A Retrospective View on EJB 2

Henry H. Liu

Note: This article is a chapter called out from my book "*Developing Enterprise Applications: An End-to-End Approach*." It is moved out of the book as not all readers are interested in this out-dated EJB 2 technology.

In the frenetic era of e-businesses more than a decade ago, the term EJB was one of the hottest buzz-words. Corporate executives often used this buzz-word to cheer up their employees and get external media attention by shouting loudly that "e-businesses run on EJBs and EJBs run on our platforms …." That fashion has faded with time - unfortunately. What happened exactly? EJB lost its battle against POJO-based frameworks like Spring and Hibernate.

Perhaps it's unfair to blame the EJB technology for its complexity. It's more fair to say that Spring, Hibernate and the like emerged – mostly because of EJB's complexity. Perhaps it's even meaningless to say what's unfair and what's fair here. What I am most interested in here is to present a retrospective view on EJB 2 by walking you through a few EJBs in their completeness rather than giving you a verbal review (and we are even going to run those EJB 2's on JBoss). By doing so, you would have a better understanding of what EJBs exactly are. Moreover, we even reap the following benefits:

- You would appreciate more for what Spring can do for your project if you knew how complex it would be if you had to use EJBs to write your enterprise application. It's even a non-trivial task to rewrite our SOBA sample with EJB 2 components.
- Many legacy enterprise applications out there still run on EJB 2. In case you would be charged for migrating your EJB 2 based product to EJB 3 (which looks more like POJOs than EJB 2's) or Spring based, or for integrating your Spring-based components with your EJB 2 legacy product, this chapter would provide you a sufficient base for you to get started.

Let's begin with a general overview of EJBs before we get concrete with coding in EJBs.

1 Overview of EJB 2

The Enterprise JavaBeans (EJB) architecture is a specification developed at Sun Microsystems (now part of Oracle) more than a decade ago. It promotes a component-based architecture for developing distributed, enterprise applications. Because it's a specification, some vendors have to implement it, and thus we have EJB server providers such as IBM's WebSphere, Oracle's WebLogic, Red Hat's Jboss, and a few others that are less widely known. Since enterprise applications are more demanding in terms of transactional integrity, security, performance, scalability, reliability, availability, maintainability, etc., the idea was that EJB containers would handle all complexities in managing EJB components while having developers focus on coding the business logic part of an EJB-based enterprise application only. This was a great idea that it would be a huge extra burden if developers had to worry about tedious implementations of data persistence, transactions, security and resource management, etc., if EJB containers did not provide these services out of the box. This idea was inscribed by J2EE with three types of EJBs: entity beans, session beans, and message-driven beans.

In this chapter, we cover entity beans and session beans only. Since the EJB spec has gone through several releases, as is shown in Table 1, we need to be clear upfront about which version of EJBs we deal with. We pick EJB 2, as that's what would come to most people's mind when talking about the complexities of EJBs. The latest EJB 3 spec has dropped entity beans completely, and some latest EJB containers do not support entity beans anymore.

Table 1 EJB evolution from 1.0 to 3.1 (note the new key features with EJB 3)

Release	**Year (JSR)**	**Comment**
EJB 1.0/1.1	2001	Initially started at IBM; later adopted by Sun Microsystems.
EJB2.0/2.1	2003(JSR 19/153)	Stateless/stateful session beans, entity beans, and message-driven beans.
EJB3.0/3.1	2006/2010 (JSR 220/318)	• Remote interfaces and message-driven beans de-supported in EJB 3.1 Lite. • Deployment descriptors replaced by annotations in bean implementation classes. • Supports POJOs, DI (dependency injection) and annotations. • Entity beans replaced by Hibernate and Java Persistence API

Based on how data persistence is managed, EJB 2 entity beans can be of either bean-managed-persistence (BMP) or container-managed-persistence (CMP). In the next two sections, we discuss BMP and CMP entity beans along with a stateless session bean that accesses those entity beans, respectively. Let's begin with the BMP entity beans next.

2 A BMP-Based EJB 2 Example

A complete EJB-based enterprise application may contain any number of EJBs and web components, accessed by multiple application clients. For illustrative purposes, we present two EJBs only: one customer entity bean and one customer session bean. We demonstrate how a session bean can access the entity bean, whether BMP-based or CMP-based. We then demonstrate how an external client can access these EJBs. The example is simple, but it illustrates the gist of how an enterprise application uses EJBs as its basic building elements in the realm of enterprise computing – much like the Spring POJOs we used to compose SOBA as we have demonstrated in the previous chapters.

The enterprise computing community agrees that one of the most challenging tasks with developing an enterprise application is to create, update, and remove data records stored in a database with the required data integrity so that the data is always consistent and synchronized with all clients. Entity beans were considered one of the most promising approaches to dealing with this challenging task. As a reflection of enterprise data stored in a database, entity beans help maintain data both in the data store and in memory so that widely distributed clients can safely access (create, query, update, and remove) the data while the EJB container provides services to help guarantee that data transactions can occur in a thread-safe, consistent, and efficient manner. To illustrate how this happens, we need to create a domain object in the database first, as is described next.

2.1 MySQL Preparation

You might wonder why we use MySQL instead of some embedded databases. First, I am always inclined to get everything as real as possible when presenting a sample. Secondly, after completing the previous chapters with the SOBA sample, I am certain that you have become comfortable with MySQL. Thirdly, I will show you not only the EJBs but also how to run the EJB examples presented in this chapter on Jboss so that you would learn how to configure a real, popular database to work on Jboss. This would help enhance your skillset further for developing enterprise applications.

First, download the EJB samples from this book's website at http://www.perfmath.com. Assuming that you have set up your MySQL environment by following the procedures described in Appendix B, execute the following commands to create the database and customer table to work with the EJB 2 examples presented in this chapter:

```
cmd>mysql –h 127.0.0.1 –u root –p <create_ejbdb.sql
cmd>mysql ejb –h 127.0.0.1 –u admin –p < create_customer.sql
```

Listings 1 and 2 show the contents of these two scripts. The first script creates a database with the name of "*ejb*" and user/password of "*admin/admin*" while the second script creates a table named "*customer*" with the columns of {*customerID, name, password, email and locked*}. The second script also creates a primary key for the *customer* table using the "*customerID*" column. The EJB examples presented in this chapter use this simple database and table to demonstrate entity and session EJBs. You might wonder why we don't use the SOBA database and the customer table therein for the EJB samples to be presented in this chapter. The reasons are twofold. One is that our customer table in the SOBA database has too many columns for the EJB examples to be presented here, and the other is that some readers might want to try out these EJB samples standalone.

Listing 1 create_ejbdb.sql

```
create database ejb;
grant usage on *.* to admin@localhost identified by 'admin';
grant all privileges on ejb.* to admin@localhost;
show databases;
```

Listing 2 create_customer.sql

```
CREATE TABLE customer (
customerID    VARCHAR(10) not null,
name          VARCHAR(30),
password      VARCHAR(30),
email         VARCHAR(30),
locked        BOOLEAN);
alter table customer  add constraint customer_pk_customerID primary key (customerID);
```

2.2 The Concept of Entity Beans

Entity beans represent enterprise data while session beans represent business logic. Unlike the session beans, a major requirement for entity beans is to maintain their consistent properties no matter how many clients are accessing a same entity bean at the same time. The EJB container and bean developers are responsible for making sure that concurrent operations from multiple clients on an entity bean occur within an appropriate transaction. This boils down to the persistence mechanism associated with an entity bean. If persistence is coded by the bean developer, it is called *bean-managed persistence* (BMP); otherwise, if the persistence is managed by the EJB container, it is called *container-managed persistence* (CMP). With BMP, the bean developer writes the database access code. If the database is a relational one, data access is typically coded in data access object (DAO) classes using JDBC, as we will see in our first EJB sample shortly. However, BMP-based entity beans can also access data in non-relational databases.

An entity bean needs to have up to four Java interfaces and one EJB class to describe it, which include:

- **Home interface and remote interface**: The home interface defines `create`, `findBy` and `home` methods, while the remote interface defines business methods. Methods defined in the home interface are also called *class methods*, since they are not specific to an instance, while methods defined in the remote interface are also called *instance methods*. We'll explain more about these methods later.
- **Local interface and local home interface**: The local interface and local home interface correspond to the home interface and remote interface, respectively. Both session beans and entity beans may have local or remote interfaces or both. Local interfaces provide clients with access in the same JVM, while remote interfaces allow clients to call EJBs in a remote or separate JVM. One significant drawback with EJBs is that remote access incurs significant RMI overhead and network traffic, leading to severe performance issue. Therefore, in general, entity beans are called by session beans as a session façade using their local interfaces for maximum performance and scalability potentials, while remote interfaces are reserved for development test only.
- **Bean Implementation Class**: The bean implementation class implements the interfaces defined by the home interface and remote/local interface of an entity bean. A bean implementation class must follow certain method naming rules as we will discuss more later when we look at the customer entity bean implementation class.

