

Developing Enterprise Java Components

5th Edition
Covers Java Persistence

Enterprise JavaBeans™ 3.0



O'REILLY®

Bill Burke & Richard Monson-Haefel

Enterprise JavaBeans™ 3.0, Fifth Edition

by Bill Burke and Richard Monson-Haefel

Copyright © 2006, 2004, 2001, 2000, 1999 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides

Production Editor: Colleen Gorman

Copyeditor: Audrey Doyle

Proofreaders: Matt Hutchinson and Rachel Monaghan

Indexer: Julie Hawks

Cover Designer: Hanna Dyer

Interior Designer: David Futato

Illustrators: Robert Romano and Jessamyn Read

Printing History:

June 1999:	First Edition.
March 2000:	Second Edition.
September 2001:	Third Edition.
June 2004:	Fourth Edition.
May 2006:	Fifth Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Enterprise JavaBeans™*, Fifth Edition, the image of a wallaby and joey, and related trade dress are trademarks of O'Reilly Media, Inc.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems. Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries. JBoss is fully owned and operated by JBoss, Inc. in the United States and other countries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00978-X

[M]

This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.

Developing Your First Beans

The primary goal of the EJB 3.0 and Java Persistence specifications was to make it as easy as possible to write and deploy an EJB-based application. Creating an application is as easy as compiling your code, jarring up your classes, and running your application server. This chapter gives an introduction to writing your first entity and session bean. You'll find that getting up and running is fairly simple.

Developing an Entity Bean

Let's start by examining how to create an entity bean. We'll implement the Cabin entity that is part of the Titan Cruises Java EE application. The Cabin entity encapsulates the data and behavior associated with a cruise ship cabin in Titan's business domain. Although you can interact with entity beans outside of an application server, we will later create a TravelAgent session bean to serve as a data access interface for creating and locating cabins.

Cabin: The Bean Class

When developing an entity bean, all we need to define is the bean class. Entities in Java Persistence are plain Java objects that are annotated with O/R mapping metadata. We started to define the Cabin entity in Chapter 2; here, we add two new methods for setting and getting the ship ID and the bed count. The ship ID identifies the ship to which the cabin belongs, and the bed count tells how many people the cabin can accommodate:

```
package com.titan.domain

import javax.persistence.*;

@Entity
@Table(name="CABIN")
public class Cabin implements java.io.Serializable{
```

```

private int id;
private String name;
private int deckLevel;
private int shipId;
private int bedCount;

@Id
@Column(name="ID")
public int getId() { return id; }
public void setId(int pk) { id = pk; }

@Column(name="NAME")
public String getName() { return name; }
public void setName(String str) {name = str; }

@Column(name="DECK_LEVEL")
public int getDeckLevel() { return deckLevel; }
public void setDeckLevel(int level) { deckLevel = level; }

@Column(name="SHIP_ID")
public int getShipId() { return shipId; }
public void setShipId(int sid) { shipId = sid; }

@Column(name="BED_COUNT")
public int getBedCount() { return bedCount; }
public void setBedCount(int bed) { bedCount = bed; }
}

```

The Cabin bean class is annotated with `@javax.persistence.Entity` and `@javax.persistence.Table`. The `@Entity` annotation tells the persistence provider that this is an entity class that is mapped to a database and that can be managed by an `EntityManager` service. The `@Table` annotation tells the EJB container to which database table the bean class should map. The bean class implements `java.io.Serializable`, but it is not required to. Having entity classes implement `Serializable` allows them to be used as the parameters and return values of the remote interface methods of a session bean. This allows you to use the same class for both persistence and data transfer.

Cabin also defines four properties: `name`, `deckLevel`, `shipId`, and `bedCount`. *Properties* are attributes of an entity bean that can be accessed by public *set* and *get* methods; they can also be accessed directly through the bean's fields. In this example, we use public *set* and *get* methods. For each property, we define how it maps to the columns in the CABIN database table with the `@javax.persistence.Column` annotation. The `getId()` property is marked as the primary key of the Cabin entity by using the `@javax.persistence.Id` annotation. The primary key denotes the identity of a particular entity bean at runtime and when it is persisted in the database.



