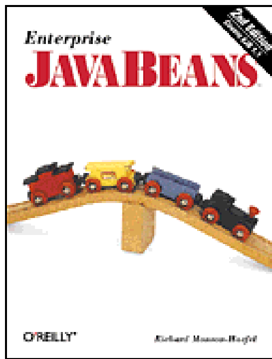


# Enterprise JavaBeans, 2nd Edition

By Richard Monson-Haefel
2nd Edition March 2000
1-56592-869-5, Order Number: 8695
350 pages, $34.95

---

# Chapter 4
# Developing Your First Enterprise Beans

**In this chapter:**

Choosing and Setting Up an EJB Server
Developing an Entity Bean
Developing a Session Bean

## Choosing and Setting Up an EJB Server

One of the most important features of EJB is that beans should work with containers from different vendors. That doesn't mean that selecting a server and installing your beans on that server are trivial processes. We'll start this chapter with a general discussion of how you select and set up a server.

The EJB server you choose should be compliant with the EJB 1.0 or EJB 1.1 specification. However, in the EJB 1.0 version of the specification, support for entity beans and container-managed persistence is optional. In EJB 1.1, support for entity beans is required. The first example in this chapter--and most of the examples in this book--assume that your EJB server supports entity beans and container-managed persistence.[1] The EJB server you choose should also provide a utility for deploying an enterprise bean. It doesn't matter whether the utility is command-line oriented or graphical, as long as it does the job. The deployment utility should allow you to work with prepackaged enterprise beans, i.e., beans that have already been developed and archived in a JAR file. Finally, the EJB server should support an SQL-standard relational database that is accessible using JDBC. For the database, you should have privileges sufficient for creating and modifying a few simple tables in addition to normal read, update, and delete capabilities. If you have chosen an EJB server that does not support an SQL standard relational database, you may need to modify the examples to work with the product you are using.

### Setting Up Your Java IDE

To get the most from this chapter, it helps to have an IDE that has a debugger and allows you to add Java files to its environment. Several Java IDEs, like Symantec's Visual Cafe, IBM's VisualAge, Inprise's JBuilder, and Sun's Forte, fulfill this simple requirement. The debugger is especially important because it allows you to walk slowly through your client code and observe how the EJB client API is used.

Once you have an IDE set up, you need to include the Enterprise JavaBeans packages. These packages include `javax.ejb` and for EJB 1.0 `javax.ejb.deployment`. You also need the JNDI packages, including `javax.naming`, `javax.naming.directory`, and `javax.naming.spi`. In addition, you will need the `javax.rmi` package for EJB 1.1. All these packages can be downloaded from Sun's Java site (*http://www.javasoft.com*) in the form of ZIP or JAR files. They may also be accessible in the subdirectories of your EJB server, normally under the *lib* directory.

# Developing an Entity Bean

There seems to be no better place to start than the Cabin bean, which we have been examining throughout the previous chapters. The Cabin bean is an entity bean that encapsulates the data and behavior associated with a real-world cruise ship cabin in Titan's business domain.

## Cabin: The Remote Interface

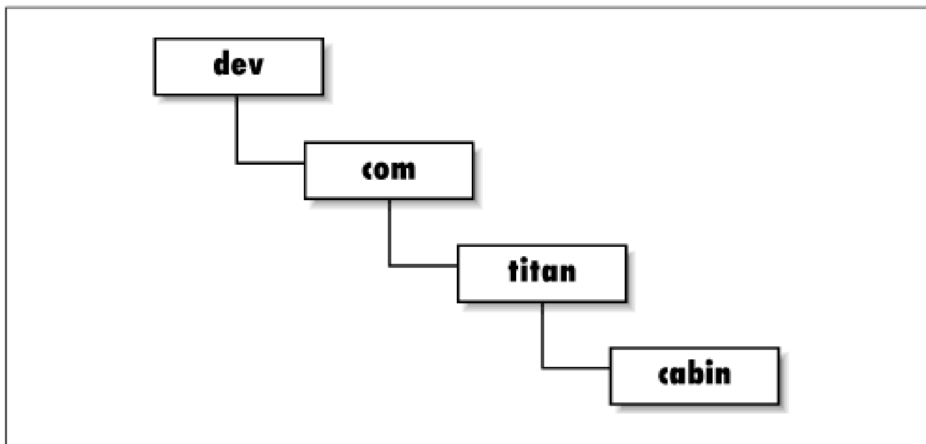
When developing an entity bean, we first want to define the bean's remote interface. The remote interface defines the bean's business purpose; the methods of this interface must capture the concept of the entity. We defined the remote interface for the Cabin bean in Chapter 2; here, we add two new methods for setting and getting the ship ID and the bed count. The ship ID identifies the ship that the cabin belongs to, and the bed count tells how many people the cabin can accommodate.

```
package com.titan.cabin;

import java.rmi.RemoteException;

public interface Cabin extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String str) throws RemoteException;
    public int getDeckLevel() throws RemoteException;
    public void setDeckLevel(int level) throws RemoteException;
    public int getShip() throws RemoteException;
    public void setShip(int sp) throws RemoteException;
    public int getBedCount() throws RemoteException;
    public void setBedCount(int bc) throws RemoteException;
}
```

The `Cabin` interface defines four properties: the `name`, `deckLevel`, `ship`, and `bedCount`. *Properties* are attributes of a bean that can be accessed by public set and get methods. The methods that access these properties are not explicitly defined in the `Cabin` interface, but the interface clearly specifies that these attributes are readable and changeable by a client.

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

**Figure 4-1. Directory structure for the Cabin bean**




## CabinHome: The Home Interface

Once we have defined the remote interface of the Cabin bean, we have defined the world's view of this simple entity bean. Next, we need to define the Cabin bean's home interface, which specifies how the bean can be created, located, and destroyed; in other words, the Cabin bean's life-cycle behavior. Here is a complete definition of the `CabinHome` home interface:

```
package com.titan.cabin;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CabinHome extends javax.ejb.EJBHome {

    public Cabin create(int id)
        throws CreateException, RemoteException;

    public Cabin findByPrimaryKey(CabinPK pk)
        throws FinderException, RemoteException;
}
```

The `CabinHome` interface extends the `javax.ejb.EJBHome` and defines two life- cycle methods: `create()` and `findByPrimaryKey()`. These methods create and locate Cabin

beans. Remove methods (for deleting beans) are defined in the `javax.ejb.EJBHome` interface, so the `CabinHome` interface inherits them. This interface is packaged in `com.titan.cabin`, just like the `Cabin` interface. It should be copied to your IDE and saved as *CabinHome.java* in the same directory as the *Cabin.java* file. If you attempt to compile the `CabinHome` interface, you will get an error stating that the `CabinPK` class could not be found. Next, we will create the `CabinPK` class to correct this problem.

## CabinPK: The Primary Key

The `CabinPK` is the primary key class of the Cabin bean. All entity beans must have a serializable primary key that can be used to uniquely identify an entity bean in the database. Here is the class definition of the `CabinPK` primary key:

```
package com.titan.cabin;

public class CabinPK implements java.io.Serializable {
    public int id;

    public int hashCode() {
        return id;
    }
    public boolean equals(Object obj) {
        if (obj instanceof CabinPK) {
            return (id == ((CabinPK)obj).id);
        }
        return false;
    }
}
```

The primary key belongs to the `com.titan.cabin` package and implements the `java.io.Serializable` interface. The `CabinPK` defines one public attribute, `id`. This `id` field is used to locate specific `Cabin` entities or records in the database at runtime. EJB 1.1 requires that we override the `Object.hashCode()` and `Object.equals()` methods; EJB 1.0 doesn't require this, but it's a good practice regardless of the version of EJB you are using. These methods ensure that the primary key evaluates properly when used with hash tables and in other situations.[2] Later, we will learn that the primary key must encapsulate attributes that match one or more container-managed fields in the bean class. In this case, the `id` field will have to match a field in the `CabinBean` class. Copy the `CabinPK` definition into your IDE, and save it to the cabin directory as *CabinPK.java* file. Compile it. Now that `CabinPK` has been compiled, you should be able to compile `CabinHome` without errors.

## CabinBean: The Bean Class

You have now defined the complete client-side API for creating, locating, removing, and using the Cabin bean. Now we need to define `CabinBean`, the class that provides the implementation on the server for the Cabin bean. The `CabinBean` class is an entity bean that uses container-managed persistence, so its definition will be fairly simple. In addition to the callback methods discussed in Chapters and , we must also define EJB implementations for most of the methods defined in the `Cabin` and `CabinHome` interfaces. Here is the complete definition of the `CabinBean` class in EJB 1.1:

```
// EJB 1.1 CabinBean
package com.titan.cabin;

import javax.ejb.EntityContext;

public class CabinBean implements javax.ejb.EntityBean {

    public int id;
    public String name;
    public int deckLevel;
    public int ship;
    public int bedCount;

    public CabinPK ejbCreate(int id) {
        this.id = id;
        return null;
    }
    public void ejbPostCreate(int id) {
        // Do nothing. Required.
    }
    public String getName() {
        return name;
    }
    public void setName(String str) {
        name = str;
    }
    public int getShip() {
        return ship;
    }
    public void setShip(int sp) {
        ship = sp;
    }
    public int getBedCount() {
        return bedCount;
    }
    public void setBedCount(int bc) {
        bedCount = bc;
    }
    public int getDeckLevel() {
        return deckLevel;
    }
    public void setDeckLevel(int level ) {
        deckLevel = level;
    }

    public void setEntityContext(EntityContext ctx) {
         // Not implemented.
    }
    public void unsetEntityContext() {
         // Not implemented.
    }
    public void ejbActivate() {
        // Not implemented.
    }
    public void ejbPassivate() {
        // Not implemented.
    }
    public void ejbLoad() {
        // Not implemented.
    }
    public void ejbStore() {
```

```
        // Not implemented.
    }
    public void ejbRemove() {
        // Not implemented.
    }
}
```

And here's the `CabinBean` class for EJB 1.0. It differs only in the return value of `ejbCreate()`:

```
// EJB 1.0 CabinBean
import javax.ejb.EntityContext;

public class CabinBean implements javax.ejb.EntityBean {

    public int id;
    public String name;
    public int deckLevel;
    public int ship;
    public int bedCount;

    public void ejbCreate(int id) {
        this.id = id;
    }
    public void ejbPostCreate(int id) {
        // Do nothing. Required.
    }
    public String getName() {
        return name;
    }
    public void setName(String str) {
        name = str;
    }
    public int getShip() {
        return ship;
    }
    public void setShip(int sp) {
        ship = sp;
    }
    public int getBedCount() {
        return bedCount;
    }
    public void setBedCount(int bc) {
        bedCount = bc;
    }
    public int getDeckLevel() {
        return deckLevel;
    }
    public void setDeckLevel(int level ) {
        deckLevel = level;
    }

    public void setEntityContext(EntityContext ctx) {
        // Not implemented.
    }
    public void unsetEntityContext() {
        // Not implemented.
    }
    public void ejbActivate(){
        // Not implemented.
```

```
    }
    public void ejbPassivate(){
        // Not implemented.
    }
    public void ejbLoad(){
        // Not implemented.
    }
    public void ejbStore(){
        // Not implemented.
    }
    public void ejbRemove(){
        // Not implemented.
    }
}
```

The `CabinBean` class belongs to the `com.titan.cabin` package, just like the interfaces and primary key class. The `CabinBean` class can be divided into four sections for discussion: declarations for the container-managed fields, the `ejbCreate()` methods, the callback methods, and the remote interface implementations.

