

**JBoss 3.0 Workbook**  
**FOR**  
**ENTERPRISE JAVA BEANS, 3<sup>rd</sup> Edition**  
**(BETA copy)**



**By**  
**Bill BURKE AND SACHA LABOUREY**

Copyright © 2002 Titan Books, Inc. All Rights Reserved.

### *Why didn't I take the blue pill?*

Bill Burke is one of the core developers of JBoss Clustering and gives regular talks at JBoss training courses and seminars on the subject. Bill has over 9 years experience implementing and using middleware in the industry. He was one of the primary developers of Iona Technology's, Orbix 2000 CORBA product and has also designed and implemented J2EE applications at Mercantec, Dow Jones, and Eigner Corporation. Besides hanging with his wonderful wife, you can find Bill cheering for the New England Patriots at CMGi Stadium with his dad.

Sacha Labourey is one of the core developers of JBoss Clustering and frequently gives JBoss training courses. He owns a master in computer science from the Swiss Federal Institute of Technology and is the founder of Cogito Informatique, a Swiss company specialized in the application servers and middlewares fields. To prove to himself he is not a computer addict, he regularly goes on trips to the Alps to extend his Rumantsh vocabulary.

# Preface

This workbook is designed to be a companion for O'Reilly's *Enterprise JavaBeans, Third Edition*, by Richard Monson-Haefel, for users of JBoss, an open-source J2EE application server. It is one of a series of workbooks that is being published by Titan-Books as an informative companion to that best-selling work.

The goal of this workbook is to provide the reader with step-by-step instructions for installing, configuring, and using JBoss and for deploying and running the examples from *Enterprise JavaBeans*.

This book is based on the production release of JBoss 3.0.0 Final and includes all the EJB 2.0 examples from the *Enterprise JavaBeans* book. All the examples in this workbook will work properly with JBoss 3.0.0 Final, but not with earlier versions of JBoss.

## Contents of This Book

This workbook is divided into three sections:

- **Server Installation and Configuration** – This section will walk you through downloading, installing and configuring JBoss. It will also provide a brief overview of the structure of the JBoss installation.
- **Exercises** – These sections contain step-by-step instructions for downloading, building, and running the example programs referred to from *Enterprise JavaBeans, Third Edition* (which, for brevity, this workbook will refer to as “the EJB book”). The text will also walk through the various deployment descriptors and source code to point out JBoss features and issues.
- **Database Configuration** – This section will be a small appendix of XML snippets for configuring a few popular JDBC drivers from different database vendors.

Because JBoss 3.0.0 Final is an EJB 2.0-compliant J2EE implementation, the EJB 1.1 exercises referred to in the EJB book are not included in this workbook.

The workbook text for each exercise depends on the amount of configuration required for the example program, but will generally also include the following information:

- Compiling and building the example code
- Deploying the EJB components to the application server
- Running the example programs and evaluating the results.

The exercises were designed to be built and executed in order. Every effort was made to remove any dependencies between exercises by including all components for each exercise within each

exercise directory, but dependencies may still exist. Following the workbook text will guide you through any of these dependencies.

Also, this workbook is not intended to be a course on database configuration or design. The exercises have been designed to work out-of-the-box with the open-source database, Hypersonic SQL, that is shipped with JBoss 3.0 and all database tables are created automatically at runtime by the application server.

## ***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/>

We will post errata here, and any updates when required to support specification or product changes. This site also contains links to many popular EJB-related sites on the internet.

I hope you find this workbook useful in your study of Enterprise JavaBeans and the JBoss open-source J2ee implementation. Comments, suggestions, and error reports on the text of this workbook or the downloaded example files are welcome and appreciated. Please e-mail them to:

[jboss-workbook@yahoogroups.com](mailto:jboss-workbook@yahoogroups.com)

To obtain more information about JBoss or the JBoss project please go to the project's web site:

<http://www.jboss.org/>

There you will find links to detailed JBoss documentation, on-line forums, and other events happening in the JBoss community. You will also be able to obtain detailed information on JBoss training, support, and consulting services.

The JBossGroup also has a number of books out on JBoss and other J2EE standards through its SAMS publications:

*JBoss Administration and Development.* By Marc Fleury and Scott Stark

*JMX: Managing J2EE with Java Management Extensions.* By Marc Fleury and Juha Lindfors

## Conventions Used in This Book

*Italics* are used for:

- ♦ Filenames and pathnames
- ♦ Names of hosts, domains, and 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 `BeanNamePK`, 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

We would like to thank Marc Fleury, the founder of JBoss, for recommending us for this book and Richard Monson-Haefel for accepting the recommendation. We would also like to thank Greg Nyberg, the author of the Weblogic edition of these workbooks. The example programs he provided in his workbook were a great starting place for us and made our lives much easier.

Special thanks also goes out to those who reviewed and critiqued this work: Dain Sundstrom, Daniel Ruflé and the rest of JBoss Group.

Finally, Bill would also like to thank his wife for putting up with all his whining and complaining and Sacha promises Sophie that he will no longer use the writing of this workbook as an excuse for being late for any of their rendez-vous.

# ***Server Installation and Configuration***

This chapter will guide you through the different steps required to install a fully working JBoss server.

You will learn about JBoss 3.0 micro-kernel architecture and the last section will show you how to install the code for the forthcoming exercises.

At any time, if you need more detailed information about JBoss configuration, do not hesitate to visit the JBoss web site (<http://www.jboss.org/>) where you will find comprehensive on-line documentation.

## **About JBoss**

JBoss is a collaborative effort of a worldwide group of developers to create an open-source, J2EE-based application server. With more than 1 million copies download in less than 12 months. JBoss is the leading J2EE application server.

JBoss implements the full J2EE stack of services (EJB, JMS, JTS/JTA, Servlets/JSP, JNDI, etc.) and also provides advanced features such as clustering, JMX, Web Services, and IIOP integration.

Since JBoss code is licensed under the LGPL (see <http://www.gnu.org/copyleft/lesser.txt>), you can freely use it at no cost in any commercial application or redistribute it.

## **Installing JBoss Application Server**

Before going any further, you need to have JDK 1.3 or higher installed and correctly configured.

To download JBoss binaries, go to the JBoss web site at <http://www.jboss.org/> and follow the “Download JBoss” link. There you will find all current JBoss binaries you can download (in both `zip` or `tar.gz` archive formats).

Once you have downloaded a JBoss binary, extract the archive in the folder of your choice. Under Windows, you can use the WinZip utility to extract the archive content. Under Unix, you can use the following command lines:

```
||| $ gunzip jboss-3.0.0.tar.gz
||| $ tar xf jboss-3.0.0.tar
```

Then, go in the `JBOSS_HOME/bin` directory and launch the run script that matches your OS:

Unix

```
||| $ run.sh
```

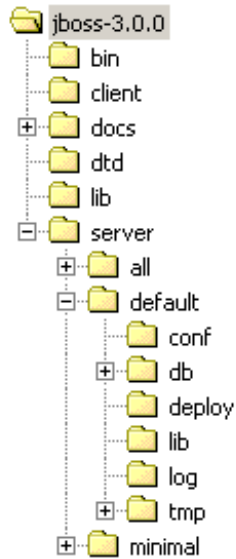
Windows

```
||| C:\jboss-3.0.0\bin>run.bat
```

That's it! You now have a fully working JBoss server!

## ***Discovering JBoss Directory Structure***

Once you have installed JBoss, the following directory structure is created:



The purpose of the various directories is discussed in Table 1.

Directory	Description
<code>bin</code>	Contains scripts to start and shutdown JBoss
<code>client</code>	Contains client-side Java libraries (JARs) required to communicate with JBoss
<code>docs</code>	Sample configuration files (database configuration, etc.)
<code>dtd</code>	DTDs for the various XML files used in JBoss
<code>lib</code>	JARs loaded at startup by JBoss and shared by all JBoss configurations
<code>server</code>	Contains various JBoss configurations. Each configuration must be in a different sub-directory. The name of the sub-directory represents the name of the configuration. By default, JBoss contains 3 configurations: <code>minimal</code> , <code>default</code> and <code>all</code> .
<code>server/all</code>	JBoss' complete configuration starting all services, including



	clustering and IIOP.
<code>server/minimal</code>	JBoss' minimal configuration starting only very basic services. This configuration cannot be used to deployed EJBs.
<code>server/default</code>	JBoss' default configuration. This configuration is used by default.
<code>server/default/conf</code>	JBoss' configuration files. You will learn more about the content of this directory in the next section.
<code>server/default/db</code>	JBoss' databases files (embedded database or JBossMQ for example)
<code>server/default/deploy</code>	JBoss' hot-deployment folder. Any file or directory dropped in this folder is automatically deployed in JBoss: EJB, WAR, EAR and even services.
<code>server/default/lib</code>	JARs loaded at startup by JBoss when starting this particular configuration.
<code>server/default/log</code>	JBoss' log files
<code>server/default/tmp</code>	JBoss' temporary files

*Table 1*

If you want to define your own configuration, create a new sub-directory under the `server` directory containing the appropriate files. To start JBoss with a given configuration uses the “-c” command line parameter:

Windows:

```
||| C:\jboss-3.0.0\bin> run.bat -c config-name
```

Unix:

```
||| $ ./run.sh -c config-name
```

## ***JBoss Configuration Files***

As described in the previous section, JBoss' `server` directory can contain any number of directories: each one being considered as a different JBoss configuration.

The `server/config-name/conf` directory contains JBoss' configuration files. The purpose of the various files is discussed in Table 2.

<b>File</b>	<b>Description</b>
<code>axis-config.xml</code>	JBoss.net (Web Services) configuration
<code>jacorb.properties</code>	JBoss IIOP configuration
<code>jbossmq-state.xml</code>	JBossMQ (JMS implementation) user configuration

<code>jboss-service.xml</code>	Definition of JBoss' services launched at startup (classloaders, JNDI, deployers, etc.)
<code>log4j.xml</code>	Log4J logging configuration
<code>login-config.xml</code>	JBoss security configuration (JBossSX)
<code>standardjaws.xml</code>	Default configuration for JBoss' CMP 1.1 engine. Contains JDBC-to-SQL mapping information for various databases, default CMP settings, logging configuration, etc.
<code>standardjboss.xml</code>	Default container configuration
<code>standardjbosscmp-jdbc.xml</code>	Same as <code>standardjaws.xml</code> but for the JBoss CMP 2.0 engine.

Table 2

## Deployment in JBoss

The deployment process in JBoss is very straightforward. In each configuration, a specific directory is constantly scanned for changes by JBoss. Its location is:

`JBoss_HOME/server/config-name/deploy`

This directory is generally informally referred to as “*the deploy directory*”.

In this directory, you can copy in particular:

- Any JAR library: the classes it contains are automatically added to the JBoss classpath
- An EJB JAR
- A WAR (Web Application aRchive)
- An EAR (Enterprise Application aRchive)
- An XML file containing JBoss MBean definitions
- A directory ending in `.jar`, `.war` or `.ear` and containing respectively the extracted content of an EJB JAR, a WAR or an EAR.

To **redeploy** any of the above files (JAR, WAR, EAR, XML, etc.), overwrite it with a more recent version. JBoss will detect the change thanks to the file timestamp, undeploy the previous file and redeploy it. To redeploy a directory, update its modification-timestamp by using a command line utility such as `touch`.

To **undeploy** a file, remove it from the deploy directory.

## JBoss Quick Internals

JBoss 3.0 is built around a few very powerful concepts allowing users to customize and fine-tune their servers for very specific needs, not limited to J2EE. This flexibility allows JBoss to be used in very different environments, ranging from embedded systems to very large clusters.

The following sections will shortly comment some of these concepts.

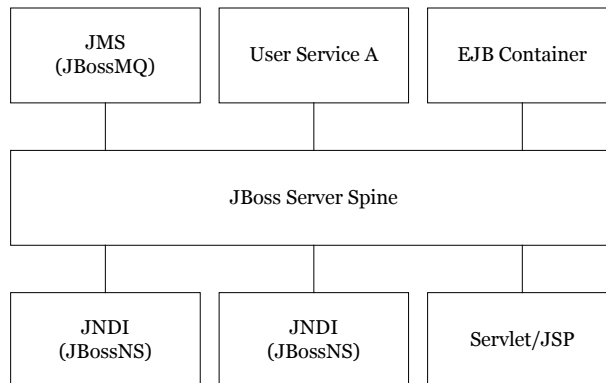
### *Micro-Kernel Architecture*

JBoss 3.0 is based on a micro-kernel design where components can be plugged at runtime to extend its behaviour.

This design particularly fits well with the J2EE platform which essentially is a service based platform. The platform contains services for persistence, transactions, security, naming, messaging, logging, etc.

While other application servers are generally built as monolithic applications containing all services of the J2EE platform at all times, JBoss takes a radically different approach. In JBoss, each of these services is hot deployed as a component running on top of the core, called the **JBoss Server Spine**. Furthermore, you are encouraged to implement your own services to run on top of JBoss.

- ❖ This means that the JBoss application server is not limited to J2EE applications and is frequently used to build any kind of applications requiring a strong and reliable base. For this reason, the JBoss core is also known as the *WebOS*.



JBoss Server Spine itself is based on the **JMX** specification (Java Management eXtensions), making any deployed component automatically manageable in a standard fashion. In the JMX terminology, a service deployed in JBoss is called an **MBean** (i.e. a Managed Bean).

- ❖ More information about the JMX specification can be found at the SUN web site <http://java.sun.com/products/JavaManagement/>

## ***Hot Deployment***

JBoss 2.x has been famous for being the first J2EE-based application server to support the hot deployment and redeployment of applications (EJB JAR, WAR and EAR). At the same time, many application servers required to restart to update an application.

Thanks to its micro-kernel architecture and revolutionary Java class loader, JBoss 3.0 pushes this logic further and is not only able to hot deploy and redeploy applications, but is now able to hot (re-)deploy any service and keep track of dependencies between services.

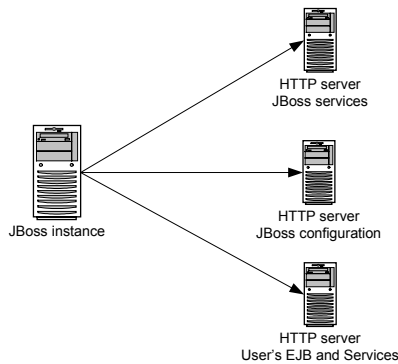
This makes JBoss usable in very demanding environments such as telecommunications systems.

## ***Net Boot***

JBoss 3.0 is able to boot from any network location, just by bootstrapping from the JBoss Server Spine. At this point, JBoss is able to download from various locations the JBoss services, configuration and any user applications (EJB, WAR or EAR) and services.

This makes deployment of new servers very easy.

- ❖ JBoss' bootstrap code weights approximately 5Ko, which makes it suitable for many embedded systems.

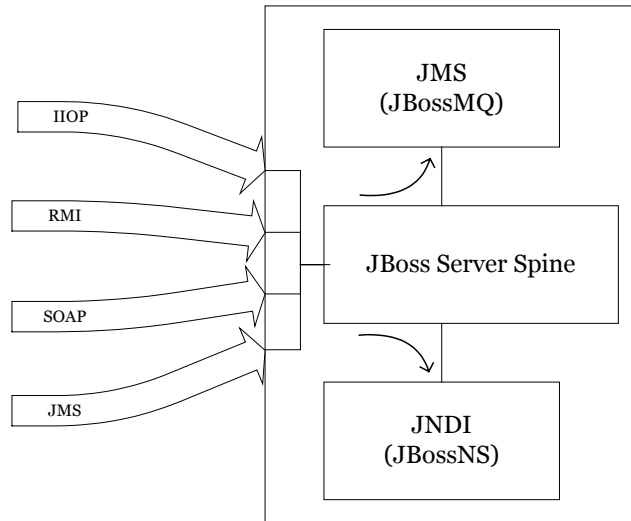


## ***Detached Invokers***

JBoss 3.0 completely detaches the protocol on which an invocation is received from the target service that serves the request. Consequently, new protocols adapters can be easily added to JBoss to support new invocation transports.

JBoss 3.0 currently supports the following invokers:

- RMI
- IIOP
- JMS
- SOAP
- HA-RMI (Clustering over RMI)

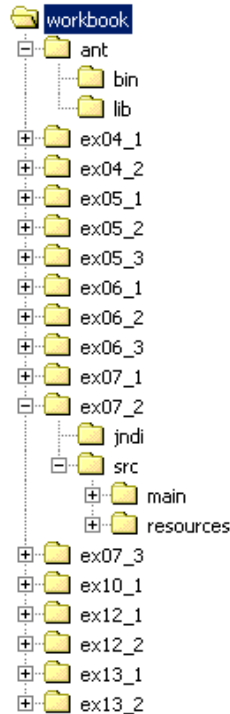


## Exercise Code Setup and Configuration

The example code for the exercises can be downloaded from <http://www.titan-books.com/>. For exercises that require a database, JBoss's default embedded database will be used. Consequently, no additional database setup is required. Appendix A will show you how to configure JBoss to use a different database if so desired.

### *Exercises directory structure*

The example code is organized as a set of directories, one for each exercise. You will find the code of each exercise in the `src/main` sub-directory and the configuration files in `src/resources`.



To help you build and run the exercises, the Ant tool is used. For each exercise, a `build.xml` is provided. It contains the Ant configuration needed to compile the classes, build the EJB JAR, deploy it to JBoss and run the client test applications. For this reason, the Ant tool is provided with the exercises and can be found in the `ant` directory.

- ❖ More information about the Ant tool can be found at the Apache Jakarta web site <http://jakarta.apache.org/ant/>

## ***Environment Setup***

For the Ant scripts to work correctly, you first need to correctly set some environment variables in the shells you will use to run the exercises:

- The `JAVA_HOME` environment variable must point to where your JDK is installed
- The `JBOSS_HOME` environment variable must point to where JBoss 3.0 is installed
- The directory containing the Ant scripts must be in your path

This generally results, depending on your platform, in the following commands:

#### Windows:

```
C:\workbook\ex04_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex04_1> set JBOSS_HOME=C:\jboss-3.0.0
C:\workbook\ex04_1> set PATH=..\ant\bin;%PATH%
```

#### Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
$ export PATH=../ant/bin:$PATH
```

In each chapter, you will find the detailed instructions on how to build, deploy and run the exercises using Ant.





## ***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 build, deploy, and execute the Cabin EJB example. Please note, that since we are using JBoss' default embedded database, there is no database configuration or creation of database tables needed. The code shown below mirrors the example code provided in Chapter 4 of the EJB book.

#### ***Startup JBoss***

Startup JBoss as described in the *Server Installation and Configuration* chapter at the beginning of this book.

#### ***Initialize the Database***

The database table for this exercise will automatically be created in JBoss' default database, HypersonicSQL, when the EJB jar is deployed.

#### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex04\_1* directory created by the extraction process
2. You must set the JAVA\_HOME and JBOSS\_HOME environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex04_1> set JAVA_HOME=C:\jdk1.3  
C:\workbook\ex04_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
||| $ export JAVA_HOME=/usr/local/jdk1.3
||| $ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex04_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

To learn more about the *Ant* utility, visit the Ant project at the Jakarta web site at <http://jakarta.apache.org/ant/index.html>.

Ant compiles the java source code, builds the ejb jar, and deploys the jar by simply copying it to JBoss' *deploy* directory. If you are watching the JBoss console window, you will notice that JBoss automatically discovers the ejb jar once it has been copied into the *deploy* directory and automatically deploys the bean.

Another particularly interesting thing about building ejb jars is that there is no special ejb compilation step. Unlike other vendors, there is no code generation of client stubs. JBoss has a lightweight mechanism for creating client proxies that is done at ejb-jar deployment. This facilitates a quick development and deployment cycle.

The *build.xml* file provided in each workbook exercise directory defines the following build tasks that can be executed by typing *ant taskname*:

- The default task(just typing *ant*) will compile, build the ejb jar, and deploy the jar into JBoss. The deployment procedure is just a simple copy into the JBoss *deploy* directory.
- *clean* removes all *.class* and *.jar* files from the working directory and will undeploy the jar from JBoss by deleting the file from JBoss' *deploy* directory.
- *compile* compiles all the java source files.
- *run.client\_xxx* will run a specific example program. Each exercise in this book will have a *run.client* rule for each example program.

## ***Examine the JBoss Specific Files***

You do not need any JBoss specific files to write a simple EJB. For an entity bean as simple as the Cabin EJB, JBoss will create the appropriate database tables within it's embedded database Hypersonic SQL by examining the *ejb-jar.xml* deployment descriptor.

- ❖ In later chapters you will learn how to map entity beans to different data sources and pre-existing database tables using JBoss specific CMP deployment descriptors.

By default, JBoss uses the `<ejb-name>` from the bean's ejb-jar deployment descriptor for the JNDI binding of the bean's HOME interface. If you do not like this default, you can override it in a *jboss.xml* file. Client's use this name to lookup an EJB's home interface. For this example, CabinEJB is bound to `CabinHomeRemote`.

#### **jboss.xml**

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>CabinEJB</ejb-name>
      <jndi-name>CabinHomeRemote</jndi-name>
    </entity>
  </enterprise-beans>
</jboss>
```

### ***Examine and Run the Client Applications***

There are two example programs that implement the sample clients provided in the EJB book:

- *Client\_1.java* creates a single Cabin bean, populates each of its attributes, then looks up the created bean with the primary key.
- *Client\_2.java* creates 99 additional Cabins with a variety of different data that will be used in subsequent exercises.

#### **Client\_1.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;

public class Client_1
{
    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
{
    return new InitialContext();
}
}

```

The `getInitialContext` method creates an `InitialContext` with no properties. Since there are no properties set, the java library that implements `InitialContext` will search the classpath for the file `jndi.properties`. Each example program in this workbook will have a `jndi` directory that contains a `jndi.properties` file. You will be executing all example programs through `ant` and it will set the classpath appropriately to reference this properties file.

Run the `Client_1` application by invoking `ant run.client_41a` at the command prompt. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output of **Client\_1** should look something like this:

```

C:\workbook\ex04_1>ant run.client_41a
Buildfile: build.xml

prepare:

```

```
compile:

ejbjar:

run.client_41a:
    [java] Master Suite
    [java] 1
    [java] 1
    [java] 3
```

*Client\_1* adds a row to the database representing the Cabin bean and does not delete it at the conclusion of the program. You cannot run this program more than once unless you stop JBoss and clean the database by invoking the ant task `clean.db` and restarting JBoss. Otherwise, you will get the following error:

```
run.client_41a:
    [java] javax.ejb.DuplicateKeyException: Entity with primary key
1 already exists
    [java] at
sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Strea
mRemoteCall.java:245)
    [java] at
sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java
:220)
    [java] at
sun.rmi.server.UnicastRef.invoke(UnicastRef.java:122)
    [java] at
org.jboss.invocation.jrmp.server.JRMPInvoker_Stub.invoke(Unknown
Source)
    [java] at
org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxy.invoke(JRMPInv
okerProxy.java:128)
    [java] at
org.jboss.invocation.InvokerInterceptor.invoke(InvokerInterceptor.ja
va:108)
    [java] at
org.jboss.proxy.TransactionInterceptor.invoke(TransactionInterceptor
.java:73)
    [java] at
org.jboss.proxy.SecurityInterceptor.invoke(SecurityInterceptor.java:
76)
```

```

[java]      at
org.jboss.proxy.ejb.HomeInterceptor.invoke (HomeInterceptor.java:185)
[java]      at
org.jboss.proxy.ClientContainer.invoke (ClientContainer.java:96)
[java]      at $Proxy0.create(Unknown Source)
[java]      at com.titan.clients.Client_1.main (Client_1.java:23)

```

Run the Client\_2 application by invoking `ant run.client_41b` at the command prompt. Don't forget to set your **JBoss\_HOME** and **PATH** environment variables.

The output of **Client\_2** should look something like this:

```

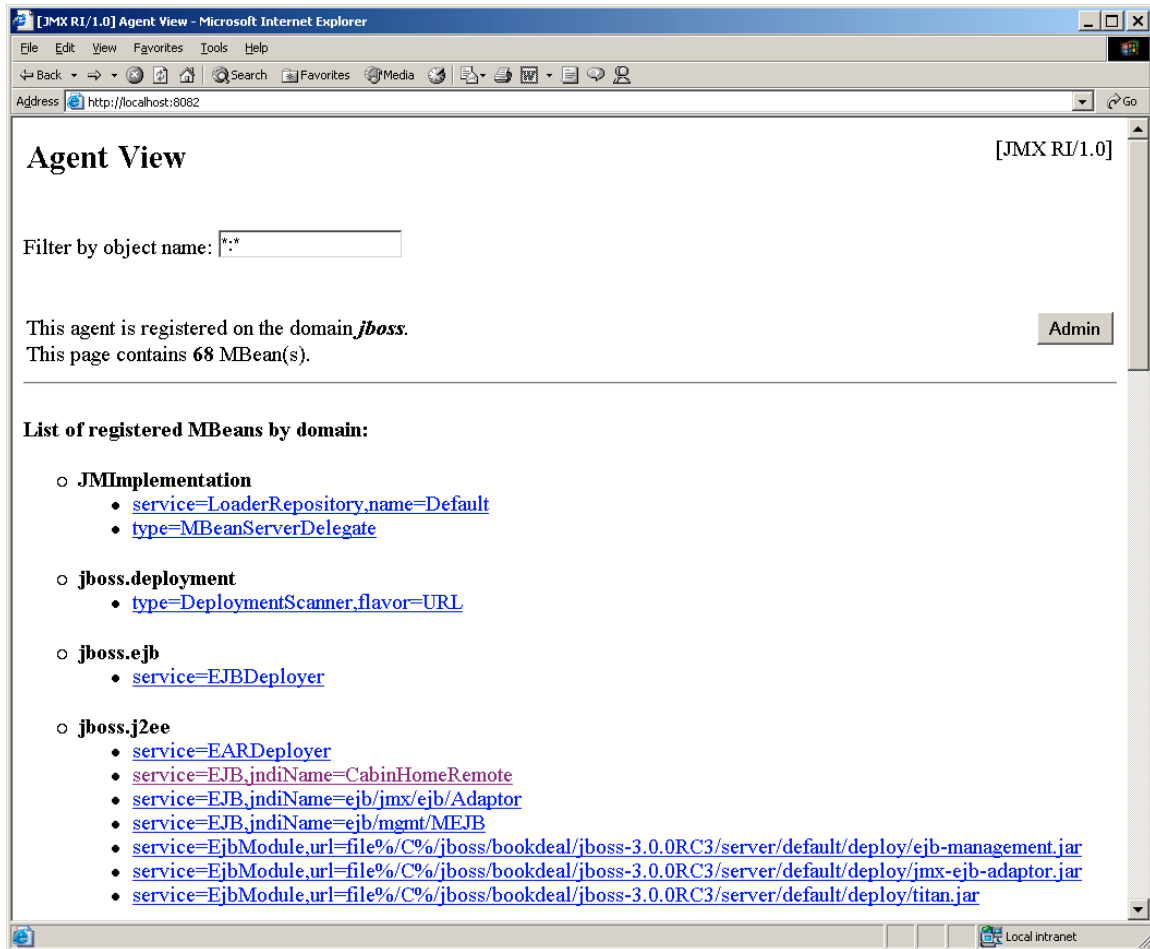
run.client_41b:
[java] PK=1, Ship=1, Deck=1, BedCount=3, Name=Master Suite
[java] PK=2, Ship=1, Deck=1, BedCount=2, Name=Suite 100
[java] PK=3, Ship=1, Deck=1, BedCount=3, Name=Suite 101
[java] PK=4, Ship=1, Deck=1, BedCount=2, Name=Suite 102
[java] PK=5, Ship=1, Deck=1, BedCount=3, Name=Suite 103
[java] PK=6, Ship=1, Deck=1, BedCount=2, Name=Suite 104
[java] PK=7, Ship=1, Deck=1, BedCount=3, Name=Suite 105
[java] PK=8, Ship=1, Deck=1, BedCount=2, Name=Suite 106
...
[java] PK=90, Ship=3, Deck=3, BedCount=3, Name=Suite 309
[java] PK=91, Ship=3, Deck=4, BedCount=2, Name=Suite 400
[java] PK=92, Ship=3, Deck=4, BedCount=3, Name=Suite 401
[java] PK=93, Ship=3, Deck=4, BedCount=2, Name=Suite 402
[java] PK=94, Ship=3, Deck=4, BedCount=3, Name=Suite 403
[java] PK=95, Ship=3, Deck=4, BedCount=2, Name=Suite 404
[java] PK=96, Ship=3, Deck=4, BedCount=3, Name=Suite 405
[java] PK=97, Ship=3, Deck=4, BedCount=2, Name=Suite 406
[java] PK=98, Ship=3, Deck=4, BedCount=3, Name=Suite 407
[java] PK=99, Ship=3, Deck=4, BedCount=2, Name=Suite 408
[java] PK=100, Ship=3, Deck=4, BedCount=3, Name=Suite 409

```

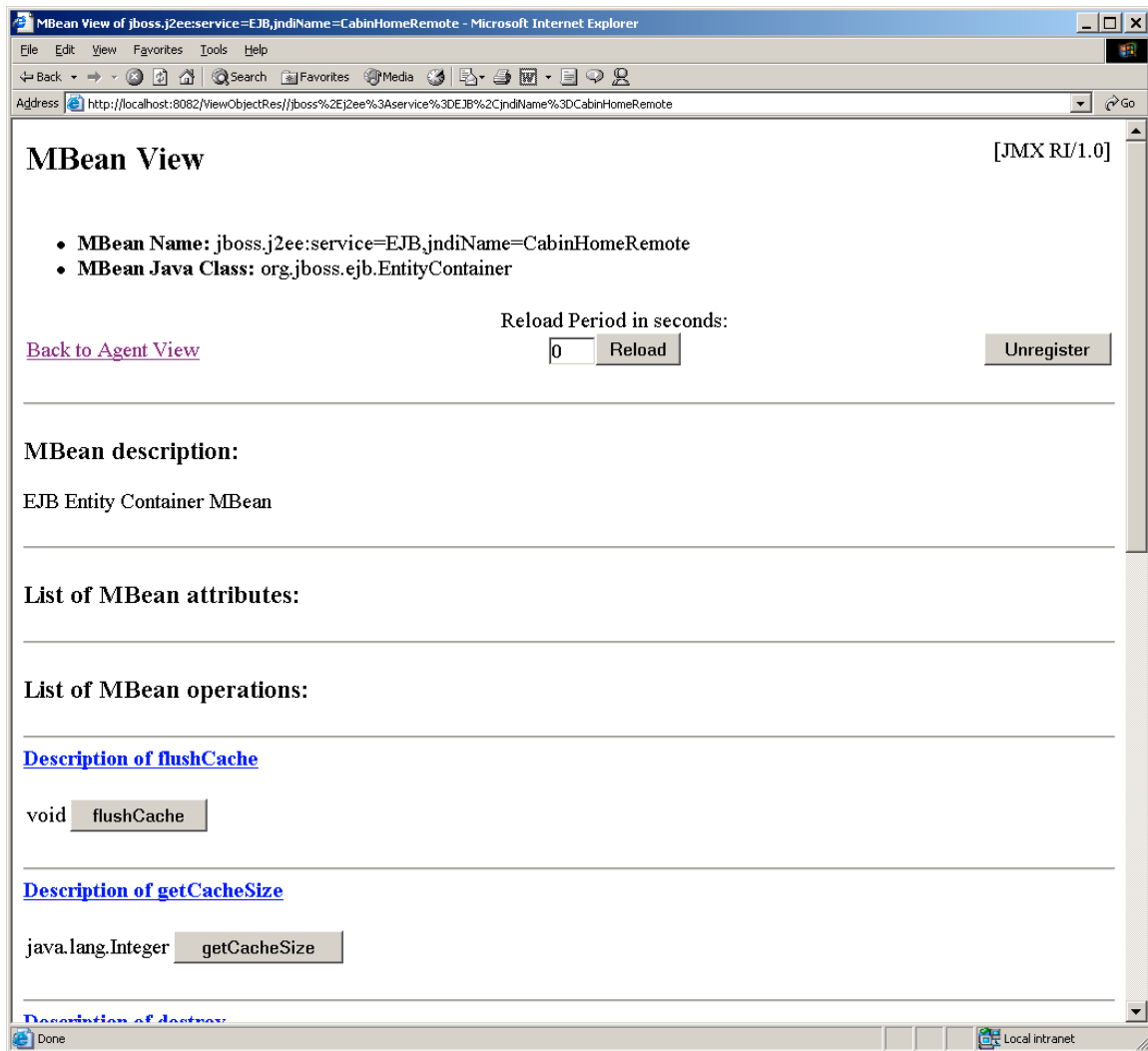
Like Client\_1, this example creates a bunch of rows in the database and does not delete them when complete. Client\_2 can only be executed once without causing `DuplicateKey` exceptions.

## ***Managing Entity Beans***

Every EJB in JBoss is deployed and managed as a JMX MBean. You can view and manage EJBs deployed within JBoss by accessing the JMX management console available at <http://localhost:8082/> through your web browser.



Click on the `service=EJB,jndiName=CabinHomeRemote` link that is shown in the picture above. Entity Beans have two management functions. You can flush the entity bean's cache or view the number of cached objects for that entity bean. To flush, click on the `flushCache` button. To view number of cached beans, click on the `getCacheSize` button.









## Exercise 4.2: A Simple Session Bean

In this exercise you will create and build the TravelAgent EJB. This simple bean illustrates the use of a stateless session bean and mirrors the code shown in Chapter 4 of the EJB book.

### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Initialize the Database***

The database should contain the 100 rows created by a successful execution of the test programs from the previous exercise, *Client\_1* and *Client\_2*.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex04\_2* directory created by the extraction process
2. You must set the JAVA\_HOME and JBOSS\_HOME environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex04_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex04_2> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex04_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## Examine the JBoss Specific Files

In this example, the *jboss.xml* deployment descriptor overrides the default JNDI binding for the deployed EJBs. *CabinEJB* is bound to *CabinHomeRemote* and the *TravelAgentEJB* is bound to *TravelAgentHomeRemote*.

### **jboss.xml**

```
||| <jboss>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <jndi-name>CabinHomeRemote</jndi-name>
        </entity>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <jndi-name>TravelAgentHomeRemote</jndi-name>
            <ejb-ref>
                <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
                <jndi-name>CabinHomeRemote</jndi-name>
            </ejb-ref>
        </session>
    </enterprise-beans>
</jboss>
```

The EJB book describes how you must use `<ejb-ref>` declarations when one EJB references another. The *TravelAgent* EJB references the *Cabin* entity bean, so the following xml is required in *ejb-jar.xml*.

### **ejb-jar.xml**

```
||| <ejb-ref>
    <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>
```

If you have a `<ejb-ref-name>` declared in your *ejb-jar.xml* file, you must have a corresponding `<ejb-ref>` declaration in your *jboss.xml* file that maps the portable jndi name used by the *TravelAgent* EJB to the real jndi name of the *Cabin* EJB.

### **jboss.xml**

```
||| <jboss>
```

```

<enterprise-beans>
  <entity>
    <ejb-name>CabinEJB</ejb-name>
    <jndi-name>CabinHomeRemote</jndi-name>
  </entity>
  <session>
    <ejb-name>TravelAgentEJB</ejb-name>
    <jndi-name>TravelAgentHomeRemote</jndi-name>
    <ejb-ref>
      <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
      <jndi-name>CabinHomeRemote</jndi-name>
    </ejb-ref>
  </session>
</enterprise-beans>
</jboss>

```

## ***Examine and Run the Client Application***

The example program in this section invokes the TravelAgent EJB to list cabins that meet a certain criteria.

### **Client\_3.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]);
}
...

```

The client code does a JNDI lookup for the TravelAgent home and does a simple create method invocation to obtain a reference to a TravelAgent EJB. The client then calls `listCabins` and receives a list of cabin names that meet the provided criteria.

Let's examine a little bit of the code in TravelAgent EJB `listCabins` method to see how it works.

### TravelAgentBean.java

```
public String [] listCabins(int shipID, int bedCount)
{
    try
    {
        javax.naming.Context jndiContext = new InitialContext();
        Object obj =
            jndiContext.lookup("java:comp/env/ejb/CabinHomeRemote");

        CabinHomeRemote home = ...
```

When a deployed EJB in JBoss wants to access JNDI all that is needed is a simple `new InitialContext()`. JBoss will automatically create an optimized, collocated reference to the JNDI server running inside the application server. The rest of `listCabins` is pretty straightforward, so let's continue by running the client application.

Run the *Client\_3* application by invoking `ant run.client_42` at the command prompt. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output of **Client\_3** should look something like this:

```
C:\workbook\ex04_2>ant run.client_42
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_42:
[java] 1,Master Suite,1
[java] 3,Suite 101,1
[java] 5,Suite 103,1
[java] 7,Suite 105,1
[java] 9,Suite 107,1
[java] 12,Suite 201,2
[java] 14,Suite 203,2
[java] 16,Suite 205,2
[java] 18,Suite 207,2
[java] 20,Suite 209,2
[java] 22,Suite 301,3
[java] 24,Suite 303,3
[java] 26,Suite 305,3
[java] 28,Suite 307,3
```



[java] 30,Suite 309,3

## ***Exercises for Chapter 5***

### **Exercise 5.1: The Remote Component Interfaces**

The example programs in Exercise 5.1 dive into some of the features of the home interface of an EJB, including the use of the remove method. They also show you how to access and use various metadata that is available through an EJB's API.

#### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

#### ***Initialize the Database***

The database should contain the 100 rows created by a successful execution of the test programs from Exercise 4.1 and 4.2.

#### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex05\_1* directory created by the extraction process
2. You must set the JAVA\_HOME and JBOSS\_HOME environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex05_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex05_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```



3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex05_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the JBoss Specific Files***

There are no new JBoss configuration files or components in this exercise.

## ***Examine and Run the Client Applications***

There are two example programs that illustrate the concepts explained in the EJB book:

- *Client\_51a.java* illustrates the use of the remove method on the Cabin EJB Home interface.
- *Client\_51b.java* illustrates the use of bean metadata methods.

The example code for *Client\_51a* and *Client\_51b* is pulled directly from the EJB book. There is no need to go into this code here because the EJB book already does a very good job of this.

Run *Client\_51a* by invoking `ant run.client_51a` at the command prompt. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. Do the same to run *Client\_51b*.  
`ant run.client_51b`.

The output of *Client\_51a* should be exactly as described in the EJB book. The output of *Client\_51b* is as follows:

```
||| C:\workbook\ex05_1>ant run.client_51b
Buildfile: build.xml

prepare:

compile:

run.client_51b:
    [java] com.titan.cabin.CabinHomeRemote
    [java] com.titan.cabin.CabinRemote
    [java] java.lang.Integer
    [java] false
```

```
||| [java] GETTING THE HOME  
||| [java] Master Suite
```

Please note that if you try to run *Client\_51a* more than once you will see an exception stating that the entity you are attempting to remove does not exist.

```
||| [java] java.rmi.NoSuchObjectException: Entity not found:  
||| primaryKey=30
```





## Exercise 5.2: The EJBObject, Handle, and Primary Key

The example programs in Exercise 5.2 explore the APIs available through the `EJBObject` and `EJBMetaData` interfaces. They will also examine how to use `Handle` and `HomeHandle` as persistent references to EJB objects and homes.

### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Initialize the Database***

The database should contain the 100 rows created by a successful execution of the test programs from Exercise 4.1 otherwise this example will not work properly.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the `ex05_2` directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex05_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex05_2> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex05_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

### ***Examine the JBoss Specific Files***

There are no new JBoss configuration files or components in this exercise.

### ***Examine and Run the Client Applications***

There are three example programs that illustrate the concepts explained in the EJB book:

- *Client\_52a.java* illustrates the use of `EJBObject` to retrieve an EJB's home interface.
- *Client\_52b.java* illustrates how to use `isIdentical` to determine if two EJB references are the same.
- *Client\_52c.java* illustrates how to use EJB handles as persistent bean references.

The example code is pulled directly from the EJB book and embellished somewhat to expand on introduced concepts. The EJB book does a pretty good job of explaining the concepts illustrated in the example programs, so further explanation of the code is not needed in this workbook.

Run *Client\_52a*, *Client\_52b*, and *Client\_52c* by invoking the appropriate ant task as you have done in previous examples: `run.client_52a`, `run.client_52b` and `run.client_52c`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.







## Exercise 5.3: The Local Component Interfaces.

The example program in Exercise 5.3 explores the use of local interfaces. The Cabin entity bean you created in Exercise 4.1 will be expanded to provide a local interface for use within the TravelAgent stateless session bean. This exercise will also describe how to modify your EJB deployment descriptors to enable local interfaces..

### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Initialize the Database***

The database should contain the 100 rows created by a successful execution of the test programs from Exercise 4.1 and 4.2.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex05\_3* directory created by the extraction process
2. You must set the JAVA\_HOME and JBOSS\_HOME environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex05_3> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex05_3> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex05_3> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## Examine the JBoss Specific Files

JBoss 3.0 has a minor bug. It requires that you use `ejb-link` when you want your bean to reference a local bean.

### *ejb-jar.xml*

```
<ejb-jar>
  <enterprise-beans>
    ...
    <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>Stateless</session-type>
      <transaction-type>Container</transaction-type>

      <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-link is required by jboss for local-refs. -->
        <ejb-link>CabinEJB</ejb-link>
      </ejb-local-ref>
    ...
  </ejb-jar>
```

If you examine the *jboss.xml* file in Exercise 5.3, you will see that you must additionally declare the JNDI binding for the Cabin EJB's local home interface as well as the binding for the remote home interface. Both `CabinHomeRemote` and `CabinHomeLocal` will be registered by JBoss into the JNDI tree.

### *jboss.xml*

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>CabinEJB</ejb-name>
      <jndi-name>CabinHomeRemote</jndi-name>
      <local-jndi-name>CabinHomeLocal</local-jndi-name>
```

```
</entity>
```

TravelAgentEJB also tells JBoss how to reference the `CabinHomeLocal` by using `ejb-local-ref-name`.

```
<session>
  <ejb-name>TravelAgentEJB</ejb-name>
  <jndi-name>TravelAgentHomeRemote</jndi-name>
  <ejb-local-ref>
    <ejb-local-ref-name>ejb/CabinHomeLocal</ejb-local-ref-name>
    <jndi-name>CabinHomeLocal</jndi-name>
  </ejb-local-ref>
</entity>
</enterprise-beans>
</jboss>
```

### ***Examine and Run the Client Applications***

The example code for `Client_53` is exactly the same as `Client_3` from Exercise 4.2.

Run *Client\_53* by invoking the appropriate ant task as you have done in previous examples: `run.client_53`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex05_3>ant run.client_53
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_53:
[java] 1,Master Suite,1
[java] 3,Suite 101,1
[java] 5,Suite 103,1
[java] 7,Suite 105,1
[java] 9,Suite 107,1
[java] 12,Suite 201,2
[java] 14,Suite 203,2
[java] 16,Suite 205,2
[java] 18,Suite 207,2
[java] 20,Suite 209,2
[java] 22,Suite 301,3
[java] 24,Suite 303,3
```



[java] 26,Suite 305,3  
[java] 28,Suite 307,3

## ***Exercises for Chapter 6***

### **Exercise 6.1: Basic Persistence in CMP 2.0**

This exercise begins the process of walking you through the intricacies of CMP 2.0. You will learn more detailed JBoss CMP 2.0 configuration mechanisms in this chapter by creating the Customer EJB described in the EJB book.

#### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

#### ***Initialize the Database***

The database table for this exercise will automatically be created in JBoss' default database, HypersonicSQL, when the EJB jar is deployed.

#### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex06\_1* directory created by the extraction process
2. You must set the JAVA\_HOME and JBOSS\_HOME environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex06_1> set JAVA_HOME=C:\jdk1.3  
C:\workbook\ex06_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3  
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex06_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## Examine the JBoss Specific Files

In this section a new JBoss CMP 2.0 deployment descriptor is introduced, *jbosscmp-jdbc.xml*. This file provides more detailed control of your bean's database mapping as well as more advanced performance tuning options.

### jbosscmp-jdbc.xml

```
||| <jbosscmp-jdbc>

    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>Hypersonic SQL</datasource-mapping>
        <create-table>true</create-table>
        <remove-table>true</remove-table>
    </defaults>

    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <table-name>Customer</table-name>
            <cmp-field>
                <field-name>id</field-name>
                <column-name>ID</column-name>
            </cmp-field>
            <cmp-field>
                <field-name>lastName</field-name>
                <column-name>LAST_NAME</column-name>
            </cmp-field>
            <cmp-field>
                <field-name>firstName</field-name>
                <column-name>FIRST_NAME</column-name>
            </cmp-field>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

```

        <cmp-field>
            <field-name>hasGoodCredit</field-name>
            <column-name>HAS_GOOD_CREDIT</column-name>
        </cmp-field>
    </entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

### The <defaults> section:

```
<datasource>java:/DefaultDS</datasource>
```

This configuration variable tells JBoss' CMP engine what database connection pool to use for the entity beans defined in this jar. It is currently configured to use the default datasource, but you can change it to your own defined data sources. Appendix A gets into more detail on how to configure your own data sources.

```
<datasource-mapping>Hypersonic SQL</datasource-mapping>
```

This variable describes the database mapping that CMP should use. Here's some other mappings you could use. *This is a non-exhaustive list.:*

```

<datasource-mapping>Oracle8</datasource-mapping>
<datasource-mapping>Oracle7</datasource-mapping>
<datasource-mapping>MS SQLSERVER</datasource-mapping>
<datasource-mapping>MS SQLSERVER2000</datasource-mapping>

```

For other available supported database mappings, please review JBoss' advanced documentation on their website at <http://www.jboss.org/>

```
<create-table>true</create-table>
```

When this configuration variable is set to true, JBoss will create the database tables for each entity bean described in this deployment descriptor if these tables do not already exist. This happens when the EJB's jar is deployed.

```
<remove-table>true</remove-table>
```

When this configuration variable is set to true, JBoss will remove/drop the tables from the database for each entity bean described in this deployment descriptor. This happens when the entity bean is undeployed.

### The <enterprise-beans> section:

There is an xml fragment <entity></entity> for each entity bean defined in this EJB jar.

```
|||      <ejb-name>CustomerEJB</ejb-name>
```

Defines the entity bean that is described in that section.

```
|||      <table-name>Customer</table-name>
```

This variable defines what database table this entity bean should map to.

```
|||      <cmp-field>
|||          <field-name>id</field-name>
|||          <column-name>ID</column-name>
|||      </cmp-field>
```

Each `<cmp-field>` section describes the mapping between an entity bean's field and the column of the database table. The `<field-name>` tag is the entity bean field name, while the `<column-name>` defines the table column name.

## ***Examine and Run the Client Applications***

There is only one single client application for this exercise, Client\_61. It is modeled after the example in the EJB book. It will create Customer EJBs in the database based on the command line parameters.

To run the client first set your JBOSS\_HOME and PATH environment variables appropriately. Then invoke the provided wrapper script to execute the program. You must supply a set of primary key, first name, and last name values of the command line as shown here:

```
Client_61 777 Bill Burke 888 Sacha Labourey
```

The output of this execution should be:

```
||| C:\workbook\ex06_1>client_61 777 Bill Burke 888 Sacha Labourey
||| Buildfile: build.xml
|||
||| prepare:
|||
||| compile:
|||
||| ejbjar:
|||
||| run.client_61:
|||     [java] 777 = Bill Burke
|||     [java] 888 = Sacha Labourey
```

The example program removes the created beans at the conclusion of operation so there will be no data in the database.









## Exercise 6.2: Dependent Value Classes in CMP 2.0

The example programs in Exercise 6.2 explores using a dependent value class to combine multiple CMP fields into a single serializable object that can be passed in and out of entity-bean methods.

### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Initialize the Database***

There is no database initialization needed.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex06\_2* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex06_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex06_2> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex06_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

### ***Examine the JBoss Specific Files***

There are no new JBoss configuration files or components in this exercise.

### ***Examine and Run the Client Applications***

The example program, *Client\_62*, shows how the `Name` dependent value class is used with the Customer EJB. The example code is pulled directly from the EJB book and embellished somewhat to expand on introduced concepts. The EJB book does a pretty good job of explaining the concepts illustrated in *Client\_62*, so further explanation of the code is not needed in this workbook.

The client application uses the new `getName` and `setName` methods of the Customer EJB to initialize, modify and display a newly created Customer bean using the `Name` dependent value class. This test bean is then removed from the database before the application finishes.

To run *Client\_62* invoke the ant task: `run.client_62`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex06_2>ant run.client_62
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_62:
    [java] 1 = Richard Monson
    [java] 1 = Richard Monson-Haefel
```





## Exercise 6.3: A Simple Relationship in CMP 2.0

The example program in Exercise 6.3 shows how to implement a simple CMP relationship between the Customer EJB and the Address EJB. Dependent values classes are also used again by the client to pass address information along to the Customer EJB.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex06\_3* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
||| C:\workbook\ex06_3> set JAVA_HOME=C:\jdk1.3
||| C:\workbook\ex06_3> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
||| $ export JAVA_HOME=/usr/local/jdk1.3
||| $ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex06_3> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

### ***Examine the JBoss Specific Files***

There are no new JBoss configuration files or components in this exercise.



## Examine and Run the Client Applications

The example program, *Client\_63*, shows how to create a Customer EJB and set the Address relation on that customer.

### AddressBean.java

```
public abstract class AddressBean implements javax.ejb.EntityBean
{
    private static final int IDGEN_START =
        (int)System.currentTimeMillis();
    private static int idgen = IDGEN_START;

    public Integer ejbCreateAddress (String street, String city,
                                    String state, String zip )
        Throws CreateException
    {
        setId(new Integer(idgen++));
        setStreet(street);
        setCity(city);
        setState(state);
        setZip(zip);
        return null;
    }
    ...
}
```

At the time of the writing of this workbook, JBoss 3.0 was in beta and automatic primary key generation had not been implemented yet. So, for this and subsequent examples, a very crude id generator was created. The code just takes the current time in milliseconds at the load of the bean and increments it by one at every `ejbCreate`. Crude, workable for these examples, but not recommended for real applications.

To run *Client\_63* invoke the ant task: `run.client_63`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex06_3>ant run.client_63
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_63:
    [java] Creating Customer 1..
    [java] Creating AddressDO data object..
```

```
[java] Setting Address in Customer 1...
[java] Acquiring Address data object from Customer 1...
[java] Customer 1 Address data:
[java] 1010 Colorado
[java] Austin,TX 78701
[java] Creating new AddressDO data object..
[java] Setting new Address in Customer 1...
[java] Customer 1 Address data:
[java] 1600 Pennsylvania Avenue NW
[java] DC,WA 20500
[java] Removing Customer 1...
```

## ***Exercises for Chapter 7***

### **Exercise 7.1: Entity Relationships in CMP 2.0: Part 1**

This exercise walks you through creating and implementing a complex set of interrelated entity beans defined in Chapter 7 of the EJB book.

#### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

#### ***Initialize the Database***

The databases table for this exercise will automatically be created in JBoss' default database, HypersonicSQL, when the EJB jar is deployed.

#### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex07\_1* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex07_1> set JAVA_HOME=C:\jdk1.3  
C:\workbook\ex07_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3  
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex07_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the JBoss Specific Files***

There are no new features introduced in JBoss specific files in this chapter. Please review Chapter 6.1 of this workbook to understand the JBoss specific files in this example. Also, this chapter implements non-performance-tuned entity beans and relies on the CMP 2.0 engine to create all database tables. Please review JBoss' advanced CMP 2.0 documentation for more in-depth configuration options at, <http://www.jboss.org/>.

## ***Examine and Run the Client Applications***

From this chapter on, so that the example code matches the code illustrated in the EJB book, we will no longer be using remote entity bean interfaces. As a result, the Customer EJB will be switch to local-only interfaces:

- `CustomerHomeRemote` became `CustomerHomeLocal`
- `CutomerRemote` became `CustomerLocal`
- Bean interface methods no longer throw `RemoteException`
- The *ejb-jar.xml* descriptor was changed to use local interfaces:

```
||| <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>
```

switches to...

```
||| <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 binding also changes as well in *jboss.xml*

```
||| <entity>
    <ejb-name>CustomerEJB</ejb-name>
    <jndi-name>CustomerHomeRemote</jndi-name>
```

```

|||      </entity>
switches to ...
|||      <entity>
|||          <ejb-name>CustomerEJB</ejb-name>
|||          <local-jndi-name>CustomerHomeLocal</local-jndi-name>
|||      </entity>

```

Since we will now be using local interfaces, the example programs no longer need to use dependent values classes to set up relationships like Customer to Address. This really simplifies the code and allows local entity beans such as Address, CreditCard, and Phone to be passed directly as parameters to Customer EJB methods.

A direct consequence of this is that we can no longer have remote clients invoking business logic on the entity beans implemented in this chapter. Therefore, all example business logic will be implemented within a stateless session bean in a number of different methods. Also, EJB containers do not allow the manipulation of a relationship collection (including iteration through the collection) outside the context of a transaction. Since all beans methods are *Required* by default in JBoss, all example test code will run within a transaction. Chapter 14 in the EJB book discusses transactions in more detail.

To execute these examples from the command line, separate distinct remote clients are implemented that get a reference to the stateless test bean and invoke the appropriate test method.

### Client\_71a

The Client\_71a example program investigates the unidirectional relationship between Customer and Address. The business logic for this example is implemented in `com.titan.test.Test71Bean` in the `test71a` method.

In method `test71a`, you will see that output is written to the `PrintWriter` created below. At the end of this method, a `String` is extracted from the `PrintWriter` and passed back to the remote client for display.

```

||| public String test71a() throws RemoteException
||| {
|||     String output = null;
|||     StringWriter writer = new StringWriter();
|||     PrintWriter out = new PrintWriter(writer);
|||     try
|||     {

```

The first part of `test71a` simply fetches the home interfaces of Customer and Address from JNDI. It then creates both a Customer and a Address.

```

|||         InitialContext jndiContext = getInitialContext();
|||         Object obj = jndiContext.lookup("CustomerHomeLocal");
|||         CustomerHomeLocal customerhome = (CustomerHomeLocal)obj;
|||
|||         obj = jndiContext.lookup("AddressHomeLocal");

```

```

AddressHomeLocal addresshome = (AddressHomeLocal)obj;

out.println("Creating Customer 71");

Integer primaryKey = new Integer(71);
CustomerLocal customer = customerhome.create(primaryKey);
customer.setName( new Name("Smith","John") );

AddressLocal addr = customer.getHomeAddress();

if (addr==null)
{
    out.println("Address reference is NULL, Creating one and
                setting in Customer..");
    addr = addresshome.createAddress("333 North Washington"
                                     , "Minneapolis"
                                     , "MN", "55401");
}

```

Simply call `customer.setHomeAddress` to set up the relationship.

```

        customer.setHomeAddress(addr);
    }

```

...

Next we modify the address directly with new information. This is the correct way to modify a unidirectional relationship that has already been set up.

```

        addr.setStreet("445 East Lake Street");
        addr.setCity("Wayzata");
        addr.setState("MN");
        addr.setZip("55432");
    }

```

...

The next bit of code is the wrong way to modify a unidirectional relationship that has already been created. The code below creates a new Address and resets the relationship by calling `customer.setHomeAddress` with the newly created Address entity. What we end up with here is a database “leak” in this situation. Since the home address is being overridden with a new Address entity, the old Address entity is orphaned and is just sitting their in the database unused and forgotten.

```

        addr = addresshome.createAddress("700 Main Street"
                                     , "St. Paul", "MN", "55302");
    }

    customer.setHomeAddress(addr);
}

```

The next bit of code shows that you can share the same entity between two different relationships. The same Address is shared between the Home Address and Billing Address relationships.

```
|||
    addr = customer.getHomeAddress();
...
    customer.setBillingAddress(addr);

    AddressLocal billAddr = customer.getBillingAddress();
    AddressLocal homeAddr = customer.getHomeAddress();
```

The Billing and Home Address should be identical beans.

```
|||
    if (billAddr.isIdentical(homeAddr))
    {
        out.println("Billing and Home are the same!");
    }
    else
    {
        out.println("Billing and Home are NOT the same!
                    BUG IN JBOSS!");
    }
}
```

Finally, `test71a` closes the `PrintWriter` and extracts the output string that will be returned to the client for display purposes.

```
|||
    }
    catch (Exception ex)
    {
        ex.printStackTrace(out);
    }
    out.close();
    output = writer.toString();

    return output;
}
```

To run *Client\_71a* invoke the ant task: `run.client_71a`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
|||
C:\workbook\ex07_1>ant run.client_71a
Buildfile: build.xml

prepare:

compile:

run.client_71a:
[java] Creating Customer 71
```

```

[java] Address reference is NULL, Creating one and setting in
Customer..
[java] Address Info: 333 North Washington Minneapolis, MN 55401
[java] Modifying Address through address reference
[java] Address Info: 445 East Lake Street Wayzata, MN 55432
[java] Creating New Address and calling setHomeAddress
[java] Address Info: 700 Main Street St. Paul, MN 55302
[java] Retrieving Address reference from Customer via
getHomeAddress
[java] Address Info: 700 Main Street St. Paul, MN 55302
[java] Setting Billing address to be the same as Home address.
[java] Testing that Billing and Home Address are the same
Entity.
[java] Billing and Home are the same!

```

## Client\_71b

The Client\_71b example program illustrates a simple one-to-one bi-directional relationship between a Customer bean and a CreditCard bean. The business logic for this example is implemented in `com.titan.test.Test71Bean` in the `test71b` method. Let's examine the code for this example.

The below code uses the default JNDI context to obtain references to the local home interfaces of the Customer and CreditCard EJBs. It also creates an instance of a Customer EJB

```

// obtain CustomerHome
InitialContext jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeLocal");
CustomerHomeLocal customerhome = (CustomerHomeLocal)obj;

obj = jndiContext.lookup("CreditCardHomeLocal");
CreditCardHomeLocal cardhome = (CreditCardHomeLocal)obj;
Integer primaryKey = new Integer(71);
CustomerLocal customer = customerhome.create(primaryKey);
customer.setName( new Name("Smith","John") );

```

Next, an instance of a CreditCard bean is created. Notice that you do not have to pass in a primary key to create the CreditCard. A primary key will be automatically generate with the crude algorithm introduced in *Chapter 6.3*.

```

// set Credit Card info
Calendar now = Calendar.getInstance();
CreditCardLocal card = cardhome.create(now.getTime(),
    "3700000000000001", "John Smith", "O'Reilly");

```

Next, the one-to-one relationship bi-directional relationship between Customer and CreditCard is created simply by calling the `setCreditCard` method of Customer EJB.



```
customer.setCreditCard(card);
```

The following code illustrates the bi-directional relationship by navigating to a Customer from a CreditCard instance and vice versa.

```
String cardname = customer.getCreditCard().getNameOnCard();
out.println("customer.getCreditCard().getNameOnCard()="
    + cardname);

Name name = card.getCustomer().getName();
String custfullname = name.getFirstName() + " " +
    name.getLastName();
out.println("card.getCustomer().getName()="+custfullname);
```

Finally, the code then illustrates how to destroy the relationship between the Customer and the CreditCard beans.

```
card.setCustomer(null);

CreditCardLocal newcardref = customer.getCreditCard();
if (newcardref == null)
{
    out.println("Card is properly unlinked from customer
bean");
}
else
{
    out.println("Whoops, customer still thinks it has a
card! BUG IN JBOSS!");
}
```

To run *Client\_71b* invoke the ant task: `run.client_71b`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex07_1>ant run.client_71b
Buildfile: build.xml

prepare:

compile:

run.client_71b:
[java] Finding Customer 71
[java] Creating CreditCard
[java] Linking CreditCard and Customer
[java] Testing both directions on relationship
```

```
[java] customer.getCreditCard().getNameOnCard()=John Smith
[java] card.getCustomer().getName()=John Smith
[java] Unlink the beans using CreditCard, test Customer side
[java] Card is properly unlinked from customer bean
[java]
```

### Client\_71c

The Client\_71c example program illustrates the proper use of a one-to-many unidirectional relationship between the customers and phone numbers. The business logic for this example is implemented in `com.titan.test.Test71Bean` in the `test71c` method. Let's examine the code for this example.

First of all, the test code locates the Customer home interface through JNDI and then finds the Customer we want to add new phone numbers to.

```
// obtain CustomerHome
InitialContext jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeLocal");
CustomerHomeLocal home = (CustomerHomeLocal)obj;

// Find Customer 71
Integer primaryKey = new Integer(71);
CustomerLocal customer = home.findByPrimaryKey(primaryKey);
```

The next bit of code relates two phone numbers to the customer and outputs the contents of the customer/phone relationship after each addition. It does this by invoking the Customer helper method `addPhoneNumber`.

```
// Display current phone numbers and types
out.println("Starting contents of phone list:");
ArrayList vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
    String ss = (String) (vv.get(jj));
    out.println(ss);
}

// add a new phone number
out.println("Adding a new type 1 phone number..");
customer.addPhoneNumber("612-555-1212", (byte)1);

out.println("New contents of phone list:");
vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
    String ss = (String) (vv.get(jj));
    out.println(ss);
}
```

```

    }

    // add a new phone number
    out.println("Adding a new type 2 phone number..");
    customer.addPhoneNumber("800-333-3333", (byte)2);

    out.println("New contents of phone list:");
    vv = customer.getPhoneList();
    for (int jj=0; jj<vv.size(); jj++)
    {
        String ss = (String) (vv.get(jj));
        out.println(ss);
    }

```

Next, the use of the `updatePhoneNumber` helper method is demonstrated. A pre-existing phone number is modified using this method.

```

    // update a phone number
    out.println("Updating type 1 phone numbers..");
    customer.updatePhoneNumber("763-555-1212", (byte)1);

    out.println("New contents of phone list:");
    vv = customer.getPhoneList();
    for (int jj=0; jj<vv.size(); jj++)
    {
        String ss = (String) (vv.get(jj));
        out.println(ss);
    }

```

Finally, the code illustrates how to remove a member of a one-to-many unidirectional relationship.

```

    // delete a phone number
    out.println("Removing type 1 phone numbers from this
               Customer..");
    customer.removePhoneNumber((byte)1);

    out.println("Final contents of phone list:");
    vv = customer.getPhoneList();
    for (int jj=0; jj<vv.size(); jj++)
    {
        String ss = (String) (vv.get(jj));
        out.println(ss);
    }

```

Note that the phone is still in the database, but it is no longer related to this customer bean.

To run *Client\_71c* invoke the ant task: `run.client_71c`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex07_1>ant run.client_71c
Buildfile: build.xml

prepare:

compile:

run.client_71c:
    [java] Starting contents of phone list:
    [java] Adding a new type 1 phone number..
    [java] New contents of phone list:
    [java] Type=1   Number=612-555-1212
    [java] Adding a new type 2 phone number..
    [java] New contents of phone list:
    [java] Type=1   Number=612-555-1212
    [java] Type=2   Number=800-333-3333
    [java] Updating type 1 phone numbers..
    [java] New contents of phone list:
    [java] Type=1   Number=763-555-1212
    [java] Type=2   Number=800-333-3333
    [java] Removing type 1 phone numbers from this Customer..
    [java] Final contents of phone list:
    [java] Type=2   Number=800-333-3333
```





## Exercise 7.2: Entity Relationships in CMP 2.0: Part 2

The example programs in Exercise 7.2 illustrate the remaining four entity-bean relationship types:

- Many-to-one unidirectional (Cruise-Ship)
- One-to-many bi-directional (Cruise-Reservation)
- Many-to-many bi-directional (Customer-Reservation)
- Many-to-many unidirectional (Cabin-Reservation)

### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Initialize the Database***

There is no database initialization needed since JBoss will create the needed tables at bean deployment.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex07\_2* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex07_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex07_2> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex07_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the JBoss Specific Files***

There are no new concepts introduced in the JBoss specific deployment descriptors.

## ***Examine and Run the Client Applications***

There are a large number of example programs in this section to demonstrate the different relationships described in the corresponding chapter in the EJB book. These exercises can be re-run as many times as you like because they clean up after themselves by removing all created

- Client\_72a demonstrates the many-to-one unidirectional Cruise-Ship relationship, as well as the sharing of a reference between different beans.
- Client\_72b demonstrates the one-to-many bi-directional Cruise-Reservation relationship as well as how to use set methods to modify reservations that are associated with a cruise.
- Client\_72c continues with the Cruise-Reservation relationship by using the `addAll` method to modify the reservations associated with a cruise.
- Client\_72d demonstrates the many-to-many bi-directional Customer-Reservation relationship.
- Client\_72e continues with the Customer-Reservation relationship by showing how to use `setCustomers` to modify the Customers for a Reservation.
- Client\_72f demonstrates the many-to-many unidirectional Cabin-Reservation relationship.

### **Client\_72a**

The business logic for this example is implemented in `com.titan.test.Test72Bean` in the `test72a` method. Client\_72a models the many-to-one unidirectional Cruise-Ship relationships shown in Figure 7-12 of the EJB book.

First the relationships described in the top half of Figure 7-12 of the EJB book are created.

```
||| cruises[0] = cruisehome.create("Cruise 1", shipA);  
||| cruises[1] = cruisehome.create("Cruise 2", shipA);  
||| cruises[2] = cruisehome.create("Cruise 3", shipA);  
||| cruises[3] = cruisehome.create("Cruise 4", shipB);
```



```

cruises[4] = cruisehome.create("Cruise 5", shipB);
cruises[5] = cruisehome.create("Cruise 6", shipB);

```

Next, Cruise 4 is switched to share a reference to Ship A. The bottom half of Figure 7-12 is now created.

```

ShipLocal newship = cruises[0].getShip();
cruises[3].setShip(newship);

```

To run *Client\_72a* invoke the ant task: `run.client_72a`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. You may re-run this example as many times as you wish.

The output should look something like this:

```

C:\workbook\ex07_2>ant run.client_72a
Buildfile: build.xml

prepare:

compile:

run.client_72a:
[java] Creating Ships
[java] PK=1001 name=Ship A tonnage=30000.0
[java] PK=1002 name=Ship B tonnage=40000.0
[java] Creating Cruises
[java] Cruise 1 is using Ship A
[java] Cruise 2 is using Ship A
[java] Cruise 3 is using Ship A
[java] Cruise 4 is using Ship B
[java] Cruise 5 is using Ship B
[java] Cruise 6 is using Ship B
[java] Changing Cruise 4 to use same ship as Cruise 1
[java] Cruise 1 is using Ship A
[java] Cruise 2 is using Ship A
[java] Cruise 3 is using Ship A
[java] Cruise 4 is using Ship A
[java] Cruise 5 is using Ship B
[java] Cruise 6 is using Ship B
[java] Removing created beans

```

### Client\_72b

The business logic for this example is implemented in `com.titan.test.Test72Bean` in the `test72b` method. *Client\_72b* models the one-to-many bi-directional Cruise-Reservation relationships shown in Figure 7-14 of the EJB book.

First the relationships described in the top half of Figure 7-14 of the EJB book are created.

```

        for (int i = 0; i < 6; i++)
        {
            CruiseLocal cruise = (i < 3) ? cruiseA : cruiseB;
            reservations[i] = reservationhome.create(cruise,new
ArrayList());
            reservations[i].setDate(date.getTime());
            reservations[i].setAmountPaid((i + 1) * 1000.0);
            date.add(Calendar.DAY_OF_MONTH, 7);
        }

```

Next, the code sets the reservations of Cruise B to be the reservations of Cruise A. This actually moves those relationships from A to B and Cruise A and Reservations 1-3 no longer have any Cruise-Reservation relationships. This behavior is illustrated in the bottom half of Figure 7-14

```

        Collection a_reservations = cruiseA.getReservations();
        cruiseB.setReservations( a_reservations );

```

To run *Client\_72b* invoke the ant task: `run.client_72b`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. You may re-run this example as many times as you wish.

The output should look something like this:

```

C:\workbook\ex07_2>ant run.client_72b
Buildfile: build.xml

prepare:

compile:

run.client_72b:
[java] Creating Cruises
[java] name=Cruise A
[java] name=Cruise B
[java] Creating Reservations
[java] Reservation date=11/01/2002 is for Cruise A
[java] Reservation date=11/08/2002 is for Cruise A
[java] Reservation date=11/15/2002 is for Cruise A
[java] Reservation date=11/22/2002 is for Cruise B
[java] Reservation date=11/29/2002 is for Cruise B
[java] Reservation date=12/06/2002 is for Cruise B
[java] Testing CruiseB.setReservations(
CruiseA.getReservations() )
[java] Reservation date=11/01/2002 is for Cruise B
[java] Reservation date=11/08/2002 is for Cruise B
[java] Reservation date=11/15/2002 is for Cruise B
[java] Reservation date=11/22/2002 is for No Cruise!
[java] Reservation date=11/29/2002 is for No Cruise!
[java] Reservation date=12/06/2002 is for No Cruise!

```

```
||| [java] Removing created beans.
```

### Client\_72c

The business logic for this example is implemented in `com.titan.test.Test72Bean` in the `test72c` method. `Client_72c` explores the use of `Collection.addAll()` in the Cruise-Reservation one-to-many bi-directional relationship shown in Figure 7-15 of the EJB book.

First the relationships described in the top half of Figure 7-15 of the EJB book are created.

```
|||         for (int i = 0; i < 6; i++)
|||         {
|||             CruiseLocal cruise = (i < 3) ? cruiseA : cruiseB;
|||             reservations[i] = reservationhome.create(cruise,new
ArrayList());
|||             reservations[i].setDate(date.getTime());
|||             reservations[i].setAmountPaid((i + 1) * 1000.0);
|||             date.add(Calendar.DAY_OF_MONTH, 7);
|||         }
```

Next, the code changes all reservations of Cruise A to be for Cruise B instead. This result of this action is best described in the bottom half of Figure 7-15

```
|||         Collection a_reservations = cruiseA.getReservations();
|||         Collection b_reservations = cruiseB.getReservations();
|||         b_reservations.addAll(a_reservations);
```

To run *Client\_72c* invoke the ant task: `run.client_72c`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. You may re-run this example as many times as you wish.

The output should look something like this:

```
||| C:\workbook\ex07_2>ant run.client_72c
||| Buildfile: build.xml
|||
||| prepare:
|||
||| compile:
|||
||| run.client_72c:
|||     [java] Creating Cruises
|||     [java] name=Cruise A
|||     [java] name=Cruise B
|||     [java] Creating Reservations
|||     [java] Reservation date=11/01/2002 is for Cruise A
|||     [java] Reservation date=11/08/2002 is for Cruise A
|||     [java] Reservation date=11/15/2002 is for Cruise A
|||     [java] Reservation date=11/22/2002 is for Cruise B
```

```

[java] Reservation date=11/29/2002 is for Cruise B
[java] Reservation date=12/06/2002 is for Cruise B
[java] Testing using b_res.addAll(a_res) to combine
reservations
[java] Reservation date=11/01/2002 is for Cruise B
[java] Reservation date=11/08/2002 is for Cruise B
[java] Reservation date=11/15/2002 is for Cruise B
[java] Reservation date=11/22/2002 is for Cruise B
[java] Reservation date=11/29/2002 is for Cruise B
[java] Reservation date=12/06/2002 is for Cruise B

```

## Client\_72d

The business logic for this example is implemented in `com.titan.test.Test72Bean` in the `test72d` method. Client\_72c explores the use of `Collection.addAll()` in the Customer-Reservation many-to-many bi-directional relationship shown in Figure 7-17 of the EJB book.

First two sets of customers are created.

```

Set lowcustomers = new HashSet();
Set highcustomers = new HashSet();
CustomerLocal[] allCustomers = new CustomerLocal[6];
for (int kk=0; kk<6; kk++) {
    CustomerLocal cust = customerhome.create(new Integer(kk));
    allCustomers[kk] = cust;
    cust.setName(new Name("Customer "+kk,""));
    if (kk<=2) {
        lowcustomers.add(cust);
    } else {
        highcustomers.add(cust);
    }
    out.println(cust.getName().getLastName());
}

```

Next, six reservations are created and related to one of the customer sets as shown in the top half of Figure 7-15 of the EJB book.

```

reservations[0] = reservationhome.create(cruiseA, lowcustomers);
reservations[0].setDate(date.getTime());
reservations[0].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[1] = reservationhome.create(cruiseA, highcustomers);
reservations[1].setDate(date.getTime());
reservations[1].setAmountPaid(5000.0);

```

Finally, the code additionally sets up Customers 4-6 for Reservation A by using `addAll`. The result of this action is best described in the bottom half of Figure 7-17

```
|||      Set customers_a = reservations[0].getCustomers();
|||      Set customers_b = reservations[1].getCustomers();
|||      customers_a.addAll(customers_b);
```

To run *Client\_72d* invoke the ant task: `run.client_72d`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. You may re-run this example as many times as you wish.

The output should look something like this:

```
||| C:\workbook\ex07_2>ant run.client_72d
||| Buildfile: build.xml
|||
||| prepare:
|||
||| compile:
|||
||| run.client_72d:
||| [java] cruise.getName()=Cruise A
||| [java] ship.getName()=Ship A
||| [java] cruise.getShip().getName()=Ship A
||| [java] Creating Customers 1-6
||| [java] Customer 0
||| [java] Customer 1
||| [java] Customer 2
||| [java] Customer 3
||| [java] Customer 4
||| [java] Customer 5
||| [java] Creating Reservations 1 and 2, each with 3 customers
||| [java] Reservation date=11/01/2002 is for Cruise A with
||| customers Customer 2 Customer 1 Customer 0
||| [java] Reservation date=11/08/2002 is for Cruise A with
||| customers Customer 5 Customer 4 Customer 3
||| [java] Performing customers_a.addAll(customers_b) test
||| [java] Reservation date=11/01/2002 is for Cruise A with
||| customers Customer 2 Customer 1 Customer 0 Customer 5 Custo
||| mer 4 Customer 3
||| [java] Reservation date=11/08/2002 is for Cruise A with
||| customers Customer 5 Customer 4 Customer 3
||| [java] Removing created beans
```

## Client\_72e

The business logic for this example is implemented in `com.titan.test.Test72Bean` in the `test72e` method. `Client_72e` explores the use of `setCustomers` to share an entire collection in the Customer-Reservation many-to-many bi-directional relationship shown in Figure 7-18 of the EJB book.

First four sets of customers are created.

```
Set customers13 = new HashSet();
Set customers24 = new HashSet();
Set customers35 = new HashSet();
Set customers46 = new HashSet();
CustomerLocal[] allCustomers = new CustomerLocal[6];
for (int kk=0; kk<6; kk++) {
    CustomerLocal cust = customerhome.create(new Integer(kk));
    allCustomers[kk] = cust;
    cust.setName(new Name("Customer "+kk, ""));
    if (kk<=2) { customers13.add(cust); }
    if (kk>=1 && kk<=3) { customers24.add(cust); }
    if (kk>=2 && kk<=4) { customers35.add(cust); }
    if (kk>=3) { customers46.add(cust); }
}
```

Next, the relationships between Customers and Reservations are set up as shown in the top half of Figure 7-18 of the EJB book.

```
reservations[0] = reservationhome.create(cruiseA, customers13);
reservations[0].setDate(date.getTime());
reservations[0].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[1] = reservationhome.create(cruiseA, customers24);
reservations[1].setDate(date.getTime());
reservations[1].setAmountPaid(5000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[2] = reservationhome.create(cruiseA, customers35);
reservations[2].setDate(date.getTime());
reservations[2].setAmountPaid(6000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[3] = reservationhome.create(cruiseA, customers46);
reservations[3].setDate(date.getTime());
reservations[3].setAmountPaid(7000.0);
```

Finally, the code sets up the relationships shown in the bottom half of Figure 7-18

```
Set customers_a = reservations[0].getCustomers();
reservations[3].setCustomers(customers_a);
```

To run *Client\_72e* invoke the ant task: `run.client_72e` Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. You may re-run this example as many times as you wish.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72e
Buildfile: build.xml

prepare:

compile:

run.client_72e:
    [java] Creating a Ship and Cruise
    [java] cruise.getName()=Cruise A
    [java] ship.getName()=Ship A
    [java] cruise.getShip().getName()=Ship A
    [java] Creating Customers 1-6
    [java] Creating Reservations 1-4 using three customers each
    [java] Reservation date=11/01/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
    [java] Reservation date=11/08/2002 is for Cruise A with
customers Customer 3 Customer 2 Customer 1
    [java] Reservation date=11/15/2002 is for Cruise A with
customers Customer 4 Customer 3 Customer 2
    [java] Reservation date=11/22/2002 is for Cruise A with
customers Customer 5 Customer 4 Customer 3
    [java] Performing reservationD.setCustomers(customersA) test
    [java] Reservation date=11/01/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
    [java] Reservation date=11/08/2002 is for Cruise A with
customers Customer 3 Customer 2 Customer 1
    [java] Reservation date=11/15/2002 is for Cruise A with
customers Customer 4 Customer 3 Customer 2
    [java] Reservation date=11/22/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
    [java] Removing created beans.
```

## Client\_72f

The business logic for this example is implemented in `com.titan.test.Test72Bean` in the `test72f` method. *Client\_72f* demonstrates removing beans in the many-to-many unidirectional Cabin-Reservation relationship as shown in Figure 7-20 of the EJB book.

First four sets of cabins are created.