Table-mapping annotations like `@Table` and `@Column` are not required; if they're omitted, they default to the unqualified name of the class and the property name, respectively. However, primary-key identification is required.

Notice that we have made the `Cabin` bean class a part of a new package named `com.titan.domain`. Place all the classes and interfaces associated with each type of bean in a package specific to the bean. Because our beans are for use by Titan Cruises, we placed these packages in the `com.titan` package hierarchy. We also created directory structures that match package structures. If you are using an IDE that works directly with Java files, create a new directory called *dev* (for development) and create the directory structure shown in Figure 4-1. Copy the `Cabin` bean class into your IDE and save its definition to the *domain* directory. Compile the `Cabin` bean class to ensure that its definition is correct. The *Cabin.class* file, generated by the IDE's compiler, should be written to the *domain* directory, the same directory as the *Cabin.java* file. The rest of the `Cabin` bean's classes will be placed in this same directory. We are done defining the `Cabin` entity.

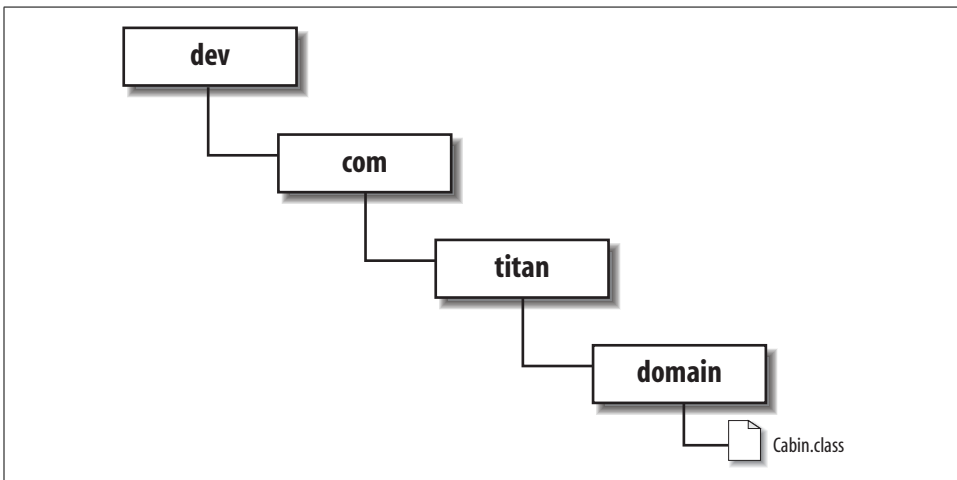


Figure 4-1. *dev* directory structure

The persistence.xml File

The Java Persistence specification requires a simple XML deployment descriptor file, *persistence.xml*, which configures basic things like the name of the `EntityManager` service that will be managing the set of entities deployed in a particular deployment package. It also defines what database the `EntityManager` service will be communicating with and may also specify additional, vendor-specific properties.

```
<persistence>
  <persistence-unit name="titan">
```

```
        <jta-data-source>java:/TitanDB</jta-data-source>
    </persistence-unit>
</persistence>
```

The `<name>` element represents the set of classes that are managed by a given `EntityManager`. The `<jta-data-source>` element defines the database that will be used to persist entities in this deployment. This `persistence.xml` file is located in a `META-INF` directory. `persistence.xml` is discussed in detail in Chapter 5.

Developing a Session Bean

Now that our `Cabin` entity is ready, we need to create a session bean that will act as the interface for interacting with the `Cabin` entity. This session bean will take on the business process and responsibilities of a travel agent and will be called the `TravelAgent EJB`.