Declared fields in a bean class can be persistent fields and property fields. These categories are not mutually exclusive. The persistent field declarations describe the fields that will be mapped to the database. A persistent field is often a property (in the JavaBeans sense): any attribute that is available using public set and get methods. Of course, a bean can have any fields that it needs; they need not all be persistent, or properties. Fields that aren't persistent won't be saved in the database. In `CabinBean`, all the fields are persistent.

The `id` field is persistent, but it is not a property. In other words, `id` is mapped to the database but cannot be accessed through the remote interface. The primary key, `CabinPK`, also contains an integer field called `id`, just like the `CabinBean`. This means that the primary key for the `CabinBean` is its `id` field because the signatures match.

The `name`, `deckLevel`, `ship`, and `bedCount` fields are persistent fields. They will be mapped to the database at deployment time. These fields are also properties because they are publicly available through the remote interface.

In the case of the Cabin bean, there was only one `create()` method, so there is only one corresponding `ejbCreate()` method and one `ejbPostCreate()` method, which is shown in both the EJB 1.1 and EJB 1.0 listings. When a client invokes a method on the home interface, it is delegated to a matching `ejbCreate()` method on the bean instance. The `ejbCreate()` method initializes the fields; in the case of the `CabinBean`, it sets the `id` field to equal the passed integer.

In the case of EJB 1.0, the `ejbCreate()` method returns `void` for container-managed persistence; this method returns the bean's primary key in bean-managed persistence. In EJB 1.1, the `ejbCreate()` method always returns the primary key type; with container-managed persistence, this method returns the `null` value. It's the container's responsibility to create the primary key. Why the change? Simply put, the change makes it easier for a bean-managed bean to extend a container-managed bean. In EJB 1.0, this is not possible because Java won't allow you to overload methods with different return values. Container-managed and bean-managed persistence was touched on in Chapter 3 and is

discussed in detail in Chapter 6.

Once the `ejbCreate()` method has executed, the `ejbPostCreate()` method is called to perform any follow-up operations. The `ejbCreate()` and `ejbPostCreate()` methods must have signatures that match the parameters and (optionally) the exceptions of the home interface's `create()` method. However, `ejbCreate()` and `ejbPostCreate()` aren't required to throw the `RemoteException` or `CreateException`. The EJB container throws these exceptions automatically at runtime if there is a problem with communications or some other system-level problem.

The `findByPrimaryKey()` method is not defined in container-managed bean classes. With container-managed beans you do not explicitly declare find methods in the bean class. Instead, find methods are generated at deployment and implemented by the container. With bean-managed beans (beans that explicitly manage their own persistence), find methods must be defined in the bean class. Our Cabin bean is a container-managed bean, so we will not need to define its find method. In Chapter 6, when you develop bean-managed entity beans, you will define the find methods in the bean classes you develop.

The business methods in the `CabinBean` match the signatures of the business methods defined in the remote interface. These include `getName()`, `setName()`, `getDeckLevel()`, `setDeckLevel()`, `getShip()`, `setShip()`, `getBedCount()`, and `setBedCount()`. When a client invokes one of these methods on the remote interface, the method is delegated to the matching method on the bean class. Again, the business methods do not throw the `RemoteException` like the matching methods in the remote interface. In both the `ejbCreate()` and remote interface methods, it is possible to define application or custom exceptions. If a custom exception is defined, both the interface method and its matching method in the bean class must throw it. We will learn more about custom exceptions in Chapter 6.

The entity context methods are responsible for setting and unsetting the `EntityContext`. The `EntityContext` is an interface implemented by the EJB container that provides the bean with information about the container, the identity of the client, transactional control, and other environmental information if the bean needs it. Because the Cabin bean is a very simple container-managed bean, this example does not use the `EntityContext`. Subsequent examples in Chapter 6 will make good use of the `EntityContext`.

The `CabinBean` class implements `javax.ejb.EntityBean`, which defines five callback methods: `ejbActivate()`, `ejbPassivate()`, `ejbLoad()`, `ejbStore()`, and `ejbRemove()`. The container uses these callback methods to notify the `CabinBean` of certain events in its life cycle. Although the callback methods are implemented, the implementations are empty. The `CabinBean` is simple enough that it doesn't need to do any special processing during its life cycle. When we study entity beans in more detail in Chapter 6, we will take advantage of these callback methods.

That's enough talk about the `CabinBean` definition. Now that you are familiar with it, copy it to your IDE, save it to the cabin directory as *CabinBean.java*, and compile it.

You are now ready to create a deployment descriptor for the `Cabin` bean. The deployment

descriptor performs a function similar to a properties file. It describes which classes make up a bean and how the bean should be managed at runtime. During deployment, the deployment descriptor is read and its properties are displayed for editing. The deployer can then modify and add settings as appropriate for the application's operational environment. Once the deployer is satisfied with the deployment information, he or she uses it to generate the entire supporting infrastructure needed to deploy the bean in the EJB server. This may include adding the bean to the naming system and generating the bean's EJB object and EJB home, persistence infrastructure, transactional support, resolving bean references, and so forth.

Although most EJB server products provide a wizard for creating and editing deployment descriptors, we will create ours directly so that the bean is defined in a vendor-independent manner. This requires some manual labor, but it gives you a much better understanding of how deployment descriptors are created. Once the deployment descriptor is finished, the bean can be placed in a JAR file and deployed on any EJB-compliant server of the appropriate version.

## EJB 1.1: The Deployment Descriptor

An XML deployment descriptor for every example in this book has already been created and is available from the download site. If you haven't downloaded the examples, do so now. The examples are packaged in a ZIP file and organized by chapter and bean, so you will need to put the *ejb-jar.xml* file from the directory *chapter4/EJB11/com/titan/cabin* in the ZIP file. When you create the JAR file to deploy the Cabin bean, this *ejb-jar.xml* file *must* be in the JAR as *META-INF/ejb-jar.xml* in order for it to be found. If it has any other name or any other location, this deployment descriptor will not be used.

Here's a quick peek at the deployment descriptor for the Cabin bean, so you can get a feel for how an XML deployment descriptor is structured and the type of information it contains:

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

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

<ejb-jar>
 <enterprise-beans>
   <entity>
      <description>
          This Cabin enterprise bean entity represents a cabin on
          a cruise ship.
      </description>
      <ejb-name>CabinBean</ejb-name>
      <home>com.titan.cabin.CabinHome</home>
      <remote>com.titan.cabin.Cabin</remote>
      <ejb-class>com.titan.cabin.CabinBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>com.titan.cabin.CabinPK</prim-key-class>
      <reentrant>False</reentrant>

      <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>deckLevel</field-name></cmp-field>
      <cmp-field><field-name>ship</field-name></cmp-field>
```

```
      <cmp-field><field-name>bedCount</field-name></cmp-field>
   </entity>
 </enterprise-beans>

 <assembly-descriptor>
   <security-role>
      <description>
          This role represents everyone who is allowed full access
          to the cabin bean.
      </description>
     <role-name>everyone</role-name>
   </security-role>

   <method-permission>
     <role-name>everyone</role-name>
     <method>
         <ejb-name>CabinBean</ejb-name>
         <method-name>*</method-name>
     </method>
   </method-permission>

   <container-transaction>
     <method>
        <ejb-name>CabinBean</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

The `<!DOCTYPE>` element describes the purpose of the XML file, its root element, and the location of its DTD. The DTD is used to verify that the document is structured correctly. This element is discussed in detail in Chapter 10 and is not important to understanding this example.

The rest of the elements are nested one within the other and are delimited by a beginning tag and ending tag. The structure is really not very complicated. If you have done any HTML coding you should already understand the format. An element always starts with <*name of tag* > tag and ends with </*name of tag* > tag. Everything in between--even other elements--is part of the enclosing element.

The first major element is the `<ejb-jar>` element, which is the root of the document. All the other elements must lie within this element. Next is the `<enterprise-beans>` element. Every bean declared in an XML file must be included in this section. This file only describes the Cabin bean, but we could define several beans in one deployment descriptor.

The `<entity>` element shows that the beans defined within this tag are entity beans. Similarly, a `<session>` element describes session beans; since the Cabin bean is an entity bean, we don't need a `<session>` element. In addition to a description, the `<entity>` element provides the fully qualified class names of the remote interface, home interface, bean class, and primary key. The `<cmp-field>` elements list all the container-managed fields in the entity bean class. These are the fields that will be persisted in the database and are managed by the container at runtime. The `<entity>` element also includes a `<reentrant>` element that can be set as `True` or `False` depending on whether the bean allows reentrant

loopbacks or not.

The next section of the XML file, after the `<enterprise-bean>` element, is enclosed by the `<assembly-descriptor>` element, which describes the security roles and transactional attributes of the bean. It may seem odd to separate this information from the `<enterprise-beans>` element, since it clearly applies to the Cabin bean, but in the scheme of things it's perfectly natural. An XML deployment descriptor can describe several beans, which might all rely on the same security roles and transactional attributes. To make it easier to deploy several beans together, all this common information is separated into the `<assembly-descriptor>` element.