2.3 EJB METHODS AND DATABASE ACCESS

The bean implementation class has to implement a rigid set of EJB methods in order to carry out certain types of database accesses. Table 2 lists those methods. The EJB methods and their corresponding database operations seem to indicate the complexity set up in the first place. It may be worthwhile to memorize this so that you would know how the newer frameworks replace this framework, because these database operations must be implemented one way or the other no matter which framework is used.

Table 2 EJB methods and their corresponding database access operations

Method	**Data operation**
ejbCreate ()	Inserts new record and returns primary key
ejbFindByPrimaryKey ()	Selects record with a given primary key and returns primary key
ejbFindXXX()	Selects one or more records and returns one or a collection of primary keys
ejbHomeXXX()	Custom database operations (select or update) not specific to any primary keys
ejbLoad()	Refreshes in-memory persistent variables as Selects
ejbStore()	Stores persistent variables as Updates
ejbRemove()	Deletes record

Next, we present the BMP-based EJB2 example. We'll explain various methods listed in Table 2 in proper context of the customer EJBs.

2.4 THE CUSTOMER ENTITY BEAN

The Data Model Class

Once again, note that the customer table has the columns of {*customerID, name, password, email, locked*}. First, in order to compose the customer entity bean, we need a Java class to represent the customer table and its columns. We use a Java class named `CustomerModel.java` to accomplish this task. Listing 3 shows this class with all its getters and setters omitted to save space. This model class encapsulates all persistence variables for the customer entity bean. Its customerID attribute would serve as the primary key that the customer entity bean would depend on for many operations. The BMP version of a customer entity bean would use a DAO class to interact with the database, which will be discussed after we take a look at how the customer entity bean is implemented next.

Listing 3 The model class for the customer entity bean (CustomerModel.java)

```
public class CustomerModel
{
    private String customerID;
    private String name;
    private String password;
    private String email;
    private boolean locked;

    public CustomerModel(String customerID, String name, String password,
            String email, boolean locked) {
        super();
        this.customerID = customerID;
        this.name = name;
        this.password = password;
        this.email = email;
        this.locked = locked;
    }
    // getters and setters omitted
}
```

The Home Interface and Remote Interface

Listing 4 shows the home interface (`CustomerHome.java`) while Listing 5 shows the remote interface (`Customer.java`) for the customer entity bean. We discuss these two interfaces next.

As shown in Listing 4, the home interface defines the following methods for the customer entity bean: `create (…)`, `findByPrimaryKey (…)`, `findByCustomerName (…)`, `findAll ()`, and `getTotalCustomers ()`. This home interface extends `EJBHome` and all its methods throw `RemoteException`. Besides, the `create` method throws `CreateException`, whereas all finder methods throw `FinderException`. The last method of `getTotalCustomers` () is neither a `create` nor a `finder` method. In EJB's parlance, it is called a *home* method. You may wonder what it means by "*home*." What it means is that unlike all other finder methods defined in the remote interface, it does not require the EJB container to instantiate the entity bean in order for its caller to use it. We will get back to this when we look at the bean implementation class later.

Listing 5 shows all getters and setters for the customer entity bean, which is kind of redundant with the customer model class shown in Listing 3. There really isn't much to talk about it except that it extends `EJBObject` and throws `RemoteException` with every one of its methods.

Listing 4 The home interface for the customer entity bean (CustomerHome.java)

```
import java.util.Collection;
import javax.ejb.*;
import java.rmi.*;

public interface CustomerHome extends EJBHome
{
   public Customer create(String name, String password, String email, boolean locked)
 throws RemoteException, CreateException;
```

```
   public Customer findByPrimaryKey (String customerID) throws RemoteException,
     FinderException;
   public Customer findByCustomerName (String name) throws RemoteException,
     FinderException;
   public Collection findAll () throws RemoteException, FinderException;

// home interface
   public int getTotalCustomers () throws RemoteException;
}
```

Listing 5 The remote interface for the customer entity bean (Customer.java)

```
import javax.ejb.*;
import java.rmi.*;

public interface Customer extends EJBObject
{
   public String getCustomerID () throws RemoteException;
   public String getName () throws RemoteException;
   public void setName (String newName) throws RemoteException;
   public String getPassword () throws RemoteException;
   public void setPassword (String newPassword) throws RemoteException;
   public String getEmail () throws RemoteException;
   public void setEmail (String newEmail) throws RemoteException;
   public boolean getLocked () throws RemoteException;
   public void setLocked (boolean locked) throws RemoteException;
}
```

Local Home Interface and Local Interface

Listings 6 and 7 show the local home interface (CustomerLocalHome.java) and the local interface (CustomerLocal.java). Compared with the (remote) home interface shown in Listing 4, the local interface extends EJBLocalHome and does not throw RemoteException as it's designed to be accessed locally in the same JVM as the caller for performance reasons. Similarly, the local interface extends EJBLocalObject and does not throw RemoteException.

Listing 6 The local home interface for the customer entity bean (CustomerLocalHome.java)

```
import java.util.Collection;
import javax.ejb.*;

public interface CustomerLocalHome extends EJBLocalHome
{
   public CustomerLocal create(String name, String password, String email,
boolean locked) throws CreateException;
   public CustomerLocal findByPrimaryKey (String customerID) throws FinderException;
   public Collection findByCustomerName (String name) throws FinderException;
   public Collection findAll () throws FinderException;
   public int getTotalCustomers () ;
}
```

Listing 7 The local interface for the customer entity bean (CustomerLocal.java)

```
import javax.ejb.*;

public interface CustomerLocal extends EJBLocalObject
{
   public String getCustomerID ();
```

```
    public String getName ();
    public void setName (String newName);
    public String getPassword ();
    public void setPassword (String newPassword);
    public String getEmail ();
    public void setEmail (String newEmail);
    public boolean getLocked ();
    public void setLocked (boolean locked);
}
```

The Customer Entity Bean Implementation Class

Listing 8 shows the bean implementation class for the customer entity bean. In addition to implementing the `EntityBean` interface and getters and setters defined in the remote and/or local interface, the bean implementation class gets heavier in implementing the methods defined in the customer home interface, because we are dealing with a BMP-based entity bean. Next, let's take a closer look at what this bean implementation class is trying to accomplish.

First, note the persistent variable defined as the attribute of `customerData`, as well as the variable `customerDAO` for DAO access. The `context` and `dirty` attributes are EJB variables. The `dirty` flag is set by the business methods that modify the persistent variables to indicate that the customer data has been modified. Note that the business methods of getters and setters are just regular getters and setters. You might wonder how they get the job done for manipulating the customer data. It's the EJB container that does all the actual work behind the scene: The EJB container calls the `ejbLoad ()` method prior to each business method call and calls the `ejbStore` () method afterwards.

Notice that the bean implementation class has many methods. Let's sort them out as follows:

- **Business methods**: These are the getters and setters implemented in the beginning part of the class.
- **Finder methods**: `ejbFindByPrimaryKey` (…), `ejbFindByCustomerName` (…), and `ejbFindAll` (). These finder methods correspond to those defined in the home interface as shown in Listing 4. Note the naming rule here that the prefix of "*ejb*" must be applied to each finder method defined in the home interface with the first letter "*F*" capitalized. The `ejbFindByPrimaryKey` () method returns the primary key while all other finder methods returns a collection of primary keys or empty, depending on if any record exists in the database that satisfies the query criterion. When a finder method returns a primary key, the EJB container looks up the entity bean in its cache with the matching primary key. If the matching entity bean is found and considered *active*, the EJB container returns the cached entity bean to the client. If not, the EJB container invokes the `ejbActivate` () method, causing `ejbLoad` () to read the persistent data from the database into the newly activated entity bean.
- **EJB home method:** `ejbHomeGetTotalCustomers` (). This is the home method we discussed previously. Note the prefix of "`ejbHome`" applied to this method. The client invokes this home method using the home or local home interface. While this method accesses the database, it does not cause the EJB container to instantiate the entity bean like the finder methods.
- **EJB methods**: `ejbCreate` (…), `ejbActivate` (), `ejbPassivate` (), `ejbLoad` (), `ejbStore` (), `ejbPostCreate` (…), `ejbRemove` (), `setEntityContext` (…) and `unsetEntityContext` (). These methods are called by the EJB container and the clients do not call them directly. Although majority of them are empty, they must be there in the bean implementation class. In addition, note how the dirty flag is used in the `ejbLoad` () and `ejbStore` () methods. This is a performance optimization technique, since a database operation is mostly considered *expensive*.
- **Helper method**: `getDAO` (). This method uses a DAO factory class to set the customerDAO attribute for the bean implementation class. Listing 9 shows the `CustomerDAOFactory.java` class. We discuss the MySQL implementation of the `CustomerDAO` interface for this entity bean following Listing