TravelAgentRemote: The Remote Interface

The first part of creating the `TravelAgent EJB` is to define its remote interface. This interface specifies what business methods a client is allowed to invoke on the `EJB`. Clients interact with the `TravelAgent`'s `createCabin()` and `findCabin()` methods to manipulate `Cabin` entities:

```
package com.titan.travelagent;

import javax.ejb.Remote;
import com.titan.domain.Cabin;

@Remote
public interface TravelAgentRemote {

    public void createCabin(Cabin cabin);
    public Cabin findCabin(int id);
}
```

There really is nothing special about the remote interface of the `TravelAgent EJB`. It looks like a plain Java interface, except that it is annotated with the `@javax.ejb.Remote` annotation. This annotation tells the `EJB` container that this particular interface is the remote business interface of the `TravelAgent EJB`. Unlike `EJB 2.1`, also notice that the business methods do not have to throw `java.rmi.RemoteException`. They can if they want to, but they don't have to.

TravelAgentBean: The Bean Class

Now that we have defined the remote interface to the `TravelAgent EJB`, we need to implement the bean class that contains the business logic for this session bean. The `TravelAgent EJB` is defined as a stateless bean. We use the `@javax.ejb.Stateless`

annotation to denote this. Although they are not required to, it is good practice for stateless session beans to implement all of their business interfaces so that the client/bean contract can be enforced by the Java compiler. In this case, the business interface is `TravelAgentRemote`. Here is the complete definition of the `TravelAgentBean` class:

```
package com.titan.travelagent;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.titan.domain.Cabin;

@Stateless
public class TravelAgentBean implements TravelAgentRemote{
    @PersistenceContext(unitName="titan") private EntityManager manager;

    public void createCabin(Cabin cabin) {
        manager.persist(cabin);
    }

    public Cabin findCabin(int pKey) {
        return manager.find(Cabin.class, pKey);
    }
}
```

A bunch of things are going on in this implementation. First, the `TravelAgentBean` class uses the `@javax.persistence.PersistenceContext` annotation to get access to the `EntityManager` service that it uses to create and find `Cabin` entities. `@PersistenceContext` tells the EJB container that it must set the `manager` field with an `EntityManager` instance. The EJB container sees that the type of the field is `javax.persistence.EntityManager` and knows that it must set the field to be a reference to an `EntityManager` service that references the Titan persistence unit we defined in the `persistence.xml` file. This field will be initialized when the bean instance is instantiated.

The next two methods interact with the `EntityManager` service to create a `Cabin` entity within the database and to locate a `Cabin` entity based on its primary key. The `createCabin()` method invokes `EntityManager.persist()`, passing in an instance of the entity that we want to create. After this operation is complete, the `Cabin` instance is attached to persistence management and is stored in the database. We'll see later on that you create the `Cabin` instance as you would any other plain Java object. The `findCabin()` method takes as its parameter a primary key of a cabin in the database. It calls the `EntityManager.find()` method, passing the `Cabin` bean class as a parameter and the actual primary key. The `Cabin` bean class parameter tells the `EntityManager` which entity it is trying to find in the database. The `findCabin()` method then returns the found `Cabin` entity back to the remote client.

titan.jar: The JAR File

The JAR file is a platform-independent file format for compressing, packaging, and delivering several files together. Based on the Zip file format and the zlib compression standards, the JAR (Java Archive) tool was originally developed to make downloads of Java applets more efficient. As a packaging mechanism, however, the JAR file format is a very convenient way to “shrink-wrap” components and other software for delivery to third parties. In EJB development, a JAR file packages all the classes and interfaces associated with a bean. Besides EJB definitions and classes, you are also allowed to package entity beans and their *persistence.xml* deployment descriptor.

Creating the JAR file for deployment is easy. Position yourself in the *dev* directory that is just above the *com/titan* directory tree and execute the following command:

```
C:\dev> jar cf titan.jar com/titan/domain/*.class
                        com/titan/travelagent/*.class
                        META-INF/persistence.xml
```

The *c* option tells the *jar* utility to create a new JAR file that contains the files indicated in subsequent parameters. It also tells the *jar* utility to stream the resulting JAR file to standard output. The *f* option tells *jar* to redirect the standard output to a new file named in the second parameter (*titan.jar*). It's important to get the order of the option letters and the command-line parameters to match. You can learn more about the *jar* utility and the *java.util.zip* package in *Java in a Nutshell* or *Learning Java* (both published by O'Reilly).