```

Set cabins13 = new HashSet();
Set cabins24 = new HashSet();
Set cabins35 = new HashSet();
Set cabins46 = new HashSet();
CabinLocal[] allCabins = new CabinLocal[6];
for (int kk=0; kk<6; kk++) {
    CabinLocal cabin = cabinhome.create(new Integer(kk));
    allCabins[kk] = cabin;
    cabin.setName("Cabin "+kk);
    if (kk<=2) { cabins13.add(cabin); }
    if (kk>=1 && kk<=3) { cabins24.add(cabin); }
    if (kk>=2 && kk<=4) { cabins35.add(cabin); }
    if (kk>=3) { cabins46.add(cabin); }
    out.println(cabin.getName());
}

```

Next, the initial relationships between Reservations and Cabins are create as shown in the top half of Figure 7-20 of the EJB book.

```

reservations[0] = reservationhome.create(cruiseA, null);
reservations[0].setCabins(cabins13);
reservations[0].setDate(date.getTime());
reservations[0].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[1] = reservationhome.create(cruiseA, null);
reservations[1].setCabins(cabins24);
reservations[1].setDate(date.getTime());
reservations[1].setAmountPaid(5000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[2] = reservationhome.create(cruiseA, null);
reservations[2].setCabins(cabins35);
reservations[2].setDate(date.getTime());
reservations[2].setAmountPaid(6000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[3] = reservationhome.create(cruiseA, null);
reservations[3].setCabins(cabins46);
reservations[3].setDate(date.getTime());
reservations[3].setAmountPaid(7000.0);

```

Finally, the code removes some of the relationships as shown at the bottom half of Figure 7-20

```

Set cabins_a = reservations[0].getCabins();
Iterator iterator = cabins_a.iterator();
while (iterator.hasNext())
{
    CabinLocal cc = (CabinLocal)iterator.next();
}

```



```

    out.println("Removing "+cc.getName()+" from cabins_a");
    iterator.remove();
}

```

To run *Client\_72f* invoke the ant task: `run.client_72f` Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables. You may re-run this example as many times as you wish.

The output should look something like this:

```

C:\workbook\ex07_2>ant run.client_72f
Buildfile: build.xml

prepare:

compile:

run.client_72f:
    [java] Creating a Ship and Cruise
    [java] cruise.getName()=Cruise A
    [java] ship.getName()=Ship A
    [java] cruise.getShip().getName()=Ship A
    [java] Creating Cabins 1-6
    [java] Cabin 0
    [java] Cabin 1
    [java] Cabin 2
    [java] Cabin 3
    [java] Cabin 4
    [java] Cabin 5
    [java] Creating Reservations 1-4 using three cabins each
    [java] Reservation date=11/01/2002 is for Cruise A with cabins
Cabin 2 Cabin 1 Cabin 0
    [java] Reservation date=11/08/2002 is for Cruise A with cabins
Cabin 3 Cabin 2 Cabin 1
    [java] Reservation date=11/15/2002 is for Cruise A with cabins
Cabin 4 Cabin 3 Cabin 2
    [java] Reservation date=11/22/2002 is for Cruise A with cabins
Cabin 5 Cabin 4 Cabin 3
    [java] Performing cabins_a collection iterator.remove() test
    [java] Removing Cabin 2 from cabins_a
    [java] Removing Cabin 1 from cabins_a
    [java] Removing Cabin 0 from cabins_a
    [java] Reservation date=11/01/2002 is for Cruise A with cabins
    [java] Reservation date=11/08/2002 is for Cruise A with cabins
Cabin 3 Cabin 2 Cabin 1

```

```
||| [java] Reservation date=11/15/2002 is for Cruise A with cabins  
Cabin 4 Cabin 3 Cabin 2  
[java] Reservation date=11/22/2002 is for Cruise A with cabins  
Cabin 5 Cabin 4 Cabin 3  
||| [java] Removing created beans
```





## Exercise 7.3: Cascade Deletes in CMP 2.0

This very short exercise demonstrates the use of automatic cascade-delete offered by CMP 2.0 containers. It does this with an example Customer bean and some other beans related to it.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex07\_3* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex07_3> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex07_3> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex07_3> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

### ***Examine the JBoss Specific Files***

There are no new JBoss configuration files or components in this exercise.

## ***Examine and Run the Client Applications***

The example program, *Client\_73*, is a simple example to demonstrate cascade-delete. The example code is pretty straightforward and needs no explanation.

To run *Client\_73* invoke the ant task: `run.client_73`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex07_3>ant run.client_73
Buildfile: build.xml

prepare:

compile:

run.client_73:
[java] Creating Customer 10078, Addresses, Credit Card, Phones
[java] Creating CreditCard
[java] customer.getCreditCard().getName()=Ringo Star
[java] Creating Address
[java] Address Info: 780 Main Street Beverly Hills, CA 90210
[java] Creating Phones
[java] Adding a new type 1 phone number..
[java] Adding a new type 2 phone number.
[java] New contents of phone list:
[java] Type=1   Number=612-555-1212
[java] Type=2   Number=888-555-1212
[java] Removing Customer EJB only
```

## *Exercises for Chapter 8*

### **Exercise 8.1:**

#### **EJB 2.0 CMP: EJB QL 1**

The exercises in this section examine some of the basic aspects of EJB QL programming and functionality. They explore basic finder methods, ejbSelect methods, and the use of the IN operation in EJB QL queries.

#### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

#### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex08\_1* directory created by the extraction process
2. You must set the JAVA\_HOME and JBOSS\_HOME environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex08_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex08_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex08_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the JBoss Specific Files***

There are no new features introduced in JBoss specific files in this chapter. Please review Chapter 6.1 of this workbook to understand the JBoss specific files in this example.

## ***Initialize the Database***

The databases table for this exercise will automatically be created in JBoss' default database, HypersonicSQL, when the EJB jar is deployed. But to initialize all database tables in this example, you must perform the `ant` task `run.initialize`.

```
||| C:\workbook\ex08_1>ant run.initialize
Buildfile: build.xml

prepare:

compile:

run.initialize:
    [java] added Bill Burke
    [java] added Sacha Labourey
    [java] added Marc Fleury
    [java] added Jane Swift
    [java] added Nomar Garciaparra
```

As in the previous section, all example business logic is implemented within a stateless session bean. If you would like to see the database initialization code, there is an `initialize` method within this stateless bean, `com.titan.test.Test81Bean`, that creates all the necessary entity beans for this exercise.

## ***Examine and Run the Client Applications***

Each example method of `Test81Bean` illustrates the example code fragments shown in the EJB book. Each `Test81Bean` method is invoked by a small, simple client application.

### **Client\_81a**



The `Client_81a` example program demonstrates a few simple finder methods that are exposed through the `CustomerHome` interface.

```
public interface CustomerHomeLocal extends javax.ejb.EJBLocalHome
{
    ...
    public CustomerLocal findByName(String lastName,
                                     String firstName)
        throws FinderException;

    public Collection findByGoodCredit()
        throws FinderException;
    ...
}
```

These finder methods are defined in the `Customer` EJB's deployment descriptor as follows:

```
<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>
</query>
<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/>
  </query-method>
</query>
<ejb-ql>
  SELECT OBJECT(c) FROM Customer c
  WHERE c.hasGoodCredit = TRUE
</ejb-ql>
</query>
```

The example also demonstrates a few `ejbSelect` methods. These `ejbSelect` methods are defined in the `Address` EJB's deployment descriptor as follows:

```
<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/>
    </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.address.AddressLocal</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer AS c
        WHERE c.homeAddress = ?1
    </ejb-ql>
</query>

```

Since `ejbSelect` methods are private to the entity bean class, custom `Home` methods were needed in the `Address` home interface to wrap and invoke the private `ejbSelect` methods.

```

public interface AddressHomeLocal extends javax.ejb.EJBLocalHome
{
    ...
    public Collection queryZipCodes(String state)
        throws FinderException;

    public Collection queryAll()
        throws FinderException;

    public CustomerLocal queryCustomer(AddressLocal addr)
        throws FinderException;
}

```

These custom `Home` methods need corresponding `ejbHome` methods defined in the `Address` bean class. All they do is just delegate to the `ejbSelect` methods that they wrap.

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

    public abstract Collection ejbSelectZipCodes(String state)
        throws FinderException;

    public abstract Collection ejbSelectAll()
        throws FinderException;

    public abstract CustomerLocal ejbSelectCustomer(AddressLocal
addr)
        throws FinderException;

    public Collection ejbHomeQueryZipCodes(String state)
        throws FinderException
    {
        return ejbSelectZipCodes(state);
    }

    public Collection ejbHomeQueryAll()
        throws FinderException
    {
        return ejbSelectAll();
    }

    public CustomerLocal ejbHomeQueryCustomer(AddressLocal addr)
        throws FinderException
    {
        return ejbSelectCustomer(addr);
    }

    ...
}
```

Custom Home methods are described briefly in Chapter 5 of the EJB book and covered in more detail in Chapter 11. As you can see, they are extremely useful in exposing private `ejbSelect` methods so that they can be invoked by test programs or business logic. All the example programs for section 8 of this workbook use the custom Home methods for this purpose.

`Client_81a` invokes and displays the output from these queries. To run it invoke the ant task: `run.client_81a`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
||| C:\workbook\ex08_1>ant run.client_81a
```

Buildfile: build.xml

prepare:

compile:

run.client\_81a:

```
[java] FIND METHODS
[java] -----
[java] SELECT OBJECT(c) FROM Customer c
[java] WHERE c.lastName = ?1 AND c.firstName = ?2
[java] Find Bill Burke using findByName
[java]     Found Bill Burke
[java]
[java] SELECT OBJECT(c) FROM Customer c
[java] WHERE c.hasGoodCredit = TRUE
[java] Find all with good credit.  Sacha has bad credit!
[java]     Bill has good credit.
[java]     Marc has good credit.
[java]     Jane has good credit.
[java]     Nomar has good credit.
[java]
[java] SELECT METHODS
[java] -----
[java] SELECT a.zip FROM Address AS a
[java] WHERE a.state = ?1
[java] show.ejbSelectZipCodes with queryZipCodes
[java]     01821
[java]     02115
[java]     02116
[java]
[java] SELECT OBJECT(a) FROM Address AS a
[java] show.ejbSelectAll with queryAll
[java]     123 Boston Road
[java]     Billerica, MA 01821
[java]
[java]     Etwa Schweitzer Strasse
[java]     Irgendwo, Switzerland 07711
[java]
[java]     Sharondale Dr.
[java]     Atlanta, GA 06660
[java]
[java]     1 Beacon Street
[java]     Boston, MA 02115
[java]
[java]     1 Yawkey Way
```

```

[java]      Boston, MA 02116
[java]
[java]      West Broad Street
[java]      Richmond, VA 23233
[java]
[java]      Somewhere
[java]      Atlanta, GA 06660
[java]
[java]
[java] SELECT OBJECT(C) FROM Customer AS c
[java] WHERE c.homeAddress = ?1
[java] show ejbSelectCustomer using Bill's address.
[java] The customer is:
[java]      Bill Burke
[java]      123 Boston Road
[java]      Billerica, MA 01821

```

### Client\_81b

The Client\_81b example program investigates some of the queries illustrated in the EJB book under the *Simple Queries with Paths* section in Chapter 8. . Please refer to the EJB book for an explanation of the details of the tested queries below. The business logic for this example is implemented in `com.titan.test.Test81Bean` in the `test81b` method.

All the EJB QL queries in this example are `ejbSelect` methods. Again, these `ejbSelect` methods are wrapped by custom Home methods. The following Customer EJB QL queries and home methods are tested within this example:

```

query:      SELECT c.lastName FROM Customer AS c
ejbSelect method: ejbSelectLastNames
custom Home method: queryLastNames
ejbHome method: ejbHomeQueryLastNames

query:      SELECT c.creditCard FROM Customer c
ejbSelect method: ejbSelectCreditCards
custom Home method: queryCreditCards
ejbHome method: ejbHomeQueryCreditCards

query:      SELECT c.homeAddress.city FROM Customer c
ejbSelect method: ejbSelectCities
custom Home method: queryCities
ejbHome method: ejbHomeQueryCities

query:      SELECT c.creditCard.creditCompany.address
            FROM Customer AS c

```

ejbSelect method:       ejbSelectCreditCompanyAddresses  
custom Home method:   queryCreditCompanyAddresses  
ejbHome method:        ejbHomeQueryCreditCompanyAddresses

query:                   SELECT c.creditCard.creditCompany.address.city  
                          FROM Customer AS c

ejbSelect method:       ejbSelectCreditCompanyCities  
custom Home method:   queryCreditCompanyCities  
ejbHome method:        ejbHomeQueryCreditCompanyCities

Client\_81b invokes and displays the output from these queries. To run it invoke the ant task:  
`run.client_81b`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex08_1>ant run.client_81b
Buildfile: build.xml

prepare:

compile:

run.client_81b:
[java] SIMPLE QUERIES with PATHS
[java] -----
[java] SELECT c.lastName FROM Customer AS c
[java]      Burke
[java]      Labourey
[java]      Fleury
[java]      Swift
[java]      Garciaparra
[java]
[java] SELECT c.creditCard FROM Customer c
[java]      5324 9393 1010 2929
[java]      5311 5000 1011 2333
[java]      5310 5131 7711 2663
[java]      5810 5881 7788 2688
[java]      5450 5441 7448 2644
[java]
[java] SELECT c.homeAddress.city FROM Customer c
[java]      Billerica
[java]      Irgendwo
[java]      Atlanta
[java]      Boston
[java]      Boston
[java]
```

```

[java] SELECT c.creditCard.creditCompany.address
[java] FROM Customer AS c
[java]      West Broad Street
[java]      Richmond, VA 23233
[java]
[java]      West Broad Street
[java]      Richmond, VA 23233
[java]
[java]      West Broad Street
[java]      Richmond, VA 23233
[java]
[java]      Somewhere
[java]      Atlanta, GA 06660
[java]
[java]      Somewhere
[java]      Atlanta, GA 06660
[java]
[java]
[java] SELECT c.creditCard.creditCompany.address.city
[java] FROM Customer AS c
[java]      Richmond
[java]      Richmond
[java]      Richmond
[java]      Atlanta
[java]      Atlanta

```

### Client\_81c

The Client\_81c example program investigates some of the queries illustrated in the EJB book under the *IN Operator* section in Chapter 8. Please refer to the EJB book for an explanation of the details of the tested queries below. The business logic for this example is implemented in `com.titan.test.Test81Bean` in the `test81c` method.

All the EJB QL queries in this example are `ejbSelect` methods. Again, these `ejbSelect` methods are wrapped by custom Home methods. The following Customer EJB QL queries and home methods are tested within this example:

```

query:          SELECT OBJECT( r )
                  FROM Customer AS c, IN( c.reservations ) AS r
ejbSelect method: ejbSelectReservations
custom Home method: queryReservations
ejbHome method:   ejbHomeQueryReservations

query:          SELECT r.cruise
                  FROM Customer AS c, IN( c.reservations ) AS r
ejbSelect method: ejbSelectCruises

```

custom Home method: queryCruises  
ejbHome method: ejbHomeQueryCruises

query: SELECT cbn.ship  
FROM Customer AS c, IN( c.reservations ) AS r,  
IN( r.cabins ) AS cbn

ejbSelect method: ejbSelectShips  
custom Home method: queryShips  
ejbHome method: ejbHomeQueryShips

Client\_81c invokes and displays the output from these queries. To run it invoke the ant task:  
run.client\_81c. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:


```
C:\workbook\ex08_1>ant run.client_81c
Buildfile: build.xml

prepare:

compile:

run.client_81c:
[java] THE IN OPERATOR
[java] -----
[java] SELECT OBJECT( r )
[java] FROM Customer AS c, IN( c.reservations ) AS r
[java]     Reservation for Alaskan Cruise
[java]     Reservation for Alaskan Cruise
[java]     Reservation for Atlantic Cruise
[java]     Reservation for Atlantic Cruise
[java]     Reservation for Alaskan Cruise
[java]
[java] SELECT r.cruise
[java] FROM Customer AS c, IN( c.reservations ) AS r
[java]     Cruise Alaskan Cruise
[java]     Cruise Alaskan Cruise
[java]     Cruise Atlantic Cruise
[java]     Cruise Atlantic Cruise
[java]     Cruise Alaskan Cruise
[java]
[java] SELECT cbn.ship
[java] FROM Customer AS c, IN( c.reservations ) AS r,
[java] IN( r.cabins ) AS cbn
[java]     Ship Queen Mary
[java]     Ship Queen Mary
```





```
[java] Ship Queen Mary
[java] Ship Queen Mary
[java] Ship Titanic
[java] Ship Titanic
[java] Ship Titanic
[java] Ship Titanic
[java] Ship Titanic
[java] Ship Titanic
[java] Ship Queen Mary
[java] Ship Queen Mary
```





## Exercise 8.2: EJB QL Continued

The example programs in Exercise 8.2 delve deeper into the complexities of EJB QL. You will learn about arithmetic and logic operators in WHERE clauses as well as other more complex WHERE clause constructs. Most example queries provided in Chapter 8 of the EJB book are demonstrated in the test programs of this section.

### ***Startup JBoss***

If you already have JBoss running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Build and Deploy the Example Programs***

The examples of this section are built in the `ex08_2` directory. There are built the same as described in other chapters.

### ***Examine the JBoss Specific Files***

There are no new features introduced in JBoss specific files in this chapter. Please review Chapter 6.1 of this workbook to understand the JBoss specific files in this example.

### ***Initialize the Database***

The databases table for this exercise will automatically be created in JBoss' default database, HypersonicSQL, when the EJB jar is deployed. But to initialize all database tables in this example, you must perform the `ant` task `run.initialize`.

```
C:\workbook\ex08_2>ant run.initialize
Buildfile: build.xml

prepare:

compile:

run.initialize:
    [java] added Bill Burke
    [java] added Sacha Labourey
    [java] added Marc Fleury
    [java] added Jane Swift
    [java] added Nomar Garciaparra
    [java] added Richard Monson-Haefel
```

As in the previous section, all example business logic is implemented within a stateless session bean. If you would like to see the database initialization code, there is an `initialize` method within this stateless bean, `com.titan.test.Test82Bean`, that creates all the necessary entity beans for this exercise.

## ***Examine and Run the Client Applications***

Each example method of `Test82Bean` illustrates the example code fragments shown in the EJB book. Each `Test82Bean` method is invoked by a small, simple client application.

### **Client\_82a**

The `Client_82a` example program investigates the queries illustrated in the EJB book under the *Using DISTINCT* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82a` method.

The code demonstrates a Customer EJB finder query that returns duplicate responses, then invokes a finder query that uses the `DISTINCT` keyword to filter out duplicates.

```
finder method:      findAllCustomersWithReservations
query:              SELECT OBJECT( cust)
                    FROM Reservation res, IN (res.customers) cust

finder method:      findDistinctCustomersWithReservations
query:              SELECT DISTINCT OBJECT( cust)
                    FROM Reservation res, IN (res.customers) cust
```

`Client_82a` invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82a`. Don't forget to set your **JBOSS\_HOME** and **PATH** environment variables.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82a
Buildfile: build.xml

prepare:

compile:

run.client_82a:
[java] USING DISTINCT
[java] -----
[java] Non-distinct:
[java] SELECT OBJECT( cust)
[java] FROM Reservation res, IN (res.customers) cust
```

```

[java]      Bill has a reservation.
[java]      Sacha has a reservation.
[java]      Nomar has a reservation.
[java]      Bill has a reservation.
[java]      Marc has a reservation.
[java]      Jane has a reservation.
[java]
[java] Distinct:
[java] SELECT DISTINCT OBJECT( cust)
[java] FROM Reservation res, IN (res.customers) cust
[java]      Bill has a reservation.
[java]      Sacha has a reservation.
[java]      Marc has a reservation.
[java]      Jane has a reservation.
[java]      Nomar has a reservation.

```

### Client\_82b

The Client\_82b example program investigates the queries illustrated in the EJB book under *The WHERE Clause and Literals* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82b` method.

The code demonstrates a various Customer and Ship EJB finder queries that show how to use string, numeric, and Boolean literals in EJB QL queries.

```

EJB:          Customer
finder method: findByAmericanExpress
query:        SELECT OBJECT( c ) FROM Customer AS c
              WHERE c.creditCard.organization = 'American Express'

```

```

EJB:          Ship
finder method: findByTonnage100000
query:        SELECT OBJECT( s ) FROM Ship AS s
              WHERE s.tonnage = 100000.0

```

```

EJB:          Customer
finder method: findByGoodCredit
query:        SELECT OBJECT( c ) FROM Customer AS c
              WHERE c.hasGoodCredit = TRUE

```

Client\_82b invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82b`.

The output should look something like this:

```

||| C:\workbook\ex08_2>ant run.client_82b
||| Buildfile: build.xml

```

```

prepare:

compile:

run.client_82b:
    [java] THE WHERE CLAUSE AND LITERALS
    [java] -----
    [java] SELECT OBJECT( c ) FROM Customer AS c
    [java] WHERE c.creditCard.organization = 'American Express'
    [java]     Jane has an American Express card.
    [java]     Nomar has an American Express card.
    [java]
    [java] SELECT OBJECT( s ) FROM Ship AS s
    [java] WHERE s.tonnage = 100000.0
    [java]     Ship Queen Mary as tonnage 100000.0
    [java]
    [java] SELECT OBJECT( c ) FROM Customer AS c
    [java] WHERE c.hasGoodCredit = TRUE
    [java]     Bill has good credit.
    [java]     Marc has good credit.
    [java]     Jane has good credit.
    [java]     Nomar has good credit.
    [java]     Richard has good credit.

```

### Client\_82c

The Client\_82c example program investigates the queries illustrated in the EJB book under *The WHERE Clause and Input Parameters* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82c` method.

The code demonstrates a Customer EJB `ejbSelect` query that uses strings as input parameters to the query, and a Cruise EJB finder method that uses a Ship EJB as an input parameter. As in previous sections, the `ejbSelect` query is wrapped in a custom Home method.

```

EJB:                Customer
ejbSelect method:    ejbSelectLastNames
custom Home method: queryLastNames
ejbHome method:      ejbHomeQueryLastNames
query:               SELECT OBJECT( c ) FROM Customer AS c
                     WHERE c.homeAddress.state = ?2
                     AND c.homeAddress.city = ?1

EJB:                Cruise
finder method:       findByShip

```

```
query:          SELECT OBJECT( crs ) FROM Cruise AS crs
                  WHERE crs.ship = ?1
```

Client\_82c invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82c`. Don't forget to set your **JBoss\_HOME** and **PATH** environment variables.

The output should look something like this:

```
||| C:\workbook\ex08_2>ant run.client_82c
    Buildfile: build.xml

prepare:

compile:

run.client_82c:
[java] THE WHERE CLAUSE AND INPUT PARAMETERS
[java] -----
[java] SELECT OBJECT( c ) FROM Customer AS c
[java] WHERE c.homeAddress.state = ?2
[java] AND c.homeAddress.city = ?1
[java] Get customers from Billerica, MA
[java]   Bill is from Billerica.
[java]
[java] SELECT OBJECT( crs ) FROM Cruise AS crs
[java] WHERE crs.ship = ?1
[java] Get cruises on the Titanic
[java]   Atlantic Cruise is a Titanic cruise.
```

### Client\_82d

The Client\_82d example program investigates the queries illustrated in the EJB book under *The WHERE Clause andCDATA Sections* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82d` method.

The code demonstrates a Reservation EJB finder method that must be enclosed in an XML CDATA section because it uses the > symbol in the query.

```
EJB:             Reservation
finder method:    findWithPaymentGreaterThan
query:           <![CDATA[
                  OBJECT( r ) FROM Rreservation r
                  WHERE r.amountPaid > ?1
                  ]]>
```



Client\_82d invokes and displays the output from this query. To run it invoke the ant task:  
`run.client_82d`.

The output should look something like this:

```
||| C:\workbook\ex08_2>ant run.client_82d
||| Buildfile: build.xml

||| prepare:

||| compile:

||| run.client_82d:
||| [java] THE WHERE CLAUSE AND CDATA Sections
||| [java] -----
||| [java] ![CDATA[
||| [java] SELECT OBJECT( r ) FROM Rreservation r
||| [java] WHERE r.amountPaid > ?1
||| [java] ]]>
||| [java]      found reservation with amount paid > 20000.0: 40000.0
```

### Client\_82e

The Client\_82e example program investigates the queries illustrated in the EJB book under *The WHERE Clause and BETWEEN* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82e` method.