Listing 8 The bean implementation class for the customer entity bean (CustomerBean.java)

```
import java.util.*;
import javax.ejb.*;

public class CustomerBean implements EntityBean {
    CustomerModel customerData;
    private CustomerDAO customerDAO = null;
    private EntityContext context;
    private boolean dirty;

    //business methods
    public String getCustomerID() {
        return customerData.getCustomerID ();
    }

    public String getName() {
        return customerData.getName ();
    }
    public void setName(String name)  {
        customerData.setName (name);
        dirty = true;
    }
    // more similar getters and setters are omitted here to save space

    public String ejbFindByPrimaryKey (String primaryKey)
        throws FinderException {
        boolean found;
        try {
            CustomerDAO dao = getDAO ();
            found = dao.dbSelectByPrimaryKey(primaryKey);
        } catch (CustomerDAOSysException ex) {
            ex.printStackTrace();
            throw new EJBException ("ejbFindByPrimaryKey: " + ex.getMessage());
        }
        if (found) {
            return primaryKey;
        } else {
            throw new ObjectNotFoundException
            ("Row for id = " + primaryKey + " not found.");
        }
    }
    public Collection ejbFindByCustomerName (String name)
            throws FinderException {
            Collection customers;
            try {
                CustomerDAO dao = getDAO ();
                customers = dao.dbSelectByCustomerName (name);
            } catch (CustomerDAOSysException ex) {
                ex.printStackTrace();
                throw new EJBException ("ejbFindByCustomerName: " +
                    ex.getMessage());
            }
            return customers;
        }
    public Collection ejbFindAll ()
            throws FinderException {
            Collection customers;
            try {
```

```
            CustomerDAO dao = getDAO ();
            customers = dao.dbSelectAll();
        } catch (CustomerDAOSysException ex) {
            ex.printStackTrace();
            throw new EJBException ("ejbFindAll: " + ex.getMessage());
        }
        return customers;
    }
public int ejbHomeGetTotalCustomers ()
        throws FinderException {
        int numOfCustomers;
        try {
            CustomerDAO dao = getDAO ();
            numOfCustomers = dao.dbCountTotalCustomers();
        } catch (CustomerDAOSysException ex) {
            ex.printStackTrace();
            throw new EJBException ("ejbHomeGetTotalCustomers: " +
            ex.getMessage());
        }
        return numOfCustomers;
    }
private CustomerDAO getDAO() throws CustomerDAOSysException {
    if (customerDAO == null) {
        customerDAO = CustomerDAOFactory.getDAO();
    }
    return customerDAO;
}
public String ejbCreate (String name, String password,
        String email, boolean locked) throws CreateException {
    CustomerModel customer = null;
    try {
        CustomerDAO dao = getDAO ();
        String key = dao.dbGetKey ();
        customer = new CustomerModel (key, name, password, email, locked);
        dao.dbInsertCustomer(customer);
    } catch (CustomerDAOSysException ex) {
        throw new CreateException ("ejbCreate: " +
            ex.getMessage());
    }
    customerData = customer;
    dirty = false;
    return customerData.getCustomerID();
}
public void setEntityContext (EntityContext context) {
    this.context = context;
}
public void unsetEntityContext () {  }
public void ejbActivate () { }
public void ejbPassivate () {
    customerDAO = null;
}
public void ejbLoad () {
    try {
        CustomerDAO dao = getDAO ();
        customerData = dao.dbLoadCustomer((String)context.getPrimaryKey()) ;
        dirty = false;
    } catch (CustomerDAOSysException ex) {
            ex.printStackTrace();
            throw new EJBException ("ejbLoad: " +
                ex.getMessage());
        }
    }
    public void ejbStore () {
        try {
            if (dirty) {
            CustomerDAO dao = getDAO ();
            dao.dbStoreCustomer(customerData) ;
            dirty = false;
            }
        } catch (CustomerDAOSysException ex) {
            ex.printStackTrace();
            throw new EJBException ("ejbStore: " +
                ex.getMessage());
        }
    }
    public void ejbPostCreate (String name, String password,
            String email, boolean locked)  {
        }
    public void ejbRemove () {
        try {
            CustomerDAO dao = getDAO ();
            dao.dbRemoveCustomer((String)context.getPrimaryKey()) ;
            dirty = false;
        } catch (CustomerDAOSysException ex) {
            ex.printStackTrace();
            throw new EJBException ("ejbRemove: " +
                ex.getMessage());
        }
    }
}
```

Listing 9 The customer DAO factory class for the BMP-based customer entity bean (CustomerDAOFactory.java)

```
import javax.naming.*;
public class CustomerDAOFactory {
    public static CustomerDAO getDAO () throws CustomerDAOSysException {
        CustomerDAO customerDAO = null;
        String customerDAOClass = "java:comp/env/CustomerDAOClass";
        String className = null;
        try {
            InitialContext ic = new InitialContext ();
            className = (String) ic.lookup(customerDAOClass);
            customerDAO = (CustomerDAO) Class.forName(className).newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new CustomerDAOSysException (
"CustomerDAOFactory.getDAO: " + "NamingException for <" + className + "> DAO class: \n" +
ex.getMessage());
        }
        return customerDAO;
    }
}
```

The MySQL Implementation of the `CustomerDAO` interface for the BMP-Based Customer Entity Bean

In order to get this BMP-based customer entity bean working, an implementation of the `CustomerDAO` interface is required. We chose MySQL for the reasons we stated earlier. However, to save space, we only show the first portion of it, and you should download the EJB samples from this book's website and look for this class for all the details you are interested in. We also omit the `CustomerDAO` interface and the `CustomerDAOSysException` class that can also be found in the download, because they are less interesting to us for this example.

For this `CustomerDAOMySQL.java` class as shown in Listing 10, note that the `dbName` is set as follows:

```
String dbName = "java:DefaultDS";
```

We point out this detail here because when we run this example later on Jboss, we would need to set up a `datasource` with exactly the same name of "`DefaultDS`." This is one of the trickier issues with running the EJB samples presented in this chapter, which calls for proper attention. Besides, we use the same `RandomID` class as we used in SOBA sample to create unique customer IDs.

Listing 10 CustomerDAOMySQL.java

```
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.sql.DataSource;
import java.sql.*;

public class CustomerDAOMySQL implements CustomerDAO{
    private Connection con = null;
    private DataSource dataSource = null;

    public CustomerDAOMySQL () throws CustomerDAOSysException {
        String dbName = "java:DefaultDS";
        try {
            InitialContext ic = new InitialContext ();
            dataSource = (DataSource) ic.lookup(dbName);
        } catch (NamingException ex) {
            throw new CustomerDAOSysException (
                    "Cannot connect to database: " +
            dbName + ":\n" + ex.getMessage ());
        }
    }
    @Override
    public String dbGetKey() {
        return (new RandomID(10)).getId();
    }
    private void getConnection () throws CustomerDAOSysException {

        try {
            con = dataSource.getConnection();
        } catch (SQLException ex) {
            throw new CustomerDAOSysException (
                "Exception during db connection: " +
                ":\n" + ex.getMessage ());
        }
    }
// ................ rest omitted to save space ................
}
```

Although a client can call an entity bean remotely, for performance reasons, the best practice is to call an entity bean through a stateless session bean as a session façade. Next, we present such a session bean for accessing the customer entity bean discussed here.

2.5 THE CUSTOMER SESSION BEAN

Given the detailed coverage of the customer entity bean in the preceding sections, let's have a brief coverage of the customer session bean that we use to call the customer entity bean. Once again, we rely on the source code listing to help explain what each of the classes does exactly.