The *jar* utility creates the file *titan.jar* in the *dev* directory. If you're interested in looking at the contents of the JAR file, you can use any standard Zip application (WinZip, PKZIP, etc.) or the command `jar tvf titan.jar`. Figure 4-2 shows the structure of this Jar file.

Creating a CABIN Table in the Database

One of the primary jobs of a deployment tool is mapping entity beans to databases. In the case of the Cabin entity, we must map its *id*, *name*, *deckLevel*, *shipId*, and *bedCount* fields to some data source. Before proceeding with deployment, you need to set up a database and create a CABIN table. You can use the following standard SQL statement to create a CABIN table that will be consistent with the examples provided in this chapter:

```
create table CABIN
(
    ID int primary key NOT NULL,
    SHIP_ID int,
    BED_COUNT int,
    NAME char(30),
    DECK_LEVEL int
)
```

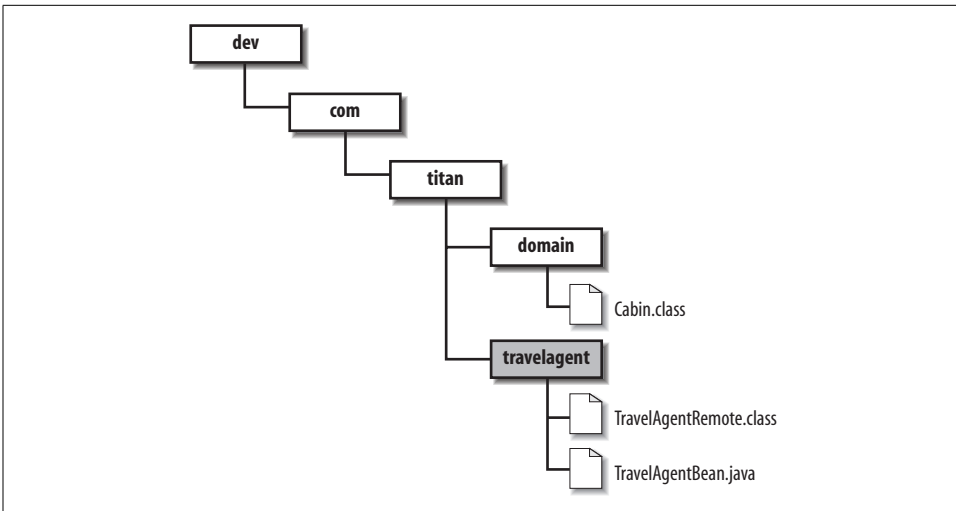


Figure 4-2. titan.jar contents

This statement creates a CABIN table that has five columns corresponding to the container-managed fields in the Cabin class. Once the table is created and connectivity to the database is confirmed, you can proceed with the deployment process.

Alternatively, most if not all persistence provider implementations will support auto-generation of your database tables. For example, you can configure JBoss to create the tables for each entity bean when the application server boots and deploys its EJBs. Other application servers will have different mechanisms to do this.

Deploying the EJB JAR

Deployment is the process of reading the bean's JAR file, changing or adding properties to the deployment descriptor, mapping the bean to the database, defining access control in the security domain, and generating any vendor-specific classes needed to support the bean in the EJB environment. Some EJB server products require you to use a set of deployment tools to deploy your EJBs to your application server. These tools may provide a graphical user interface or a set of command-line programs.

A deployment tool reads the JAR file and looks for annotated classes as well as any possible XML deployment descriptors so that it can determine which EJBs and entities are being deployed. In a graphical deployment wizard, the metadata of each EJB and entity is presented using a set of property sheets similar to those used in environments such as Visual Basic .NET, PowerBuilder, and JBuilder.