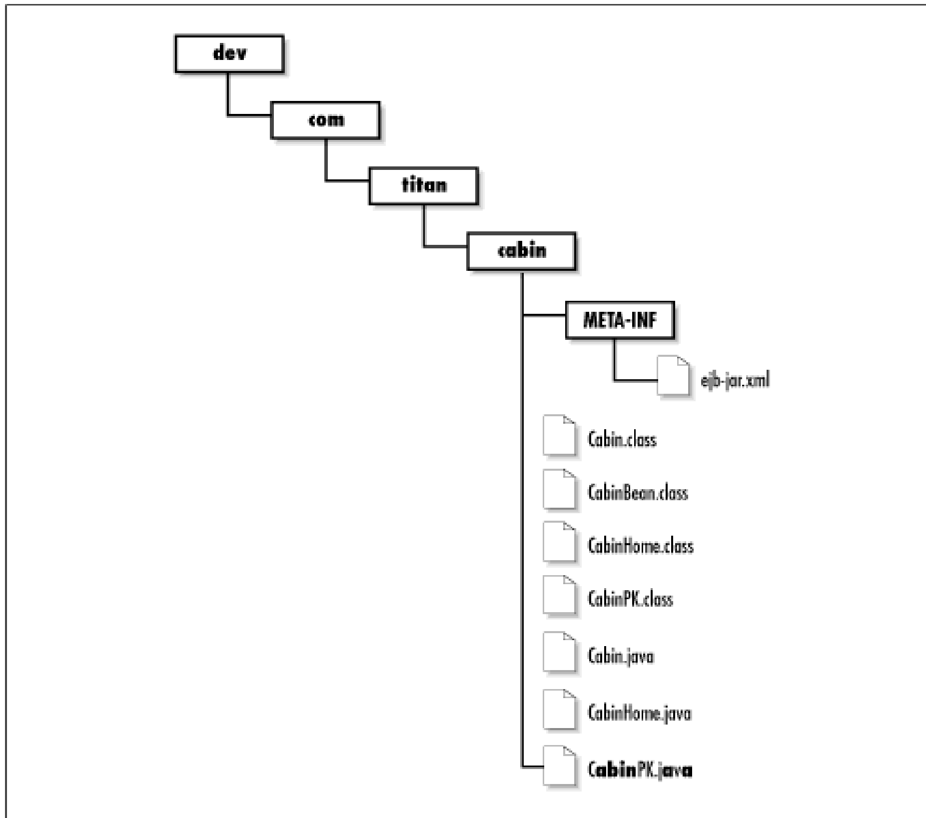
There is another reason (perhaps a more important reason) for separating information about the bean itself from the security roles and transactional attributes. The EJB 1.1 specification clearly defines the responsibilities of different participants in the development and deployment of beans. We don't address these development roles in this book because they are not critical to learning the fundamentals of EJB. For now, it's enough to know that the person who develops the bean and the person who assembles the beans into an application have separate responsibilities and therefore separate parts of the XML deployment descriptor. The bean developer is responsible for everything within the `<enterprise-beans>` element; the bean assembler is responsible for everything within the `<assembly-descriptor>`. In our example, we're playing both roles, developing the beans and assembling them. But in real life, you might buy a set of beans developed by a third-party vendor, who would have no idea how you intend to use the beans, what your security requirements are, etc.

The `<assembly-descriptor>` contains the `<security-role>` elements and their corresponding `<method-permission>` elements, which were described in Chapter 3 under "Security." In this example there is one security role, `everyone`, which is mapped to all the methods in the Cabin bean using the `<method-permission>` element. (The `*` in the `<method-name>` element means "all methods").

The container-transaction element declares that all the methods of the Cabin bean have a `Required` transactional attribute. Transactional attributes are explained in more detail in Chapter 8, but for now it means that all the methods must be executed within a transaction. The deployment descriptor ends with the enclosing tab of the `<ejb-jar>` element.

Copy the Cabin bean's deployment descriptor into the same directory as the class files for the Cabin bean files (*Cabin.class*, *CabinHome.class*, *CabinBean.class*, and *CabinPK.class*) and save it as *ejb-jar.xml*. You have now created all the files you need to package your EJB 1.1 Cabin bean. Figure 4-2 shows all the files that should be in the *cabin* directory.

**Figure 4-2. The Cabin bean files (EJB 1.1)**




## EJB 1.0: The Deployment Descriptor

Here is a Java application that instantiates, populates, and serializes a `DeploymentDescriptor` for the EJB 1.0 Cabin bean:

```
package com.titan.cabin;

import javax.ejb.deployment.EntityDescriptor;
import javax.ejb.deployment.ControlDescriptor;
import javax.naming.CompoundName;
import com.titan.cabin.CabinBean;
import java.util.Properties;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;

public class MakeDD {

    public static void main(String [] args) {
        try {
          if (args.length <1){
            System.out.println("must specify target directory");
            return;
          }

        EntityDescriptor cabinDD = new EntityDescriptor();

        cabinDD.setEnterpriseBeanClassName("com.titan.cabin.CabinBean");
```

```
        cabinDD.setHomeInterfaceClassName("com.titan.cabin.CabinHome");
        cabinDD.setRemoteInterfaceClassName("com.titan.cabin.Cabin");
        cabinDD.setPrimaryKeyClassName("com.titan.cabin.CabinPK");

        Class beanClass = CabinBean.class;
        Field [] persistentFields = new Field[5];
        persistentFields[0] = beanClass.getDeclaredField("id");
        persistentFields[1] = beanClass.getDeclaredField("name");
        persistentFields[2] = beanClass.getDeclaredField("deckLevel");
        persistentFields[3] = beanClass.getDeclaredField("ship");
        persistentFields[4] = beanClass.getDeclaredField("bedCount");

        cabinDD.setContainerManagedFields(persistentFields);

        cabinDD.setReentrant(false);

        CompoundName jndiName = new CompoundName("CabinHome",
                                                 new Properties());
        cabinDD.setBeanHomeName(jndiName);

        ControlDescriptor cd = new ControlDescriptor();

        cd.setIsolationLevel(ControlDescriptor.TRANSACTION_READ_COMMITTED);
        cd.setTransactionAttribute(ControlDescriptor.TX_REQUIRED);

        cd.setRunAsMode(ControlDescriptor.CLIENT_IDENTITY);

        cd.setMethod(null);
        ControlDescriptor [] cdArray = {cd};
        cabinDD.setControlDescriptors(cdArray);

        String fileSeparator =
            System.getProperties().getProperty("file.separator");
        if (! args[0].endsWith(fileSeparator))
            args[0] += fileSeparator;

        FileOutputStream fos = new FileOutputStream(args[0]+"CabinDD.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(cabinDD);
        oos.flush();
        oos.close();
        fos.close();

        } catch (Throwable t) { t.printStackTrace();}
    }
}
```

Copy this definition into your IDE, save it in the *cabin* directory, and compile it. When you run the application, `MakeDD`, use the path to the *cabin* directory, where all the other Cabin bean files are stored, as a command-line parameter:

```
\dev % java com.titan.cabin.MakeDD com/titan/cabin

F:\..\dev>java com.titan.cabin.MakeDD com\titan\cabin
```

If you run this application, you should end up with a file called *CabinDD.ser* in the *com/titan/cabin* directory. This is your serialized `DeploymentDescriptor` for the Cabin bean. Now that you know that the application works properly, let's look at it in more detail. We begin with the creation of the `EntityDescriptor`:

```
EntityDescriptor cabinDD = new EntityDescriptor();
```

An entity descriptor is a `DeploymentDescriptor` that has been extended to support entity beans. If we were creating a `DeploymentDescriptor` for a session bean, we would use the `SessionDescriptor` subclass. Notice that we are not extending `EntityDescriptor` to create a special cabin `DeploymentDescriptor`. We are using the `EntityDescriptor` class provided by the EJB package `javax.ejb.deployment`.

The `EntityDescriptor` describes the classes and interfaces used by the Cabin bean. The next section of code sets the names of the bean class, the home interface, the remote interface, and the primary key. All of these set methods are defined in the `EntityDescriptor`'s superclass, `DeploymentDescriptor`, except for `setPrimaryKeyClassName()`; this method is defined in the `EntityDescriptor` class.

```
cabinDD.setEnterpriseBeanClassName("com.titan.cabin.CabinBean");
cabinDD.setHomeInterfaceClassName("com.titan.cabin.CabinHome");
cabinDD.setRemoteInterfaceClassName("com.titan.cabin.Cabin");
cabinDD.setPrimaryKeyClassName("com.titan.cabin.CabinPK");
```

When the bean is deployed, the deployment tools will read these properties so that the tools can locate the bean interfaces and primary key class and generate all the supporting code, such as the EJB object and EJB home.

The next section is a little more complicated. Our Cabin bean is going to be a container-managed entity bean, which means that the container will automatically handle persistence. To handle persistence, the container must know which of the `CabinBean`'s fields it is responsible for. Earlier, it was decided that the `id`, `name`, `deckLevel`, `ship`, and `bedCount` fields were all persistent fields in the `CabinBean`. The following code tells the `EntityDescriptor` that these fields are container managed by using the Reflection API to pass an array of `Field` objects to `setContainerManagedFields()`:

```
Class beanClass = CabinBean.class;
Field [] persistentFields = new Field[5];
persistentFields[0] = beanClass.getDeclaredField("id");
persistentFields[1] = beanClass.getDeclaredField("name");
persistentFields[2] = beanClass.getDeclaredField("deckLevel");
persistentFields[3] = beanClass.getDeclaredField("ship");
persistentFields[4] = beanClass.getDeclaredField("bedCount");

cabinDD.setContainerManagedFields(persistentFields);
```

Although the code tells the `EntityDescriptor` which fields are container- managed, it doesn't describe how these fields will map to the database. The actual mapping of the fields to the database depends on the type of database and on the EJB server used. The mapping is vendor- and database-dependent, so we won't worry about it just now. When the bean is actually deployed in some EJB server, the deployer will map the container-managed fields to whatever database is used.