This customer session bean consists of the following classes:

- **Home interface (Listing 11 CustomerSessionHome.java)**: This is the home interface for the customer session bean. As shown in Listing 11, it differs from the home interface of the customer entity bean shown in Listing 4 that it has only one method of **create** (), since it is a stateless session bean. It does not have those finder methods and the home method.
- **Remote interface (Listing 12 CustomerSession.java)**: This is the remote interface for the customer session bean. The remote interface for the customer entity bean contains simple getters and setters, while the remote interface for this customer session bean contains seven business methods that do not necessarily match those of the customer entity bean's remote interface. We'll compare more next when we discuss the bean implementation class for the customer session bean.
- **Bean implementation class (Listing 13 CustomerSessionBean.java)**: This is the bean implementation class for the customer session bean. It implements all seven methods defined in its remote interface. In addition, it implements the five EJB methods of `ejbCreate` (), `ejbRemove` (), `ejbActivate` (), `ejbPassivate` () and `setSessionContext` (…). We'll explain exactly how these methods are called in a client when we discuss the test client for this EJB example shortly.

We also need a Customer VO class and an Exception class, which are discussed following the end of Listing 13.

Listing 11 The home interface for the customer session bean (CustomerSessionHome.java)

```
import javax.ejb.*;
import java.rmi.*;

public interface CustomerSessionHome extends EJBHome{
    CustomerSession create () throws RemoteException, CreateException;
}
```

Listing 12 The remote interface for the customer session bean (CustomerSession.java)

```
package com.ejb2.test;

import java.util.*;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface CustomerSession extends EJBObject {
    public int getTotalCustomers() throws RemoteException;

    public void createCustomer(CustomerVO customer) throws CreateException,
    FinderException, RemoteException;

    public void changePassword(String name, String password)
            throws CustomerCredentialException, CustomerException,
            FinderException, RemoteException;

    public void **changeEmail**(String name, String email)
            throws CustomerCredentialException, CustomerException,
            FinderException, RemoteException;

    public boolean **authenticateCustomer**(String name, String password)
            throws CustomerCredentialException, FinderException,
            RemoteException;
    public String **findByPrimaryKey**(String customerID) throws FinderException,
    RemoteException;

    public boolean **getLocked**(String name)
            throws CustomerCredentialException, FinderException,
            RemoteException;

    public Collection **getCustomers**() throws FinderException,
    RemoteException;
    public CustomerVO **getCustomerByID**(String id) throws FinderException, RemoteException;
}
```

Listing 13 The bean implementation class for the customer session bean (CustomerSessionBean.java)

```
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
public class CustomerSessionBean implements SessionBean {
    private CustomerLocalHome **customerHome**;
    public int getTotalCustomers () {
        return customerHome.getTotalCustomers();
    }
    public void **createCustomer** (CustomerVO customer)
    throws CreateException, FinderException {
        if (customer.getName() == null || customer.getPassword () == null
            || customer.getEmail () == null) {
            throw new CreateException ("Customer data is null");
        }
        if (customer.getName() == "" || customer.getPassword () == ""
                || customer.getEmail () == "") {
                throw new CreateException ("Customer fields cannot be empty");
            }
        Collection customers = **customerHome.findByCustomerName** (customer.getName ());
        if (customers.size() == 0) {
            **customerHome.create** (customer.getName () , customer.getPassword (),
                    customer.getEmail (), customer.getLocked());
        } else {
            throw new CreateException (
                    "Customer name already in use.");
        }
    }
    public void **changePassword** (String name, String password)
            throws CustomerCredentialException, CustomerException, FinderException {
                if (password.equals("")) {
                    throw new CustomerException ("Password cannot be empty");
                }
                Collection customers = **customerHome.findByCustomerName** (name);
            if (customers.size() == 1) {
                Iterator i = c.iterator();
                CustomerLocal cust = (CustomerLocal) i.next ();
                cust.setPassword(password);
            } else {
                throw new CustomerCredentialException (
                        "Cannot find customer " + name);
            }
        }
public void changeEmail (String name, String email)
        throws CustomerCredentialException, CustomerException, FinderException {
            // similar to  changePassword
        }
public boolean authenticateCustomer (String name, String password)
        throws CustomerCredentialException, FinderException {
    boolean valid = false;
            Collection customers = customerHome.findByCustomerName (name);
            if (customers.size() == 1) {
                Iterator customer = customers.iterator();
                CustomerLocal cust = (CustomerLocal) customer.next ();
                if (!cust.getPassword().equals(password)) {
                    throw new CustomerCredentialException (
                            "Incorrect password for customer " + name);
                } else {
                    valid = true;
                }
            } else {
                throw new CustomerCredentialException (
                        "Cannot find customer " + name);
            }
            return valid;
        }
public String findByPrimaryKey (String customerID)
        throws FinderException {
            CustomerLocal customer = customerHome.findByPrimaryKey
                (customerID);
            if (customer != null) {
                return customer.getCustomerID();
            } else {
                return "not found";
            }
        }
public boolean getLocked (String name)
        throws CustomerCredentialException, FinderException {
            Collection customers = customerHome.findByCustomerName (name);
            if (customers.size() == 1) {
                Iterator customer = customers.iterator();
                CustomerLocal cust = (CustomerLocal) customer.next ();
                return cust.getLocked();
            } else {
                throw new CustomerCredentialException (
                        "Cannot find customer " + name);
            }
        }
public Collection getCustomers () throws FinderException {
    ArrayList customerList = new ArrayList ();
    Collection all = customerHome.findAll();
```

```
        Iterator customer = all.iterator();
        while (customer.hasNext()) {
            CustomerLocal customerEJB = (CustomerLocal) customers.next();
            CustomerVO customer = new CustomerVO (customerEJB.getName(),
                    customerEJB.getPassword(), .getEmail(),
                    customerEJB.getLocked());
            customerList.add (customer);
          }
          return customerList;
    }
    public CustomerVO getCustomerByID (String id)
          throws FinderException {
            CustomerLocal  cust = customerHome.findByPrimaryKey(id);
            CustomerVO customerVO = new CustomerVO (cust.getName(),
                  cust.getPassword(), cust.getEmail(), cust.getLocked());
            return customerVO;
          }
    public CustomerSessionBean () {}
    public void ejbCreate () {
        try {
            Context initial = new InitialContext ();
            Object objref = initial.lookup("java:comp/env/ejb/Customer");
            customerHome = (CustomerLocalHome) objref;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new EJBException (ex.getMessage());
        }
    }
    public void ejbRemove () {}
    public void ejbActivate () {}
    public void ejbPassivate () {}
    public void setSessionContext (SessionContext sc) {}
}
```

2.6 THE CUSTOMER VO AND EXCEPTION CLASSES

A *value object* (VO), or more properly *data transfer object* (DTO), is a helper class that contains getters and setters only for the attributes it represents. It is used to transfer data between layers or tiers. It encapsulates data into a *serializable* object so that it can be transported across JVMs in Java's context. The EJB examples presented in this chapter uses a `CustomerVO` class, which is shown in Listing 14.

Besides the `CustomerVO` helper class, the EJB samples presented in this chapter also require two application exception classes of `CustomerCredentialException.java` and `CustomerException.java`, as shown in Listings 15 and 16, respectively. Application exceptions are usually recoverable non-fatal errors caused by the mis-use of the application by the client. The EJB container does not perform any special error handling when an application exception occurs other than propagates it back to the client, indicating the error made by the client and giving the client a chance to re-try.

In contrast, system exceptions are generally unrecoverable, for example, a failure to obtain a database connection, the failure to load a class, or inserting a record with a conflicting key or null key, etc. System exceptions need to be investigated and fixed by the developer.

With our simple EJB examples, we have two application exception classes as stated above. If the input username/password pair does not match what is stored in the database, a `CustomerCredentialException` is thrown. A `CustomerException` is thrown for all other non-credential related errors associated with the EJB or underlying database.

We need an EJB deployment descriptor and a test client to run this EJB example, which are discussed next.

Listing 14 The customer VO class (CustomerVO.java)

```
public class CustomerVO implements java.io.Serializable {
    private String name;
    private String password;
    private String email;
    private boolean locked;
    public CustomerVO(String name, String password, String email, boolean locked) {
        super();
        this.name = name;
        this.password = password;
        this.email = email;
        this.locked = locked;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
// all other getters and setters omitted here to save space
}
```

Listing 15 The customer credential exception class (CustomerCredentialException.java)

```
public class CustomerCredentialException extends Exception {
    public CustomerCredentialException() {
    }

    public CustomerCredentialException(String msg) {
        super(msg);
    }
}
```

Listing 16 The customer exception class (CustomerException.java)

```
public class CustomerException extends Exception {
    public CustomerException() {
    }

    public CustomerException(String msg) {
        super(msg);
    }
}
```

2.7 The Deployment Descriptor for the BMP-Based Customer Entity and Session Beans

Listing 17 shows the EJB deployment descriptor for this BMP-based EJB example. The first part is for describing the session and entity enterprise beans, while the second part is the assembly descriptor, which is Jboss specific. The <session> element describes the customer session bean, while the <entity> element describes the customer entity bean. The `<home>`, `<remote>`, `<ejb-`

class>, <local-home> and <local> elements all must be specified with the fully qualified class name. The <ejb-local-ref> element defines the local access to the customer entity bean from the customer session bean.