Some EJB servers, like JBoss, do not require any special vendor-generated classes, so there is no need for a deployment tool. JBoss, for instance, just requires you to put your *ejb* JAR in a *deploy/* directory. The application server examines the JAR file at

runtime when the server boots up to determine which EJB containers must be created and attached to the runtime.

Creating a Client Application

Now that the Cabin entity and TravelAgent EJBs have been deployed, we want to access them from a remote client. In this section, we create a remote client that connects to the EJB server, locates the EJB remote interface for the TravelAgent EJB, and interacts with the TravelAgent EJB to create and pull Cabin entities from the database. The following code shows a Java application that creates a new Cabin entity, sets its name, deckLevel, shipId, and bedCount properties, and then locates it again using its primary key:

```
package com.titan.clients;

import com.titan.travelagent.TravelAgentRemote;
import com.titan.domain.Cabin;

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

public class Client {
    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref = jndiContext.lookup("TravelAgentBean/remote");
            TravelAgentRemote dao = (TravelAgentRemote)
                PortableRemoteObject.narrow(ref, TravelAgentRemote.class);

            Cabin cabin_1 = new Cabin();
            cabin_1.setId(1);
            cabin_1.setName("Master Suite");
            cabin_1.setDeckLevel(1);
            cabin_1.setShipId(1);
            cabin_1.setBedCount(3);

            dao.createCabin(cabin_1);

            Cabin cabin_2 = dao.findCabin(1);
            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 (javax.naming.NamingException ne){ne.printStackTrace();}
    }
}
```

```

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

    Properties p = new Properties();
    // ... Specify the JNDI properties specific to the vendor.
    return new javax.naming.InitialContext(p);
}
}

```

To access an enterprise bean, a client starts by using JNDI to obtain a directory connection to a bean's container. JNDI is an implementation-independent API for directory and naming systems. Every EJB vendor must provide a directory service that is JNDI-compliant. This means that they must provide a JNDI service provider, which is a piece of software analogous to a driver in JDBC. Different service providers connect to different directory services—not unlike JDBC, where different drivers connect to different relational databases. The `getInitialContext()` method uses JNDI to obtain a network connection to the EJB server.

The code used to obtain a JNDI context depends on which EJB vendor you use. Consult your vendor's documentation to find out how to obtain a JNDI context appropriate to your product. For example, the code used to obtain a JNDI context in WebSphere might look something like the following:

```

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

    java.util.Properties properties = new java.util.Properties();
    properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
    properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
    return new InitialContext(properties);
}
}

```

The same method developed for JBoss would be different:

```

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

    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");
    p.put(Context.URL_PKG_PREFIXES,
        " org.jboss.naming:org.jnp.interfaces");
    p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
    return new javax.naming.InitialContext(p);
}
}

```

Once a JNDI connection is established and a context is obtained from the `getInitialContext()` method, the context can be used to look up the remote interface of the `TravelAgentEJB`:

```

Object ref = jndiContext.lookup("TravelAgentBean/remote");

```

Throughout this book, we'll use lookup names like "TravelAgentBean/remote" for remote client applications. The actual name you use to do a lookup may be different, depending on the requirements of your vendor. You will need to bind a lookup name to the EJB server's naming service, and some vendors may require a special directory path or provide a default binding.

If you are using a standard Java EE component (Servlet, JSP, EJB, or Java EE Application Client), you will not need to set the properties explicitly when creating a JNDI `InitialContext`, no matter which EJB vendor you are using. That's because the JNDI properties can be configured at deployment time and are applied automatically. A Java EE component would obtain its `InitialContext` as follows:

```
public static Context getInitialContext()  
    throws javax.naming.NamingException {  
  
    return new javax.naming.InitialContext();  
}
```