The next line tells the `EntityDescriptor` that the Cabin bean is nonreentrant. We discussed the problems associated with reentrant beans in Chapter 3. Entity beans are not reentrant by default, but it never hurts to make this explicit.

```
cabinDD.setReentrant(false);
```

The following code uses the JNDI API to set the lookup name of the bean in the EJB server's directory structure. In Chapter 3, we saw that Enterprise JavaBeans requires servers to support the use of JNDI for organizing beans in a directory structure. Later, when we create a client application, the name we assign to the Cabin bean will be used to locate and obtain a remote reference to the bean's EJB home.

```
CompoundName jndiName = new CompoundName("CabinHome", new Properties());
cabinDD.setBeanHomeName(jndiName);
```

We have created a directory entry that places the bean under the name *CabinHome*. Although it makes sense to assign names that reflect the organization of your beans, you can give the EJB home any lookup name you like. We could have used other names assigned to the Cabin bean, like *HomeCabin* or just *cabin*.

Next, we create a `ControlDescriptor` to set the bean's transactional and security attributes:

```
ControlDescriptor cd = new ControlDescriptor();

cd.setIsolationLevel(ControlDescriptor.TRANSACTION_READ_COMMITTED);
cd.setTransactionAttribute(ControlDescriptor.TX_REQUIRED);

cd.setRunAsMode(ControlDescriptor.CLIENT_IDENTITY);

cd.setMethod(null);
ControlDescriptor [] cdArray = {cd};
cabinDD.setControlDescriptors(cdArray);
```

After creating the `ControlDescriptor`, we set its transactional attributes. This includes setting the transactional context and isolation level. Transactions are fairly complicated and are discussed in detail in Chapter 8. Essentially, we are saying that the bean must be executed in a transaction and that the bean is not accessible by any other client while executing a transaction. Next, we set the runAs mode of the bean. The runAs mode determines how the bean's methods will execute at runtime. In this case, the methods will be executed under the identity that invoked the bean. This means that any other beans or resources accessed by the method will be validated based on the client's identity. Then we set the methods that the `ControlDescriptor` represents and add the `ControlDescriptor` to the `EntityDescriptor`. Here we set the method to `null`, which means that the `ControlDescriptor` is the default for all methods of the Cabin bean. Any method that doesn't have its own `ControlDescriptor` uses the default `ControlDescriptor` for the bean. In this case, we only specify a default descriptor. Once all the properties on the `ControlDescriptor` have been set, it is added to the `EntityDescriptor`.

Finally, we serialize the `EntityDescriptor` with all its Cabin bean properties to a file called *CabinDD.ser*. This serialized `EntityDescriptor` should be saved to the same directory that holds all the other files for the Cabin bean, the *dev/com/titan/cabin* directory.

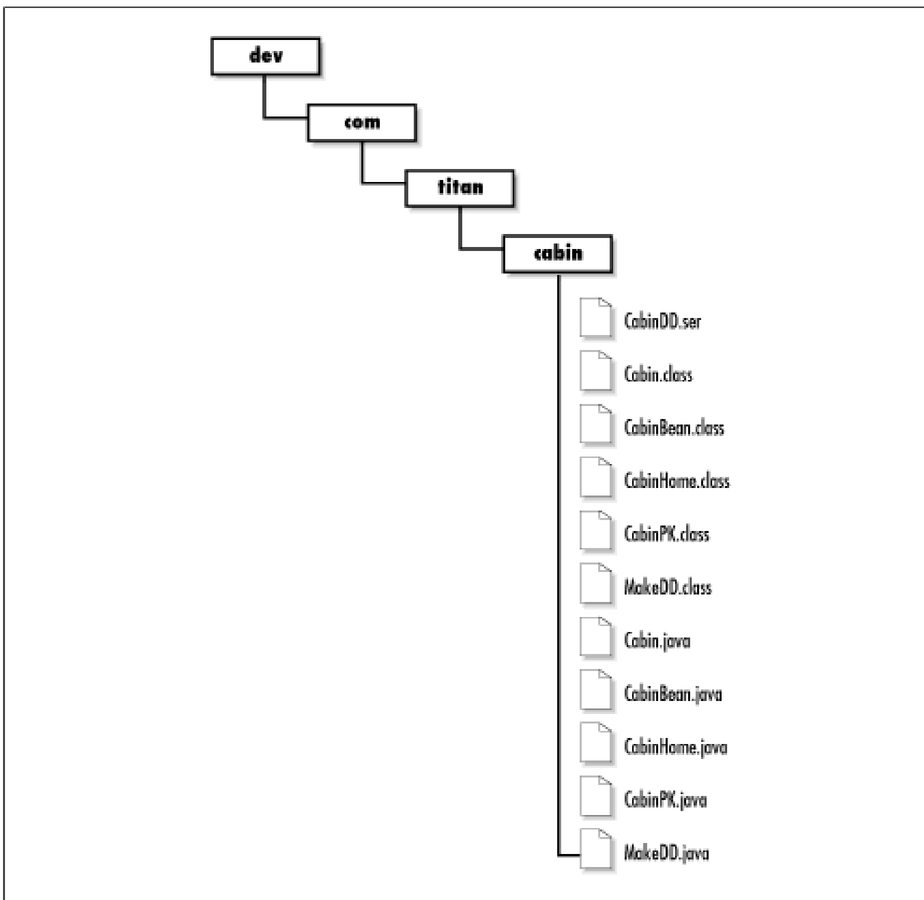
```
String fileSeparator = System.getProperties().getProperty("file.separator");
if (! args[0].endsWith(fileSeparator))
    args[0] += fileSeparator;
```

```
FileOutputStream fos = new FileOutputStream(args[0]+"CabinDD.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(cabinDD);
oos.flush();
oos.close();
fos.close();
```

The first part of the serialization section simply determines whether the path ends with a file separator for that operating system. If it doesn't, the code adds one. The second part serializes the deployment descriptor to a file called *CabinDD.ser* in the directory passed in at the command line.

You have now created everything you need to package your EJB 1.0 Cabin bean for deployment. Figure 4-3 shows all the files that should be in the *cabin* directory.

**Figure 4-3. The Cabin bean files (EJB 1.0)**




## cabin.jar: The JAR File

The JAR file is a platform-independent file format for compressing, packaging, and delivering several files together. Based on ZIP file format and the ZLIB compression standards, the JAR ( Java archive) packages and tool were 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. The original JavaBeans component architecture depends on JAR files for packaging, as does Enterprise JavaBeans. The goal in using the JAR file format in EJB is to package all the classes and interfaces associated with a bean, including the deployment descriptor into one file. The process of creating an EJB JAR file is slightly different between EJB 1.1 and EJB 1.0.

### EJB 1.1: Packaging the Cabin bean

Now that you have put all the necessary files in one directory, creating the JAR file is easy. Position yourself in the *dev* directory that is just above the *com/titan/cabin* directory tree, and execute the command:

```
\dev % jar cf cabin.jar com/titan/cabin/*.class META-INF/ejb-jar.xml
```

```
F:\..\dev>jar cf cabin.jar com\titan\cabin\*.class META-INF\ejb-jar.xml
```

You might have to create the META-INF directory first and copy *ejb-jar.xml* into that directory. 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 (*cabin.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* by David Flanagan, or *Learning Java?* (formerly *Exploring Java?*), by Pat Niemeyer and Jonathan Knudsen (both published by O'Reilly).

The *jar* utility creates the file *cabin.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 you can use the command *jar tvf cabin.jar*.

### EJB 1.0: Packaging the Cabin bean

In addition to the bean's classes and deployment descriptor, the JAR file contains a *manifest* generated by the *jar* utility. The manifest essentially serves as a README file, describing the contents in a way that's useful for any tools that work with the archive. We need to add an entry into the JAR file to specify which file contains our serialized deployment descriptor. To do this, we add two simple lines to the manifest by creating an ASCII text file named *manifest* :

```
Name: com/titan/cabin/CabinDD.ser
Enterprise-Bean: True
```

That's it! When we run the *jar* utility, we will tell it to use our manifest information when it build the JAR. The manifest for this bean is now complete. A manifest is always organized as a set of name-value pairs that describe the files in the JAR file. In this case, we need to point to the location of the serialized deployment descriptor and define the JAR as an EJB JAR. The first line points to the serialized `EntityDescriptor`, *CabinDD.ser*, for the Cabin bean. Notice that forward slashes ("/") must be used as path separators; this could be confusing if you are used to the Windows environment. The next line of the manifest

identifies the JAR as an EJB JAR. Most EJB server deployment tools check for this name-value pair before trying to read the JAR file's contents. Save the manifest in the *cabin* directory where all the other Cabin bean files are located. It should be saved as the file name *manifest* with no extension.

Now that you have put all the necessary files in one directory, creating the JAR file is easy. Position yourself in the *dev* directory that is just above the *com/titan/cabin* directory tree, and execute the following command:

```
\dev % jar cmf com/titan/cabin/manifest cabin.jar com/titan/cabin/*.class \
com/titan/cabin/*.ser

F:\..\dev>jar cmf com\titan\cabin\manifest cabin.jar com\titan\cabin\*.class
com\titan\cabin\*.ser
```

If you want, you may remove the *MakeDD.class* file from the JAR archive, since it's not a standard EJB class and the EJB deployment tools do not use it. Leaving it there will not impact deployment of the Cabin bean.