For the customer entity bean, the persistence-type is specified as "*Bean*" as this is a BMP-based entity bean. In addition, it has a resource-ref element, which defines the data source for the customer entity bean. This entry must match the *datasource* defined on Jboss. We'll discuss how to set up Jboss to run this EJB example shortly.

The assembly descriptor element defines method permissions for these two customer EJBs according to how Jboss works (as you see that EJB itself is a standard, but its implementation may introduce vendor-specific features, which makes it problematic when you run an EJB application on EJB containers provided by different vendors). This assembly descriptor must stay in the ejb-jar.xml file and cannot be moved to the jboss.xml file, which is introduced next.

Listing 18 shows the *jboss.xml* file, which specifies that these two EJBs should run in the designated security domain. Once again, this is EJB vendor specific. We'll get back to this when we discuss the test client for this EJB example in the next section.

Listing 17 The EJB deployment descriptor for the BMP-based EJB example (ejb-jar.xml)

```
<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>CustomerSessionBean</display-name>
            <ejb-name>CustomerSession</ejb-name>
            <home>com.ejb2.test.CustomerSessionHome</home>
            <remote>com.ejb2.test.CustomerSession</remote>
            <ejb-class>com.ejb2.test.CustomerSessionBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>

            <ejb-local-ref>
                <ejb-ref-name>ejb/Customer</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <local-home>com.ejb2.test.CustomerLocalHome</local-home>
                <local>com.ejb2.test.CustomerLocal</local>
                <ejb-link>Customer</ejb-link>
            </ejb-local-ref>

            <security-identity>
                <description></description>
                <use-caller-identity></use-caller-identity>
            </security-identity>
        </session>
        <entity>
            <display-name>CustomerBean</display-name>
            <ejb-name>Customer</ejb-name>
            <home>com.ejb2.test.CustomerHome</home>
            <remote>com.ejb2.test.Customer</remote>
            <local-home>com.ejb2.test.CustomerLocalHome</local-home>
            <local>com.ejb2.test.CustomerLocal</local>
            <ejb-class>com.ejb2.test.CustomerBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>java.lang.String</prim-key-class>
            <reentrant>False</reentrant>
        <env-entry>
            <env-entry-name>CustomerDAOClass</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>com.ejb2.test.CustomerDAOMySQL</env-entry-value>
        </env-entry>
        <security-identity>
            <description></description>
            <use-caller-identity></use-caller-identity>
        </security-identity>

        <resource-ref>
            <res-ref-name>DefaultDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>
    </entity>
</enterprise-beans>
<assembly-descriptor>
    <method-permission>
        <!--unchecked/ -->
        <role-name>JBossAdmin</role-name>
        <method>
            <ejb-name>Customer</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <method-permission>
        <!--unchecked/ -->
        <role-name>JBossAdmin</role-name>
        <method>
            <ejb-name>CustomerSession</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <container-transaction>
        <method>
            <description>getTotalCustomers</description>
            <ejb-name>CustomerSession</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getTotalCustomers</method-name>
            <method-params />
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
        <description></description>
        <method>
            <description>createCustomer</description>
            <ejb-name>CustomerSession</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>createCustomer</method-name>
            <method-params>
                <method-param>CustomerVO</method-param>
            </method-params>
        </method>
        <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Listing 18 jboss.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
    <security-domain>java:/jaas/jmx-console</security-domain>
    <enterprise-beans>
        <session>
            <ejb-name>CustomerSession</ejb-name>
            <local-jndi-name>com.ejb2.test.CustomerSessionLocal</local-jndi-name>
        </session>
        <entity>
            <ejb-name>Customer</ejb-name>
            <local-jndi-name>com.ejb2.test.CustomerLocal</local-jndi-name>
        </entity>
    </enterprise-beans>
</jboss>
```

2.8 INSTALLING JBOSS TO RUN THE BMP-BASED CUSTOMER EJB EXAMPLE

To run this EJB example, download Jboss 5 or 6 App Server and unzip it to your system. I have verified that the EJB 2 examples presented in this chapter work both on *jboss-5.0.1.GA* and *jboss-6.1.0.Final*. However, you need to perform the following tasks to configure your Jboss App Server:

1 Modify the *mysql-ds.xml* file in the *default/conf* folder as shown in Listing 19. The *jndi-name* needs to be set to *DefaultDS*, as many Jboss services expect this name. If you want to change it to something like *MySQLDS*, you will have to manually change the settings in many xml files, which is too tedious and problematic. In addition, you need to specify the matching connection URL, username and password. The "*ejb*" part in the connection-url element is the name of the database we created earlier for this sample.
2 Rename *hsqldb-ds.xml* to something like *hsqldb-ds.xml_save* to prevent the EJB container from loading this datasource. Note that if you rename it to something like *hsqldb-ds_save.xml* or anything else with the ".xml" extension, Jboss will still load it as Jboss tends to load resources defined in any file with the ".xml" extension.
3 Edit the *standardjbosscmp-jdbc.xml* file in the *default/conf* folder to add the line of `<datasource-mapping>mySQL</datasource-mapping>` below the `<datasource>` element.
4 Make sure you have the *mysql-connector-java-5.1.18-bin.jar* file in the *default/lib* folder.
5 Make sure you have set your JAVA_HOME and ANT_HOME environment variable.
6 Open up a command prompt and change to the *bin* folder of your Jboss install. Execute the following two commands to set the Jboss *env* variables (note that you need to replace my jboss path with your own):

```
bin>set JBOSS_HOME=c:\mspc\app\jboss-6.1.0.Final
bin>set JBOSS_SERVER_CONFIG=default
```

7 Make sure that your *jmx-console-roles.properties* file in the *default/conf/props* folder contains the following line (recall `JBossAdmin` role defined in the *ejb-jar.xml* file shown in Listing 17):

```
admin=JBossAdmin,HttpInvoker
```

8 Make sure that your *jmx-console-users.properties* file in the *default/conf/props* folder contains the following line:

```
admin=admin
```

9 Execute the command of "*run –b 0.0.0.0*" to start up your Jboss server. Next, we describe how to compile, package and run this EJB 2 sample with a test client introduced following Listing 19.

Items #7 and #8 are associated with the *application-policy* for the *jmx-console* security-domain, defined in the *login-config.xml* file in the *default/conf* folder (recall how the security domain is defined in Listing 18 *jboss.xml*):

```
<application-policy name="jmx-console">
<authentication>
<login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
   flag="required">
   <module-option name="usersProperties">props/jmx-console-
   users.properties</module-option>
    <module-option name="rolesProperties">props/jmx-console-
   roles.properties</module-option>
 </login-module>
 </authentication>
 </application-policy>
```

Jboss recommends using a stronger authentication mechanism in production environment.

Listing 19 mysql-ds.xml (in default/conf) for Jboss app server 5 and 6

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
 <local-tx-datasource>
  <jndi-name>DefaultDS</jndi-name>
  <connection-url>jdbc:mysql://localhost:3306/ejb</connection-url>
  <driver-class>com.mysql.jdbc.Driver</driver-class>
  <user-name>root</user-name>
  <password>MySql5522</password>
  <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter</exception-sorter-class-
name>
  <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional) -->
  <metadata>
    <type-mapping>mySQL</type-mapping>
  </metadata>
 </local-tx-datasource>
</datasources>
```

2.9 RUNNING THE BMP-BASED EJB2 EXAMPLE

Listing 20 shows the test driver for this BMP-based customer EJB example. First, note how the initial context set up in a Jboss specific way in the `getInitialContext` () method. Then, in the `bmpTest` () method, the below standard protocol is followed:

1 Look up the Customer Entity Bean with the statement of

```
Object obj = ctx.lookup ("Customer"),
```

which uses the bean name of Customer defined in the deployment descriptor shown in Listing 17.

2 Create a handle to the home interface of the customer entity bean using the following statement:

```
CustomerHome home = (CustomerHome)
          PortableRemoteObject.narrow(obj, CustomerHome.class)
```

3 Call EJB's methods to create and find customers using the statements like `home.create` (…), `home.findCustomerByPrimaryKey` (…), etc. Note that you need to replace the primary key hard-coded into this test client with one that exists in your own database.

For the `testSession` (…) method, the protocol is similar except that:

- It looks up the `CustomerSession` bean instead of the `Customer` entity bean.
- It uses the business methods defined in the customer session bean's remote interface rather than its home interface.

To run the ejb2BMPDriver test client, follow the below procedure:

1 Open up a command prompt and change to the project folder of this EJB 2 example in your Eclipse workspace.
2 Set the JBOSS_HOME and JBOSS_SERVER_CONFIG environment variable as shown in the previous section.
3 If your Jboss server is not up and running, start it up by following the instructions given in the preceding section.
4 Execute the following command to deploy this EJB 2 sample to the *default* folder of your Jboss server:

```
cmd>%ANT_HOME%\bin\ant deploy
```

5 Wait until the `Customer` entity bean and `CustomerSession` bean have been deployed successfully in the EJB container of your Jboss server. You can use the Jboss admin console as shown in Figure 1 to verify if the EJBs and the `DefaultDS` datasource have been successfully loaded into your Jboss EJB container. You can log into the Jboss admin console at http://localhost:8080 with the *admin/admin* username/password.
6 Execute the following command to execute the test client:

```
cmd>%ANT_HOME%\bin\ant run-ejb2BMPDriver
```

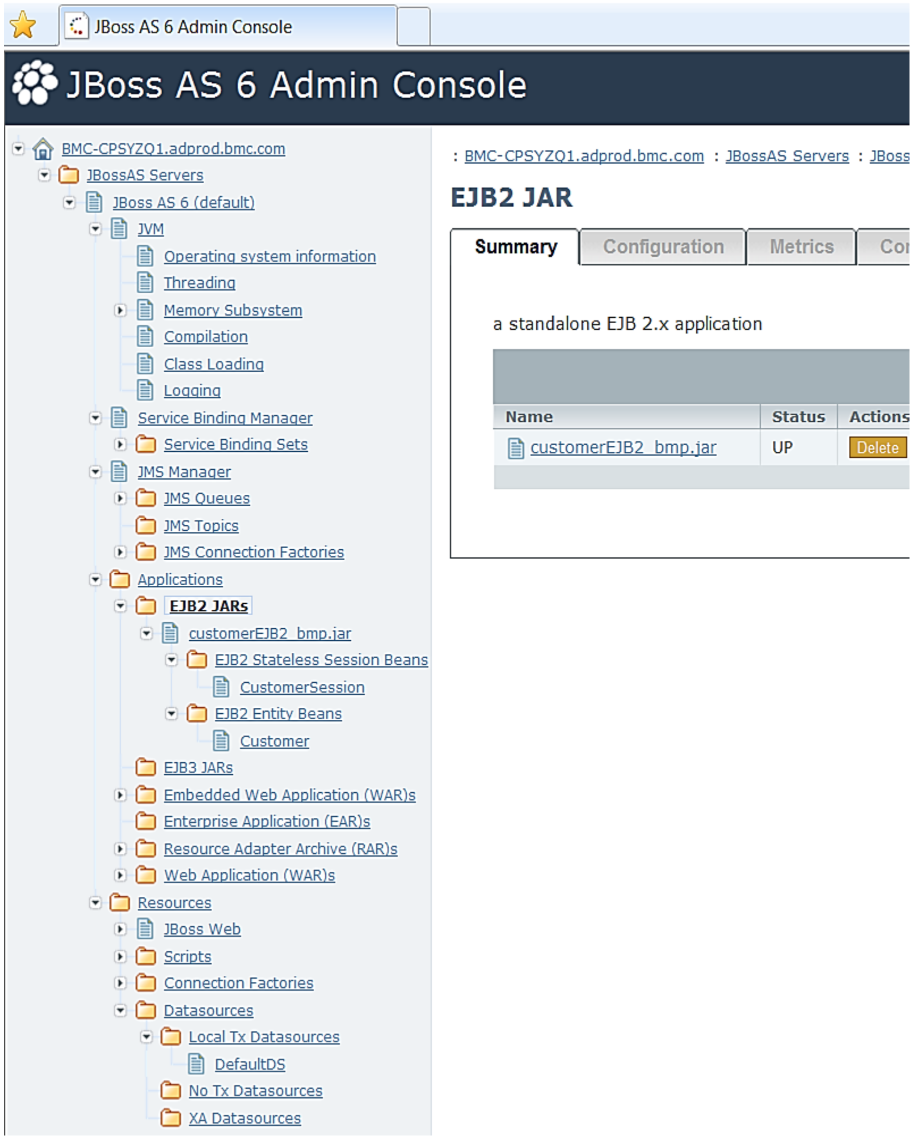
If everything goes well, you should see your console output similar to what I got in my environment as shown in Listing 21. Note that the `bmpTest` method took 281 milliseconds while the `sessionTest` method took 47 milliseconds. Does this mean that calling the customer entity bean by its local interface via a session bean is much faster than calling its remote interface directly in the client? This is left as an exercise at the end of this chapter.

Before wrapping up our BMP-based EJB 2 example and proceeding to the CMP-based EJB 2 example next, keep in mind that when you modify the client program, you don't have to re-deploy the EJBs. You only need to run "*ant compile*" and re-run the test client. However, after you modify the EJB classes, you need to redeploy the EJBs by running "*ant deploy*."

Listing 20 The test driver for the BMP-based customer EJB example

```
package com.ejb2.test;
import java.util.*;
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;

public class ejb2BMPDriver {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        System.out.println("testing Customer EJB2");
        InitialContext ctx = null;
        try {
            ctx = getInitialContext();
            String customerID = bmpTest(ctx);
            sessionTest(ctx, customerID);
        } catch (Exception ex) {
            System.out.println("Error in establishing initial context: "
                    + ex.getMessage());
        }
        long endTime = System.currentTimeMillis();
        System.out.println("total time: " + (endTime - startTime) / 1000.0
                + " seconds");
    }
    public static String bmpTest(InitialContext ctx) {
        System.out
                .println("testing CustomerEJB2/BMP by calling the entity bean directly");
        long startTime = System.currentTimeMillis();
        String customerID = null;
        try {
            Object obj = ctx.lookup("Customer");
            CustomerHome home = (CustomerHome) PortableRemoteObject.narrow(obj,
                    CustomerHome.class);
            String customerName = "c" + (new RandomID(10).getId());
            boolean locked = false;
            System.out.println("testing CustomerEJB2<BMP>: creating customer with name "
                    + customerName);
            Customer customer = home.create(customerName, "password",
                    "customer0@abc.com", locked);
            customerID = customer.getCustomerID();
            System.out.println("customer with ID " + customerID + " created");
            Collection <Customer> customers = home.findAll();
            String primaryKey = null;
            System.out.println ("findAll found : " + customers.size() + " customers");
            Iterator<Customer> cust = customers.iterator();
            while (cust.hasNext ()) {
                Customer c = (Customer) cust.next();
                primaryKey = c.getCustomerID();
                System.out.println(c.getName() + ": "
                        + c.getPassword());
            }
            System.out.println("testing CustomerEJB2: findByPrimaryKey");
            Customer findPK = home.findByPrimaryKey(primaryKey);
            findPK.setPassword("New Password");
            System.out.println("name = " + findPK.getName());

            System.out.println(home.getTotalCustomers()
                    + " customers in database.");

        } catch (CreateException ex) {
            System.err.println("Error creating customer: " + ex.getMessage());
        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception: "
                    + ex.getMessage());
        }
        System.out.println("bmpTest execution time: "
                + (System.currentTimeMillis() - startTime) / 1000.0
                + " seconds");
        return customerID;
    }
    public static void sessionTest(InitialContext ctx, String customerID) {
        System.out.println("\nsesionBeanTest:");
        long startTime = System.currentTimeMillis();
        try {
            Object obj = ctx.lookup("CustomerSession");
    CustomerSessionHome home = (CustomerSessionHome) PortableRemoteObject
            .narrow(obj, CustomerSessionHome.class);
    String customerName = (new RandomID(10).getId());
    System.out.println("testing CustomerEJB2: creating customer_"
            + customerName);