This is simpler and more portable than configuring JNDI properties for simple Java clients. All Java EE components use the same JNDI naming system that enterprise beans use to look up any service. Specifically, they require that EJB references be bound to the "java:comp/env/ejb/" namespace. For example, for a different Java EE component like a servlet, here's all we need to look up the `TravelAgentEJB`:

```
Object ref = jndiContext.lookup("java:comp/env/ejb/TravelAgentRemote");
```

At deployment time, you would use the vendor's deployment tools to map that JNDI name to the `TravelAgentEJB`'s remote interface. In later sections, we'll see that special annotations allow you to inject EJB references directly into the bean class. We already saw an example of one of them when the `EntityManager` service was injected into the `TravelAgentBean` class. In this book, Java client applications will need to use explicit parameters for JNDI lookups. As an alternative, you could use a special Java EE component called a Java EE Application Client, but this type of component is outside the scope of this book. For more information about Java EE Application Client components, consult the Java EE 5 specification.

The Client application uses the `PortableRemoteObject.narrow()` method to narrow the `Object ref` to a `TravelAgentRemote` reference:

```
Object ref = jndiContext.lookup("TravelAgentRemote");  
CabinHomeRemote home = (TravelAgentRemote)  
    PortableRemoteObject.narrow(ref, TravelAgentRemote.class);
```

The `PortableRemoteObject.narrow()` method was first introduced in EJB 1.1 and continues to be used on remote clients in EJB 3.0. It is needed to support the requirements of RMI over IIOP. Because CORBA supports many different languages, casting is not native to CORBA (some languages don't have casting). Therefore, to get a remote reference to `TravelAgentRemote`, we must explicitly narrow the object returned from `lookup()`.

The name used to find the TravelAgent EJB's remote interface is set by a vendor-specific default value, a vendor-specific annotation or deployment descriptor, or the deployer using a deployment wizard if one exists for that EJB product. The JNDI name is entirely up to the person deploying the bean; it can be the same as the bean name set in the XML deployment descriptor, or something completely different.

Creating a new Cabin entity

In this example, you finally see how Cabin entities are created. Java's `new()` operator is used to allocate an instance of the Cabin bean class. Nothing magical is happening. The client initializes the properties locally on the Cabin bean instance. The `id`, `name`, `deckLevel`, `shipId`, and `bedCount` of the Cabin entity are set:

```
Cabin cabin_1 = new Cabin();
Cabin_1.setId(1);
cabin_1.setName("Master Suite");
cabin_1.setDeckLevel(1);
cabin_1.setShipId(1);
cabin_1.setBedCount(3);
```

The Cabin entity does not get inserted into the database when you allocate it on the client. The instance must be passed to the TravelAgent EJB where it will be created in the database when the `EntityManager.persist()` method is called.

Figure 4-3 shows how the relational database table we created should look after this code has been executed. It should contain one record.

ID	NAME	SHIP_ID	BED_COUNT	DECK_LEVEL
1	Master Suite	1	3	1

Figure 4-3. CABIN table with one cabin record

The client locates Cabin entity beans by passing the primary key to the `TravelAgentRemote`'s `findCabin()` method. As you saw before, this session bean interacts with the `EntityManager` service to find the bean in the database. The `TravelAgent` passes back an instance of a `Cabin` with that primary key. This is possible because we defined the `Cabin` bean class to implement the `java.io.Serializable` interface, which allowed the `Cabin` bean instance to be marshaled across the wire back to the client.

We can now interrogate the Cabin bean instance locally to get the Cabin entity's name, deckLevel, shipId, and bedCount:

```
Cabin cabin_2 = dao.findCabin(1);
System.out.println(cabin_2.getName());
System.out.println(cabin_2.getDeckLevel());
System.out.println(cabin_2.getShipId());
System.out.println(cabin_2.getBedCount());
```

We are ready to create and run the Client application. Compile the Client application and deploy the Cabin entity into the container system. Then run the Client application. The output should look something like this:

```
Master Suite
1
1
3
```

Congratulations! You just created and used your first stateless session bean and entity bean. Of course, the Client application doesn't do much, but this is a good first step to learning how to implement EJBs and entities.