Two separate Ship EJB finder methods demonstrate how to use the BETWEEN keyword in a WHERE clause.

```
EJB:      Ship
finder method:  findByTonnageBetween
query:      SELECT OBJECT( s ) FROM Ship s
            WHERE s.tonnage BETWEEN 80000.00 and 130000.00
```

```
EJB:      Ship
finder method:  findByTonnageNotBetween
query:      SELECT OBJECT( s ) FROM Ship s
            WHERE s.tonnage NOT BETWEEN 80000.00 and 130000.00
```

Client\_82e invokes and displays the output from these queries. To run it invoke the ant task:  
`run.client_82e`.

The output should look something like this:

```
||| C:\workbook\ex08_2>ant run.client_82e
||| Buildfile: build.xml
```

```

prepare:

compile:

run.client_82e:
    [java] THE WHERE CLAUSE AND BETWEEN
    [java] -----
    [java] SELECT OBJECT( s ) FROM Ship s
    [java] WHERE s.tonnage BETWEEN 80000.00 and 130000.00
    [java]      Queen Mary has tonnage 100000.0
    [java]
    [java] SELECT OBJECT( s ) FROM Ship s
    [java] WHERE s.tonnage NOT BETWEEN 80000.00 and 130000.00
    [java]      Titanic has tonnage 200000.0

```

### Client\_82f

The Client\_82f example program investigates the queries illustrated in the EJB book under *The WHERE Clause and IN* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82f` method.

The code uses two Customer EJB finder methods. One finder queries for all customers living in Georgia or Massachusetts. The other finder queries for all customers that do not live in these two states.

```

EJB:           Customer
finder method: findInStates
query:         SELECT OBJECT( c ) FROM Customer c
               WHERE c.homeAddress.state IN ( 'GA', 'MA' )

```

Client\_82f invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82f`.

The output should look something like this:

```

C:\workbook\ex08_2>ant run.client_82f
Buildfile: build.xml

prepare:

compile:

run.client_82f:
    [java] THE WHERE CLAUSE AND IN
    [java] -----
    [java] SELECT OBJECT( c ) FROM Customer c
    [java] WHERE c.homeAddress.state IN ( 'GA', 'MA' )

```

```

[java]      Bill
[java]      Marc
[java]      Jane
[java]      Nomar
[java]
[java] SELECT OBJECT( c ) FROM Customer c
[java] WHERE c.homeAddress.state NOT IN ( 'GA', 'MA' )
[java]      Sacha

```

### Client\_82g

The Client\_82g example program investigates the queries illustrated in the EJB book under *The WHERE Clause and IS NULL* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82g` method.

There are two Customer EJB finder methods. One finder selects all customers that have a `null` home address. The other finder selects all customers that do not have a `null` address.

```

EJB:          Customer
finder method: findHomeAddressIsNull
query:        SELECT OBJECT( c ) FROM Customer c
              WHERE c.homeAddress IS NULL

```

```

EJB:          Customer
finder method: findHomeAddressIsNotNull
query:        SELECT OBJECT( c ) FROM Customer c
              WHERE c.homeAddress IS NOT NULL

```

Client\_82g invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82g`.

The output should look something like this:

```

C:\workbook\ex08_2>ant run.client_82g
Buildfile: build.xml

prepare:

compile:

run.client_82g:
[java] THE WHERE CLAUSE AND IS NULL
[java] -----
[java] SELECT OBJECT( c ) FROM Customer c
[java] WHERE c.homeAddress IS NULL
[java]      Richard

```

```

[java]
[java] SELECT OBJECT( c ) FROM Customer c
[java] WHERE c.homeAddress IS NOT NULL
[java]      Bill
[java]      Sacha
[java]      Marc
[java]      Jane
[java]      Nomar

```

## Client\_82h

The Client\_82h example program investigates the queries illustrated in the EJB book under *The WHERE Clause and IS EMPTY* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82h` method.

The code uses two different Cruise EJB finder methods to illustrate the use of `IS EMPTY`. One finder will return all the Cruises that do not have Reservations. The other finder method queries all Cruises that have Reservations.

- ❖ Unfortunately, as of the writing of this workbook, `IS NOT EMPTY` does not work properly and will throw a runtime exception. The invocation of this test is commented out in `test82h`.

```

EJB:           Cruise
finder method: findEmptyReservations
query:         SELECT OBJECT( crs ) FROM Cruise crs
                WHERE crs.reservations IS EMPTY

```

```

EJB:           Cruise
finder method: findNotEmptyReservations
query:         SELECT OBJECT( crs ) FROM Cruise crs
                WHERE crs.reservations IS NOT EMPTY

```

Client\_82h invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82h`.

The output should look something like this:

```

C:\workbook\ex08_2>ant run.client_82h
Buildfile: build.xml

prepare:

compile:

run.client_82h:

```

```

[java] THE WHERE CLAUSE AND IS EMPTY
[java] -----
[java] SELECT OBJECT( crs ) FROM Cruise crs
[java] WHERE crs.reservations IS EMPTY

```

## Client\_82i

The Client\_82i example program investigates the queries illustrated in the EJB book under *The WHERE Clause and MEMBER OF* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82i` method.

Two Cruise EJB finder methods demonstrate how to use EJB QL to find whether or not an entity is a member of a relationship.

```

EJB:           Cruise
finder method: findMemberOf
query:         SELECT OBJECT( crs ) FROM Cruise crs,
               IN (crs.reservations) res, Customer cust
               WHERE cust = ?1 ANT cust MEMBER OF res.customers

```

```

EJB:           Cruise
finder method: findNotMemberOf
query:         SELECT OBJECT( crs ) FROM Cruise crs,
               IN (crs.reservations) res, Customer cust
               WHERE cust = ?1 ANT cust NOT MEMBER OF res.customers

```

Client\_82i invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82i`.

The output should look something like this:

```

C:\workbook\ex08_2>ant run.client_82i
Buildfile: build.xml

prepare:

compile:

run.client_82i:
[java] THE WHERE CLAUSE AND MEMBER OF
[java] -----
[java] SELECT OBJECT( crs ) FROM Cruise crs,
[java] IN (crs.reservations) res, Customer cust
[java] WHERE cust = ?1 ANT cust MEMBER OF res.customers
[java] Use Bill Burke
[java]   Bill is member of Alaskan Cruise

```

```

[java]      Bill is member of Atlantic Cruise
[java]
[java] SELECT OBJECT( crs ) FROM Cruise crs,
[java] IN (crs.reservations) res, Customer cust
[java] WHERE cust = ?1 ANT cust NOT MEMBER OF res.customers
[java] Use Nomar Garciaparra
[java]      Nomar is not member of Atlantic Cruise

```

### Client\_82j

The Client\_82j example program investigates the queries illustrated in the EJB book under *The WHERE Clause and LIKE* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82j` method.

One Customer EJB finder method is used to query all Customers with a hyphenated name.

```

EJB:           Customer
finder method: findHyphenatedLastNames
query:         SELECT OBJECT( c ) FROM Customer c
               WHERE c.lastName LIKE '%-%'

```

Client\_82j invokes and displays the output from these queries. To run it invoke the ant task: `run.client_82j`.

The output should look something like this:

```

C:\workbook\ex08_2>ant run.client_82j
Buildfile: build.xml

prepare:

compile:

run.client_82j:
[java] THE WHERE CLAUSE AND LIKE
[java] -----
[java] SELECT OBJECT( c ) FROM Customer c
[java] WHERE c.lastName LIKE '%-%'
[java]      Monson-Haefel

```

### Client\_82k

The Client\_82k example program investigates the queries illustrated in the EJB book under *The WHERE Clause and Functional Expressions* section in Chapter 8. The business logic for this example is implemented in `com.titan.test.Test82Bean` in the `test82k` method.

One Customer EJB finder method demonstrates the use of a couple of functional expressions.

```
EJB:           Customer
finder method: findByLastNameLength
query:         SELECT OBJECT( c ) FROM Customer c
               WHERE LENGTH(c.lastName) > 6 AND
               LOCATE(c.lastName, 'Monson') > -1
```

Client\_82k invokes and displays the output from these queries. To run it invoke the ant task:  
`run.client_82k`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82k
Buildfile: build.xml

prepare:

compile:

run.client_82k:
[java] THE WHERE CLAUSE AND FUNCTIONAL EXPRESSIONS
[java] -----
[java] SELECT OBJECT( c ) FROM Customer c
[java] WHERE LENGTH(c.lastName) > 6 AND
[java] LOCATE(c.lastName, 'Monson') > -1
[java]      Labourey
[java]      Garciaparra
[java]      Monson-Haefel
```

### Advanced JBoss QL

In section *Problems with EJB QL* of Chapter 8 of the EJB book, Richard Monson-Haefel talks about some of the limitations of EJB QL. In the JBoss CMP 2.0 implementation, EJB QL is just a subset of a larger JBoss query language. Dain Sundstrom, the architect of the CMP 2.0 engine, did a great job of filling in some of the gaps in the EJB QL spec. Features like `ORDER BY` and the allowance of parameters within `IN` and `LIKE` clauses are just a few of the enhancements Dain has implemented. Please review his advanced CMP 2.0 documentation available at the JBoss website, <http://www.jboss.org/>, for more information on these cool features.

## Exercises for Chapter 10

### Exercise 10.1: A BMP Entity Bean

In this exercise, you will build and examine a simple EJB that uses *bean-managed persistence* (BMP) to synchronize the state of the bean with a database. You will also build a client application to test this Ship BMP bean.

#### Startup JBoss

If JBoss is already running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

#### Initialize the Database

As in the CMP examples, the state of the entity beans will be stored in the database that is embedded in JBoss. Whereas JBoss was able to create all tables for CMP beans, it cannot be done for BMP since the deployment descriptors do not contain any persistence information (object to relational mapping for example). The bean is in fact the only one that knows how to load, store, remove and find data. The persistence mapping is not described in a configuration file, but instead, embedded in the bean code.

This means that the database environment for BMP always needs to be explicitly built. To make this task easier for the BMP Ship example, two *home methods* have been defined in its home interface.

- ❖ Entity beans can define *home methods* that perform operations related to the EJB component semantic but that are not linked to any particular bean instance. As an analogy, you can compare this concept with class static methods: while static methods semantic is generally highly related to the class semantic, they are not associated to any particular class instance. Don't worry if this is not very clear for you yet: in chapter 11 of the EJB book, you will learn all about home methods.

Here is a partial view of the home interface of the Ship EJB:

```
public interface ShipHomeRemote extends javax.ejb.EJBHome {  
    ...  
    public void makeDbTable () throws RemoteException;  
    public void deleteDbTable () throws RemoteException;  
}
```



It defines two home methods. The first one will create the table needed by the Ship EJB in the JBoss embedded database and the second will drop it.

The implementation of the `makeDbTable` home method is essentially a `CREATE TABLE` SQL statement:

```
public void ejbHomeMakeDbTable () throws SQLException
{
    PreparedStatement ps = null;
    Connection con = null;
    try
    {
        con = this.getConnection ();

        System.out.println("Creating table SHIP...");
        ps = con.prepareStatement ("CREATE TABLE SHIP ( " +
                                   "ID INT PRIMARY KEY, " +
                                   "NAME CHAR (30), " +
                                   "TONNAGE DECIMAL (8,2), " +
                                   "CAPACITY INT" +
                                   ") " );

        ps.execute ();
        System.out.println("...done!");
    }
    finally
    {
        try { ps.close (); } catch (Exception e) {}
        try { con.close (); } catch (Exception e) {}
    }
}
```

The `deleteDbTable` home method only differs by the SQL statement that is executed:

```
...
System.out.println("Dropping table SHIP...");
ps = con.prepareStatement ("DROP TABLE SHIP");
ps.execute ();
System.out.println("...done!");
...
```

You will see how to call these methods in a subsequent section.

### ***Examine the EJB Standard Files***

The Ship EJB source code requires no modification to run in JBoss, so the standard EJB deployment descriptor is very simple:

### ***ejb-jar.xml (part I)***

```
...
<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 DB</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>
...
```

This first part of the deployment descriptor essentially tells the container that the Ship bean:

- is named `ShipEJB`,
- has a persistence type set to `Bean` because it is a BMP bean,
- declares a reference to a data source named `jdbc/titanDB`.

Since the bean directly manages the persistence logic, the deployment descriptor does not contain any persistence information. In contrast, this information would have been mandatory for a CMP EJB.

The second part of the deployment descriptor declares the transactional and security attributes of the Ship bean:

### ***ejb-jar.xml (part II)***

```
...
<assembly-descriptor>
  <security-role>
    <description>
```

```

        This role represents everyone who is allowed full
        access to the Ship EJB.
    </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>

</ejb-jar>

```

All methods on the Ship bean require a transaction. If no transaction is active while a method invocation enters the container, a new one will be started.

- ❖ Transactions in entity beans are always managed by the container and never directly by the bean. Thus, all work done on transactional resources, such as databases, will implicitly be part of the transactional context of the container.

## ***Examine the JBoss Specific Files***

If you do not include a *jboss.xml* specific deployment descriptor with your bean, JBoss will make the following decisions at deployment time:

- It will bind the Ship bean in the public JNDI tree under `/ShipEJB` (which is the name given to the bean in its associated `ejb-jar.xml` deployment descriptor).
- It will link the `jdbc/titanDB` datasource expected by the bean to `java:/DefaultDS` which represents a default datasource (to the embedded database).

Unless you require different settings, you do not need to provide a *jboss.xml* file. While this naming convention is generally useful for quick-prototyping, it will not satisfy more complex

deployment situations. Furthermore, using a JBoss specific deployment descriptor enables you to fine tune a container for a particular situation.

In JBoss, there is a one-to-one mapping between a bean and a container, and each container can be configured independently. If you take a look at the *JBOSS\_HOME/server/default/conf/standardjboss.xml* file, you will find all default container settings that are predefined in JBoss (standard BMP, standard CMP, clustered BMP, etc.)

When you write a JBoss specific deployment descriptor, you can either:

- Not specify any container configuration. In this case the default configuration found in *standardjboss.xml* is used by JBoss.
- Create a brand new container configuration. In this case the default settings are not used at all.
- Modify an existing configuration. In this case the defaults settings are first loaded from the existing configuration found in *standardjboss.xml* and merged with the differences indicated in the *jboss.xml* deployment descriptor. This solution allows you to make minor modifications to the default container with minimal writing in your deployment descriptor.

This last option is used in the Ship bean deployment descriptor to test its behaviour with different commit options. As outlined below, this new configuration only defines a single setting (*commit-option*). All others are inherited from the *Standard BMP EntityBean* configuration declared in the *standardjboss.xml* file.

#### ***jboss.xml (part I)***

```
<?xml version="1.0" encoding="Cp1252"?>

<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Standard BMP EntityBean</container-name>
      <commit-option>A</commit-option>
    </container-configuration>
  </container-configurations>
</jboss>
```

Since multiple EJBs could be defined in a single deployment descriptor, the role of the *<ejb-name>* tag is to link the definitions from the *ejb-jar.xml* and *jboss.xml* files. This tag can be considered as the bean **identifier**. The *<jndi-name>* tag determines the name under which the client applications will be able to lookup the home of our EJB, in this case *ShipHomeRemote*.

You can also see how our bean references a specific configuration thanks to the *<configuration-name>* tag.

### ***jboss.xml (part II)***

```
<enterprise-beans>
  <entity>
    <ejb-name>ShipEJB</ejb-name>
    <jndi-name>ShipHomeRemote</jndi-name>
    <resource-ref>
      <res-ref-name>jdbc/titanDB</res-ref-name>
      <jndi-name>java:/DefaultDS</jndi-name>
    </resource-ref>
    <configuration-name>Standard BMP EntityBean
    </configuration-name>
  </entity>
</enterprise-beans>
</jboss>
```

The Ship bean BMP implementation explicitly needs a database connection. The acquisition of this resource is done in the getConnection method:

### ***ShipBean.java***

```
private Connection getConnection () throws SQLException
{
    try
    {
        Context jndiCtx = new InitialContext ();
        DataSource ds =
            (DataSource)jndiCtx.lookup ("java:comp/env/jdbc/titanDB");
        return ds.getConnection ();
    }
    ...
}
```

The bean expects to find a datasource bound to the java:comp/env/jdbc/titanDB JNDI name. That is why the ejb-jar.xml file contains the following declaration:

### ***ejb-jar.xml***

```
...
<resource-ref>
  <description>DataSource for the Titan DB</description>
  <res-ref-name>jdbc/titanDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
...
```

Then the jboss.xml maps the jdbc/titanDB datasource name to the actual name that has been defined in JBoss:

### ***jboss.xml***

```
|||      ...
|||      <resource-ref>
|||          <res-ref-name>jdbc/titanDB</res-ref-name>
|||          <jndi-name>java:/DefaultDS</jndi-name>
|||      </resource-ref>
|||      ...
```

In any default JBoss installation, `java:/DefaultDS` represents the embedded database.

## ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex10\_1* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
||| C:\workbook\ex10_1> set JAVA_HOME=C:\jdk1.3
||| C:\workbook\ex10_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
||| $ export JAVA_HOME=/usr/local/jdk1.3
||| $ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex10_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the Client Application***

In the “Initialize the Database” section, you saw how the bean implements the home methods that create and drop the table in the database. Let’s see now how the client application will call these home methods:

### ***Client\_101.java***

```
public class Client_101
{
    public static void main (String [] args)
    {
        try
        {
            Context jndiContext = getInitialContext ();

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

            // We check if we have to build the database schema...
            //
            if ( (args.length > 0) &&
                args[0].equalsIgnoreCase ("CreateDB") )
            {
                System.out.println ("Creating database table...");
                home.makeDbTable ();
            }
            // ... or if we have to drop it...
            //
            else if ( (args.length > 0) &&
                args[0].equalsIgnoreCase ("DropDB") )
            {
                System.out.println ("Dropping database table...");
                home.deleteDbTable ();
            }
            else
            ...
        }
    }
}
```

You can see that depending on the first argument found on the command line, either `CreateDB` or `DropDB`, the client application will call the corresponding home method.

If nothing is specified on the command line, the client will test our BMP bean:

### ***Client\_101.java***

```
...
else
{
    // ... standard behaviour
    //
    System.out.println ("Creating Ship 101..");
    ShipRemote ship1 = home.create (new Integer
```

```

        (101), "Edmund Fitzgerald");

    ship1.setTonnage (50000.0);
    ship1.setCapacity (300);

    Integer pk = new Integer (101);

    System.out.println ("Finding Ship 101 again..");
    ShipRemote ship2 = home.findByPrimaryKey (pk);

    System.out.println (ship2.getName ());
    System.out.println (ship2.getTonnage ());
    System.out.println (ship2.getCapacity ());

    System.out.println ("ship1.equals (ship2) == " +
        ship1.equals (ship2));

    System.out.println ("Removing Ship 101..");
    ship2.remove ();
}
...

```

The client application first creates a new bean and calls some remote methods on it to set its tonnage and capacity. Then, it finds it again by calling `findByPrimaryKey` and compares both bean references for equality. Since they represent the same bean instance, they must be equal. We have dropped the exception handling part because it deserves no specific comments.

## ***Run the Client Application***

Testing the BMP bean is a three steps process:

1. Creating the database table
2. Testing the bean (possibly many times)
3. Dropping the database table

For each of these steps, a different ANT target is available.

### **Creating the Database Table**

To drop the table, use the `createdb_101` ant target:

```

C:\workbook\ex10_1>ant createdb_101
Buildfile: build.xml

```



```

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

createdb_101:
    [java] Creating database table...

```

On the JBoss side, the BMP bean displays the following lines:

```

...
12:31:42,584 INFO    [STDOUT] Creating table SHIP...
12:31:42,584 INFO    [STDOUT] ...done!
...

```

Once this step has been performed, the actual testing of the BMP bean can take place.

- ❖ **Note.** If you are having trouble creating the database, shutdown JBoss. Then run the Ant build target: `clean.db`. This will remove all database files and allow you to start from fresh.

## Testing the BMP bean

To test the BMP bean, use the `run.client_101` ANT target:

```

C:\workbook\ex10_1>ant run.client_101
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

```

```
run.client_101:
    [java] Creating Ship 101..
    [java] Finding Ship 101 again..
    [java] Edmund Fitzgerald
    [java] 50000.0
    [java] 300
    [java] ship1.equals (ship2) == true
    [java] Removing Ship 101..
```

Even though this is not particularly related to BMP beans, let's focus on an interesting issue that occurs when the bean is first created and initialized by the client application:

```
ShipRemote ship1 = home.create (new Integer
                                (101), "Edmund Fitzgerald");

ship1.setTonnage (50000.0);
ship1.setCapacity (300);
```

Interestingly enough this piece of code generates 3 different transactions on the server side. The client does not implicitly start any transaction in its code. The transaction only starts when the invocation enters the bean container and commits when the invocation leaves the container. Thus, when the client performs 3 calls, each one is executed in its own transactional context.

Let's examine what it means for our BMP bean:

```
14:36:31,730 INFO [STDOUT] ejbCreate() pk=101 name=Edmund
Fitzgerald
14:36:31,780 INFO [STDOUT] ejbStore() pk=101
14:36:31,840 INFO [STDOUT] setTonnage()
14:36:31,840 INFO [STDOUT] ejbStore() pk=101
14:36:31,860 INFO [STDOUT] setCapacity()
14:36:31,860 INFO [STDOUT] ejbStore() pk=101
```

As you can see, `ejbStore` is called at the end of each transaction! Consequently, for these 3 lines of code, the bean is stored 3 times. Worst of all, after any method invocation, the container has no way of knowing whether the state of the bean has been modified. Given that there is no read-only method concept in EJBs, `getXXX` calls also result in `ejbStore` being called:

```
15:03:19,301 INFO [STDOUT] getName()
15:03:19,311 INFO [STDOUT] ejbStore() pk=101
15:03:19,331 INFO [STDOUT] getTonnage()
15:03:19,331 INFO [STDOUT] ejbStore() pk=101
15:03:19,371 INFO [STDOUT] getCapacity()
15:03:19,371 INFO [STDOUT] ejbStore() pk=101
```

During the whole execution of the test program, `ejbStore` has been called 7 times.

Remember that the bean is currently running in commit option A. Which means that JBoss is allowed to cache bean instances and reuse them in other transactions. We can see that for two reasons:

1. the `findByPrimaryKey` call is never displayed in the log. The container first checks if the cache already contains an instance for the given primary key. Since this is the case, there is no need to invoke the method on the bean implementation.
2. `ejbLoad` is never called for our bean. At the start of each new transaction the bean is already in cache and there is no need to reload it from the database.

In some situations, commit option A is not viable. For example, if some other system accesses the database and modifies its content, fresh bean instances have to be reloaded from the database at the start of each new transaction to be able to see the modifications. To see this difference of behaviour, change the commit option in *jboss.xml* to C:

### ***jboss.xml***

```
...
    <container-configurations>
        <container-configuration>
            <container-name>Standard BMP EntityBean</container-name>
            <commit-option>C</commit-option>
        </container-configuration>
    </container-configurations>
```

If you run the tests again, the output is as follow:

```
14:41:29,798 INFO [STDOUT] ejbCreate() pk=101 name=Edmund
Fitzgerald
14:41:30,449 INFO [STDOUT] ejbStore() pk=101
14:41:30,539 INFO [STDOUT] ejbLoad() pk=101
14:41:30,599 INFO [STDOUT] setTonnage()
14:41:30,609 INFO [STDOUT] ejbStore() pk=101
14:41:30,659 INFO [STDOUT] ejbLoad() pk=101
14:41:30,669 INFO [STDOUT] setCapacity()
14:41:30,679 INFO [STDOUT] ejbStore() pk=101
14:41:30,709 INFO [STDOUT] ejbFindByPrimaryKey() primaryKey=101
14:41:30,729 INFO [STDOUT] ejbLoad() pk=101
14:41:30,750 INFO [STDOUT] getName()
14:41:30,750 INFO [STDOUT] ejbStore() pk=101
14:41:30,780 INFO [STDOUT] ejbLoad() pk=101
14:41:30,790 INFO [STDOUT] getTonnage()
14:41:30,800 INFO [STDOUT] ejbStore() pk=101
14:41:30,840 INFO [STDOUT] ejbLoad() pk=101
14:41:30,850 INFO [STDOUT] getCapacity()
14:41:30,860 INFO [STDOUT] ejbStore() pk=101
14:41:30,880 INFO [STDOUT] ejbLoad() pk=101
14:41:30,900 INFO [STDOUT] ejbStore() pk=101
```

```
||| 14:41:30,910 INFO [STDOUT] ejbRemove() pk=101
```

Now, in addition to the `ejbStore` calls already present in commit option A, we have `ejbLoad` calls at the start of each new transaction as well as the `ejbFindByPrimaryKey` call that reaches the bean implementation because it cannot be resolved within the cache.

- ❖ JBoss includes a proprietary “commit option D”. This option is a compromise between commit option A and C. For commit option D, the bean instance can be cached across transactions. But, a configurable timeout value indicates when this cached data is stale and must be reloaded from the database. This option is very useful when you want the advantages of commit option A, but want cached entities to be updated periodically just in case an external system has modified the data directly.

## Dropping the Database Table

Once all tests have been performed, it is possible to clean the database environment associated with the BMP bean by removing the unused table. This is the role of the `dropdb_101` target:

```
||| C:\workbook\ex10_1>ant dropdb_101
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

dropdb_101:
    [java] Dropping database table...
```

On the JBoss side, the BMP bean logs the following lines:

```
||| ...
14:40:34,339 INFO [STDOUT] Dropping table SHIP...
14:40:34,349 INFO [STDOUT] ...done!
||| ...
```

## Possible Optimizations

As you have seen during the execution of the client application, the Ship bean performs many `ejbLoad` and `ejbStore` operations. There are two reasons behind this behavior:

1. Many transactions are started
2. The Ship bean BMP code is not optimized

The first point can be improved in different ways:

- Add less fine-grained methods that return all attributes of the bean in a single data object or add a new create method with many parameters allowing creating and initializing the bean in a single call.
- Use a stateless session bean (façade pattern) that will start the transaction and perform all steps in a single transaction
- Start a transaction in our client application (by using a `UserTransaction` object)

BMP code optimization is a wide topic. Here are some tricks that are frequently used:

- Use an `isModified` flag in your bean that will be activated each time the state of the bean is modified (within `setXXX` methods for example). In the implementation of `ejbStore`, only perform the actual database call if `isModified` is set to `true`. With this optimization, all the `ejbStore` calls resulting from the `getXXX` invocations from our test application will detect that no data has been modified and will not try to synchronize with the database.
- Detect which fields are actually modified during a transaction and only update these particular fields in the database. This is actually useful for beans with lots of fields or with fields containing large data. For example, in the case of the Ship BMP bean, each `setXXX` call updates all fields of the database whilst only one is effectively modified.

Note that many of these optimizations are automatically performed by default in any decent CMP engine.

## *Exercises for Chapter 12*

### **Exercise 12.1: Stateless Session Bean**

In this exercise, you will build and examine a stateless session bean, `ProcessPaymentEJB`, that writes payment information to the database. You will also build a client application to test this `ProcessPayment` bean.

The bean will directly insert the payment information data in the database, without using an intermediary entity bean.

#### ***Examine the EJB***

This example is based on the `Customer` and `Address` EJBs and their related data objects that have been used in chapter 6.3. As no modification has been made to these EJBs, this example focuses on the `ProcessPayment` stateless session bean.

The `ProcessPayment` has a very simple remote interface. It gives the choice to process a payment either by check, cash, or credit card. Each possibility is handled by a different method:

#### ***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;
    ...
}
```

Each method accepts, as its first parameter, a `CustomerRemote` interface. This will allow the `ProcessPayment` EJB to get any information it needs about the customer.