## 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 bean, we must map its `id`, `name`, `deckLevel`, `ship`, and `bedCount` (the bean's container-managed 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,
  SHIP_ID int,
  BED_COUNT int,
  NAME char(30),
  DECK_LEVEL int
)
```

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

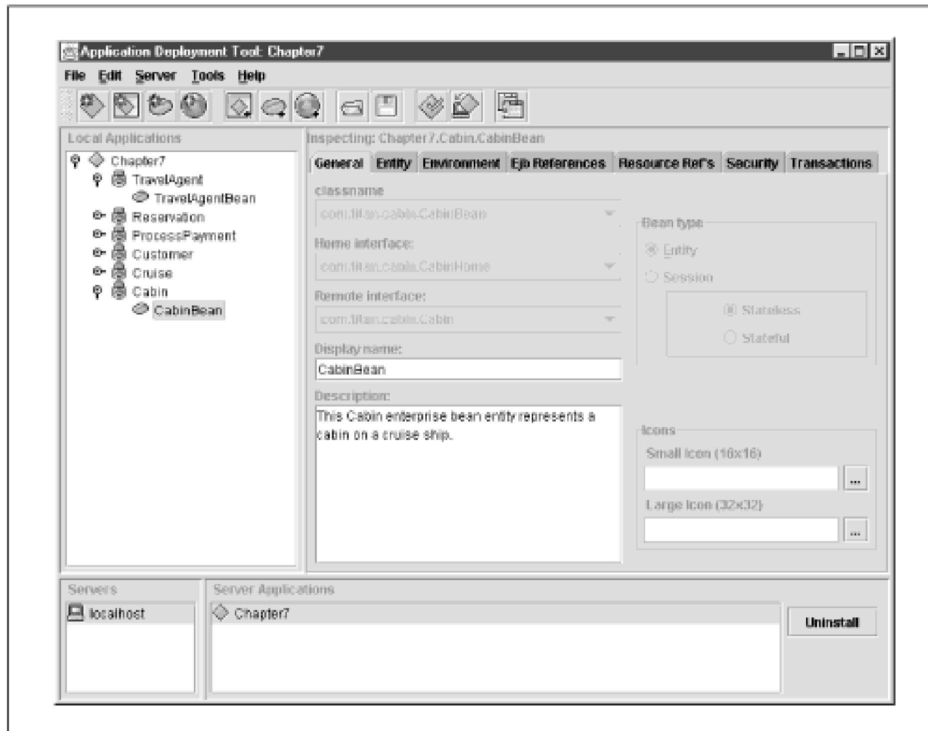
## Deploying the Cabin Bean

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 vendor-specific classes needed to support the bean in the EJB environment. Every EJB server product has its own deployment tools, which may provide a graphical user interface, a set of command-line programs, or both. Graphical deployment "wizards" are the easiest deployment tools to work with.

### EJB 1.1 deployment tools

A deployment tool reads the JAR file and looks for the *ejb-jar.xml* file. In a graphical deployment wizard, the deployment descriptor elements will be presented in a set of property sheets similar to those used to customize visual components in environments like Visual Basic, PowerBuilder, JBuilder, and Symantec Café. Figure 4-4 shows the deployment wizard used in the J2EE Reference Implementation.

**Figure 4-4. J2EE Reference Implementation's deployment wizard**

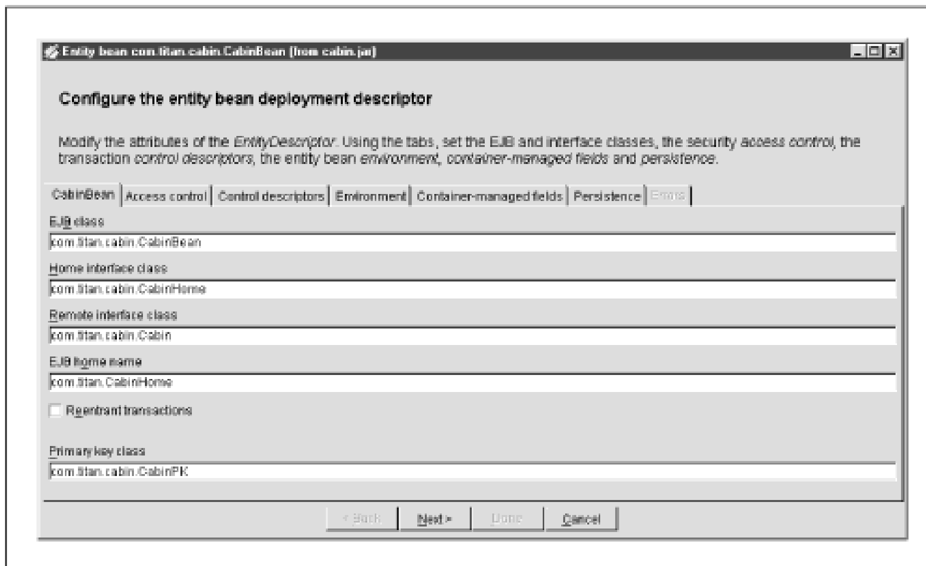



The J2EE Reference Implementation's deployment wizard has fields and panels that match the XML deployment descriptor. You can map security roles to users groups, set the JNDI look up name, map the container-managed fields to the database, etc.

### EJB 1.0 deployment tools

A deployment tool reads the JAR file and uses the manifest to locate the bean's serialized deployment descriptor. Once the deployment descriptor file has been located, it is deserialized into an object and its properties are read by invoking its get methods. In a graphical deployment wizard, these properties will be presented to the deployer in a set of property sheets, similar to those used to customize visual components in environments like Visual Basic, PowerBuilder, JBuilder, and Symantec Cafe. Figure 4-5 shows the deployment wizard used in BEA's WebLogic EJB server.

**Figure 4-5. WebLogic deployment wizard**




The WebLogic deployment wizard has fields and panels that match properties and deployment classes specified in the `javax.ejb.deployment` package. The "CabinBean" tab, for example, contains text fields for each of the interfaces and classes that we described in the Cabin bean's `EntityDescriptor`, the *CabinDD.ser*. There is also an "Access control" tab that corresponds to the `AccessControlEntry` class, and a "Control descriptors" tab that corresponds to the `ControlDescriptor` class. In addition, there is a "Container-managed fields" tab that shows the container-managed fields we defined when creating the *CabinDD.ser*. Graphical deployment wizards provided by other EJB products will look different but provide the same kinds of features.

At this point, you can choose to change deployment information, such as the transactional isolation level, to change the Cabin bean's JNDI name, or to deselect one of the container-managed fields. You can also add properties to the deployment descriptor, for example, by setting the `AccessControlEntry`s for the methods and adding environment properties. The *CabinDD.ser* that we created should have specified the minimum information that most EJB servers need to deploy the bean without changes. It is likely that all you will need to do is specify the persistence mapping from the container-managed fields to the `CABIN` table in the relational database.

Different EJB deployment tools will provide varying degrees of support for mapping container-managed fields to a data source. Some provide very robust and sophisticated graphical user interfaces, while others are simpler and less flexible. Fortunately, mapping the `CabinBean`'s container-managed fields to the `CABIN` table is a fairly straightforward process. Read the documentation for the deployment tool provided by your EJB vendor to determine how to do this mapping. Once you have finished the mapping, you can complete the deployment of the bean and prepare to access it from the EJB server.

## Creating a Client Application

Now that the Cabin bean has been deployed in the EJB server, we want to access it from a

remote client. When we say remote, we are not necessarily talking about a client application that is located on a different computer, just one that is not part of the EJB server. In this section, we will create a remote client that will connect to the EJB server, locate the EJB home for the Cabin bean, and create and interact with several Cabin beans. The following code shows a Java application that is designed to create a new Cabin bean, set its `name`, `deckLevel`, `ship`, and `bedCount` properties, and then locate it again using its primary key:

```
package com.titan.cabin;

import com.titan.cabin.CabinHome;
import com.titan.cabin.Cabin;
import com.titan.cabin.CabinPK;

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

public class Client_1 {
    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref =
                jndiContext.lookup("CabinHome");
                CabinHome home = (CabinHome)
            // EJB 1.0:Use Java cast instead of narrow( )
                PortableRemoteObject.narrow(ref,CabinHome.class);
            Cabin cabin_1 = home.create(1);
            cabin_1.setName("Master Suite");
            cabin_1.setDeckLevel(1);
            cabin_1.setShip(1);
            cabin_1.setBedCount(3);

            CabinPK pk = new CabinPK();
            pk.id = 1;

            Cabin cabin_2 = home.findByPrimaryKey(pk);
            System.out.println(cabin_2.getName());
            System.out.println(cabin_2.getDeckLevel());
            System.out.println(cabin_2.getShip());
            System.out.println(cabin_2.getBedCount());

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

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

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

To access an enterprise bean, a client starts by using the JNDI package 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 directory services that are 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 method `getInitialContext()` contains logic that uses JNDI to obtain a network connection to the EJB server.

The code used to obtain the JNDI `Context` will be different depending on which EJB vendor you are using. You will need to research your EJB vendor's requirements for obtaining a JNDI `Context` appropriate to that product.

The code used to obtain a JNDI `Context` in Gemstone/J, for example, might look something like the following:

```
public static Context getInitialContext() throws javax.naming.NamingException {
   Properties p = new Properties();
   p.put(com.gemstone.naming.Defaults.NAME_SERVICE_HOST,"localhost");
   String port = System.getProperty("com.gemstone.naming.NameServicePort",
                                    "10200");
   p.put(com.gemstone.naming.Defaults.NAME_SERVICE_PORT, port);
   p.put(Context.INITIAL_CONTEXT_FACTORY,"com.gemstone.naming.GsCtxFactory");
   return new InitialContext(p);
}
```

The same method developed for BEA's WebLogic Server would be different:

```
public static Context getInitialContext()
   throws javax.naming.NamingException {
   Properties p = new Properties();
   p.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.TengahInitialContextFactory");
   p.put(Context.PROVIDER_URL, "t3://localhost:7001");
   return new javax.naming.InitialContext(p);
}
```

Once a JNDI connection is established and a context is obtained from the `getIntialContext()` method, the context can be used to look up the EJB home of the Cabin bean:

### EJB 1.1: Obtaining a remote reference to the home interface

The previous example uses the `PortableRemoteObject.narrow()` method as prescribed in EJB 1.1:

```
Object ref = jndiContext.lookup("CabinHome");
CabinHome home = (CabinHome)
// EJB 1.0: Use Java cast instead of narrow()
    PortableRemoteObject.narrow(ref,CabinHome.class);
```

The `PortableRemoteObject.narrow()` method is new to EJB 1.1. 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 `CabinHome`, we must explicitly narrow the object returned from `lookup()`. This has the same effect as casting and is explained in more detail in Chapter 5.

The name used to find the Cabin bean's EJB home is set by the deployer using a deployment wizard like the one pictured earlier. 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.

### EJB 1.0: Obtaining a remote reference to the home interface

In EJB 1.0, you do not need to use the `PortableRemoteObject.narrow()` method to cast objects to the correct type. EJB 1.0 allows the use of Java native casting to narrow the type returned by the JNDI API to the home interface type. When you see the `PortableRemoteObject` being used, replace it with Java native casting as follows:

```
CabinHome home = (CabinHome)jndiContext.lookup("CabinHome");
```

To locate the EJB home, we specify the name that we set using the `DeploymentDescriptor.setBeanHomeName(String name)` method in the `MakeDD` application earlier. If this lookup succeeds, the `home` variable will contain a remote reference to the Cabin bean's EJB home.

