogic-ejb-jar.xml* file and change the “max beans” and “initial beans” elements to the value `1` to specify a single message-driven bean instance. Re-build and re-deploy your EJBs using *ant dist* and *ant redeploy* and re-run the producer program to request five more reservations. The reservations should be processed sequentially in a true first-in-first-out manner, albeit more slowly.

Refresh the console display if you need to, and note that the monitoring display shows five messages received by each of the two queues in our system:

Figure 61: Monitoring destinations after running producer application



Name	Consumers	Messages	Messages Received	Bytes	Bytes Pending
Ticket Topic	0	0	0	0	0
Ticket Queue	1	0	5	0	0
Reservation Queue	15	0	5	0	0

Two more optional experiments will demonstrate key differences between JMS Queues and JMS Topics. In Exercise 13.1 you observed the behavior of the JMS Topic in the presence of none, one, and multiple *JmsClient_1* listening programs. Repeat these experiments with the new *JmsClient_TicketConsumer* program, following these steps:

1. Stop the copy of the consumer program you currently have running. There is now no listener active for the *Ticket Queue* JMS Queue.
2. Run the producer program to generate a few reservations. Everything should work normally from the point of view of *JmsClient_ReservationProducer* and the *ReservationProcessor* EJB, but no confirmation messages appear because no consumer program is running.
3. Re-start *JmsClient_TicketConsumer*. It should quickly display the confirmation messages for the reservations created in Step 1, demonstrating that unlike the JMS Topic in the previous exercise, messages in a JMS Queue will wait patiently for delivery until a listener is available.
4. Open another command-prompt or telnet window and start another copy of *JmsClient_TicketConsumer*. There are now two consumers listening on the same queue.
5. Run the producer program to generate 10 reservations. Unless you are very unlucky, at least some of the 10 confirmation messages should appear in each of the two consumer output windows. None of the confirmation messages should appear in both outputs, demonstrating that each message is delivered to one and only one consumer listening on the Queue. Contrast this result with what you saw in Exercise 13.1, where both clients listening on the same JMS Topic received every message published to the Topic.

Finally, you can use the *JmsClient_ReservationProducer* application as a crude load generator to examine the performance of WebLogic and your EJB components. Ensure that your deployment descriptor for the *ReservationProcessor* EJB is configured for at least 50 beans in the instance pool and run the producer program to create 100, 200, or even 1000 reservations in rapid succession. Output messages will be flying by in the WebLogic server log, and the consumer applications will be displaying confirmation messages at a rapid pace.

Feel free to experiment with some of the following parameters to observe the changes in system performance:

- The `<max-beans-in-free-pool>` element for the *ReservationProcessor* EJB – As we pointed out a little earlier, setting this to a low value will eliminate multi-processing of incoming reservation requests.
- The `<max-beans-in-free-pool>` element for the *ProcessPayment* EJB – Setting this to a low value will limit the number of simultaneous calls to this bean and may reduce performance.
- Maximum messages in the *Titan Queue Factory* JMS Connection Factory – Change this value from the suggested 500 down to a small number such as 20. This reduction may cause the producer program to slow down, as it is forced to wait to add new messages to the queue, but it is unlikely to affect the overall processing time.
- The Default Delivery Mode in the Titan Queue Factory – Changing this from Persistent to Non-Persistent will prevent Messages being written to the JMS Store when enqueued (or deleted when dequeued) and performance may improve significantly – at the risk of losing messages if the server crashes with messages in the Queue.
- Execute threads for the server – Too few threads will undoubtably slow the processing, but too many can cause unnecessary context switching. Try values like 5, 15, 30, and 100.
- Maximum size of the *titan-jdbcPool* database connection pool – Depending on the performance of your database, a small pool can easily limit the performance of your system. Try values as high as 15 or 20, but don't forget to increase the number of execute threads at the same time, to ensure that there are always more execute threads than possible pool connections.

A detailed discussion of performance testing, tuning, and WebLogic best practices is beyond the scope of this workbook. BEA's on-line documentation includes basic performance guidelines (see *http://edocs.bea.com/wls/docs61/perform/index.html*), and a recently-published book on J2EE and WebLogic Server by Girdley, et. al. includes WebLogic configuration best practices.

Examine and Run the Client JSP Pages

There are no JSP-based examples in this exercise.

That's it! We hope this workbook has met your expectations and, along with the O'Reilly EJB book, has given you a strong foundational understanding of EJB 2.0 and WebLogic Server. Write us at the address provided on the web site (*www.titan-books.com*) and let us know how it went – We're very interested in hearing from you.

Now get out there and build something!

About the Author

Greg Nyberg has over 15 years of experience in the design and development of object-oriented systems, and specializes in large mission-critical systems using BEA WebLogic. Mr. Nyberg is the founder of and a frequent speaker at the Minneapolis BEA Users' Group, and has spoken at the BEA eWorld conference and other national conferences numerous times. Mr. Nyberg has also authored and delivered training classes in C++, Forte, Java, and J2EE technologies, and understands the importance of "hands-on" application of new technology during the learning process. In this workbook, Mr. Nyberg applies his experience as an architect and his pragmatic "How does it work and what does it do for me?" approach to the new EJB 2.0 specification and its implementation in WebLogic

About the Series

Each of the books in this series is a server-specific companion to the third edition of Richard Monson-Haefel's best-selling and award-winning *Enterprise JavaBeans* (O'Reilly 2001), available at *http://www.titan-books.com/* and at all major retail outlets. It guides the reader step by step through the exercises called out in that work, explains how to build and deploy working solutions in a particular application server, and provides useful hints, tips, and warnings.

These workbooks are published by Titan Books in the context of a friendly agreement with O'Reilly and Associates, the publishers of *Enterprise JavaBeans*, to provide serious developers with the best possible foundation for success in EJB development on their chosen platforms.

Colophon

This book is set in the legible and attractive Georgia font. Manuscripts were composed and edited in Microsoft Word, and converted to PDF format for download and printing with Adobe Acrobat.