- ❖ It is possible to use EJB remote interfaces as parameters of other EJB methods because they extend `EJBObject`, which in turn extends from `java.rmi.Remote`. Objects implementing either `Remote` or `Serializable` are perfectly valid RMI types. For the EJB container, it makes strictly no difference.

The last parameter is always the amount of the transaction. Then, depending on the payment method that is chosen, a specific data object containing the details of the transaction is passed in parameter.

A data object is a `Serializable` object that can be passed by value back and forth between a client and a remote server. Most of the time, it contains minimal behaviour and is more frequently a simple data container. For example, the `CheckDO` contains the check number and bar code:

#### ***CheckDO.java***

```
public class CheckDO implements java.io.Serializable
{
    public String checkBarCode;
    public int checkNumber;

    public CheckDO (String barCode, int number)
    {
        this.checkBarCode = barCode;
        this.checkNumber = number;
    }
}
```

Let's now focus on the `ProcessPayment` EJB implementation. Each remote method will first perform some validity tests about the payment. Then, all methods will eventually call the same private method: `process`. This method contains the persistence logic used to insert the process payment information into the database. For example, this is how it is implemented by the `byCredit` method:

#### ***ProcessPaymentBean.java***

```
public boolean byCredit (CustomerRemote customer,
                        CreditCardDO card,
                        double amount)
throws PaymentException
{
    if (card.expiration.before (new java.util.Date ()))
    {
        throw new PaymentException ("Expiration date has passed");
    }
    else

```

```

    {
        return process (getCustomerID (customer),
                        amount,
                        CREDIT,
                        null,
                        -1,
                        card.number,
                        new java.sql.Date (card.expiration.getTime ()));
    }
}

```

The method first checks if the credit card has expired. In this case, an *application exception* is raised. Otherwise, the method simply delegates the insertion of the payment information in the database to the `process` private method. Note that some parameters that are passed to `process` are meaningless. The fourth parameter for example represents the check bar code which, in the case of a credit card payment, is not available. Instead, a dummy value is passed.

The `process` method is very similar to the `ejbCreate` method of the BMP example in chapter 10. It simply gets a datasource connection, creates a `PreparedStatement` and inserts the payment information into the `PAYMENT` table:

### ***ProcessPaymentBean.java***

```

...
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.setString (6, creditNumber);
ps.setDate (7, creditExpDate);

int retVal = ps.executeUpdate ();
if (retVal!=1)
{
    throw new EJBException ("Payment insert failed");
}

return true;

```



||| ...

Note that the returned value is not very significant. Either `true` is returned or an application exception is thrown. Consequently, the signature of `process` could be changed to return `void`.

## ***Examine the EJB Standard Deployment Descriptor***

The `ProcessPayment` standard deployment descriptor is very similar to the one of the previous examples.

### ***ejb-jar.xml***

```
||| ...
<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>
||| ...
```

Note that the `session-type` tag of the `ProcessPaymentEJB` bean is set to `Stateless` and that the `transaction-type` tag is set to `Container`. This means that the container will automatically handle the transaction management and enlistment of transactional resources with the running transaction for our bean. You will learn in chapter 14 of the EJB book how this could be handled by the EJB itself (session beans and message driven beans only).

The descriptor contains a reference to a data source that it will use to store the payments. The usage of this data source is identical to the one made by the BMP example in chapter 10.

### ***ProcessPaymentBean.java***

```
private Connection getConnection () throws SQLException
{
    try
    {
        InitialContext jndiCtx = new InitialContext ();

        DataSource ds = (DataSource)
            jndiCtx.lookup ("java:comp/env/jdbc/titanDB");

        return ds.getConnection ();
    }
    catch(NamingException ne)
    {
        throw new EJBException (ne);
    }
}
```

An environment property, `minCheckNumber`, is also specified. Environment properties provide a very flexible way to parameterize the behavior of a bean at deployment time. The `env-entry` for `minCheckNumber` gives the type of the property (`java.lang.Integer`) and a default value (2000). The ProcessPayment EJB will access the value of this property through its JNDI ENC:

### ***ProcessPaymentBean.java***

```
...
InitialContext jndiCtx = new InitialContext ();

Integer value = (Integer) jndiCtx.lookup
    ("java:comp/env/minCheckNumber");
...
```

One very interesting point to note is that although the ProcessPayment bean works with Customer beans (each remote method has a Customer interface as its first parameter), no reference (`ejb-ref` or `ejb-local-ref` tag) is declared in the deployment descriptor to this EJB. This is because the ProcessPayment bean will not directly find or create Customer beans through the CustomerRemoteHome interface, but, instead, directly receive Customer beans from the client application. Thus, from the ProcessPayment EJB point of view, the Customer is a standard Remote Java object.

## ***Examine the JBoss Deployment Descriptors***

The JBoss specific deployment descriptor for the ProcessPayment bean is very simple. It only maps the datasource to the embedded database in JBoss:

### ***jboss.xml***

```
||| <session>
|||   <ejb-name>ProcessPaymentEJB</ejb-name>
|||   <jndi-name>ProcessPaymentHomeRemote</jndi-name>
|||   <resource-ref>
|||     <res-ref-name>jdbc/titanDB</res-ref-name>
|||     <jndi-name>java:/DefaultDS</jndi-name>
|||   </resource-ref>
||| </session>
```

The `res-ref-name` in *jboss.xml* maps to the same `res-ref-name` in *ejb-jar.xml*.

### ***Startup JBoss***

If JBoss is already running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex12\_1* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
||| C:\workbook\ex12_1> set JAVA_HOME=C:\jdk1.3
||| C:\workbook\ex12_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
||| $ export JAVA_HOME=/usr/local/jdk1.3
||| $ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex12_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Initialize the Database***

As with previous examples, you will use the relational database that is embedded in JBoss to store payment information. Since the deployment descriptor of a stateless session bean does not contain any information about the database schema that is needed by the bean, JBoss will not be able to automatically create the database table like for CMP beans. Instead, you will have to create it yourself.

To facilitate this task, the trick used for the BMP example of the previous chapter is applied once again. The `ProcessPayment` bean introduced in the EJB book is extended with two new methods that handle this task: `makeDbTable` and `dropDbTable`.

Here is a partial view of the remote interface of the `ProcessPayment` EJB:

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

    public void makeDbTable () throws RemoteException;
    public void deleteDbTable () throws RemoteException;
}
```

It defines two home methods: the first one will create the table needed by the `ProcessPayment` EJB in the JBoss embedded database and the second will drop it.

The implementation of the `makeDbTable` method is essentially a `CREATE TABLE` SQL statement:

```
public void makeDbTable ()
{
    PreparedStatement ps = null;
    Connection con = null;

    try
    {
        con = this.getConnection ();

        System.out.println("Creating table PAYMENT...");
        ps = con.prepareStatement ("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" +
                                   ") " );

        ps.execute ();
    }
}
```

```

        System.out.println("...done!");
    }
    catch (SQLException sql)
    {
        throw new EJBException (sql);
    }
    finally
    {
        try { ps.close (); } catch (Exception e) {}
        try { con.close (); } catch (Exception e) {}
    }
}

```

The `deleteDbTable` home method only differs by the SQL statement that is executed:

```

public void dropDbTable ()
{
    ...
    System.out.println("Dropping table PAYMENT...");
    ps = con.prepareStatement ("DROP TABLE PAYMENT");
    ps.execute ();
    System.out.println("...done!");
    ...
}

```

You will see how to call these methods in the next section.

## ***Examine the Client Applications***

This example comes with two examples. The first one will simply prepare and create a Customer bean that will be used by the second test program to insert data into the `PAYMENT` table.

### **Client\_121a**

You can run the first test application by using the `run.client_121a` ant target. This will simply create a single Customer bean that is needed by the next client application:

```

C:\workbook\ex12_1>ant run.client_121a
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
        ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;
        ..\ant\lib\ant.jar;
    [echo]

prepare:

```

```

compile:

ejbjar:

run.client_121a:
    [java] Creating Customer 1..
    [java] Creating AddressDO data object..
    [java] Setting Address in Customer 1...
    [java] Acquiring Address data object from Customer 1...
    [java] Customer 1 Address data:
    [java] 1010 Colorado
    [java] Austin,TX 78701

```

### Client\_121b

In the “Initialize the Database” section, you saw the implementation of the methods that create and drop the `PAYMENT` table in the database. Let’s see now how the client application will call these methods:

#### Client\_121b.java

```

if ( (args.length > 0) && args[0].equalsIgnoreCase ("CreateDB") )
{
    System.out.println ("Creating database table...");
    procpay.makeDbTable () ;
}
// ... or if we have to drop it...
//
else if ( (args.length > 0) && args[0].equalsIgnoreCase ("DropDB") )
{
    System.out.println ("Dropping database table...");
    procpay.dropDbTable () ;
}
else
{
    ...

```

You can see that depending on the first argument found on the command line, either `CreateDB` or `DropDB`, the client application will call the corresponding home method. If nothing is specified, the client will test the `ProcessPayment` stateless session bean.

To create the database table, you can use the `createdb_121` Ant target. It will call the test application with the appropriate command line argument.

```

C:\workbook\ex12_1>ant createdb_121
Buildfile: build.xml

init:

```

```

[echo] build.compiler = classic
[echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
        ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;
        ..\ant\lib\ant.jar;
[echo]

prepare:

compile:

ejbjar:

createdb_121:
    [java] Looking up home interfaces..
    [java] Creating database table...

```

On the JBoss console, the following lines are displayed:

```

INFO [STDOUT] Creating table PAYMENT...
INFO [STDOUT] ...done!

```

- ❖ Note. If you are having trouble creating the database, shutdown JBoss. Then run the Ant build target: `clean.db`. This will remove all database files and allow you to start from fresh.

Now that the database environment is ready, let's focus on the code of the client application that will actually test the `PaymentProcess EJB`.

First, a reference to the remote home of the `ProcessPayment EJB` is acquired from a newly created JNDI context:

```

Context jndiContext = getInitialContext ();

System.out.println ("Looking up home interfaces..");
Object ref = jndiContext.lookup ("ProcessPaymentHomeRemote");

ProcessPaymentHomeRemote procpayhome = (ProcessPaymentHomeRemote)
PortableRemoteObject.narrow (ref,ProcessPaymentHomeRemote.class);

```

With this home, a remote reference to the stateless session bean can be created:

```

ProcessPaymentRemote procpay = procpayhome.create ();

```

Then a remote home reference for the `Customer EJB` is acquired and used to find the `Customer bean` that has been created in the previous example:

```

ref = jndiContext.lookup ("CustomerHomeRemote");
CustomerHomeRemote custhome = (CustomerHomeRemote)
PortableRemoteObject.narrow (ref,CustomerHomeRemote.class);

```

```
||| CustomerRemote cust = custhome.findByPrimaryKey (new Integer (1));
```

The ProcessPayment EJB can now be tested by executing 3 payments, one using each method: cash, check and credit card.

```
||| System.out.println ("Making a payment using byCash()..");
    procpay.byCash (cust,1000.0);

    System.out.println ("Making a payment using byCheck()..");
    CheckDO check = new CheckDO ("010010101101010100011", 3001);
    procpay.byCheck (cust,check,2000.0);

    System.out.println ("Making a payment using byCredit()..");
    Calendar expdate = Calendar.getInstance ();
    expdate.set (2005,1,28); // month=1 is February
    CreditCardDO credit = new CreditCardDO ("3700000000000002",
                                             expdate.getTime (),
                                             "AMERICAN_EXPRESS");
    procpay.byCredit (cust,credit,3000.0);
```

Finally, a check payment is executed with a low check number. The ProcessPayment EJB must refuse the payment and raise an application exception.

```
||| System.out.println ("Making a payment using byCheck() with a low
                        check number..");
    CheckDO check2 = new CheckDO ("111000100111010110101", 1001);
    try
    {
        procpay.byCheck (cust,check2,9000.0);
        System.out.println("Problem! The PaymentException has
                           not been raised!"); }
    catch (PaymentException pe)
    {
        System.out.println ("Caught PaymentException: "+
                           pe.getMessage ());
    }

    procpay.remove ();
```

You can launch this test by using the run.client\_121b Ant target:

```
||| C:\workbook\ex12_1>ant run.client_121b
    Buildfile: build.xml

    init:
        [echo] build.compiler = classic
        [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
```



```

        ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;
        ..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

run.client_121b:
    [java] Looking up home interfaces..
    [java] Making a payment using byCash()..
    [java] Making a payment using byCheck()..
    [java] Making a payment using byCredit()..
    [java] Making a payment using byCheck() with a low check
number..
    [java] Caught PaymentException: Check number is too low. Must
be at least 2000

```

At the same time, on the JBoss console, the following lines are output:

```

INFO  [STDOUT] process() with customerID=1 amount=1000.0
INFO  [STDOUT] process() with customerID=1 amount=2000.0
INFO  [STDOUT] process() with customerID=1 amount=3000.0

```

Once the tests have been performed, you can drop the table by using the `dropdb_121` Ant target:

```

C:\workbook\ex12_1>ant dropdb_121
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
        ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;
        ..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

dropdb_121:
    [java] Looking up home interfaces..

```

```
||| [java] Dropping database table...
```

At the same time, on the JBoss console, the following lines are output:

```
||| INFO [STDOUT] Dropping table PAYMENT...  
||| INFO [STDOUT] ...done!
```

- ❖ To run the examples of the next section, do **not** drop the table: since the ProcessPayment EJB will be used as part of a large example, it still needs its own table to work correctly.





## Exercise 12.2: Stateful Session Bean

In this exercise, you will build and examine a stateful session bean, `TravelAgent`, that coordinates work for booking a trip on a ship. You will also build a client application to test this EJB.

This example will not strictly follow the example from the EJB book though. Instead of simplifying the beans and their relationships like the EJB book, we will use the beans implemented in chapters 6 and 7 and thus take advantage of the CMP 2.0 features of JBoss.

### *Examine the EJB*

This exercise is mainly based on the EJBs from exercise 7.3 and does not contain much material that hasn't been covered in the previous sections. Nevertheless, a few modifications have been made:

- The Customer EJB again has a remote home and bean interfaces (like in chapter 6) and exposes its relationship with the Address EJB in the remote interface through a new data object: `AddressDO`.
- The Cabin EJB has a new create method that takes several parameters
- The Reservation EJB now has a new create method that takes several parameters and has a local reference to the Customer EJB.

The role of the `TravelAgent` bean is to perform all activities needed to book a successful trip. Thus, like in the preceding example, this session bean will act as a coordinator between different EJBs and will group several actions on different beans in the same transaction. In this case though, the bean maintains a conversational state with the client i.e. each client has a dedicated bean on the server.

If you remember the previous example with stateless session beans, the home create method was not allowed to have parameters. In the case of stateful session beans though, since a conversational state exists, create methods can have parameters to initialize the bean state and several create methods can co-exist in the home interface.

In the `TravelAgent` home interface though, a single create method is defined:

```
public interface TravelAgentHomeRemote extends javax.ejb.EJBHome
{
    public TravelAgentRemote create (CustomerRemote cust)
        throws RemoteException, CreateException;
}
```

And if you take a look at the remote interface, you can see that methods are correlated around an identical state:

```
public interface TravelAgentRemote extends javax.ejb.EJBObject
```

```

{
    public void setCruiseID (Integer cruise)
        throws RemoteException, FinderException;

    public void setCabinID (Integer cabin)
        throws RemoteException, FinderException;

    public TicketDO bookPassage (CreditCardDO card, double price)
        throws RemoteException, IncompleteConversationalState;

    public String [] listAvailableCabins (int bedCount)
        throws RemoteException, IncompleteConversationalState;
}

```

Calling `setCruiseId`, if no conversational state between the client and the server existed, would make no sense. The role of this method is simply to populate this conversational state so that future calls can use this data in their processing.

Since this exercise is based on the beans implemented in chapter 6 and 7 that contain full relationships between them, the needed database schema is not exactly the same as the one in the EJB book. Consequently, since the `listAvailableCabins` method performs direct SQL calls, it has to be rewritten to take this new database schema into account:

```

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 ();

```

If you remember the previous examples, we added a method (either `home` or `remote`) to our EJB to be able to initialize our test environment. As you can guess, the same trick has been used for this example. The `TravelAgent` EJB remote interface has been extended with one method:

```

public interface TravelAgentRemote extends javax.ejb.EJBObject
{
    ...
    // Mechanism for building local beans for example programs.
}

```

```

    //
    public void buildSampleData () throws RemoteException;
}

```

This method will remove any Customer, Cabin, Ship, Cruise and Reservation EJB from the database and recreate a basic environment. Let's follow step by step this initialization.

First we acquire references to the remote home of the Customer EJB and local home to the Cabin, Ship, Cruise and Reservation EJBs.

```

public Collection buildSampleData ()
{
    Collection results = new ArrayList ();

    try
    {
        System.out.println ("TravelAgentBean::buildSampleData()");

        Object obj = jndiContext.lookup
            ("java:comp/env/ejb/CustomerHomeRemote");
        CustomerHomeRemote custhome = (CustomerHomeRemote)
            javax.rmi.PortableRemoteObject.narrow (obj,
                CustomerHomeRemote.class);

        CabinHomeLocal cabinhome =
            (CabinHomeLocal)jndiContext.lookup
                ("java:comp/env/ejb/CabinHomeLocal");
        ShipHomeLocal shiphome =
            (ShipHomeLocal)jndiContext.lookup
                ("java:comp/env/ejb/ShipHomeLocal");
        CruiseHomeLocal cruisehome =
            (CruiseHomeLocal)jndiContext.lookup
                ("java:comp/env/ejb/CruiseHomeLocal");
        ReservationHomeLocal reshome =
            (ReservationHomeLocal)jndiContext.lookup
                ("java:comp/env/ejb/ReservationHomeLocal");
    }
}

```

Then any existing bean is deleted from the database:

```

// we first clean the db by removing any customer, cabin,
// ship, cruise and reservation beans.
//
removeBeansInCollection (custhome.findAll());
results.add ("All customers have been removed");
removeBeansInCollection (cabinhome.findAll());
results.add ("All cabins have been removed");
removeBeansInCollection (shiphome.findAll());
results.add ("All ships have been removed");
removeBeansInCollection (cruisehome.findAll());

```

```

results.add ("All cruises have been removed");
removeBeansInCollection (reshome.findAll());
results.add ("All reservations have been removed");

```

`removeBeansInCollection` is a simple method that iterate over the collection and call the `remove` method on each `EJBObject` or `EJBLocalObject`.

Two customers and two ships are created:

```

// We now set our new basic environment
//
System.out.println ("Creating Customers 1 and 2...");
CustomerRemote customer1 =
    custhome.create (new Integer (1));
customer1.setName ( new Name ("Burke","Bill") );
results.add ("Customer with ID 1 created (Burke Bill)");

CustomerRemote customer2 =
    custhome.create (new Integer (2));
customer2.setName ( new Name ("Labourey","Sacha") );
results.add("Customer with ID 2 created (Labourey Sacha)");

System.out.println ("Creating Ships A and B...");
ShipLocal shipA = shiphome.create (new Integer (101),
    "Nordic Prince", 50000.0);
results.add("Created ship with ID 101...");
ShipLocal shipB = shiphome.create (new Integer (102),
    "Bohemian Rhapsody", 70000.0);
results.add("Created ship with ID 102...");

```

After each significant step, a message is added to the `results` collection. At the end of the method, this object will be returned to the caller so that he knows what has happened on the server.

Ten cabins are then created on each ship:

```

System.out.println ("Creating Cabins on the Ships...");
ArrayList cabinsA = new ArrayList ();
ArrayList cabinsB = new ArrayList ();
for (int jj=0; jj<10; jj++)
{
    CabinLocal cabinA = cabinhome.create (new Integer
        (100+jj), shipA, "Suite 10"+jj,1,1);
    cabinsA.add(cabinA);
    CabinLocal cabinB = cabinhome.create (new Integer
        (200+jj), shipB, "Suite 20"+jj,2,1);
    cabinsB.add(cabinB);
}
results.add("Created cabins on Ship A with IDs 100-109");

```



```
||| results.add("Created cabins on Ship B with IDs 200-209");
```

Some cruises are quickly organized for each ship:

```
||| CruiseLocal cruiseA1 = cruisehome.create ("Alaska Cruise",
||| shipA);
||| CruiseLocal cruiseA2 = cruisehome.create ("Norwegian
||| Fjords", shipA);
||| CruiseLocal cruiseA3 = cruisehome.create (
||| "Bermuda or Bust", shipA);
||| results.add("Created cruises on ShipA with IDs "+
||| cruiseA1.getId()+" "+cruiseA2.getId()+
||| ", "+cruiseA3.getId());
|||
||| CruiseLocal cruiseB1 = cruisehome.create ("Indian Sea
||| Cruise", shipB);
||| CruiseLocal cruiseB2 = cruisehome.create (
||| "Australian Highlights", shipB);
||| CruiseLocal cruiseB3 = cruisehome.create (
||| "Three-Hour Cruise", shipB);
||| results.add ("Created cruises on ShipB with IDs "+
||| cruiseB1.getId ()+" "+cruiseB2.getId ()+
||| ", "+cruiseB3.getId ());
```

And some reservations are made for these cruises:

```
||| ReservationLocal res =
||| reshome.create (customer1, cruiseA1,
||| (CabinLocal) (cabinsA.get (3)),
||| 1000.0, new Date ());
||| res = reshome.create (customer1, cruiseB3,
||| (CabinLocal) (cabinsB.get (8)),
||| 2000.0, new Date ());
||| res = reshome.create (customer2, cruiseA2,
||| (CabinLocal) (cabinsA.get (5)),
||| 2000.0, new Date ());
||| res = reshome.create (customer2, cruiseB3,
||| (CabinLocal) (cabinsB.get (2)),
||| 2000.0, new Date ());
|||
||| results.add ("Made reservation for Customer 1 on Cruise "+
||| cruiseA1.getId ()+" for Cabin 103");
||| results.add ("Made reservation for Customer 1 on Cruise "+
||| cruiseB3.getId ()+" for Cabin 208");
||| results.add ("Made reservation for Customer 2 on Cruise "+
||| cruiseA2.getId ()+" for Cabin 105");
||| results.add ("Made reservation for Customer 2 on Cruise "+
||| cruiseB3.getId ()+" for Cabin 202");
```

We will show, in a subsequent section, how we call this method to set up our environment.

## ***Examine the EJB Standard Deployment Descriptor***

The *ejb-jar.xml* file mostly contains definitions that you have already seen in previous examples (entity beans, relationships, ProcessPayment stateless session bean, etc.) Only two things have been added.

First, the Customer EJB now has both local and remote interfaces:

```
<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>
```

This means that the Customer EJB can be used by local clients as well as remote clients. Note that both the remote and local interfaces do not declare the same methods. For example, entity relationships are not exposed through the remote interface since this is illegal. They are only exposed by the local interface.

The second addition is the new TravelAgent stateful session bean that is the heart of this exercise:

```
<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>
  ...
</session>
```

As you can see, the declaration of a stateful session bean only differs from a stateless by the `session-type` tag set to `Stateful`.

What follows here is the declaration of all beans referenced by the TravelAgent EJB:

```
<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-link>ProcessPaymentEJB</ejb-link>
</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-link>CustomerEJB</ejb-link>
</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-link>CabinEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
  <ejb-ref-name>ejb/ShipHomeLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>
    com.titan.cabin.ShipHomeLocal
  </local-home>
  <local>com.titan.cabin.ShipLocal</local>
  <ejb-link>ShipEJB</ejb-link>
</ejb-local-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-link>CruiseEJB</ejb-link>
    </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-link>ReservationEJB</ejb-link>
    </ejb-local-ref>
    <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>

```

## Examine the JBoss Deployment Descriptor

The *jboss.xml* deployment descriptor mainly contains the JNDI name mapping found in the previous examples. The only new entry is the TravelAgent EJB definition:

```

<session>
    <ejb-name>TravelAgentEJB</ejb-name>
    <jndi-name>TravelAgentHomeRemote</jndi-name>
    <resource-ref>
        <res-ref-name>jdbc/titanDB</res-ref-name>
        <jndi-name>java:/DefaultDS</jndi-name>
    </resource-ref>
</session>

```

The JNDI name for the TravelAgent is defined and the datasource JNDI ENC name used by the `listAvailableCabins` method is mapped to the embedded database.

The `listAvailableCabins` method directly executes SQL statements against the database. Thus, it has to know precisely the name of the SQL tables and fields to use in its query. But while the field-to-column mapping of all CMP beans is already defined in the *jbossCMP-jdbc.xml* file, the fields and tables used by relationships between these beans are not defined. Consequently, JBoss will use arbitrary names for these tables. Instead of using these arbitrary names in the SQL query, *jbossCMP-jdbc.xml* is extended to precisely define into which tables and columns the relationships will be mapped. We will only map the relationships used in the SQL query though: Cabin-Ship, Cabin-Reservation and Cruise-Reservation.

The Cabin-Reservation relationship is a many-to-many relationship:

```

    <ejb-relation>

```

```

<ejb-relation-name>Cabin-Reservation</ejb-relation-name>
<relation-table-mapping>
  <table-name>RESERVATION_CABIN_LINK</table-name>
  <create-table>true</create-table>
  <remove-table>true</remove-table>
</relation-table-mapping>
<ejb-relationship-role>
  <ejb-relationship-role-name
    >Cabin-has-many-Reservations<
  /ejb-relationship-role-name>
  <key-fields>
    <key-field>
      <field-name>id</field-name>
      <column-name>CABIN_ID</column-name>
    </key-field>
  </key-fields>
</ejb-relationship-role>
<ejb-relationship-role>
  <ejb-relationship-role-name
    >Reservation-has-many-Cabins<
  /ejb-relationship-role-name>
  <key-fields>
    <key-field>
      <field-name>id</field-name>
      <column-name>RESERVATION_ID</column-name>
    </key-field>
  </key-fields>
</ejb-relationship-role>
</ejb-relation>

```

Many-to-many relationships always need an intermediate table. The name of this table is defined in the `table-name` tag. Then, for each role of the relationship, the mapping between the CMR field of the bean and the column in the table is done through the `field-name` and `column-name` tags.

The last two relationship mappings that need to be defined are one-to-many relationships:

```

<ejb-relation>
  <ejb-relation-name>Cabin-Ship</ejb-relation-name>
  <foreign-key-mapping/>
  <ejb-relationship-role>
    <ejb-relationship-role-name
      >Ship-has-many-Cabins<
    /ejb-relationship-role-name>
    <key-fields>
      <key-field>
        <field-name>id</field-name>

```

```

        <column-name>SHIP_ID</column-name>
    </key-field>
</key-fields>
</ejb-relationship-role>
<ejb-relationship-role>
    <ejb-relationship-role-name>
        >Cabin-has-a-Ship<
    </ejb-relationship-role-name>
    <key-fields/>
</ejb-relationship-role>
</ejb-relation>

<ejb-relation>
    <ejb-relation-name>Cruise-Reservation</ejb-relation-name>
    <foreign-key-mapping/>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            >Cruise-has-many-Reservations<
        </ejb-relationship-role-name>
        <key-fields>
            <key-field>
                <field-name>id</field-name>
                <column-name>CRUISE_ID</column-name>
            </key-field>
        </key-fields>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            >Reservation-has-a-Cruise<
        </ejb-relationship-role-name>
        <key-fields/>
    </ejb-relationship-role>
</ejb-relation>

```

For each relationship identified by an `ejb-relation-name` tag (the name must be the same as the one declared in *ejb-jar.xml*), the mapping of the CMR field to a table column is defined by the `field-name` and `column-name` tags.

## Startup JBoss

If JBoss is already running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

## Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *ex12\_2* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
C:\workbook\ex12_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex12_2> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex12_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Initialize the Database***

Since the example uses the ProcessPayment EJB from the previous example, the database must contain the PAYMENT table.

If you have not removed this table, then you are fine. Otherwise, you need to recreate this table. In short, this means that you need to go in the *ex12\_1* directory and execute Ant using the *createdb\_121* target. For more detailed instructions, refer to the detailed steps of the previous example.

For all other entity beans that will be used, because they use CMP, the needed tables will be automatically created at deployment time by the container.

## ***Examine the Client Applications***

Three client applications are available for this example. The first one will simply call the *buildSampleData* method of the TravelAgent bean. To run this application, uses the *run.client\_122a* Ant target:

```

C:\workbook\ex12_2>ant run.client_122a
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

run.client_122a:
    [java] Calling TravelAgentBean to create sample data..
    [java] All customers have been removed
    [java] All cabins have been removed
    [java] All ships have been removed
    [java] All cruises have been removed
    [java] All reservations have been removed
    [java] Customer with ID 1 created (Burke Bill)
    [java] Customer with ID 2 created (Labourey Sacha)
    [java] Created ship with ID 101...
    [java] Created ship with ID 102...
    [java] Created cabins on Ship A with IDs 100-109
    [java] Created cabins on Ship B with IDs 200-209
    [java] Created cruises on ShipA with IDs -47396262,
    -47396261, -47396260
    [java] Created cruises on ShipB with IDs -47396259,
    -47396258, -47396257
    [java] Made reservation for Customer 1 on
    Cruise -47396262 for Cabin 103
    [java] Made reservation for Customer 1 on
    Cruise -47396257 for Cabin 208
    [java] Made reservation for Customer 2 on
    Cruise -47396261 for Cabin 105
    [java] Made reservation for Customer 2 on
    Cruise -47396257 for Cabin 202

```

Now that you have prepared the environment, you can use the two other client applications.

The first application allows you to book a passage while the second allows you to list the cabins for a specific Cruise having a desired number of beds.



Let's look at the code behind the first tool. The code first gets a remote home to the TravelAgent and Customer EJBs:

```
public static void main(String [] args) throws Exception {

    if (args.length != 4) {
        System.out.println("Usage: java " +
            "com.titan.clients.Client_122b" +
            "<customerID> <cruiseID> <cabinID> <price>");
        System.exit(-1);
    }

    Integer customerID = new Integer(args[0]);
    Integer cruiseID = new Integer(args[1]);
    Integer cabinID = new Integer(args[2]);
    double price = new Double(args[3]).doubleValue();

    Context jndiContext = getInitialContext();
    Object obj = jndiContext.lookup("TravelAgentHomeRemote");
    TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
        javax.rmi.PortableRemoteObject.narrow(obj,
            TravelAgentHomeRemote.class);

    obj = jndiContext.lookup("CustomerHomeRemote");
    CustomerHomeRemote custhome = (CustomerHomeRemote)
        javax.rmi.PortableRemoteObject.narrow(obj,
            CustomerHomeRemote.class);
```

With the home references, it gets a reference to the customer whose ID has been given on the command line. If no customer with this id exists, an exception is thrown.

```
// Find a reference to the Customer for which to book a cruise
System.out.println("Finding reference to Customer "+customerID);
CustomerRemote cust = custhome.findByPrimaryKey(customerID);
```

The application then creates a TravelAgent stateful session bean, and gives it, as part of the transactional state, the reference to the customer, the cruise ID and the Cabin ID.

```
// Start the Stateful session bean
System.out.println("Starting TravelAgent Session...");
TravelAgentRemote tagent = tahome.create(cust);

// Set the other bean parameters in agent bean
System.out.println("Setting Cruise and Cabin information in
TravelAgent..");
tagent.setCruiseID(cruiseID);
tagent.setCabinID(cabinID);
```

Then it is possible to book the passage thanks to a dummy credit card:

```

// Create a dummy CreditCard for this
//
Calendar expdate = Calendar.getInstance();
expdate.set(2005,1,5);
CreditCardDO card = new CreditCardDO("3700000000000002",
                                     expdate.getTime(),
                                     "AMERICAN EXPRESS");

// Book the passage
//
System.out.println("Booking the passage on the Cruise!");
TicketDO ticket = tagent.bookPassage(card,price);

System.out.println("Ending TravelAgent Session...");
tagent.remove();

System.out.println("Result of bookPassage:");
System.out.println(ticket.description);
}

```

Let's now test this client application by booking Suite 201 for Mr. Bill Burke for the "Three-Hour cruise" on the "Bohemian Rhapsody" ship:

- ❖ It is not particularly easy to give command line parameters to Ant in a usual fashion. To make this task easier, scripts that accept usual command line parameters are available in the [ex12\\_2](#) directory.

To book a passage, use the BookPassage.bat (Windows) or BookPassage (Unix) script:

```

BookPassage.bat <customerID> <cruiseID> <cabinID> <price>
Or
./BookPassage <customerID> <cruiseID> <cabinID> <price>

```

```

C:\workbook\ex12_2>BookPassage 1 -47396257 201 2000.0
Buildfile: build.xml

init:
[echo] build.compiler = classic
[echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
[echo]

prepare:

compile:

```

```

ejbjar:

run.client_122b:
    [java] Finding reference to Customer 1
    [java] Starting TravelAgent Session...
    [java] Setting Cruise and Cabin information in TravelAgent..
    [java] Booking the passage on the Cruise!
    [java] Ending TravelAgent Session...
    [java] Result of bookPassage:
    [java] Bill Burke has been booked for the Three-Hour Cruise
cruise on ship Bohemian Rhapsody.
    [java] Your accommodations include Suite 201 a 2 bed cabin on
deck level 1.
    [java] Total charge = 2000.0

```

The last application allows you to get a list of available cabins for a specific cruise having a desired number of beds. First, the application tests if it has been called with the correct number of command line arguments and gets a remote home reference to the TravelAgent EJB:

```

public static void main(String [] args) throws Exception {

    if (args.length != 2) {
        System.out.println("Usage: java " +
            "com.titan.clients.Client_122c" +
            " <cruiseID> <bedCount>");
        System.exit(-1);
    }

    Integer cruiseID = new Integer(args[0]);
    int bedCount = new Integer(args[1]).intValue();

    Context jndiContext = getInitialContext();
    Object obj = jndiContext.lookup("TravelAgentHomeRemote");
    TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
        javax.rmi.PortableRemoteObject.narrow(obj,
            TravelAgentHomeRemote.class);

```

Since the method we want to call on our stateful session bean is not really dedicated to a specific instance but is instead making an SQL query in the database, we create a TravelAgent bean with a dummy Customer reference (it will never be used):

```

// Start the Stateful session bean
System.out.println("Starting TravelAgent Session...");
TravelAgentRemote tagent = tahome.create(null);

// Set the other bean parameters in agent bean

```

```

        System.out.println("Setting Cruise information in
TravelAgent..");
        tagent.setCruiseID(cruiseID);

```

Then, the application asks for a list of all available cabins with a desired number of beds on a particular cruise and displays the result (if any):

```

        String[] results = tagent.listAvailableCabins (bedCount);

        System.out.println("Ending TravelAgent Session...");
        tagent.remove();

        System.out.println("Result of listAvailableCabins:");
        for (int kk=0; kk<results.length; kk++) {
            System.out.println(results[kk]);
        }
    }
}

```

To launch this application, you can use the ListCabins.bat (Windows) or ListCabins (Unix) script:

```

ListCabins.bat <cruiseID> <bedCount>
Or
./ListCabins <cruiseID> <bedCount>

```

Let's now ask the system for a list of cabins that have two beds available on the same cruise Mr. Bill Burke made a reservation:

```

C:\workbook\ex12_2>ListCabins -47396257 2
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

run.client_122c:
    [java] Starting TravelAgent Session...
    [java] Setting Cruise information in TravelAgent..
    [java] Ending TravelAgent Session...

```

```
[java] Result of listAvailableCabins:  
[java] 200,Suite 200,1  
[java] 203,Suite 203,1  
[java] 204,Suite 204,1  
[java] 205,Suite 205,1  
[java] 206,Suite 206,1  
[java] 207,Suite 207,1  
[java] 209,Suite 209,1
```

Suite 201, that had been previously booked for Mr. Bill Burke is not shown as available, which is correct.

## *Exercises for Chapter 13*

### **Exercise 13.1: JMS as a Resource**

This exercise is entirely based on the beans implemented in the previous exercise (12.2). You will modify the TravelAgent EJB so it publishes a text message to a JMS topic when a reservation has been completed.

You will learn how to create a new JMS topic in JBoss and configure your bean to use JMS as a resource. You will also build a client application that will subscribe to this topic and display any published message. To complete new reservations, you will use the client application of the previous example.

#### ***Startup JBoss***

If JBoss is already running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

#### ***Initialize the Database***

Since this exercise will use the same database environment as the previous exercise, make sure that it is correctly set. In particular, the `RESERVATION` table, used by the `ProcessPayment` stateless session bean must be created.

If you have not removed this table, then you are fine. Otherwise, you need to recreate this table by making Ant execute the `createdb_121` target:

```
C:\workbook\ex13_1>ant createdb_121
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
        ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;
        ..\ant\lib\ant.jar;
    [echo]

prepare:
```

```

compile:

ejbjar:

createdb_121:
    [java] Looking up home interfaces..
    [java] Creating database table...

```

On the JBoss console, the following lines are displayed:

```

INFO  [STDOUT] Creating table PAYMENT...
INFO  [STDOUT] ...done!

```

For all other entity beans that will be used, because they rely on CMP, the needed tables will be automatically created at deployment time by the container.

## Create a New JMS Topic

Since the TravelAgent EJB will publish messages in a JMS topic, you will have to create this new topic in JBoss.

This exercise will walk you through two different ways for creating a new JMS topic: deploying through an XML configuration file, and through the JBoss JMX HTTP connector.

### Adding a JMS Topic Through a Configuration File

The first and most common way to set up a JMS topic is through an XML configuration file. As you have learned in the installation chapter of JBoss, every component in JBoss is a JMX MBean that can be hot-deployed. This part of the exercise shows you how to define a JMX MBean definition for a new JMS topic.

The JMX configuration file can be found in the `ex13_1/src/resources/services` folder:

#### *jbossmq-titantopic-service.xml*

```

<server>
  <mbean code="org.jboss.mq.server.jmx.Topic"
        name="jboss.mq.destination:service=Topic,
            name=titan-TicketTopic">
    <depends optional-attribute-name="DestinationManager"
      >jboss.mq:service=DestinationManager</depends>
  </mbean>
</server>

```

Each set of MBeans within a JMX configuration file must be defined within a `server` tag. An MBean itself is declared within an `mbean` tag. The only MBean declaration in this file defines the actual JMS topic you will use for the example code in this chapter. First the MBean class representing the JMS topic is declared (`org.jboss.mq.server.jmx.Topic`) along with the

JMX name of the MBean. JMX names can include any amount of key-value parameters to further describe the MBean. For JMS topic MBeans, a single parameter is useful: `name`. This is where the name of the JMS topic is defined (“`titan-TicketTopic`”). Copying this file into the JBoss deploy directory will hot-deploy this JMS topic and it will be ready for use.

One thing to note is that the `DestinationManager` MBean needs to be deployed by the application server before any Queue or Topic is deployed. This dependency is declared with the `depends` tag in `jbossmq-titantopic-service.xml`. The `titan-TicketTopic` will not be deployed until the `DestinationManager` MBean has finished initializing and is ready to service new Queues and Topics.

The `make-topic` Ant target has been defined for deploying the `jbossmq-titantopic-service.xml`. It simply copies this XML file into JBoss’s deploy directory:

```
C:\workbook\ex13_1>ant make-topic
Buildfile: build.xml

make-topic:
    [copy] Copying 1 file to C:\jboss-3.0.0\server\default\deploy
```

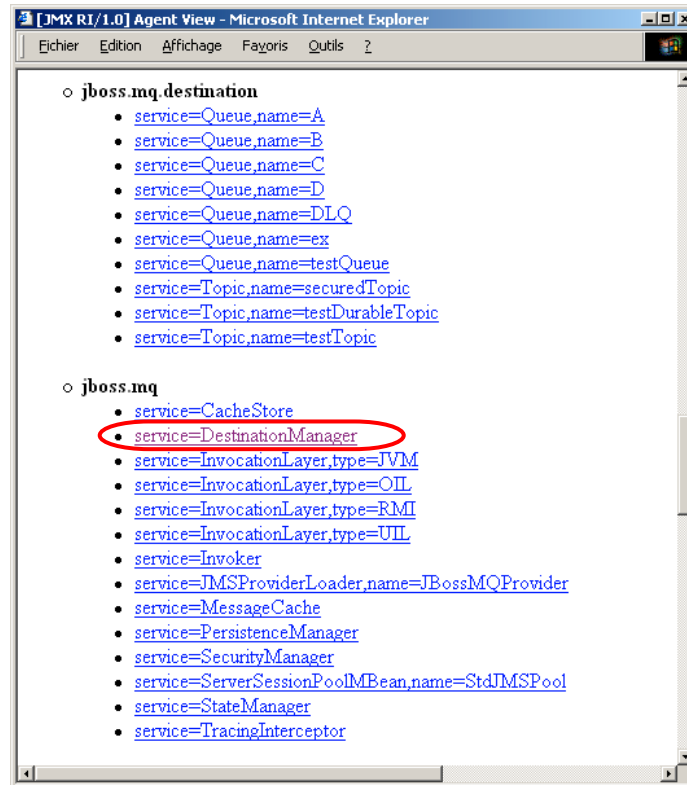
On the server side, the following lines are displayed:

```
[MainDeployer] Starting deployment of package: file:/C:/jboss-
3.0.0/server/default/deploy/jbossmq-titantopic-service.xml
[titan-TicketTopic] Creating
[titan-TicketTopic] Created
[titan-TicketTopic] Starting
[titan-TicketTopic] Bound to JNDI name: topic/titan-TicketTopic
[titan-TicketTopic] Started
[MainDeployer] Successfully completed deployment of package:
file:/C:/jboss-3.0.0/server/default/deploy/jbossmq-titantopic-
service.xml
```

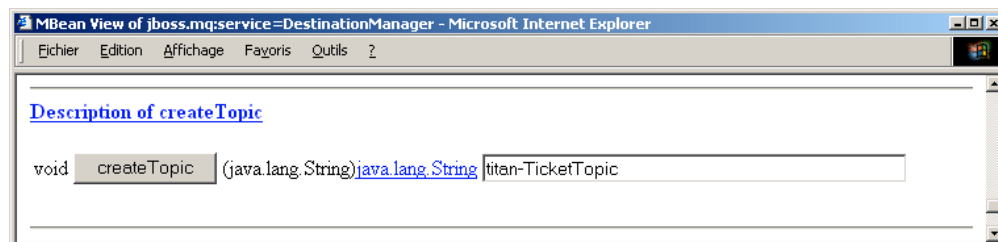
### Adding a JMS Topic Through the JMX HTTP Connector

Although an XML configuration file is the preferred way to deploy a JMS topic, you can also create one through JBoss’s JMX HTTP connector. A topic created in this way only lives in JBoss until the application server is shut down, but this can be useful for quick tests and such. To create a JMS topic in this way, you must first open your browser and go to `http://localhost:8082/`. This screen allows you to browse through all deployed JBoss JMX MBeans. Scroll down to the `jboss.mq` section. You can use the MBean service `DestinationManager` in this section to deploy new JMS topics and runtime:

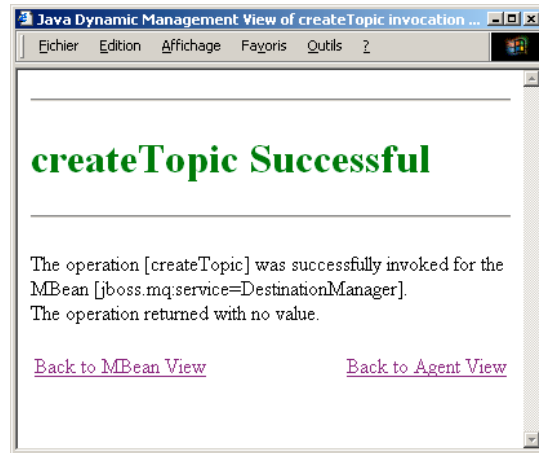




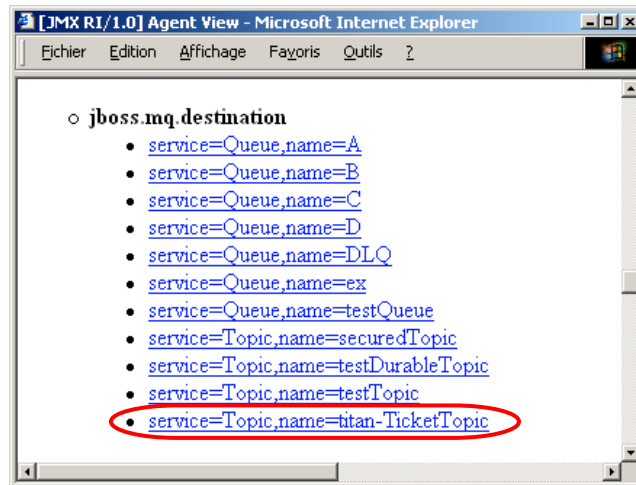
Once you click on the [service=DestinationManager](#) link, you get a list of the MBean's attributes and operations. One of the operations, [createTopic](#), allows you to create a new JMS topic:



You can write the name of the new JMS topic in the text area and then click on the [createTopic](#) button. As a result, the JMS topic is created and a status message is displayed:



To see your new JMS topic MBean, go back to the homepage of the JMX HTTP connector and search for the `jboss.mq.destination` domain. You should be able to see your new topic MBean:



Note that whatever way you choose to create your topic, you can use the JMX HTTP connector to see that status of your topics and queues.

### ***Examine the EJB Standard Files***

The `ejb-jar.xml` deployment descriptor is equivalent to the one from Exercise 12.2 except for the TravelAgent EJB. The definition for this bean has been extended to reference the JMS topics you created above.

```

<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>
  ...
  <resource-ref>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <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>

```

A reference to a `TopicConnectionFactory` is declared in the same way as a reference to a `DataSource`. The definition contains the name of the resource (`jms/TopicFactory`), the class of the resource (`javax.jms.TopicConnectionFactory`) and whether the container or the bean performs the authentication.

## Examine the JBoss Specific Files

The `TravelAgentEJB` definition within `jboss.xml` needs to be modified as well to describe the JMS topic references declared in `ejb-jar.xml`.

### ***jboss.xml***

```

...
<session>
  <ejb-name>TravelAgentEJB</ejb-name>
  <jndi-name>TravelAgentHomeRemote</jndi-name>
  <resource-ref>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
  <resource-ref>
    <res-ref-name>jms/TopicFactory</res-ref-name>

```

```

    <jndi-name>java:/JmsXA</jndi-name>
  </resource-ref>
  <resource-env-ref>
    <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
    <jndi-name>topic/titan-TicketTopic</jndi-name>
  </resource-env-ref>
</session>
...

```

The `resource-ref` entry from the `ejb-jar.xml` is mapped in the `jboss.xml` file to the `java:/JmsXA` JNDI name. If you take a look at the JBossMQ default configuration file (found in `JBOSS_HOME/server/default/deploy/jms-service.xml`), you will see that the XA connection manager is bound to this name by default.

The last part of the TravelAgent EJB descriptor in `jboss.xml` maps the `jms/TicketTopic` name from the JNDI ENC of the bean to the `topic/titan-TicketTopic` JNDI name. This name corresponds to the JMS topic you just created a few sections earlier.

## Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the `ex13_1` directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```

C:\workbook\ex13_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex13_1> set JBOSS_HOME=C:\jboss-3.0.0FINAL

```

Unix:

```

$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.0.0

```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```

C:\workbook\ex13_1> set PATH=..\ant\bin;%PATH%

```

Unix:

```

$ export PATH=../ant/bin:$PATH

```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

You will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the Client Applications***

In this exercise, two client applications will be used. You can find the code for these clients in the *ex13\_1/src/main/com/titan/clients* directory.

The first application is the one we used in exercise 12.2 to make reservations (*run.client\_122b*). We will not review the details of this application.

The second application is new (*JmsClient\_1*). It will subscribe to the *titan-TicketTopic* JMS topic and display all messages that are published on it.

The application first gets an *InitialContext*, and lookups its *TopicConnectionFactory* and *Topic*.

### ***JmsClient\_1.java***

```
Context jndiContext = getInitialContext();

TopicConnectionFactory factory = (TopicConnectionFactory)
    jndiContext.lookup("ConnectionFactory");

Topic topic = (Topic) jndiContext.lookup("topic/titan-
TicketTopic");
```

The name of the JMS topic is the same as the one you created in a previous section whereas the name of the *TopicConnectionFactory* is **not** the same as the one used by the *TravelAgent* EJB.

If you remember, the *ConnectionFactory* used by the EJB, *java:/JmsXA*, was in the *private* JNDI space of the JBoss JVM (because of the *java:* prefix). Thus, the client application is not able to lookup this name from its JVM. For external applications, JBoss binds a set of *ConnectionFactory* within the public JNDI tree, each dedicated to a particular message transport protocol.

JBossMQ supports several different kinds of message transport/invoke layers that include:

- JVM: hyper-efficient invocation layer using standard Java method invocation. Used for in-JVM JMS clients. External clients cannot use this invocation layer.
- RMI: RMI-based invocation layer
- OIL – Optimized Invocation Layer: this layer uses custom TCP/IP sockets to obtain good network performance and small memory footprint.
- UIL: for client applications that cannot accept network connections originating from the server.

Each of these invocation layer has its own *ConnectionFactory* that is bound in JNDI:

Invocation Layer	JNDI name
JVM	java:/ConnectionFactory and java:/XAConnectionFactory (with XA support)
RMI	RMIConnectionFactory and RMIXAConnectionFactory (with XA support)
OIL	ConnectionFactory and XAConnectionFactory (with XA support)
UIL	UILConnectionFactory and UILXAConnectionFactory (with XA support)