### Creating a new Cabin bean

Once we have a remote reference to the EJB home, we can use it to create a new `Cabin` entity:

```
Cabin cabin_1 = home.create(1);
```

We create a new `Cabin` entity using the `create(int id)` method defined in the home interface of the Cabin bean. When this method is invoked, the EJB home works with the EJB server to create a Cabin bean, adding its data to the database. The EJB server then creates an EJB object to wrap the Cabin bean instance and returns a remote reference to the EJB object to the client. The `cabin_1` variable then contains a remote reference to the Cabin bean we just created.

> **NOTE:** In EJB 1.1, we don't need to use the `PortableRemoteObject.narrow()` method to get the EJB object from the home reference, because it was declared as returning the `Cabin` type; no casting was required. We don't need to explicitly narrow remote references returned by `findByPrimaryKey()` for the same reason.

With the remote reference to the EJB object, we can update the `name`, `deckLevel`, `ship`, and `bedCount` of the `Cabin` entity:

```
Cabin cabin_1 = home.create(1);
cabin_1.setName("Master Suite");
cabin_1.setDeckLevel(1);
cabin_1.setShip(1);
cabin_1.setBedCount(3);
```

Figure 4-6 shows how the relational database table that we created should look after executing this code. It should contain one record.

**Figure 4-6. CABIN table with one cabin record**

CABIN TABLE

| ID | NAME | SHIP_ID | BED_COUNT | DECK_LEVEL |
|---|---|---|---|---|
| 1 | Master Suite | 1 | 3 | 1 |
| | | | | |
| | | | | |
| | | | | |

After an entity bean has been created, a client can locate it using the `findByPrimaryKey()` method in the home interface. First, we create a primary key of the correct type, in this case `CabinPK`, and set its field `id` to equal the `id` of the cabin we want. (So far, we only have one cabin available to us.) When we invoke this method on the home interface, we get back a remote reference to the EJB object. We can now interrogate the remote reference returned by `findByPrimaryKey()` to get the `Cabin` entity's `name`, `deckLevel`, `ship`, and `bedCount`:

```
CabinPK pk = new CabinPK();
pk.id = 1;

Cabin cabin_2 = home.findByPrimaryKey(pk);
System.out.println(cabin_2.getName());
System.out.println(cabin_2.getDeckLevel());
System.out.println(cabin_2.getShip());
System.out.println(cabin_2.getBedCount());
```

Copy and save the `Client_1` application to any directory, and compile it. If you haven't started your EJB server and deployed the Cabin bean, do so now. When you're finished, you're ready to run the `Client_1` in your IDE's debugger so that you can watch each step of the program. Your output should look something like the following:

```
Master Suite
1
1
3
```

You just created and used your first entity bean! Of course, the client application doesn't do much. Before going on to create session beans, create another client that adds some test data to the database. Here we'll create `Client_2` as a modification of `Client_1` that populates the database with a large number of cabins for three different ships:

```
package com.titan.cabin;

import com.titan.cabin.CabinHome;
import com.titan.cabin.Cabin;
import com.titan.cabin.CabinPK;
```

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

public class Client_2 {

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

            Object ref =
                jndiContext.lookup("CabinHome");
            CabinHome home = (CabinHome)
            // EJB 1.0: Use Java native cast
                PortableRemoteObject.narrow(ref,CabinHome.class);
            // Add 9 cabins to deck 1 of ship 1.
            makeCabins(home, 2, 10, 1, 1);
            // Add 10 cabins to deck 2 of ship 1.
            makeCabins(home, 11, 20, 2, 1);
            // Add 10 cabins to deck 3 of ship 1.
            makeCabins(home, 21, 30, 3, 1);

            // Add 10 cabins to deck 1 of ship 2.
            makeCabins(home, 31, 40, 1, 2);
            // Add 10 cabins to deck 2 of ship 2.
            makeCabins(home, 41, 50, 2, 2);
            // Add 10 cabins to deck 3 of ship 2.
            makeCabins(home, 51, 60, 3, 2);

            // Add 10 cabins to deck 1 of ship 3.
            makeCabins(home, 61, 70, 1, 3);
            // Add 10 cabins to deck 2 of ship 3.
            makeCabins(home, 71, 80, 2, 3);
            // Add 10 cabins to deck 3 of ship 3.
            makeCabins(home, 81, 90, 3, 3);
            // Add 10 cabins to deck 4 of ship 3.
            makeCabins(home, 91, 100, 4, 3);

            for (int i = 1; i <= 100; i++){
                CabinPK pk = new CabinPK();
                pk.id = i;
                Cabin cabin = home.findByPrimaryKey(pk);
                System.out.println("PK = "+i+", Ship = "+cabin.getShip()
                  + ", Deck = "+cabin.getDeckLevel()
                  + ", BedCount = "+cabin.getBedCount()
                  + ", Name = "+cabin.getName());
            }

        } 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 javax.naming.Context getInitialContext()
       throws javax.naming.NamingException{
    Properties p = new Properties();
```

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

  public static void makeCabins(CabinHome home, int fromId, int toId,
                                int deckLevel, int shipNumber)
    throws RemoteException, CreateException {

    int bc = 3;
    for (int i = fromId; i <= toId; i++) {
        Cabin cabin = home.create(i);
        int suiteNumber = deckLevel*100+(i-fromId);
        cabin.setName("Suite "+suiteNumber);
        cabin.setDeckLevel(deckLevel);
        bc = (bc==3)?2:3;
        cabin.setBedCount(bc);
        cabin.setShip(shipNumber);
    }
  }
}
```

Copy this code into your IDE, save, and recompile the `Client_2` application. When it compiles successfully, run it. There's lots of output--here are the first few lines:

```
PK = 1, Ship = 1, Deck = 1, BedCount = 3, Name = Master Suite
PK = 2, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 100
PK = 3, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 101
PK = 4, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 102
PK = 5, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 103
PK = 6, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 104
PK = 7, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 105
...
```

You now have 100 cabin records in your `CABIN` table, representing 100 cabin entities in your EJB system. This provides a good set of test data for the session bean we will create in the next section, and for subsequent examples throughout the book.

# Developing a Session Bean

Session beans act as agents to the client, controlling workflow (the business process) and filling the gaps between the representation of data by entity beans and the business logic that interacts with that data. Session beans are often used to manage interactions between entity beans and can perform complex manipulations of beans to accomplish some task. Since we have only defined one entity bean so far, we will focus on a complex manipulation of the Cabin bean rather than the interactions of the Cabin bean with other entity beans. In Chapter 7, after we have had the opportunity to develop other entity beans, interactions of entity beans within session beans will be explored in greater detail.

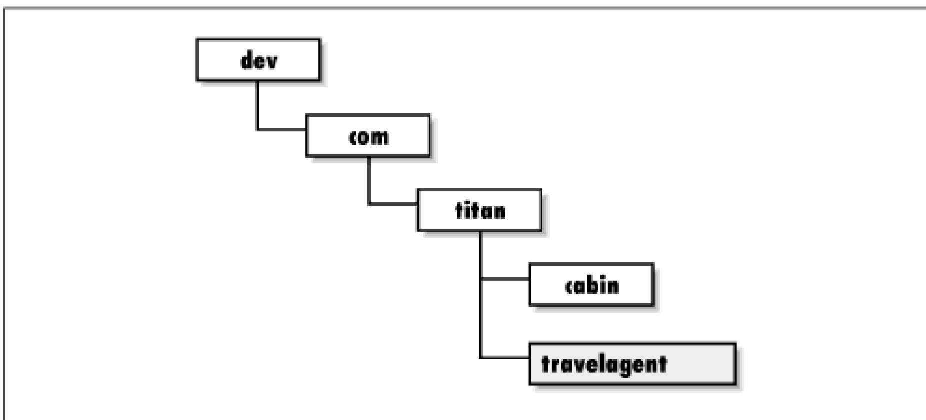
Client applications and other beans use the Cabin bean in a variety of ways. Some of these uses were predictable when the Cabin bean was defined, but most were not. After all, an entity bean represents data--in this case, data describing a cabin. The uses to which we put that data will change over time--hence the importance of separating the data itself from the workflow. In Titan's business system, for example, we will need to list and report on cabins in ways that were not predictable when the Cabin bean was defined. Rather than change the

Cabin bean every time we need to look at it differently, we will obtain the information we need using a session bean. Changing the definition of an entity bean should only be done within the context of a larger process--for example, a major redesign of the business system.

In Chapters and , we talked hypothetically about a TravelAgent bean that was responsible for the workflow of booking a passage on a cruise. This session bean will be used in client applications accessed by travel agents throughout the world. In addition to booking tickets, the TravelAgent bean also provides information about which cabins are available on the cruise. In this chapter, we will develop the first implementation of this listing behavior in the TravelAgent bean. The listing method we develop in this example is admittedly very crude and far from optimal. However, this example is useful for demonstrating how to develop a very simple stateless session bean and how these session beans can manage other beans. In Chapter 7, we will rewrite the listing method. This "list cabins" behavior is used by travel agents to provide customers with a list of cabins that can accommodate the customer's needs. The Cabin bean does not directly support the kind of list, nor should it. The list we need is specific to the TravelAgent bean, so it's the Travel- Agent bean's responsibility to query the Cabin beans and produce the list.

Before we get started, we will need to create a development directory for the TravelAgent bean, as we did for the Cabin bean. We name this directory *travelagent* and nest it below the *com/titan* directory, which also contains the *cabin* directory (see Figure 4-7).

**Figure 4-7. Directory structure for the TravelAgent bean**




## TravelAgent: The Remote Interface

As before, we start by defining the remote interface so that our focus is on the business purpose of the bean, rather than its implementation. Starting small, we know that the TravelAgent will need to provide a method for listing all the cabins available with a specified bed count for a specific ship. We'll call that method `listCabins()`. Since we only need a list of cabin names and deck levels, we'll define `listCabins()` to return an array of `String`s. Here's the remote interface for `TravelAgent`:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
```

```
import javax.ejb.FinderException;

public interface TravelAgent extends javax.ejb.EJBObject {

    // String elements follow the format "id, name, deck level"
    public String [] listCabins(int shipID, int bedCount)
        throws RemoteException;
}
```

Copy the `TravelAgent` interface definition into your IDE, and save it to the *travel- agent* directory. Compile the class to ensure that it is correct.

## TravelAgentHome: The Home Interface

The second step in the development of any bean is to create the home interface. The home interface for a session bean defines the create methods that initialize a new session bean for use by a client.

Find methods are not used in session beans; they are used with entity beans to locate persistent entities for use on a client. Unlike entity beans, session beans are not persistent and do not represent data in the database, so a find method would not be meaningful; there is no specific session to locate. A session bean is dedicated to a client for the life of that client (or less). For the same reason, we don't need to worry about primary keys; since session beans don't represent persistent data, we don't need a key to access that data.

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface TravelAgentHome extends javax.ejb.EJBHome {
    public TravelAgent create()
        throws RemoteException, CreateException;
}
```

In the case of the TravelAgent bean, we only need a simple `create()` method to get a reference to the bean. Invoking this `create()` method returns a `Travel-Agent` remote reference that the client can use for the reservation process. Copy the `TravelAgentHome` definition into your IDE and save it to the *travelagent* directory. Compile the class to ensure that it is correct.