    CustomerVO customer = new CustomerVO(customerName, "password",
            "customer0@abc.com", false);
    CustomerSession customerSession = home.create();
    try {
        customerSession.createCustomer(customer);
    } catch (CreateException ex) {
        System.out.println("createCustomer error: " + ex.getMessage());
    }
    System.out.println("testing CustomerEJB2: getCustomers");
    Collection<CustomerVO> customers = customerSession.getCustomers();
    String name = null;
    String password = null;
    int count = 0;
    Iterator<CustomerVO> cust = customers.iterator();
    while (cust.hasNext()) {
        CustomerVO c0 = (CustomerVO) cust.next();
        if (count < 1) {
            name = c0.getName();
            password = c0.getPassword();
            count = 1;
            System.out.println(name + ": " + password);
        }
    }
    System.out.println("testing getCustomerByID:");
    try {
        CustomerVO customerVO = customerSession
                .getCustomerByID(customerID);
        System.out.println("getCustomerByID result: " + customerID);
    } catch (FinderException ex) {
        System.out
                .println("findByPrimaryKey error: " + ex.getMessage());
    }
    System.out.println("testing autheticateCustomer:");
    try {
        customerSession.authenticateCustomer(name, password);
        System.out.println("authenticating customer succeeded.");
    } catch (CustomerCredentialException ex) {
        System.out.println("authenticateCustomer error: "
                + ex.getMessage());
    }
    System.out.println("testing getTotalCustomers:");
    int totalNumOfCustomers = customerSession.getTotalCustomers();
    System.out.println("total number of customers in database: "
            + totalNumOfCustomers);
} catch (CreateException ex) {
    System.err.println("Error creating object.");
    System.err.println(ex.getMessage());
} catch (Exception ex) {
    System.err.println("Caught an unexpected exception.");
    System.err.println(ex.getMessage());
}
```

```
        long endTime = System.currentTimeMillis();
        System.out.println("sessionTest execution time: "
                + (endTime - startTime) / 1000.0 + " seconds");
    }
    protected static InitialContext getInitialContext() throws NamingException {
        Hashtable<String, String> env = new Hashtable<String, String>();
        env.put("java.naming.factory.initial",
                "org.jboss.security.jndi.JndiLoginInitialContextFactory");
        env.put("java.naming.provider.url", "localhost:1099");
        env.put(Context.SECURITY_PRINCIPAL, "admin");
        env.put(Context.SECURITY_CREDENTIALS, "admin");
        return new InitialContext(env);
    }
}
```



EJB2 JAR

a standalone EJB 2.x application

Name	Status	Actions
customerEJB2_bmp.jar	UP	Delete

Figure 1 Jboss admin console showing the BMP-based customer entity bean and the customer session bean have been deployed in the Jboss EJB containter successfully together with the MySQL datasource of DefaultDS

Listing 21 Output of running the BMP-based EJB 2 example test client

run-ejb2BMPDriver:

```
    [java] testing Customer EJB2
    [java] testing CustomerEJB2/BMP by calling the entity bean directly
    [java] testing CustomerEJB2<BMP>: creating customer with name c519394520
    [java] customer with ID 464591459 created
    [java] findAll found : 7 customers
    [java] c520356325: password
……
    [java] 602732326: New Password
    [java] testing CustomerEJB2: findByPrimaryKey
    [java] name = 602732326
    [java] 7 customers in database.
    [java] bmpTest execution time: 0.281 seconds
    [java]
    [java] sesionBeanTest:
    [java] testing CustomerEJB2: creating customer_555536964
    [java] testing CustomerEJB2: getCustomers
    [java] c520356325: password
    [java] testing getCustomerByID:
    [java] getCustomerByID result: 464591459
    [java] testing autheticateCustomer:
    [java] authenticating customer succeeded.
    [java] testing getTotalCustomers:
    [java] total number of customers in database: 8
    [java] sessionTest execution time: 0.047 seconds
    [java] total time: 0.453 seconds

BUILD SUCCESSFUL
Total time: 1 second
```

3 A CMP-Based EJB 2 Example

Given what we have covered on BMP-based entity beans, it's a lot easier to explain CMP-based entity beans. In fact, the home and remote interfaces in general do not have to change much when converting an entity bean from BMP-based to CMP-based. The differences are in the details of the bean implementation class and the deployment descriptor, as discussed in this section. Since we have given detailed explanations on the concepts of the entity beans and session beans in the previous sections, textual descriptions in this section will be brief. We will only focus on the changes caused by converting from BMP-based to CMP-based entity bean.

3.1 THE CUSTOMER ENTITY BEAN

For the CMP-based customer entity bean, we have eliminated the remote home interface and remote interface, as the customer entity bean will be accessed by the custom session bean locally only. Therefore, we only list the local home interface and local interface in Listings 22 and 23, respectively. These two interfaces are identical to their counterparts from the BMP-based customer entity bean interfaces shown in Listings 6 and 7, respectively.

However, the persistence mechanism change has caused drastic changes to the customer bean implementation class from BMP-based to CMP-based. By comparing Listing 24 for the CMP-based customer bean implementation class with Listing 8 for the BMP-based customer bean implementation class, we notice the following changes:

- The bean implementation class is declared abstract now.
- The size of the source code for the customer bean implementation class has dropped from 174 lines to 56 lines, which means much less coding for the bean developer.
- The # of entity bean variables has dropped from four to one, with only the `EntityContext` variable retained while having all other three of `CustomerModel`, `CustomerDAO` and `dirty` flag eliminated.

- The business methods are now purely getters and setters without having the dirty flag set explicitly for the setters. However, all getters and setters must be declared *abstract* and the container would generate the Java persistence attributes from these methods. For example, for the `getName` ()/`setName` (`String name`) pair, the container would generate an attribute of *name* typed *String*. It's mandatory to follow this naming convention so that the container would generate proper attributes.
- The `ejbCreate` method is much simpler now. It does not have to get involved with the `DAO` and `CustomerModel` classes any more, as the container has taken away the functions associated with these classes, with the help of those abstract getters and setters. Incidentally, all the DAO related classes are gone now.
- All other EJB methods like `ejbActivate`, `ejbPassivate`, `ejbLoad`, `ejbStore`, `jbPostCreate`, and `ejbRemove` are essentially dummies now.
- You might notice that all EJB finder methods are gone from the customer bean implementation class. They will be generated by the container with the help of EJB Query Language (QL). We will discuss more about this when we get to the section on the deployment descriptor for the CMP-based entity bean later.

Next, we discuss the customer session bean following the end of Listing 24.

Listing 22 Local home interface for the CMP-based customer entity bean

```
import java.util.Collection;
import javax.ejb.*;

public interface CustomerLocalHome extends EJBLocalHome
{
   public CustomerLocal create(String name, String password, String email,
boolean locked) throws CreateException;
   public CustomerLocal findByPrimaryKey (String customerID) throws FinderException;
   public Collection findByCustomerName (String name) throws FinderException;
   public Collection findAll () throws FinderException;
   public int getTotalCustomers () ;
}
```

Listing 23 Local interface for the CMP-based customer entity bean

```
import javax.ejb.*;

public interface CustomerLocal extends EJBLocalObject
{
   public String getCustomerID ();
   public String getName ();
   public void setName (String newName);
   public String getPassword ();
   public void setPassword (String newPassword);
   public String getEmail ();
   public void setEmail (String newEmail);
   public boolean getLocked ();
   public void setLocked (boolean locked);
}
```

Listing 24 Bean implementation class for the CMP-based customer entity bean

```
import java.util.*;
import javax.ejb.*;
public abstract class CustomerBean implements EntityBean {
     private EntityContext context;
```

```
    public abstract String getCustomerID();
    public abstract void setCustomerID(String id);
    public abstract String getName();
    public abstract void setName(String name);
    public abstract String getPassword();
    public abstract void setPassword(String password);
    public abstract String getEmail();
    public abstract void setEmail(String email);
    public abstract boolean getLocked();
    public abstract void setLocked(boolean locked);
    public abstract Collection ejbSelectTotalCustomers () throws FinderException;