```

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 topic/titan-
TicketTopic...");
connect.start();

```

The end of the client application code is identical to the one presented in the EJB book.

## Run the Client Applications

As you redeployed the *titan.jar* file, JBoss dropped and recreated the database tables, destroying any content that was present. For this reason, you must make Ant execute the `run.client_122a` target to repopulate the database.

- ❖ Even if `run.client_122a` target originates from exercise 12.2, it has been duplicated in the `ex13_1` directory to facilitate your work.

In order for your new application to receive the message published on the JMS topic, you have to start it first:

```

C:\workbook\ex13_1>ant client_131
Buildfile: build.xml

init:

```

```

[echo] build.compiler = classic
[echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
[echo]

prepare:

compile:

ejbjar:

client_131:
    [java] Listening for messages on topic/titan-TicketTopic...

```

This means that the client application has successfully subscribed to the topic and is waiting for messages.

Now you need to make some reservations exactly like you did in exercise 12\_2. Open a new shell and use the `BookPassage` script to make some dummy reservations:

```

C:\workbook\ex13_1>BookPassage 1 131735436 101 3000
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

run.client_122b:
    [java] Finding reference to Customer 1
    [java] Starting TravelAgent Session...
    [java] Setting Cruise and Cabin information in TravelAgent..
    [java] Booking the passage on the Cruise!
    [java] Result of bookPassage:
    [java] Bob Smith has been booked for the Alaska Cruise cruise
on ship Nordic Prince.
    [java] Your accommodations include Suite 101 a 1 bed cabin on
deck level 1.
    [java] Total charge = 3000.0

```

If you take a look at the JMS subscriber windows you started before, the following lines should appear:

```
||| [java] Listening for messages on topic/titan-TicketTopic...  
||| [java]  
||| [java] RESERVATION RECEIVED:  
||| [java] Bob Smith has been booked for the Alaska Cruise cruise on  
||| ship Nordic Prince.  
||| [java] Your accommodations include Suite 101 a 1 bed cabin on deck  
||| level 1.  
||| [java] Total charge = 3000.0
```

As the client application is using a non-durable subscription for a topic, all the messages sent while the application is not running are definitively lost.

To see the “many-to-many” nature of JMS topic, you can launch several JMS listener applications at the same time. They will all receive the messages sent to the topic.







## Exercise 13.2: The Message-Driven Bean

This exercise is an extension of the previous exercise (13.1). You will add a Message Driven Bean (MDB), `ReservationProcessor`, that will play the same role as the `TravelAgent` EJB but will receive its booking orders through a JMS queue instead of standard (synchronous) RMI invocations.

To test the MDB, you will build a new client application that will make massive reservations in batch using the JMS queue that is bound to the MDB. You will also build a second client application that will listen on another queue to receive booking confirmations.

You will learn how to create a new JMS queue in JBoss and configure a Message Driven Bean (MDB).

### *Startup JBoss*

If JBoss is already running there is no reason to restart it. Otherwise please review the *Server Installation and Configuration* chapter at the beginning of this book.

### *Initialize the Database*

Since this exercise will use the same database environment as the previous exercise, make sure that it is correctly set. In particular, the `RESERVATION` table, used by the `ProcessPayment` stateless session bean must be created.

If you have not removed this table, then you are fine. Otherwise, you need to recreate this table by making Ant execute the `createdb_121` target:

```
C:\workbook\ex13_2>ant createdb_121
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
        ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;
        ..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

createdb_121:
```

```
||| [java] Looking up home interfaces..  
||| [java] Creating database table...
```

On the JBoss console, the following lines are displayed:

```
||| INFO [STDOUT] Creating table PAYMENT...  
||| INFO [STDOUT] ...done!
```

For all other entity beans that will be used, because they rely on CMP, the needed tables will be automatically created at deployment time by the container.

## Create a New JMS Queue

This exercise requires two different JMS queues: one for the ReservationProcessor MDB and one to receive booking confirmations.

The process of adding new JMS queues to JBoss is very similar to the process of adding new JMS topics that you have seen in the previous exercise. Again, you will see two different ways to create a new JMS queue in JBoss: by deploying a configuration file and through the JMX HTTP connector.

### Adding a JMS Queue Through a Configuration File

The first and most common way to set up a JMS Queue is through an XML configuration file. As you have learned in the installation chapter of JBoss, every component in JBoss is a JMX MBean that can be hot-deployed. This part of the exercise shows you how to define a JMX MBean definition for a new JMS Queue.

The JMX configuration file can be found in the `ex13_2/src/resources/services` folder:

#### *jbossmq-titanqueues-service.xml*

```
||| <server>  
|||   <mbean code="org.jboss.mq.server.jmx.Queue"  
|||       name="jboss.mq.destination:service=Queue,  
|||           name=titan-ReservationQueue">  
|||     <depends optional-attribute-name="DestinationManager"  
|||       >jboss.mq:service=DestinationManager</depends>  
|||   </mbean>  
|||   <mbean code="org.jboss.mq.server.jmx.Queue"  
|||       name="jboss.mq.destination:service=Queue,  
|||           name=titan-TicketQueue">  
|||     <depends optional-attribute-name="DestinationManager"  
|||       >jboss.mq:service=DestinationManager</depends>  
|||   </mbean>  
||| </server>
```

Each set of MBeans must be defined within a `server` tag and each MBean is declared with an `mbean` tag. Since we need two different queues, we define two MBeans, one for each queue. The MBean class that represents a JMS queue is `org.jboss.mq.server.jmx.Queue` and takes in its `name` property, the name of the JMS queue to be created: `titan-ReservationQueue` and `titan-TicketQueue`. Copying this file into the JBoss deploy directory will hot-deploy this JMS queue and it will be ready for use.

One thing to note is that the `DestinationManager` MBean needs to be deployed by the application server before any Queue or Topic is deployed. This dependency is declared with the `depends` tag in `jbossmq-titanqueues-service.xml`. The `titan-ReservationQueue` and `titan-TicketQueue` will not be deployed until the `DestinationManager` MBean has finished initializing and is ready to service new Queues and Topics.

The `make-queues` Ant target has been defined to deploy the `jbossmq-titanqueues-service.xml` XML file in the JBoss' deploy folder:

```
C:\workbook\ex13_2>ant make-queues
Buildfile: build.xml

make-queues:
    [copy] Copying 1 file to C:\jboss-3.0.0\server\default\deploy
```

On the server side, the following lines are displayed:

```
[MainDeployer] Starting deployment of package: file:/C:/jboss-
3.0.0/server/default/deploy/jbossmq-titanqueues-service.xml
[titan-ReservationQueue] Creating
[titan-ReservationQueue] Created
[titan-TicketQueue] Creating
[titan-TicketQueue] Created
[titan-ReservationQueue] Starting
[titan-ReservationQueue] Bound to JNDI name:
                        queue/titan-ReservationQueue
[titan-ReservationQueue] Started
[titan-TicketQueue] Starting
[titan-TicketQueue] Bound to JNDI name: queue/titan-TicketQueue
[titan-TicketQueue] Started
[MainDeployer] Successfully completed deployment of package:
file:/C:/jboss-3.0.0/server/default/deploy/jbossmq-titanqueues-
service.xml
```

### Adding a JMS Queue Through the JMX HTTP Connector

To add a new JMS queue through the JMX HTTP connector, see the detailed instructions from the previous exercise. The only difference is that instead of using the `createTopic` operation of the JBossMQ server, you will use the `createQueue` operation.

Remember that queues or topics created within the JMX HTTP Connector only live until the application server is shut down. This is why an XML configuration file is preferable.

## Examine the EJB Standard Files

The `ejb-jar.xml` file is based on the one of the previous exercise. The only real difference is that a new EJB has been added: the `ReservationProcessor` MDB:

### *ejb-jar.xml*

```
<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>
```

The MDB descriptor defines the EJB with Container Managed Transactions (CMT), auto-acknowledgement of messages, and that messages will be received from a queue (and not from a topic).

The descriptor also contains a `message-selector` tag that allows the MDB to only receive messages that fulfill a given condition.

Then, a set of `ejb-ref` entries is used to reference all beans that are used by the `ReservationProcessor` beans during their execution:

```
<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-link>ProcessPaymentEJB</ejb-link>
</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-link>CustomerEJB</ejb-link>
```

```

</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-link>CruiseEJB</ejb-link>
</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-link>CabinEJB</ejb-link>
</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-link>ReservationEJB</ejb-link>
</ejb-local-ref>
<security-identity>
  <run-as><role-name>everyone</role-name></run-as>
</security-identity>

```

Since the MDB will send a confirmation message to a queue once the booking has been successful, it needs a reference to a `javax.jms.QueueConnectionFactory`. This is the intent of the `resource-ref` that is located at the end of the MDB descriptor.

Note that there is a difference with the previous exercise. While this bean will send messages to a queue, its descriptor does not contain a `resource-env-ref` entry that references the destination queue. The reason for this is that while, in the previous example, the destination was fixed and set at deployment time, in this exercise, the destination is not fixed and not even known by the MDB. It is the client application that knows the destination and it transmits it to the MDB by serializing the JMS queue object as part of the JMS message.

```

<resource-ref>
  <res-ref-name>jms/QueueFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>

```

```
||| </resource-ref>
||| </message-driven>
```

## ***Examine the JBoss Specific Files***

Since no modifications have been made to the CMP entity beans, the `jbosscomp-jdbc.xml` file is identical to the one of the previous exercise.

The `jboss.xml` file needs to be modified to take in account the new `ReservationProcessor` EJB:

### ***jboss.xml***

```
||| <message-driven>
|||   <ejb-name>ReservationProcessorEJB</ejb-name>
|||   <destination-jndi-name
|||     >queue/titan-ReservationQueue<
|||     /destination-jndi-name>
|||   <resource-ref>
|||     <res-ref-name>jms/QueueFactory</res-ref-name>
|||     <jndi-name>java:/JmsXA</jndi-name>
|||   </resource-ref>
||| </message-driven>
```

`destination-jndi-name` maps the MDB to an existing JMS destination in the deployment environment. You should recognize the name of one of the two JMS queues you just created: `titan-ReservationQueue`.

- ❖ By default, each MDB EJB deployed in JBoss can serve up to 15 concurrent messages.

`resource-ref` maps the `ConnectionFactory` name used by the `ReservationProcessor` EJB to an actual factory in the deployment environment. This mapping is identical to the one you made in the previous exercise for the `TravelAgent` EJB.

## ***Build and Deploy the Example Programs***

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the `ex13_12` directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
||| C:\workbook\ex13_2> set JAVA_HOME=C:\jdk1.3
||| C:\workbook\ex13_2> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```



Unix:

```
||| $ export JAVA_HOME=/usr/local/jdk1.3
||| $ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\ex13_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

## ***Examine the Client Applications***

In this exercise, you will use two client applications at the same time. The first application will produce large amounts of passage booking JMS messages destined for the ReservationProcessor MDB EJB. The second client application will listen to a JMS queue for booking confirmation messages and display them as they come in.

The message producer client application will first get from the command line the cruise ID on which it has to make bookings and the number of bookings to do:

### ***JmsClient\_ReservationProducer.java***

```
||| public static void main (String [] args) throws Exception
||| {
|||
|||     if (args.length != 2)
|||         throw new Exception ("Usage: java
||| JmsClient_ReservationProducer <CruiseID> <count>");
|||
|||     Integer cruiseID = new Integer (args[0]);
|||     int count = new Integer (args[1]).intValue ();
```

It then looks up a *QueueConnectionFactory* and two JMS queues from the JBoss naming service. The first queue is the one bound to the ReservationProcessor MDB and to which passage booking messages will be sent. The second queue is not directly used, as you will see later.

```
||| QueueConnectionFactory factory = (QueueConnectionFactory)
|||     jndiContext.lookup ("ConnectionFactory");
|||
||| Queue reservationQueue = (Queue)
```

```

jndiContext.lookup ("queue/titan-ReservationQueue");
Queue ticketQueue = (Queue)
jndiContext.lookup ("queue/titan-TicketQueue");

QueueConnection connect = factory.createQueueConnection ();
QueueSession session = connect.createQueueSession
(false, Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender (reservationQueue);

```

The client application is now ready to send `count` passage booking JMS messages in batch. For this, it creates a JMS `MapMessage` and first assigns the second JMS queue that has been previously looked up from the `JMSReplyTo` property of the message. The ReservationProcessor MDB will use this queue to send a confirmation message. Then, the different information needed to book the passage is set in the message: Cruise ID, Customer ID, Cabin ID, price, credit card number and expiration date, etc. Note that only basic data types (String, Integer, etc) can be stored in a `MapMessage`.

One interesting property that is set on the JMS message **header** is `MessageFormat`. If you remember the MDB deployment descriptor, this property is used in the `message-selector` tag to constrain the messages that the MDB wants to receive.

```

for (int i = 0; i < count; i++)
{
    MapMessage message = session.createMapMessage ();

    // Used in ReservationProcessor to send Tickets back out
    message.setJMSReplyTo (ticketQueue);

    message.setStringProperty ("MessageFormat", "Version 3.4");

    message.setInt ("CruiseID", cruiseID.intValue ());
    // either Customer 1 or 2, all we've got in database
    message.setInt ("CustomerID", i%2 + 1);
    // cabins 100-109 only
    message.setInt ("CabinID", i%10 + 100);
    message.setDouble ("Price", (double)1000 + i);

    // the card expires in about 30 days
    Date expDate = new Date (System.currentTimeMillis () +
                             30*24*60*60*1000L);

    message.setString ("CreditCardNum", "5549861006051975");
    message.setLong ("CreditCardExpDate", expDate.getTime ());
    message.setString ("CreditCardType",
                      CreditCardDO.MASTER_CARD);

    System.out.println ("Sending reservation message #" + i);
}

```

```

        sender.send (message);
    }

    connect.close ();
}

```

Once all messages are sent, the application terminates. Since messages are sent asynchronously, the application may terminate before the ReservationProcessor EJB has processed all of the same messages.

The second client application is very similar to the client application you have implemented in the previous exercise (13.1). This time though, it will not subscribe to a topic but to a queue.

To receive JMS messages, the client application class implements the `javax.jms.MessageListener` interface that defines the `onMessage` method. The main method simply creates an instance of the class and uses a trick to make the main thread indefinitely waits.

### ***JmsClient\_TicketConsumer.java***

```

public class JmsClient_TicketConsumer
    implements javax.jms.MessageListener
{

    public static void main (String [] args) throws Exception
    {
        new JmsClient_TicketConsumer ();

        while(true) { Thread.sleep (10000); }
    }
}

```

The constructor is very simple JMS code that subscribes the client application to the JMS queue and waits for incoming messages:

```

public JmsClient_TicketConsumer () throws Exception
{
    Context jndiContext = getInitialContext ();

    QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup ("ConnectionFactory");

    Queue ticketQueue = (Queue)
        jndiContext.lookup ("queue/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 ();
}

```

When a message arrives in the queue, the `onMessage` method of the client application is called. The method will simply display the content of the ticket:

```

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 (JMSEException displayed)
    {
        displayed.printStackTrace ();
    }
}

```

## ***Run the Client Applications***

As you redeployed the *titan.jar* file, JBoss dropped and recreated the database tables, destroying any content that was present. For this reason, you must make Ant execute the `run.client_122a` target to repopulate the database.

- ❖ Even if `run.client_122a` target originates from exercise 12.2, it has been duplicated in the `ex13_2` directory to facilitate your work.

You will next launch the client application that will receive the tickets as passage confirmations. For this, you can use the `run.client_132` Ant target:

```

C:\workbook\ex13_2>ant run.client_132
Buildfile: build.xml

init:
[echo] build.compiler = classic
[echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
[echo]

prepare:

```

```

compile:

ejbjar:

run.client_132:
    [java] Listening for messages on titan-TicketQueue...

```

Then, you can start the client application that will book several passages. You can do this by using the `BookInBatch` script. The usage is as follow:

```
BookInBatch <cruiseID> <count>
```

Where `cruiseID` is the id of a cruise in the database (resulting of the execution of the `run.client_122a` Ant target) and `count` is the number of passages to book.

Let's try to book 100 passages:

```

C:\workbook\ex13_2>BookInBatch 142844880 100
Buildfile: build.xml

init:
    [echo] build.compiler = classic
    [echo] java.class.path = c:\jdk1.3.1\lib\tools.jar;
    ..\ant\lib\jaxp.jar;..\ant\lib\crimson.jar;..\ant\lib\ant.jar;
    [echo]

prepare:

compile:

ejbjar:

run.bookinbatch:
    [java] Sending reservation message #0
    [java] Sending reservation message #1
    [java] Sending reservation message #2
    [java] Sending reservation message #3
    ...
    [java] Sending reservation message #98
    [java] Sending reservation message #99

```

Shortly after, the client application that is listening to its JMS queue for booking confirmations will display the following lines:

```

run.client_132:
    [java] Listening for messages on titan-TicketQueue...
    [java] *****
    [java] Bob Smith has been booked for the Alaska Cruise cruise
           on ship Nordic Prince.

```

```

[java] Your accommodations include Suite 100 a 1 bed cabin on
       deck level 1.
[java] Total charge = 1000.0
[java] *****
[java] *****
[java] Joseph Stalin has been booked for the Alaska Cruise
       cruise on ship Nordic Prince.
[java] Your accommodations include Suite 101 a 1 bed cabin on
       deck level 1.
[java] Total charge = 1001.0
[java] *****
[java] *****
[java] Bob Smith has been booked for the Alaska Cruise cruise
       on ship Nordic Prince.
[java] Your accommodations include Suite 102 a 1 bed cabin on
       deck level 1.
[java] Total charge = 1002.0
[java] *****
...
[java] *****
[java] Joseph Stalin has been booked for the Alaska Cruise
       cruise on ship Nordic Prince.
[java] Your accommodations include Suite 109 a 1 bed cabin on
       deck level 1.
[java] Total charge = 1099.0
[java] *****

```

Here we are! The loop is closed.

Note that as we are using a queue to receive booking confirmation messages. Consequently, you could have started the client application that receives these messages much later. The confirmation messages sent by the ReservationProcessor MDB would have been stored on the server until the client applications starts and begins to listen to the queue.

# Appendix A

## Appendix A: Database Configuration

Appendix A describes how to set up database pools for data sources other than the default JBoss embedded database. It also illustrates how to set up your EJBs to use these database pools. The example will show how to configure an Oracle connection pool for use with JBoss. The example program from Chapter 6.1 has been modified to work with this Oracle connection pool.

### *Setup the Database*

First of all you must download the JDBC driver classes for your database. Copy your database's JDBC jar file to *\$JBOSS\_HOME/lib*. For example, the Oracle JDBC class files are contained in *classes12.zip*.

The JBoss distribution includes example db connection pool files under the *\$JBOSS\_HOME/docs/examples/jca* directory. For this chapter, we have copied the *oracle-service.xml* configuration file to *exAppendixA/titandb-service.xml* and modified it accordingly.

To deploy this db connection pool, the *titandb-service.xml* file must be copied to the *\$JBOSS\_HOME/service/default/deploy* directory. Config files like this must have a *service.xml* at the end the file or JBoss will not deploy it.

- ❖ Database connection pools are among the many things that can be hot-deployed in JBoss by plopping the pool's xml configuration file in the deploy directory.

Let's examine some of the configuration parameters defined in this file.

#### **titandb-service.xml**

```
<mbean
code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
name="jboss.jca:service=LocalTxCM,name=OracleDS">
...
<depends optional-attribute-name = "ManagedConnectionFactoryName">
  <!--embedded mbean-->
  <mbean code= "org.jboss.resource.connectionmanager.RARDeployment"
name="jboss.jca:service=LocalTxDS,name=OracleDS">

    <attribute name="JndiName">OracleDS</attribute>
```

The `JndiName` attribute is the name of the db connection pool within JNDI. You can find this pool in JNDI with the `java:/OracleDS`. The class of this bound object is `javax.sql.DataSource`.

```
<attribute name="ManagedConnectionFactoryProperties">
  <properties>
    <config-property name="ConnectionURL" type="java.lang.String">
      jdbc:oracle:thin:@localhost:1521:FABCONWB
    </config-property>
  </properties>
</attribute>
```

The `ConnectionURL` attribute tells the Oracle JDBC driver how to connect to the database. Consult your database JDBC manuals on how to obtain this URL. It may be different per database type.

```
<config-property name="DriverClass" type="java.lang.String">
  oracle.jdbc.driver.OracleDriver
</config-property>
```

The `DriverClass` attribute tells JBoss and the base JDBC classes the name of Oracle's JDBC driver class to instantiate and use.

```
<config-property name="UserName" type="java.lang.String">
  scott
</config-property>
<config-property name="Password" type="java.lang.String">
  tiger
</config-property>
</properties>
</attribute>
```

Finally, the `UserName` and `Password` attributes are used when connecting to the Oracle database.

Ok, so the first part of this file describes the JDBC driver and how to connect to it. The second part of *titandb-service.xml* describes the attributes of the connection pool.

```
...
<depends optional-attribute-name="ManagedConnectionPool">
  <!--embedded mbean-->
  <mbean
code="org.jboss.resource.connectionmanager.JBossManagedConnectionPool"
name="jboss.jca:service=LocalTxPool,name=OracleDS">

    <attribute name="MinSize">0</attribute>
    <attribute name="MaxSize">50</attribute>
```



The `MinSize` and `MaxSize` attributes tell JBoss the initial JDBC connections to have in the connection pool and the maximum connections that are allowed.

```
<attribute name="BlockingTimeoutMillis">5000</attribute>
```

Whenever the available connections are maxed out in the pool, a thread interested in obtaining a new connection will block until a connection is released back into the pool.

`BlockingTimeoutMillis` is the maximum time a thread will wait for a connection until it aborts and throws an exception.

```
<attribute name="IdleTimeoutMinutes">15</attribute>
```

When a connection has been idle in the connection pool for more than `IdleTimeoutMinutes` it will be closed and released from the connection pool.

```
<!--criteria indicates if Subject (from security domain) or app
supplied
    parameters (such as from getConnection(user, pw)) are used
to distinguish
    connections in the pool. Choices are
    ByContainerAndApplication (use both),
    ByContainer (use Subject),
    ByApplication (use app supplied params only),
    ByNothing (all connections are equivalent, usually if
adapter supports
    reauthentication)-->
<attribute name="Criteria">ByContainer</attribute>
</mbean>
```

## ***Examine the JBoss Specific Files***

The example code for this section has been borrowed from Chapter 6.1 of this workbook. It is fairly easy to configure the EJBs from this chapter to use the Oracle connection pool you created above. Simply point the datasource to `java:/OracleDS` and use the Oracle8 database mapping.

### **jbosscomp-jdbc.xml**

```
<jbosscomp-jdbc>

<defaults>
  <datasource>java:/OracleDS</datasource>
  <datasource-mapping>Oracle8</datasource-mapping>
  <create-table>true</create-table>
  <remove-table>true</remove-table>
```

```

</defaults>

<enterprise-beans>
  <entity>
    <ejb-name>CustomerEJB</ejb-name>
    <table-name>Customer</table-name>
    <cmp-field>
      <field-name>id</field-name>
      <column-name>ID</column-name>
    </cmp-field>
    <cmp-field>
      <field-name>lastName</field-name>
      <column-name>LAST_NAME</column-name>
    </cmp-field>
    <cmp-field>
      <field-name>firstName</field-name>
      <column-name>FIRST_NAME</column-name>
    </cmp-field>
    <cmp-field>
      <field-name>hasGoodCredit</field-name>
      <column-name>HAS_GOOD_CREDIT</column-name>
    </cmp-field>
  </entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

## Startup JBoss

You must restart JBoss for this example for JBoss to recognize the JDBC jar file you copied into the lib directory. Please review the *Server Installation and Configuration* chapter at the beginning of this book if you don't remember you to start JBoss.

## Build and Deploy the Example Programs

To build and deploy the example for this chapter, you must configure the *titandb-service.xml* file described above to conform to the database you are using.

Perform the following steps:

1. Open a command prompt or shell terminal and change directories down to the *exAppendixA* directory created by the extraction process
2. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.0 is installed. Here are some examples:

Windows:

```
||| C:\workbook\exAppendixA> set JAVA_HOME=C:\jdk1.3
```

```
||| C:\workbook\exAppendixA> set JBOSS_HOME=C:\jboss-3.0.0FINAL
```

Unix:

```
||| $ export JAVA_HOME=/usr/local/jdk1.3
||| $ export JBOSS_HOME=/usr/local/jboss-3.0.0
```

3. Add *ant* to your execution path. *Ant* is the build utility

Windows:

```
||| C:\workbook\exAppendixA> set PATH=..\ant\bin;%PATH%
```

Unix:

```
||| $ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing *ant*. *Ant* is the build utility. It uses *build.xml* to figure out what to compile and how to build your jars.

You will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server. The build script also copies *titandb-service.xml* to the *deploy* directory as well. This causes the customer database pool to be deployed.

## ***Examine and Run the Client Applications***

There is only one single client application for this exercise, *Client\_61*. It is modeled after the example in the EJB book. It will create Customer EJBs in the database based on the command line parameters.

To run the client first set your JBOSS\_HOME and PATH environment variables appropriately. Then invoke the provided wrapper script to execute the program. You must supply a set of primary key, first name, and last name values of the command line as shown here:

```
Client_61 777 Bill Burke 888 Sacha Labourey
```

The output of this execution should be:

```
||| 777 = Bill Burke
||| 888 = Sacha Labourey
```

The example program removes the created beans at the conclusion of operation so there will be no data in the database.