## TravelAgentBean: The Bean Class

Using the remote interface as a guide, we can define the `TravelAgentBean` class that implements the `listCabins()` method. The following code contains the complete definition of `TravelAgentBean` for this example. Copy the `TravelAgentBean` definition into your IDE and save it to the *travelagent* directory. Compile the class to ensure that it is correct. EJB 1.1 and EJB 1.0 differ significantly in how one bean locates another, so I have provided separate TravelAgentBean listings for each version.

### EJB 1.1: TravelAgentBean

Here's the code for the EJB 1.1 version of the `TravelAgentBean`:

```
package com.titan.travelagent;

import com.titan.cabin.Cabin;
import com.titan.cabin.CabinHome;
import com.titan.cabin.CabinPK;
import java.rmi.RemoteException;
import javax.naming.InitialContext;
import javax.naming.Context;
import java.util.Properties;
import java.util.Vector;

public class TravelAgentBean implements javax.ejb.SessionBean {

   public void ejbCreate() {
   // Do nothing.
   }
   public String [] listCabins(int shipID, int bedCount) {

        try {
            javax.naming.Context jndiContext = new InitialContext();
            Object obj = jndiContext.lookup("java:comp/env/ejb/CabinHome");

            CabinHome home = (CabinHome)
                javax.rmi.PortableRemoteObject.narrow(obj, CabinHome.class);

            Vector vect = new Vector();
            CabinPK pk = new CabinPK();
            Cabin cabin;
            for (int i = 1; ; i++) {
               pk.id = i;
               try {
                  cabin = home.findByPrimaryKey(pk);
                } catch(javax.ejb.FinderException fe) {
                    break;
                }
                // Check to see if the bed count and ship ID match.
                if (cabin.getShip() == shipID &&
                    cabin.getBedCount() == bedCount) {
                  String details =
                    i+","+cabin.getName()+","+cabin.getDeckLevel();
                  vect.addElement(details);
                }
            }

            String [] list = new String[vect.size()];
            vect.copyInto(list);
            return list;

       } catch(Exception e) {throw new EJBException(e);}
   }

   private javax.naming.Context getInitialContext()
   throws javax.naming.NamingException {
      Properties p = new Properties();
      // ... Specify the JNDI properties specific to the vendor.
      return new javax.naming.InitialContext(p);
   }

   public void ejbRemove(){}
```

```
   public void ejbActivate(){}
   public void ejbPassivate(){}
   public void setSessionContext(javax.ejb.SessionContext cntx){}
}
```

Examining the `listCabins()` method in detail, we can address the implementation in pieces, starting with the use of JNDI to locate the `CabinHome`:

```
javax.naming.Context jndiContext = new InitialContext();

Object obj = jndiContext.lookup("java:comp/env/ejb/CabinHome");

CabinHome home = (CabinHome)
     javax.rmi.PortableRemoteObject.narrow(obj, CabinHome.class);
```

Beans are clients to other beans, just like client applications. This means that they must interact with other beans in the same way that client applications interact with beans. In order for one bean to locate and use another bean, it must first locate and obtain a reference to the bean's EJB home. This is accomplished using JNDI, in exactly the same way we used JNDI to obtain a reference to the `CabinHome` in the client application we developed earlier. In EJB 1.1, all beans have a default JNDI context called the environment context, which was discussed a little in Chapter 3. The default context exists in the name space (directory) called `"java:comp/env"` and its subdirectories. When the bean is deployed, any beans it uses are mapped into the subdirectory `"java:comp/env/ejb"`, so that bean references can be obtained at runtime through a simple and consistent use of the JNDI default context. We'll come back to this when we take a look at the deployment descriptor for the TravelAgent bean below.

Once the EJB home of the Cabin bean is obtained, we can use it to produce a list of cabins that match the parameters passed. The following code loops through all the Cabin beans and produces a list that includes only those cabins with the ship and bed count specified:

```
Vector vect = new Vector();
CabinPK pk = new CabinPK();
Cabin cabin;
for (int i = 1; ; i++) {
   pk.id = i;
   try {
       cabin = home.findByPrimaryKey(pk);
   } catch(javax.ejb.FinderException fe){
       break;
   }
   // Check to see if the bed count and ship ID match.
   if (cabin.getShip() == shipID && cabin.getBedCount() == bedCount) {
      String details = i+","+cabin.getName()+","+cabin.getDeckLevel();
      vect.addElement(details);
   }
}
```

This method simply iterates through all the primary keys, obtaining a remote reference to each Cabin bean in the system and checking whether its `ship` and `bedCount` match the parameters passed in. The `for` loop continues until a `FinderException` is thrown, which would probably occur when a primary key is used that isn't associated with a bean. (This isn't the most robust code possible, but it will do for now.) Following this block of code, we

simply copy the `Vector`'s contents into an array and return it to the client.

While this is a very crude approach to locating the right Cabin beans--we will define a better method in Chapter 7--it is adequate for our current purposes. The purpose of this example is to illustrate that the workflow associated with this listing behavior is not included in the Cabin bean nor is it embedded in a client application. Workflow logic, whether it's a process like booking a reservation or obtaining a list, is placed in a session bean.

### EJB 1.0: TravelAgentBean

Here's the code for the EJB 1.0 version of the `TravelAgentBean`:

```
package com.titan.travelagent;

import com.titan.cabin.Cabin;
import com.titan.cabin.CabinHome;
import com.titan.cabin.CabinPK;
import java.rmi.RemoteException;
import javax.naming.InitialContext;
import javax.naming.Context;
import java.util.Properties;
import java.util.Vector;

public class TravelAgentBean implements javax.ejb.SessionBean {

   public void ejbCreate() {
   // Do nothing.
   }
   public String [] listCabins(int shipID, int bedCount)
      throws RemoteException {
        try {
            Context jndiContext = getInitialContext();
            CabinHome home = (CabinHome)jndiContext.lookup("CabinHome");

            Vector vect = new Vector();
            CabinPK pk = new CabinPK();
            Cabin cabin;
            for (int i = 1; ; i++) {
               pk.id = i;
               try {
                   cabin = home.findByPrimaryKey(pk);
               } catch(javax.ejb.FinderException fe) {
                   break;
               }
               // Check to see if the bed count and ship ID match.
               if (cabin.getShip() == shipID &&
                   cabin.getBedCount() == bedCount) {
                 String details =
                   i+","+cabin.getName()+","+cabin.getDeckLevel();
                 vect.addElement(details);
               }
            }

            String [] list = new String[vect.size()];
            vect.copyInto(list);
            return list;

       } catch (javax.naming.NamingException ne) {
```

```
            throw new RemoteException("Unable to locate CabinHome",ne);
        }
    }

    private javax.naming.Context getInitialContext()
    throws javax.naming.NamingException {
        Properties p = new Properties();
        // ... Specify the JNDI properties specific to the vendor.
        return new javax.naming.InitialContext(p);
    }

    public void ejbRemove(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void setSessionContext(javax.ejb.SessionContext cntx){}
}
```

The most significant difference between this code and the EJB 1.1 code is the use of JNDI to locate the `CabinHome`:

```
Context jndiContext = getInitialContext();
CabinHome cabinHome = (CabinHome)jndiContext.lookup("CabinHome");
```

Beans interact with other beans in the same way that clients interact with beans. In order for one bean to locate and use another bean, it must first locate and obtain a reference to the bean's EJB home. This is accomplished using JNDI, in exactly the same way we used JNDI to obtain a reference to the `CabinHome` in the client application we developed earlier. If you take a close look at the method `getInitialContext()`, you will discover that it is exactly the same as the `getInitialContext()` method in the client classes defined earlier. The only difference is that the method is not static. You will need to change this code to match the correct settings for your EJB server. Once the EJB home of the Cabin bean is obtained, we can use it to produce our list of cabins that match the parameters passed.

The logic for finding beans with cabins that match the desired parameters is the same in EJB 1.1 and EJB 1.0. Again, it's a crude approach: we will define a better method in Chapter 7. Our purpose here is to demonstrate that the workflow associated with this listing behavior is not included in the Cabin bean nor is it embedded in a client application. Workflow logic, whether it's a process like booking a reservation or obtaining a list, is placed in a session bean.

## EJB 1.1: TravelAgent Bean's Deployment Descriptor

The TravelAgent bean uses an XML deployment descriptor similar to the one used for the Cabin entity bean. Here is the *ejb-jar.xml* file used to deploy the TravelAgent. In Chapter 10, you will learn how to deploy several beans in one deployment descriptor, but for now the TravelAgent and Cabin beans are deployed separately.