    public int ejbHomeGetTotalCustomers ()
            throws FinderException {
            return ejbSelectTotalCustomers ().size();
        }
    public String ejbCreate (String name, String password,
            String email, boolean locked) throws CreateException {
            String newKey = (new RandomID(10)).getId();
            setCustomerID (newKey);
            setName (name);
            setPassword (password);
            setEmail (email);
            setLocked (locked);
            return newKey;
        }
    public void setEntityContext (EntityContext context) {
        this.context = context;
    }
    public void unsetEntityContext () { }
    public void ejbActivate () { }
    public void ejbPassivate () { }
    public void ejbLoad () { }
    public void ejbStore () { }
    public void ejbRemove () { }
    public void ejbPostCreate (String name, String password,
            String email, boolean locked)  {
        }
}
```

3.2 THE CUSTOMER SESSION BEAN

The customer session bean home interface, remote interface and bean implementation class for the new CMP-based customer entity bean remain the same as for the previous BMP-based EJB example, since we have not added any new business methods. Therefore, they are not listed here. You can refer back to Listings 11 through 13 for the home interface, remote interface and bean implementation class for the stateless customer session bean to be used with this CMP-based EJB 2 example.

3.3 THE DEPLOYMENT DESCRIPTOR

In addition to the same *jboss.xml* file as shown in Listing 25, we need to add more to the deployment descriptor that we used for the previous BMP-based EJB 2 example. Listing 26 shows the new descriptor for the CMP-based EJB 2 example. The <session> element for the customer session bean remains the same, as we are using the same customer session bean as for the BMP-based EJB 2 example. We highlighted the newly added tags to identify the changes caused by converting the customer entity bean from BMP-based to CMP-based (you can compare this listing with Listing 17 to verify the changes). Here is a summary of what have been changed and added:

- The `persistence-type` is now changed from `Bean` to `Container` to signify that this bean is CMP-based.
- An abstract persistence schema is defined for the customer entity bean. The name of this schema is "customers*"*.
- The `<primkey-field>` tag and `<cmp-field>` tags are used to define the primary key field and all other fields.
- Two `<query>` tags are added: one for the `ejbSelectTotalCustomers` method and the other for the `findByCustomerName` method. The `ejbSelectTotalCustomers` method is declared in the `CustomerBean.java` file as an abstract method, which requires this mapping defined with the `<method-name>` and `<ejb-ql>` tags here. The `findByCustomerName` method is defined in the local home interface of the customer entity bean, which requires this mapping defined with the `<method-name>` and `<ejb-ql>` tags here as well. Note that we must use the "table" name of `customers` as defined by the `<abstract-schema-name>` tag in those queries.

However, it's not sufficient to have this ejb-jar.xml file alone. We must have an EJB-vendor specific file, which, in the case of Jboss, is the `jbosscmp-jdbc.xml` file as shown in Listing 27. It contains a `<defaults>` element, which defines a way for the container to find which `datasource` to use.

With all these pieces in place, we are ready to run this CMP-based EJB example. The required test client program and the procedure for running this example on Jboss are given in the next section.

Listing 25 The jboss.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
    <security-domain>java:/jaas/jmx-console</security-domain>
    <enterprise-beans>
        <session>
            <ejb-name>CustomerSession</ejb-name>
            <local-jndi-name>com.ejb2.test.CustomerSessionLocal</local-jndi-name>
        </session>
        <entity>
            <ejb-name>Customer</ejb-name>
            <local-jndi-name>com.ejb2.test.CustomerLocal</local-jndi-name>
        </entity>
    </enterprise-beans>
</jboss>
```

Listing 26 The deployment descriptor for the CMP-based EJB 2 Example

```
<!DOCTYPE ejb-jar PUBLIC
  "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
  "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>CustomerSessionBean</display-name>
            <ejb-name>CustomerSession</ejb-name>
            <home>com.ejb2.test.CustomerSessionHome</home>
            <remote>com.ejb2.test.CustomerSession</remote>
            <ejb-class>com.ejb2.test.CustomerSessionBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <ejb-local-ref>
                <ejb-ref-name>ejb/Customer</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.ejb2.test.CustomerLocalHome</local-home>
            <local>com.ejb2.test.CustomerLocal</local>
            <ejb-link>Customer</ejb-link>
        </ejb-local-ref>
        <security-identity>
            <description></description>
            <use-caller-identity></use-caller-identity>
        </security-identity>
    </session>
    <entity>
        <display-name>CustomerBean</display-name>
        <ejb-name>Customer</ejb-name>
        <local-home>com.ejb2.test.CustomerLocalHome</local-home>
        <local>com.ejb2.test.CustomerLocal</local>
        <ejb-class>com.ejb2.test.CustomerBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>

        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>customers</abstract-schema-name>
        <cmp-field>
            <description>no description</description>
            <field-name>name</field-name>
        </cmp-field>
        <cmp-field>
            <description>no description</description>
            <field-name>password</field-name>
        </cmp-field>
        <cmp-field>
            <description>no description</description>
            <field-name>email</field-name>
        </cmp-field>
        <cmp-field>
            <description>no description</description>
            <field-name>customerID</field-name>
        </cmp-field>
        <cmp-field>
            <description>no description</description>
            <field-name>locked</field-name>
        </cmp-field>
        <primkey-field>customerID</primkey-field>
        <security-identity>
            <description></description>
            <use-caller-identity></use-caller-identity>
        </security-identity>
        <query>
            <description></description>
            <query-method>
                <method-name>ejbSelectTotalCustomers</method-name>
                <method-params />
            </query-method>
            <ejb-ql>select object (c) from customers c</ejb-ql>
        </query>
        <query>
            <query-method>
```

```
                    <method-name>findByCustomerName</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <ejb-ql>select distinct object (c) from customers c
where c.name = ?1</ejb-ql>
            </query>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
            <method-permission>
            <role-name>JBossAdmin</role-name>
            <method>
                <ejb-name>Customer</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <role-name>JBossAdmin</role-name>
            <method>
                <ejb-name>CustomerSession</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <method>
                <description>getTotalCustomers</description>
                <ejb-name>CustomerSessionBean</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>getTotalCustomers</method-name>
                <method-params />
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    <container-transaction>
            <description></description>
            <method>
                <description>createCustomer</description>
                <ejb-name>CustomerSession</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>createCustomer</method-name>
                <method-params>
                    <method-param>CustomerVO</method-param>
                </method-params>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Listing 27 The jbosscmp-jdbc.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jbosscmp-jdbc PUBLIC
  "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
  "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
```

```
<jbosscmp-jdbc>
    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>mySQL</datasource-mapping>
        <create-table>false</create-table>
        <remove-table>false</remove-table>
        <pk-constraint>true</pk-constraint>
        <read-ahead>
            <strategy>on-load</strategy>
            <page-size>4</page-size>
            <eager-load-group>*</eager-load-group>
        </read-ahead>
    </defaults>
</jbosscmp-jdbc>
```

3.4 Running the CMP-Based EJB 2 Example

Before you run this CMP-based EJB2 example on an EJB container like Jboss, you need to install and configure Jboss by following the instructions given in §2.8. You also need to remove the jar file for the previous BMP-based EJB 2 example in the `default/deploy` folder as it will conflict with this example.

We need a new driver for this CMP-based EJB 2 example, though, which is shown in Listing 28. This test driver is essentially the same as the one we discussed in §2.9 for the BMP-based EJB 2 example except that the `bmpTest` method is commented out since we are not interested in calling the customer session bean remotely any more. For this reason, we would not list this test driver here.

To run this example, simply follow the similar procedure given in §2.9 except that the command to issue should use the name of this test driver as follows:

```
cmd>%ANT_HOME%\bin\ant run-ejb2CMPDriver
```

If everything goes well in your environment, you should get your console output similar to what I got in my environment as shown in Listing 28. This wraps up our CMP-based EJB 2 example.

Listing 28 Output of running the CMP-based EJB 2 example

```
run-ejb2CMPDriver:
run-ejb2CMPDriver:
    [java] testing Customer EJB2
    [java] sesionBeanTest:
    [java] testing CustomerEJB2: creating customer_645425785
    [java] testing CustomerEJB2: getCustomers
    [java] c520356325: password
    [java] testing CustomerEJB2: getCustomerByID
    [java] getCustomerByID result: 645425785
    [java] testing getTotalCustomers:
    [java] total number of customers in database: 11
    [java] sessionTest execution time: 0.203 seconds
    [java] total time: 0.297 seconds
BUILD SUCCESSFUL
Total time: 1 second
```

4 Summary

This chapter reviewed the EJB 2 framework retrospectively with two working examples: one uses a BMP-based customer entity bean and the other uses a CMP-based customer entity bean. I hope you do not feel that this is irrelevant material, as Spring emerged as a lighter-weight framework out of a heavy-weight framework like EJB 2. This should motivate you to continue to hone your Spring skills for a more rewarding personal career path.

5 Recommended Reading

Since EJB 2 is outdated technology, I do not recommend anyone to spend too much time on it. If you are on EJB, you should move forward to EJB 3, which has abandoned the entity bean completely and replaced it with Hibernate, JPA and annotations. The gap between Spring and EJB 3 is much smaller now, which is a tremendous benefit for enterprise Java application developers.

6 Exercises

1 In §2.9, the BMP-based EJB 2 example run shows that the method of calling the customer entity bean directly via its remote interface took 281 milliseconds while the method of calling its local interface via a stateless session bean took 47 milliseconds. Is that large difference caused by local versus remote access to the customer entity bean?

2 Implement the same EJB sample with Spring 3.2 as we introduced in this book. You should use the same customer table created on MySQL. Quantify the simplifications from using Spring over EJB 2.