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

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

```
    <ejb-name>TravelAgentBean</ejb-name>
    <home>com.titan.travelagent.TravelAgentHome</home>
    <remote>com.titan.travelagent.TravelAgent</remote>
    <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>

    <ejb-ref>
      <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>com.titan.cabin.CabinHome</home>
      <remote>com.titan.cabin.Cabin</remote>
   </ejb-ref>
  </session>
 </enterprise-beans>

 <assembly-descriptor>
   <security-role>
      <description>
         This role represents everyone who is allowed full access
         to the cabin bean.
      </description>
     <role-name>everyone</role-name>
   </security-role>

   <method-permission>
     <role-name>everyone</role-name>
     <method>
         <ejb-name>TravelAgentBean</ejb-name>
         <method-name>*</method-name>
     </method>
   </method-permission>

   <container-transaction>
     <method>
        <ejb-name>TravelAgentBean</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

Other than the `<session-type>` and `<ejb-ref>` elements, this XML deployment descriptor should make sense since it uses many of the same elements as the Cabin bean's. The `<session-type>` element can be `Stateful` or `Stateless` to indicate which type of session bean is used.

The `<ejb-ref>` element is used at deployment time to map the bean references used within the TravelAgent bean. In this case, the `<ejb-ref>` element describes the Cabin bean, which we already deployed. The `<ejb-ref-name>` element specifies the name that must be used by the TravelAgent bean to obtain a reference to the Cabin bean's home. The `<ejb-ref-type>` tells the container what kind of bean it is, `Entity` or `Session`. The `<home>` and `<remote>` elements specify the fully qualified interface names of the Cabin's home and remote bean interfaces.

When the bean is deployed, the `<ejb-ref>` will be mapped to the Cabin bean in the EJB

server. This is a vendor-specific process, but the outcome should always be the same. When the TravelAgent does a JNDI lookup using the context name `"java:comp/env/ejb/CabinHome"` it will obtain a remote reference to the Cabin bean's home. The purpose of the `<ejb-ref>` element is to eliminate network specific and implementation specific use of JNDI to obtain bean references. This makes a bean more portable because the network location and JNDI service provider can change without impacting the bean code or even the XML deployment descriptor.

## EJB 1.0: The TravelAgent Beans' Deployment Descriptor

Deploying the TravelAgent bean is essentially the same as deploying the Cabin bean, except we use a `SessionDescriptor` instead of an `EntityDescriptor`. Here is the definition of the `MakeDD` for creating and serializing a `SessionDescriptor` for the `TravelAgentBean`:

```
package com.titan.travelagent;

import javax.ejb.deployment.*;
import javax.naming.CompoundName;
import java.util.*;
import java.io.*;

public class MakeDD {

    public static void main(String [] args) {
        try {

            if (args.length <1) {
                System.out.println("must specify target directory");
                return;
            }

            SessionDescriptor sd = new SessionDescriptor();

            sd.setEnterpriseBeanClassName(
                "com.titan.travelagent.TravelAgentBean");
            sd.setHomeInterfaceClassName(
                "com.titan.travelagent.TravelAgentHome");
            sd.setRemoteInterfaceClassName(
                "com.titan.travelagent.TravelAgent");

            sd.setSessionTimeout(300);
            sd.setStateManagementType(SessionDescriptor.STATELESS_SESSION);

            ControlDescriptor cd = new ControlDescriptor();
            cd.setIsolationLevel(ControlDescriptor.TRANSACTION_READ_COMMITTED);
            cd.setMethod(null);
            cd.setRunAsMode(ControlDescriptor.CLIENT_IDENTITY);
            cd.setTransactionAttribute(ControlDescriptor.TX_REQUIRED);
            ControlDescriptor [] cdArray = {cd};
            sd.setControlDescriptors(cdArray);

            CompoundName jndiName =
                new CompoundName("TravelAgentHome", new Properties());
            sd.setBeanHomeName(jndiName);

            String fileSeparator =
                System.getProperties().getProperty("file.separator");
```

```
            if(! args[0].endsWith(fileSeparator))
                args[0] += fileSeparator;

            FileOutputStream fis =
                new FileOutputStream(args[0]+"TravelAgentDD.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fis);
            oos.writeObject(sd);
            oos.flush();
            oos.close();
            fis.close();
        } catch(Throwable t) { t.printStackTrace(); }
    }
}
```

The `MakeDD` definition for the TravelAgent bean is essentially the same as the one for the Cabin bean. The difference is that we are using a `SessionDescriptor` instead of an `EntityDescriptor` and the bean class names and JNDI name are different. We do not specify any container-managed fields because session beans are not persistent.

After instantiating the `javax.ejb.SessionDescriptor`, the `MakeDD` application sets the remote interface and bean class names:

```
sd.setEnterpriseBeanClassName("com.titan.travelagent.TravelAgentBean");
sd.setHomeInterfaceClassName("com.titan.travelagent.TravelAgentHome");
sd.setRemoteInterfaceClassName("com.titan.travelagent.TravelAgent");
```

Next, we set two properties that control session timeouts (what happens if the bean is idle) and state management:

```
sd.setSessionTimeout(300);
sd.setStateManagementType(SessionDescriptor.STATELESS_SESSION);
```

`setSessionTimeout()` specifies how many seconds the session should remain alive if it is not being used. In `MakeDD` we specify 300 seconds. This means that if no method is invoked on the session for over five minutes, it will be removed and will no longer be available for use.[3] If a method is invoked on a bean that has timed out, a `javax.ejb.ObjectNotFoundException` will be thrown. Once a stateful session bean has timed out, all of its accumulated state is lost. When a session bean times out, the client must create a new TravelAgent bean by invoking the `TravelAgentHome.create()` method. The `setStateManagement()` method determines whether the bean is stateful or stateless. At this point in it its development, the `TravelAgentBean` doesn't have any conversational state that needs to be maintained from one method to the next, so we make it a stateless session bean, which is more efficient. Both of these methods are unique to session descriptors; there are no corresponding methods in the `EntityDescriptor` class.

The next section specifies the default `ControlDescriptor` for the `TravelAgentBean`. These settings are the same as those used in the Cabin bean. The isolation level determines the visibility of the data being accessed. Chapter 8 explores isolation levels in more detail. The transactional attribute, `TX_REQUIRED`, tells the EJB server that this bean must be included in the transactional scope of the client invoking it; if the client is not in a transaction, a new transaction must be created for the method invocation, as follows:

```
ControlDescriptor cd = new ControlDescriptor();
```

```
cd.setIsolationLevel(ControlDescriptor.TRANSACTION_READ_COMMITTED);
cd.setMethod(null);
cd.setRunAsMode(ControlDescriptor.CLIENT_IDENTITY);
cd.setTransactionAttribute(ControlDescriptor.TX_REQUIRED);
ControlDescriptor [] cdArray = {cd};
sd.setControlDescriptors(cdArray);
```

The next section creates a JNDI name for `TravelAgent`'s EJB home. When we use JNDI to look up the `TravelAgentHome`, this will be the name we specify:

```
CompoundName jndiName = new CompoundName("TravelAgentHome",new Properties());
```

Finally, the `MakeDD` serializes the `SessionDescriptor` to a file named *TravelAgentDD.ser* and saves it to the *travelagent* directory.

You will need to compile and run the `MakeDD` class before continuing:

```
\dev % java com.titan.travelagent.MakeDD com/titan/travelagent

F:\..\dev>java com.titan.travelagent.MakeDD com\titan\travelagent
```

## EJB 1.1: The JAR File

To place the TravelAgent bean in a JAR file, we use the same process we used for the Cabin bean. We shrink-wrap the TravelAgent bean class and its deployment descriptor into a JAR file and save to the *com/titan/travelagent* directory:

```
\dev % jar cf cabin.jar com/titan/travelagent/*.class META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\travelagent\*.class META-INF\ejb-jar.xml
```

You might have to create the META-INF directory first, and copy *ejb-jar.xml* into that directory. The TravelAgent bean is now complete and ready to be deployed.

## EJB 1.0: The JAR File

To place the TravelAgent bean in a JAR file, we use the same process we used for the Cabin bean. First, we have to create a manifest file, which we save in the *com/titan/travelagent* directory:

```
Name: com/titan/travelagent/TravelAgentDD.ser
Enterprise-Bean: True
```

Now that the manifest is ready, we can shrink-wrap the TravelAgent bean so that it's ready for deployment:

```
\dev % jar cmf com/titan/travelagent/manifest \
TravelAgent.jar com/titan/travelagent/*.class com/titan/travelagent/*.ser

F:\..\dev>jar cmf com\titan\travelagent\manifest TravelAgent.jar
com\titan\travelagent\*.class com\titan\travelagent\*.ser
```

The TravelAgent bean is now complete and ready to be deployed.

## Deploying the TravelAgent Bean

To make your TravelAgent bean available to a client application, you need to use the deployment utility or wizard of your EJB server. The deployment utility reads the JAR file to add the TravelAgent bean to the EJB server environment. Unless your EJB server has special requirements, it is unlikely that you will need to change or add any new attributes to the bean. You will not need to create a database table for this example, since the TravelAgent bean is using only the Cabin bean and is not itself persistent. Deploy the TravelAgent bean and proceed to the next section.

## Creating a Client Application

To show that our session bean works, we'll create a simple client application that uses it. This client simply produces a list of cabins assigned to ship 1 with a bed count of 3. Its logic is similar to the client we created earlier to test the Cabin bean: it creates a context for looking up `TravelAgentHome`, creates a TravelAgent bean, and invokes `listCabins()` to generate a list of the cabins available. Here's the code:

```
package com.titan.travelagent;

import com.titan.cabin.CabinHome;
import com.titan.cabin.Cabin;
import com.titan.cabin.CabinPK;

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

public class Client_1 {
    public static int SHIP_ID = 1;
    public static int BED_COUNT = 3;

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

           Object ref = (TravelAgentHome)
               jndiContext.lookup("TravelAgentHome");
           TravelAgentHome home = (TravelAgentHome)
               // EJB 1.0: Use Java cast instead of narrow()
               PortableRemoteObject.narrow(ref,TravelAgentHome.class);

           TravelAgent reserve = home.create();

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

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

        } catch(java.rmi.RemoteException re){re.printStackTrace();}
          catch(Throwable t){t.printStackTrace();}
```

```
  }
  static public Context getInitialContext() throws Exception {
    Properties p = new Properties();
    // ... Specify the JNDI properties specific to the vendor.
    return new InitialContext(p);
  }
}
```

The output should look like this:

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

You have now successfully created the first piece of the TravelAgent session bean: a method that obtains a list of cabins by manipulating the Cabin bean entity.

---

1. Chapter 9 discusses how to work with servers that don't support entity beans. Chapter 6 includes a discussion of bean-managed persistence, which you can use if your server doesn't support container-managed persistence.

2. In Chapters and , we discuss implementing the `hashCode()`and `equals()`methods in more detail.

3. Whether a session timeout is measured from creation time (the time the session bean is created) or from the time of last activity (when the last business method is invoked) is not clearly described in EJB 1.0. As a result, some vendors set the timeout relative to one of these two events (creation or last activity). Consult your vendor's documentation to determine your EJB server's timeout policy.

**Back to: Enterprise JavaBeans, 2nd Edition**

---



*© 2000, O'Reilly & Associates, Inc.*
*webmaster@oreilly.com*